The Cranium Network Interface Architecture: Support for Message Passing on Adaptive Packet Routing Networks

by

Neil R. McKenzie

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

University of Washington

1997

Approved by_____

(Chairperson of Supervisory Committee)

_____

_____

Program Authorized
to Offer Degree Computer Science and Engineering

Date January 31, 1997

In presenting this dissertation in partial fulfillment of the requirements for the Doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to University Microfilms, 1490 Eisenhower Place, P.O. Box 975, Ann Arbor, MI 48106, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform."

Signature_____

Date_____

University of Washington

Abstract

The Cranium Network Interface Architecture: Support for Message Passing on Adaptive Packet Routing Networks

by Neil R. McKenzie

Chairperson of Supervisory Committee: Professor Carl Ebeling
Department of Computer Science
and Engineering

Cranium is a network interface architecture for message passing in a scalable parallel computer system. Cranium provides the following features:

- *Support for adaptive networks.* Cranium is compatible with adaptive networks that permit packets to overtake other packets in transit. The network interface notifies the processor after an entire message is received, independent of the arrival order of its individual packets.

- *User-level bus-master DMA for both low latency and high throughput.* Cranium connects at the processor-memory bus to bypass the bottleneck at the I/O bus, without incurring the complexity of a design that is tightly coupled to the processor. Direct access by user programs avoids the overhead of operating system calls.

- *Support for both short and long messages.* Message traffic in networks is bi-modal: most messages are short, but long messages consume most network bandwidth. Cranium directly supports messages ranging in length from a cache line up to an MMU page. Buffered communication for small messages is supported via a queue that is mapped into the user's address space. For large

messages, unbuffered communication is implemented using automatic-receive DMA to place packet data into the user's memory without copying.

Cranium provides high communication performance as well as excellent speedup of parallel application programs. The Cranium application program interface introduces minimal software overhead, yet provides functionality similar to a heavily-layered approach such as Intel NX.

Analysis of Cranium's performance shows that the end-to-end latency of a short message is 60 to 100 clock cycles; throughput is 90% of peak with messages as short as 2K bytes. Cranium is evaluated empirically using a suite of hand-coded parallel benchmark programs that run on the Talisman multiprocessor simulator and the Chaos network simulator.

To demonstrate that a Cranium chip can be fabricated, we describe Teschio, an ASIC implementation of Cranium. Teschio is estimated to require fewer than 400,000 gates and support a clock frequency of 100 MHz. Many implementation tradeoffs are explored in the design of Teschio: performance of the memory system, depth of FIFO buffers and support for multiple user contexts.

Table of Contents

List of Figures vii

List of Tables xi

Chapter 1: Introduction 1

1.1 MPP computers and scalable networks 4
1.2 The communication bottleneck 5
1.3 The role of the network interface in message passing 7
1.4 Packet length and routing algorithms 11
1.5 Performance of networks and network interfaces 13
1.6 Related work 16
1.7 The thesis 17

Chapter 2: Network interfaces 20

2.1 The physical interface 21
2.1.1 Coupling the physical interface with the processing node . . . 21
2.1.2 Data movement 25
2.1.3 Notification and dispatch 28
2.1.4 Argument checking and protection 29
2.1.5 Fault handling 34
2.2 The logical interface 35
2.2.1 Systolic communication 36
2.2.2 Remote memory 36
2.2.3 Send/receive communication 40
2.2.4 Protocol support 40
2.3 Analysis 42
2.4 Summary 45

Chapter 3: The Cranium network interface architecture 47
3.1 Design goals . . . 47
3.2 The difficulty of interfacing with adaptive routers . . . 48
3.3 Cranium implementation-independent architecture . . . 51
3.3.1 Send channels and auto-receive channels . . . 52
3.3.2 Queuing channels . . . 53
3.3.3 Protection . . . 55
3.3.4 Error handling in Cranium . . . 57
3.4 Cranium implementation-dependent architecture . . . 58
3.4.1 The Cranium scheduler . . . 58
3.4.2 Support for cache coherence . . . 59
3.4.3 Multiple user contexts . . . 61
3.4.4 Gather-scatter support . . . 63
3.5 Summary . . . 65

Chapter 4: The Cranium software interface 67
4.1 Sending a message under Cranium . . . 68
4.2 Receiving a message under Cranium . . . 70
4.2.1 Unbuffered communication . . . 70
4.2.2 Buffered communication . . . 70
4.3 Synchronization . . . 71
4.4 Interrupts and error diagnostics . . . 74
4.5 Comparison with other message passing interfaces . . . 74
4.5.1 Intel NX . . . 74
4.5.2 Active messages . . . 77
4.6 Summary . . . 78

Chapter 5: The test environment 80
5.1 Talisman . . . 82
5.1.1 Talisman's timing model . . . 83
5.2 Chaos network simulator . . . 86
5.3 Modeling the behavior of Cranium in the simulator . . . 89

5.3.1 Injection and delivery of packets 90
5.4 Implementation: integrating the simulators 90
5.5 Running the combined simulator 94
5.6 Host execution performance of the combined simulator 95
5.7 Summary 98

Chapter 6: Evaluation of Cranium **100**
6.1 Performance analysis 101
6.1.1 Latency of a single packet 103
6.1.2 Throughput 105
6.1.3 Broadcast 108
6.2 Empirical evaluation of Cranium 112
6.2.1 Goals 113
6.2.2 Suite of parallel benchmark programs 114
6.2.3 Benchmark implementation 120
6.2.4 Benchmark measurements 121
6.2.5 Determining maximum speedup and efficiency 123
6.2.6 Determining the significance of the communication cost 126
6.2.7 Putting it all together 128
6.2.8 Performance of the communication system 128
6.2.9 Summary 132
6.3 Comparing Cranium against other network interface styles 134
6.3.1 Modifying Cranium to emulate other network interfaces 135
6.3.2 Analytical evaluation of the modifications 136
6.3.3 Empirical evaluation of the modifications 137
6.3.4 Summary 139
6.4 Related work 140
6.4.1 Study #1: CM-5 vs. Paragon 140
6.4.2 Study #2: CM-5 vs. J-machine vs. Star-T 141
6.4.3 Study #3: Meerkat vs. Delta 141
6.5 Summary 142

Chapter 7: Teschio: A VLSI chip implementation of Cranium 144
7.1 Teschio system environment 146
7.1.1 Environment of a single node 146
7.1.2 The ADU bus 147
7.1.3 The P link 148
7.1.4 Node mapping 149
7.1.5 Data redundancy 149
7.2 Teschio internal structure 151
7.2.1 Core module 153
7.2.2 Inbox and outbox 155
7.3 Internal micro-operations of Teschio 158
7.3.1 Command and status interface with the host processor 159
7.3.2 Performing the table lookup functions 160
7.3.3 Sending a packet 162
7.3.4 Receiving a packet 164
7.3.5 Updating the internal data structures 167
7.4 Timing analysis 170
7.4.1 Design guidelines 171
7.4.2 Latency 171
7.4.3 Throughput 175
7.4.4 Summary 178
7.5 Fabrication parameters for Teschio 178
7.6 Extensions to Teschio 181
7.6.1 Packet scheduling and traffic shaping 181
7.6.2 Fast context switching 182
7.6.3 Gather-scatter support 183
7.7 Summary 185

Chapter 8: Conclusions 187
8.1 Cranium architecture 187
8.2 Cranium application program interface 189
8.3 Test environment 189

8.4 Performance analysis . . . 190
8.5 Empirical results . . . 190
8.6 Teschio . . . 191
8.7 Contributions of this dissertation . . . 191
8.8 Future work . . . 192
8.8.1 The Chaos-LAN project . . . 192
8.8.2 Additional performance studies . . . 194
8.9 Closing thoughts . . . 194
8.9.1 Message passing vs. shared memory . . . 194
8.9.2 The road ahead . . . 195

Bibliography **196**

Appendix A: The Cranium application programmer's interface **208**
A.1 Application program interface to the OS for message passing . . . 210
A.1.1 Initialization information . . . 210
A.1.2 Allocation and deallocation of DMA buffers . . . 211
A.1.3 Interrupt handler . . . 212
A.2 Application program interface to Cranium . . . 212
A.2.1 Packet header . . . 212
A.2.2 Command word . . . 214
A.2.3 Cranium general interrupt mask . . . 217
A.2.4 Queue interface . . . 219
A.2.5 Cranium register map . . . 222
A.3 Interface between Cranium and the operating system . . . 223
A.3.1 Initializing and terminating a user message-passing context . . . 225
A.3.2 Context switch . . . 225
A.4 Examples of message passing using the Cranium API . . . 226
A.4.1 Initialization . . . 227
A.4.2 Sending a message . . . 227
A.4.3 Receiving a message . . . 230
A.4.4 Discussion . . . 232

Appendix B: Measurements of programs in the parallel benchmark suite **234**

Appendix C: Measurements of communication cost in Gauss under modifications to Cranium **242**

Appendix D: Description of low-level details of Teschio **243**

D.1 Handshaking signals and protocol of the P link 243

D.2 Node mapping 245

D.3 Sending and receiving a single-packet message 247

List of Figures

1.1 Overview of a scalable parallel computer system 4
1.2 One processing node of a scalable parallel computer system 5
1.3 Layers in a message passing system 9
1.4 Processor overhead in message passing 14
1.5 Performance of networks and network interfaces 15

2.1 Possible locations of the network interface in the processing node of a multicomputer 22
2.2 Implementation of argument checking and protection in the communication system 30
2.3 Taxonomy of the receive interface 41

3.1 Architecture of a network interface to support remote memory with an out-of-order network 50
3.2 Cranium architecture 51
3.3 Cranium packet format 53
3.4 Organization of circular buffer in DRAM for queuing channel 54
3.5 Protection for safe user-level access via mapping tables 56
3.6 The performance implications of timesharing on a multicomputer . . 62
3.7 Gather-scatter support in the network interface 64

4.1 Flow diagram for sending a message 69
4.2 Flow diagram for receiving a message in an auto-receive channel . . . 70
4.3 Flow diagram for receiving a message in the user queue 71
4.4 Using local synchronization to support auto-receive 72
4.5 Synchronization for auto-receive using a global barrier 73
4.6 Using barrier synchronization in a communication phase where every node is both a message source and destination 73

5.1 Scheduling threads in Talisman 84
5.2 Chaotic routing chip: external interfaces and internal buffering 88
5.3 Data paths in the chaotic routing chip 89
5.4 Integration of Talisman and Chaos simulators 91
5.5 Chaos network animation 92
5.6 State machine for communication between Talisman and Chaos . . . 93

6.1 Latency of a single packet under the model "Cranium auto, cold" . . 104
6.2 Throughput of a point-to-point message 106
6.3 Sliding window protocol 107
6.4 A hypercube topology for tree broadcast 109
6.5 Latency of optimal tree broadcast a single packet message 110
6.6 Throughput of broadcasting a 1 Kbyte message 112
6.7 Non-overlapping communication vs. overlapping communication . . . 122
6.8 Maximum speedup and efficiency of benchmark programs 124
6.9 Significance of communication cost 127
6.10 Efficiency of the benchmark suite when the cost of communication is considered 127
6.11 Raw performance of the communication system in bytes per cycle . . 129
6.12 Performance normalized to the maximum achievable throughput . . . 129
6.13 Effective performance of the communication system in bytes per cycle 132
6.14 Communication performance of Cranium under modifications M1, M2 and M3 138

7.1 Simple block diagram of Teschio 145
7.2 One node of a multicomputer system based on Teschio 146
7.3 ADU bus timing 147
7.4 Using two-dimensional parity 150
7.5 Structural diagram of Teschio 152
7.6 Inbox and outbox 156
7.7 Handshaking between inbox, outbox and surrounding environment . . 158
7.8 Accepting a command word from the host processor 161
7.9 The table lookup functions 162

7.10 The core module handler for injecting a packet into the outbox 163
7.11 Decoding and validating the header of a packet 165
7.12 Determining the proper queue for handling the arriving packet 166
7.13 Determining the proper automatic-receive channel for handling the arriving packet 167
7.14 Updating the queue structure 168
7.15 Updating the send channel and auto-receive channel information . . . 169
7.16 Sequence of micro-operations involved in a packet send operation and the minimum number of cycles taken at each step. 172
7.17 Sequence of micro-operations involved in a packet receive operation and the minimum number of cycles taken at each step. 174
7.18 An architecture for a hybrid MPP-SMP system that uses multiple Teschio chips per processing node 183
7.19 Supporting the scatter operation in Teschio 184

8.1 Overview of the Chaos-LAN research project 193

A.1 Schematic view of the interactions among the user application program, the operating system and the network interface 209
A.2 C structure describing the Cranium initialization information 210
A.3 C structure for DMA buffer allocation for Cranium 212
A.4 C structure describing the Cranium packet header 213
A.5 C structure for the Cranium command word 215
A.6 C structure for the Cranium general interrupt mask 218
A.7 C structure for the Cranium queue pointer interface 220
A.8 C structure for the format of packets in the queue 221
A.9 C structure for the Cranium register map 222
A.10 C structure describing the Cranium interface to the operating system 224
A.11 Example code for initialization and the two communication examples 228
A.12 Example code to send a message 229
A.13 Example code to receive a message into the user queue 231
A.14 Example code to receive a message into an automatic channel 233

D.1 Organization of the P link 244
D.2 Node naming scheme used by Teschio 246
D.3 Organization of bit fields placed on ADU bus by the processor to issue a send or receive command 248
D.4 Timing of a send or receive command on the ADU bus 248
D.5 Timing diagram for sending a single packet message 250
D.6 Timing diagram for receiving a single packet message 251

List of Tables

2.1 Comparison of existing network interfaces 43
2.2 Number of memory operations per word per message for the three data movement types Systolic, PIO and DMA 44

3.1 Packet reordering example 49
3.2 Attributes of the Cranium network interface architecture 52

4.1 Comparison of message passing interfaces NX, AM and Cranium . . . 79

5.1 Host configuration data 95
5.2 Slowdown of combined simulator 96

6.1 Latency of a single packet message 103
6.2 Input data set sizes 121
6.3 Maximum aggregate instructions per cycle executed when $p = 16$ . . 125
6.4 Throughput of a long message under the modifications to Cranium . 136

7.1 Description of fields in one channel of the channel array 154
7.2 Description of fields in one channel of the queue channel array 154
7.3 Impact of FIFO size on throughput (4K byte messages) 177
7.4 Total count of the number of bits of memory in Teschio 180

A.1 List of data structures to support message passing under Cranium . . 209
A.2 Interrupt mask state for a Cranium auto-receive channel 217
A.3 Barrier state transition table 223

B.1 Measurements of dense matrix multiplication (DMM). 236
B.2 Calculated instructions per cycle, speedup and efficiency of DMM. . . 236
B.3 Significance of communication, actual throughput, percentage of ideal throughput and effective throughput of DMM. 236

B.4 Measurements of fast Fourier transform (FFT). 237
B.5 Calculated instructions per cycle, speedup and efficiency of FFT. . . . 237
B.6 Significance of communication, actual throughput, percentage of ideal throughput and effective throughput of FFT. 237
B.7 Measurements of Gaussian elimination (Gauss). 238
B.8 Calculated instructions per cycle, speedup and efficiency of Gauss. . . 238
B.9 Significance of communication, actual throughput, percentage of ideal throughput and effective throughput of Gauss. 238
B.10 Measurements of Jacobi successive over-relaxation (Jacobi). 239
B.11 Calculated instructions per cycle, speedup and efficiency of Jacobi. . . 239
B.12 Significance of communication, actual throughput, percentage of ideal throughput and effective throughput of Jacobi. 239
B.13 Measurements of Jacobi successive over-relaxation without global combine (JacNoGC). 240
B.14 Calculated instructions per cycle, speedup and efficiency of JacNoGC. 240
B.15 Significance of communication, actual throughput, percentage of ideal throughput and effective throughput of JacNoGC. 240
B.16 Measurements of bucket sort (Sort). 241
B.17 Calculated instructions per cycle, speedup and efficiency of Sort. . . . 241
B.18 Significance of communication, actual throughput, percentage of ideal throughput and effective throughput of Sort. 241

C.1 Cost of communication in Gauss under Cranium modifications M1, M2, M3 and M1+M2. 242

ACKNOWLEDGMENTS

I thank the Department of Computer Science and Engineering at the University of Washington for its high quality and continuing commitment to excellence. Everyone in CSE between the years 1987 and 1996 played some role in my education. I can't possibly name everyone with whom I crossed paths, hence the blanket inclusion. However, some people played a bigger role than others, whom I wish to thank individually; I hope I do not overlook any of the principal players.

Two research groups at UW-CSE provided the most significant contributions to my education: the Chaos network and communication group, and the computer architecture and distributed systems group. Within the Chaos group I would like to thank my reading committee of Carl Ebeling, Larry Snyder and Kevin Bolding, and fellow students Donald "DCI" Chinn, Sung-Eun Choi, Melanie Fulgham, Magda Konstantinidou, Thu Nguyen and Bill Yost. In the computer architecture and systems group I wish to thank fellow students Craig Anderson, Rob Bedichek, Jeff Chase, Ed Felten, Dave "Don Pardo" Keppel, Alex Klaiber, Dylan McNamee and Chandu Thekkath. In particular, Ed Felten's excellent dissertation on message passing provided a wealth of information and the initial excitement to propel me into this arena. Rob Bedichek provided the Talisman multiprocessor simulator, upon which my performance results are based. Rob, Alex, Craig, Pardo and Dylan all inspired many invigorating conversations on message passing and network interface design. Often, our disagreements helped reveal the deepest truths. Fortunately, the design space is rich enough to accommodate all of us!

Research groups outside UW also played an important role in my dissertation work. I thank Greg Buzzard, Ian Robinson, John Wilkes and the rest of the Hamlyn network interface development group at Hewlett-Packard Lab-

oratories in Palo Alto, California and Al Davis at the University of Utah for providing an excellent sanity check for my ideas.

People connected with the Laboratory for Integrated Systems at UW-CSE played an important role in my master's project, the UW MacTester. I thank Tod Amon, Mary Bailey, Bill Barnard, Gaetano Borriello, Pai Chou, Darren Cronquist, Paul Franklin, Soha Hassoun, Scott Hauck, Ken Hines, Ted Kehl, Brian Lockyear, Larry McMurchie, Ross Ortega, Elizabeth Walkup and Wayne Winder for their support.

Fellow graduate students provided important friendships and made my life within the confines of Sieg Hall more bearable than it otherwise might have been. Thank you, Jim and Chris Ahrens, Juan Alemany, Franz Amador, Ruth Anderson, Greg Barnes, Virgil Bourassa, Lauren Bricker, Reid Brown, Suzanne Bunton, Brad Chamberlain, Travis Craig, Chris Fisher, Dan Kerns, Tracy Kimbrel, Eric Koldinger, Tony LaMarca, Calvin Lin, Greg Linden, Vasily Litvinov, E. Christopher Lewis, Gus Lopez, Ian McDuff, Dorothy Neville, Ruben Ortega, Sean Sandys, Jason Secosky, Eric Selberg, Jean Schweitzer, Kevin Sullivan, Radhika Thekkath, Wendy Thrash, Derrick Weathersby and Ken Whaley. I also thank the faculty and staff at UW CSE, in particular: Brian Bershad, Frankye Jones, Warren Jessup, Stephen Lee, Mark Murray, Larry Ruzzo, Alan Shaw and John Zahorjan.

Deep thanks are due to my current employer, MERL — A Mitsubishi Electric Research Laboratory in Cambridge, Massachusetts. MERL has provided me with a wonderful opportunity to earn a living while finishing the writing of this dissertation. Thanks in particular to Les Belady, John Howard, Hugh Lauer and Randy Osborne for making this dream possible. Thanks also go to our former housemates Dave and Lisa Hubbell, for holding down the fort while I was busy jet-setting between Seattle and New England during 1994-95.

Almost none of the above would have been possible without the financial and emotional support of my parents Keith and Della McKenzie. I am inspired by the life of my late paternal grandfather Ira McKenzie, a farmer whose hard work made it possible to survive the Great Depression and the Dust Bowl. Thankfully, he was able to attend our wedding in Seattle in 1991.

Finally, I thank my wife Angel Thalls. This dissertation is dedicated to you, your love, your support, and your optimism. It's been a long, strange trip from Santa Cruz, California to Arlington, Massachusetts by way of Seattle and Providence. It's been an amazing, crazy, unbelievable time. I don't know how we managed, but we did. May the future be as exhilarating as the past nine years!

To Angel.

Chapter 1

INTRODUCTION

Sit down before fact like a little child, and be prepared to give up every preconceived notion, follow humbly wherever and to whatever abyss Nature leads, or you shall learn nothing.

– *T. H. Huxley*

Digital logic technology continues on a path of sustained increase in performance and circuit density that is matched by its sustained decrease in power consumption and cost. The current rate of improvement in the performance of high-end processors is an impressive 55 percent per year; indeed this rate of increase exceeds the historical trend of 35 percent per year [1]. This trend is expected to continue unabated for the next two decades. Today's personal computers that cost a few thousand dollars outperform multi-million-dollar supercomputers from the 1970s.

Nevertheless, there still exist problems in computer science whose computational requirements exceed that of any single processor personal computer or workstation, and will continue to do so for many years to come. For example:

- *Real-time volume rendering.*

  Volume rendering is the transformation of the three-dimensional data set (such as a set of medical scans collected by MRI or computed tomography) into a two-dimensional representation for viewing on a high-resolution display. Rendering in real-time means that changes to the viewpoint or the data set affect the rendered image immediately. For the appearance of smooth motion, successive two-dimensional views (frames) must be created and displayed ten to twenty times per second. For high-resolution data sets consisting of 512^3 volume elements (*voxels*), the computational requirements of real-time, smooth motion volume rendering are on the order of 10 billion operations per second.

Currently, this rate of computation is two to three orders of magnitude beyond the capabilities of even the fastest personal computers and workstations that do not contain special-purpose hardware to accelerate volume rendering.

- *Finite element modeling and analysis.* Applications of large finite element computations are ubiquitous in the physical sciences such as meteorology (weather prediction), physics, chemistry, astronomy, civil and mechanical engineering, and so on. Finite element problems can require as many as 10^{15} operations. Today's fastest personal computers can execute operations at a rate of roughly 10^8 operations per second. At this rate a fast PC would take approximately 10^7 seconds or about 4 months to execute 10^{15} operations. Very little science would be accomplished if single processor computers were the only choice for solving problems of this magnitude.

Clearly, there exist many interesting problems in the sciences whose computational requirements are beyond the capabilities of today's PCs and workstations. The capability of PCs in the next decade or two may become sufficient to solve these problems, but researchers and surgeons need to be able to solve their problems now, not ten or twenty years from now.

The most obvious and least expensive approach to achieve the required amounts of computational performance is the construction of a parallel computer using standard off-the-shelf parts. For instance, it is possible to buy several personal computers and connect them together with a local area network. A parallel program performs both computation and communication. Communication is necessary to distribute the computation, the initial data set and the user's input parameters. While the program is running, different processors often need to share intermediate results and synchronize.

The challenge of parallel computing is achieving *scalability*. If we buy two personal computers, we expect their combination to provide twice the computing performance of a single personal computer. In general we expect to achieve *linear speedup*; if we have N processors, then we want the parallel version of the program to run in 1/N of the time of the single processor version. The difficulty in making a parallel computer from a collection of personal computers with a local area network is that some programs achieve very poor speedups. The reason is that communication be-

comes the performance bottleneck. Here is an example. Assume that we are using a top-of-the-line personal computer from current technology; at the time of this writing, such a PC uses a Intel Pentium Pro processor running at 200 megahertz. For a particular computation it executes roughly 100 million instructions per second. In a parallel version of the program, it must communicate an intermediate result after every 100 instructions. The communication network is a standard Ethernet; it takes approximately 1/1000 of a second to perform a communication. In that time the Pentium Pro can execute 100,000 instructions. A parallel program that alternates between the computation and communication portions of the program would spend 99.9 percent of its time communicating and only 0.1 percent computing. Slow communication defeats the purpose of making the parallel program run faster than on a single processor. It becomes necessary to design a communication infrastructure that provides the required performance.

It is relatively easy to achieve linear speedup if the number of processors is small. In a symmetric multiprocessor (SMP), all processors are plugged into the common memory bus of the system. The symmetry of an SMP system is that the access time to shared memory for each processor is the same; any program can run on any of the processors and achieve the same performance. The limitation of a shared bus is that it is like an old-style telephone party line; only one phone call can occupy the wire at any given time. As the number of processors increases, the communication requirements increase, but the bandwidth of the network does not increase. The performance of an SMP system is limited to no more than a few dozen times that of a uniprocessor system, no matter how many processors are available.

To achieve scalability to hundreds or thousands of processors, the communication infrastructure must allow many or all of the processors to be involved in communication at the same time. The modern telephone network demonstrates an example of a scalable system. Millions of people can talk on the phone at the same time making separate calls and separate connections. Except in rare circumstances, a new call can be initiated from any phone without having to wait for other customers of the phone system to terminate their calls. The phone system is based on a multitude of point-to-point links spanning the end users and the intermediate switching systems, rather than being based on a single shared wire. When new users are added to the system, there is no effect on the ability of other users to make and receive calls. By analogy,

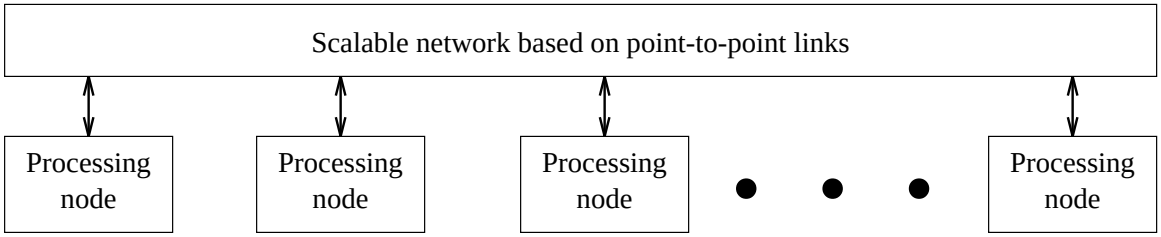


Figure 1.1: Overview of a scalable parallel computer system

in a scalable parallel computer each processor is a customer of the network. Each processor should be able to communicate with any other processor in the network with minimal or no interference from other communication currently in progress.

1.1 MPP computers and scalable networks

Figure 1.1 presents an overview of a scalable parallel computer system, consisting of a scalable network and a collection of identical *processing nodes*. Scalable parallel architectures have been referred to in the literature as MPP (massively parallel processor) machines [2, 3, 4] to distinguish them from SMP machines. The remainder of this dissertation is concerned only with scalable MPP architectures.

Many different types of scalable networks have been developed for parallel computer systems. There exists an extensive literature on this subject; for example, refer to the proceedings of the Parallel Computer Routing and Communication Workshop [5]. Scalable networks are constructed from a single basic circuit called the *router* or *routing node*. The router is replicated potentially hundreds or thousands of times. It is impractical for each router to connect directly to all the other routers in the network; in general, each router has only a few ports that connect directly to other routers. The number of ports is called the *degree* of the routing node. In most network designs, the degree of the routing node is a small constant value (e.g. four, five, six or seven). For ease of construction it is helpful if the configuration can be extended in an obvious way, in a fashion similar to the way Lego bricks are stacked together, in either two or three dimensions [6]. The physical links between neighboring routers are constructed using metal wires or in some cases using optical fibers. Information propagates through the network by passing from neighbor to neighbor through these links.

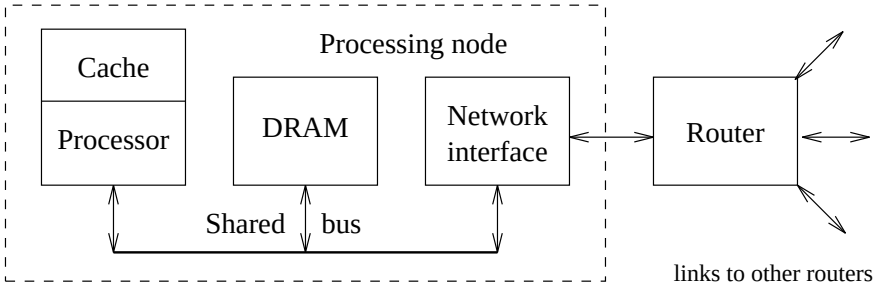


Figure 1.2: One processing node of a scalable parallel computer system

Each processor in an MPP system connects to one of the routers in the network through a separate circuit called the *network interface*. The network interface is the combination of hardware and software that provides the "glue" between the router and the processor. Figure 1.2 shows a block diagram of a single processing node in an MPP system. The processing node consists of a processor, cache memory, one or more memory modules consisting of dynamic RAM and a network interface, that are all connected together on a local shared bus. (In most MPP systems the network interface connects to the processor-memory bus rather than an I/O bus; see Section 2.1.1.) Note that the number of routers may be greater than the number of processors; not every router is required to be attached to a processing node.

A processor communicates with other processors by injecting packets of information into the network through the network interface. Each packet contains a header field plus a payload field; the header contains the address of the destination processor and the payload contains the application-specific information (i.e. the message to be communicated). Once a packet is in the network, it may take one or more routing hops before it arrives at the routing node directly connected to the destination processing node. The router then delivers the packet to this processing node through its network interface.

1.2 The communication bottleneck

A scalable point-to-point network is necessary but it is not a sufficient mechanism for providing high-performance communication in an MPP computer system. The term *communication bottleneck* describes the typical situation where one aspect of the communication system severely limits the overall communication performance.

Traditionally, the bottleneck was due to the network itself. This bottleneck persists in low-cost, general-purpose networks such as Ethernet, which was used in the example on page 3. However, extensive research and development of the kinds of networks as described in Section 1.1 have yielded a variety of moderate cost solutions providing scalable performance. These scalable network designs have largely eliminated the network as the source of the communication bottleneck. Today, the problem lies in the software required to perform communication, the network interface hardware, or both. In particular, software overhead greatly reduces the performance of message passing. *Software overhead* is the number of clock cycles taken to execute all the extra processor instructions needed to access remote memory locations that would not need to be executed if the memory were local to the processor. Some software overhead is intrinsic to the parallel application program that performs communication, but a large percentage is due to the mismatch between what the program requires and what the network interface hardware actually provides. For instance, the program might transmit only small messages, whereas the hardware provides efficient support only for large messages, or vice versa. Some interfaces capture arriving messages into a queue, which can be inefficient for large messages. Other interfaces place arriving messages directly into memory. If two or more messages are destined for the same memory at roughly the same time, then the interface can present a race condition or cause data to be lost. Invariably a software layer must be added to the network interface to ensure the correct functionality at the cost of inflating the software overhead. An important goal in network interface design is to anticipate the kinds of communication requirements of programs and provide direct hardware support for the most frequently used operations, to minimize the software overhead and achieve the highest communication performance.

The purpose of this dissertation is to advance the state of the art of the architecture and implementation of network interface circuits. The network interface ties together many disparate levels of the complete parallel system. At the hardware level, it ties together the processor, the memory and the network router. It provides services for the operating system: argument checking, protection, atomicity and restartability. It provides a medium of communication for application programs. It provides a software interface to the network for application programmers and the authors of run-time libraries and compilers. It is therefore not surprising that network interface

design is difficult. The complexity of integrating all the levels of a scalable parallel computer makes the design of the network interface a daunting task. A lack of design skill at any of these levels will result in a network interface that delivers poor performance at best, or fail to produce a workable network interface at all. As the performance of networks and processors increases, the performance of the network interface becomes an increasingly significant bottleneck in the overall performance of the parallel system.

We devise a structured methodology for the design of network interfaces for MPP computers by examining the network interfaces in existing designs. We extract the key abstractions of these designs, by considering the logical design as well as the physical design. These abstractions are independent of any particular compiler, programming language, operating system, processor instruction set, network or hardware implementation technology. Once the abstractions are understood, then a new type of network interface can be synthesized. We set out to create the simplest network interface design that provides all of the necessary functionality, one that provides the best possible communication performance over a wide range of possible uses and environments.

The remainder of this chapter is organized as follows. We take an in-depth look into the role of the network interface in the message passing system of an MPP computer in Section 1.3. In Section 1.4 we examine some additional issues in scalable network design: packet length and routing algorithms. In Section 1.5 we comment on the performance issues in networks and network interfaces. We briefly introduce related work in Section 1.6. In particular, we comment on two examples from the literature concerning only the software aspect of interprocessor communication. There has been significant progress towards reducing this aspect of the communication bottleneck in recent years, but there are fundamental limitations to these kinds of software-only approaches. Much more detail on related work is provided in Chapter 2. Finally we state the thesis of this dissertation which concerns the architecture and implementation of a network interface for an MPP computer system.

1.3 The role of the network interface in message passing

In general, scalable parallel computers use a style of communication known as *message passing* [7, 8, 9]. The invocation of each message is stated explicitly in the program,

by means of send and receive commands[1]. A simple message consists of a block of data, whose size may range from a single bit to millions of bits. A simple message is uni-directional and asynchronous. After the sender places the message into the network, it does not wait for the arrival of the message at the receiver.

The network interface occupies a crucial layer in the message passing system. The *message passing system* consists of all the layers of hardware and software described in Figure 1.3. It provides the abstraction that the application program itself sends and receives data, as indicated by the dashed line. The application program accesses the network by means of a message passing library and its associated application program interface (API). The message passing library interacts with the operating system, which acts as an intermediary between the library code and the network interface hardware. The sender's network interface places data into the network and the receiver's network interface takes data off the network. The message passing library handles the low-level details of message passing: buffer allocation, packetization and re-assembly, protocol processing and synchronization.

A typical message passing protocol for a simple message consists of the following steps:

1. The application program running on the sending processing node prepares a block of data to be sent, perhaps by allocating a message buffer and writing the outgoing message data into it.

2. The application program running on the sending processing node executes a send command. The send command is realized by one of the following operations: a function call, a system trap, a read or write operation using a special processor register or memory operand, or a special opcode in the processor instruction set.

3. The arguments of the send command are checked. A typical send command

[1] There exist scalable computers that share some attributes of SMP machines, in that they use a shared-memory style of communication supported directly in the network interface hardware. The underlying architecture is otherwise an MPP, based on a scalable, point-to-point interconnection network. Without loss of generality, we assume that the explicit message passing paradigm is supported directly in hardware and in the programming model for the system. In Chapter 2, we argue that the shared memory paradigm can be considered just another message passing model.

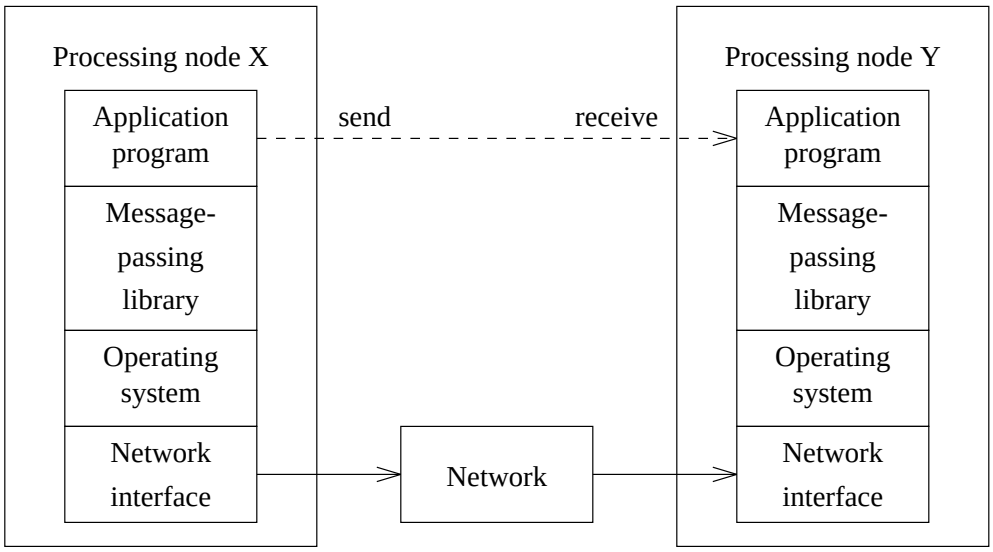


Figure 1.3: Layers in a message passing system

specifies the identifier of the receiving processing node, the address of a data buffer or immediate data, the size of the data, and some optional information specifying the message protocol or a message tag to be matched at the receiver. If the destination identifier, the buffer address or the message tag are out of range, then the send operation is rejected.

4. The message passing system partitions the message into packets and the network interface injects each packet into the network. This partitioning step is not necessary for a network that permits arbitrarily long packets (i.e. a wormhole network); this issue is discussed in Section 1.4.

5. Packets that contain the message travel through the network and arrive at the receiving processing node.

6. The network interface at the receiving processing node accepts the packets as they arrive from the network and places them in a data buffer.

7. The receiving processing node executes code representing a receive command. In some systems, the receive command is explicitly executed by the application program. In some systems, receiving a message happens automatically by means of hardware, an interrupt handler, or a combination of both.

8. The application program at the receiving processing node is notified that the message has arrived, whereupon code is dispatched to perform any necessary post-processing of the message.

Arbitrarily complicated message protocols can be built on top of simple messages. A *message protocol* [10] is an agreement between the sender and the receiver concerning the size, format and sequence of the message. Typically, message passing systems provide a number of different message protocols for the application programmer. One example is a bi-directional, synchronous message protocol that involves a round trip from sender to receiver back to the sender. Message protocols are needed to prevent information loss and deadlock. Information may be lost if the receiver does not allocate sufficient buffer space for the incoming message. Deadlock occurs when a sending processing node is never allowed to send. It can happen when two or more processing nodes act as both senders and receivers and there is a cyclic dependency. Message protocols represent a significant source of processor overhead because it is often necessary to copy data and exchange synchronization messages. The programmer, run-time library or compiler must choose the appropriate message protocol for each message. An incorrect protocol choice may cause the program to achieve poor speedups or manifest race conditions that cause unreliable operation.

To make message passing efficient, a network interface design provides direct hardware support for many aspects of the message passing system. Three important types of support that have been explored in both research and commercial parallel systems are support for argument checking, data movement and protocol processing, described as follows.

- Argument checking is a significant source of software overhead in a message passing system. If argument checking is performed only in software, it is by code that runs at supervisor level, as part of the operating system (i.e. in a device driver). All interactions between the application program and the operating system involve hundreds or thousands of processor instructions and are fundamentally slow. Support for argument checking in the network interface hardware can dramatically reduce this source of overhead.

- Hardware support for data movement is called Direct Memory Access (DMA). DMA reduces the transfer time per message word for two reasons. First, DMA

can take advantage of a fast pipelined memory bus mode called burst mode. Second, DMA allows packets to be sent or received while the processor attends to other tasks. This ability to overlap communication with computation greatly reduces the overhead of communication.

- Network interfaces can also provide direct support for message protocol processing. Supported protocols run efficiently. However, supporting every conceivable protocol in hardware is impossible in practice. The best strategy for the designers of the network interface is to identify the most common message protocols and support them directly, and emulate the others in software.

The challenge to the designer of the network interface hardware is to manifest all three types of support, ensure that the software layers of the message passing system are able to take advantage of these hardware support features correctly and efficiently, yet avoid designing a circuit that is needlessly complicated and difficult to construct.

1.4 Packet length and routing algorithms

Two issues in network design have an impact on network interface design. The first issue is packet length. Some networks are able to accept and deliver packets of arbitrary length. This style of network is generally called *wormhole*. In effect, the network temporarily builds a dedicated circuit between the router connected to the sending processing node and the router connected to the destination processing node. After the head of the "worm" arrives at the destination node, a packet payload of arbitrary length follows. When the end of the packet (i.e. the tail of the worm) passes through the network, the network links used by this packet are freed up and can be subsequently allocated to other packets. A contrasting style to wormhole is a network that assumes that the packet length is constant (or, more generally, the network places an upper bound on the packet length). An advantage of using fixed-length packets is that it simplifies some design issues related to packet buffering both in the network and in the network interface. A packet can be stored completely in a buffer within a single node and thereby it does not occupy any of the network links. This packet can be forwarded to the next routing node when there is available buffer

space at that node. Hence, a network that routes fixed-length packets is also called *store-and-forward*. This property allows a wider variety of packet routing algorithms to be used by the network than wormhole does, as we will discuss below. A tradeoff is that some messages cannot be contained in a single packet. Either the processor or the network interface at the sending node must subdivide long messages into a series of packets, to be injected and delivered by the network separately. This aspect is called packetization and re-assembly. An efficient network interface design based on DMA for fixed-length packets performs packetization and re-assembly automatically.

The second issue is the effect of the network routing algorithm on the arrival order of packets comprising a long message. This issue is relevant only when considering packetization and re-assembly; a wormhole packet that contains an entire message does not require re-assembly. Since a scalable network is constructed from point-to-point links, there are a multitude of paths from the sending node to the receiving node. The network routing algorithm determines the routes taken by each packet in the message. There are many different routing algorithms. One class of routing algorithms is known as *oblivious* or deterministic, wherein the path taken by the packet is unambiguously determined by the addresses of the source node and the destination node. Under oblivious routing, each packet in the message follows the one in front along the same path; packets arrive in the same order as they were injected. There are also *adaptive* routing algorithms. In an adaptive routing algorithm [11, 12], the packets comprising a long message choose different paths, depending on the instantaneous level of congestion encountered at each router. Adaptivity often increases the throughput of the network because it increases the number of paths that can be taken and thereby helps distribute the workload. Adaptive routing has several other advantages over oblivious routing. It improves the opportunities for fault tolerance, the ability to operate in the presence of failed routers or processors [13]. It also simplifies some issues concerning the ability of the operating system to manage multiple user contexts, by saving the state of the network and later re-injecting packet traffic. A network and parallel programming system that depends on in-order packet delivery makes it very difficult to provide this capability in an efficient way. Some examples of adaptive network routers that have been realized in silicon are the CM-5 network router [14] and the Chaos router [15].

A potential disadvantage of adaptivity is that packets may arrive in a different order than they are sent. Multiple pathways allow packets to pass one another in the network; packets travel at different rates due to the local instantaneous congestion encountered along the way. Out-of-order arrival complicates the task of re-assembly and notification to the processor that an entire message has arrived. Unlike in the case of an oblivious router, the arrival of the last packet in the message is not a guarantee that the entire message has arrived. The impact is to increase the amount of processor overhead needed to handle adaptivity, increase the complexity of the design and implementation of the network interface, or both.

Because there are many advantages to using adaptivity in a scalable network for a parallel computer, its important to understand how to construct network interfaces that work well with them. A well-designed network interface for an adaptive router should be able to minimize any disadvantage posed by packets that arrive out-of-order.

1.5 Performance of networks and network interfaces

There are two types of metrics that describe the performance of a network: transfer time and capacity. *Latency* describes the time it takes to initiate and deliver a single minimum length packet. *Bandwidth* is the capacity of a data channel, the maximum quantity of data per unit time it is able to transfer. *Throughput* is the achieved rate of transfer over the data channel, often described in terms of a percentage of the bandwidth of the channel. If messages are sufficiently large, then the throughput has the most significant effect on the overall time it takes to transfer the message. For sufficiently small messages, the capacity of the channel is not a concern, so the end-to-end latency makes the most significant contribution to the transfer time. Time is measured in terms of clock cycles. The length of a clock cycle continues to decrease as the underlying hardware technology improves. A clock cycle in a high-performance network in current systems is roughly 10 nanoseconds, representing a clock frequency of 100 megahertz. Today's state-of-the-art scalable networks have latencies on the order of a few tens of clock cycles; a small message traverses the network in less than a microsecond.

Ideally, the performance of message passing is the same as that of the underlying communication network. In practice, the layers of the message passing system add a

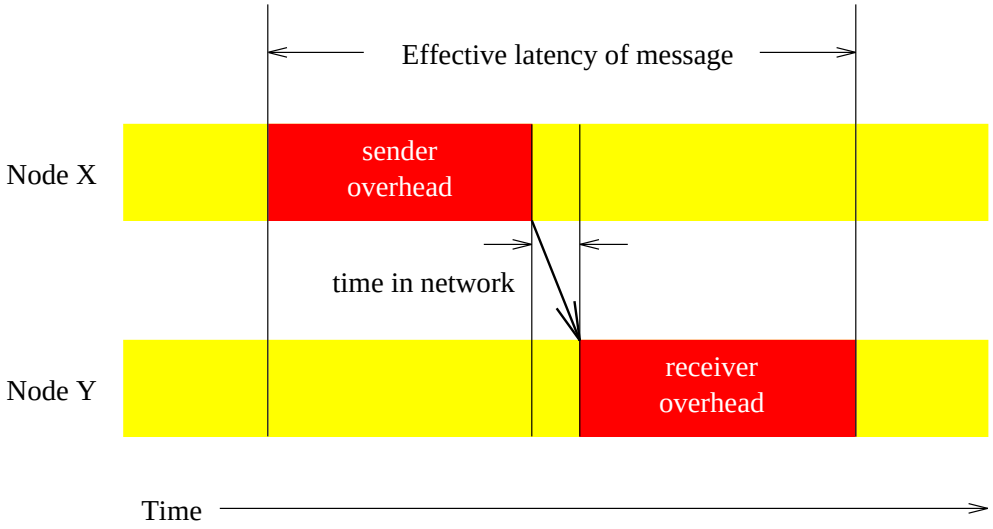


Figure 1.4: Processor overhead in message passing

considerable amount of latency. The message passing library, the operating system and the network interface all contribute to the increase in latency that results in the loss of performance. In some systems the processor overhead introduced by the message passing system overwhelms the latency of the network, as shown in Figure 1.4. The problem is that the processor ends up performing all of the required operations: argument checking, data movement, notification and dispatch, and protocol processing.

Figure 1.5 illustrates the difference between the performance of networks and that of network interfaces. In both subfigures (a) and (b), transfer time is plotted as a function of message size. Lower curves in the graph represent lower elapsed times and consequently higher performance. Figure 1.5a plots curves B1 and B2 to represent the end-to-end throughput of two different networks. Note that B2 has higher performance than B1 because both have the same Y-intercept (representing the latency of a minimum-length message) and the slope of B2 is less than that of B1. Figure 1.5b demonstrates the effect of coupling a network interface with the network. The network interface attached to B1 has an intrinsic latency of L1 and provides a total transfer time indicated by the curve NI1. Likewise, curve NI2 represents the total transfer time bounded by L2 and B2. NI1 has lower intrinsic latency, and NI2 has higher intrinsic throughput. Even if the network interface represented by NI1 were connected to the higher bandwidth network, its performance curve would not

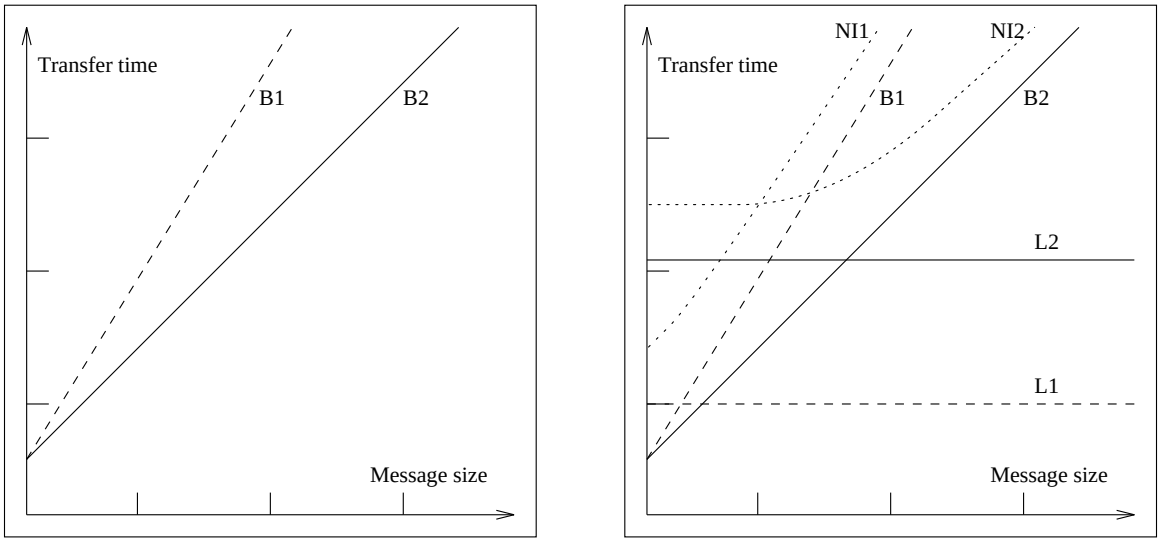


Figure 1.5: Performance of networks and network interfaces. Transfer time is plotted as a function of message size. The lower the graph, the better the performance. Subfigure (a) shows two graphs B1 and B2 that represent the transfer time of two different networks. Subfigure (b) demonstrates the effect of network interfaces on the overall latency and throughput of these networks. The intrinsic latencies of two different network interfaces are described by the curves L1 and L2. The curve NI1 describes the performance of the network when combined with a low latency network interface; it is bounded by the curves B1 and L1. The curve NI2 shows a competing approach whose network interface is optimized for high throughput; its curve is bounded by B2 and L2.

change, because the throughput of the network interface itself is the limiting factor. Basic network interface designs that contain a single transfer mechanism must make the choice between high throughput and low latency – no single transfer mechanism has ever been shown to perform well for both cases.

Choosing between high throughput and low latency is difficult because different parallel programs have different requirements. Some programs involve bi-modal message traffic, in that they require both long and short messages. In one study of message traffic in a typical set of parallel program benchmarks, 98% of all messages contained fewer than 40 bytes [16]. The same study showed that these small messages

comprised 55% of the total bytes communicated. The other 45% of the total bytes came from the 2% of the messages that were 40 bytes or longer. Many other studies confirm the bi-modality of message traffic. In general, the majority of messages are small, but a significant amount of the total number of bytes transferred comes from large messages.

1.6 Related work

There are many network interfaces that have been designed for scalable parallel computers. Some examples used in commercial systems are the network interfaces in the Thinking Machines CM-1 and CM-5 [2, 14], the Paragon and Teraflops multicomputers by Intel [17, 18], and the Cray Research T3D and T3E [19, 20, 21]. In all of these systems, the network interface is implemented using a separate chip, and the computing node is based on a standard processor architecture such as the SPARC, the DEC Alpha or the Intel Pentium. Some examples from academic research projects are the MIT Message-Driven Processor (MDP) [22], the CMU iWarp [23] and the Caltech Mosaic [24]. In all of these projects, the network interface is *tightly coupled* with the processor; i.e. the network interface and the processor are placed on the same silicon. The motivation for the tight coupling is to reduce the overhead associated with short messages. For instance, the MDP provides a hardware technique for fast notification and dispatch. Notification and dispatch for an arriving message take only three processor instructions, unlike the tens or hundreds of instructions required in conventional designs.

Two other related projects address the performance problem in message passing using software-only techniques: *active messages* and *protocol compilers*. The motivation for active messages [25] is to improve the performance of data movement and software handler dispatch. In an active message, values from the payload of the arriving packet are incorporated immediately into the computation instead of being stored in memory as an intermediate step. This technique represents significant savings in overhead if the network interface does not use DMA. An active message includes the address of the receiving node's software handler routine in the message itself, and the overhead of dispatch is thereby reduced to a small fixed cost. Protocol compilers address the protocol choice problem. Parachute [10] is one such protocol compiler that analyzes message passing patterns in a parallel program; it automatically gen-

erates a new program where the optimal protocol is selected for each message in the program. A protocol compiler can only choose from the existing protocols in the message passing library; it does not re-implement the protocols within the library.

Software-only techniques provide a significant savings in overhead over previous implementations of message passing libraries. However, they cannot solve the performance problems inherent in the network interface hardware. For instance, most network interface designs implement only one message protocol directly. Other message protocols must be realized by software emulation. Generally speaking, one protocol works well for small messages, and another one does so for large messages. Since most programs involve some kind of mix of small and large messages, all messages of the "wrong type" will be inefficient. This deficiency cannot be cured by either active messages or a protocol compiler.

The tension between low latency support and high throughput support motivates the use of two different mechanisms to deal with the two cases independently. The challenge is to provide a unified approach that avoids needless complexity. The architecture presented in this dissertation indeed uses two different mechanisms for receiving packets but only one send mechanism. By contrast, most network interface designs that contain two different mechanisms use two completely separate strategies for short and long messages. For instance, the MIT Alewife interface [26] provides a message passing interface for long messages and a shared memory interface for short messages. The iWarp chip [23] also provides two separate strategies, one for large messages called memory communication and the other for small messages called systolic communication. These designs therefore become more complex than is necessary. We believe that our design represents the minimum complexity that is required to achieve the desired performance.

1.7 The thesis

The issues presented in Sections 1.3, 1.4 and 1.5 demonstrate the intrinsic difficulty associated with the design and implementation of network interfaces in scalable computers. A network interface design needs to present a convenient programming model to the software developer, provide the flexibility to work with a variety of modern scalable networks, and yield the low latency and high throughput of these networks. It also needs to be simple to design and independent from any given network or

processor design so that successive implementations can be produced to match the current 55 per cent increase in processor performance per year.

This dissertation presents an architecture for network interfaces in scalable parallel computer systems. This network interface architecture, called *Cranium*, satisfies the following criteria:

- Cranium is a flexible solution that works with a wide variety of processor and network designs. In particular, it is designed to work with adaptive routing algorithms that allow packets to pass one another in the network and arrive in a different order than they were injected. The architecture uses fixed-size packets with a payload the size of a processor cache line. A wormhole network could be used given the restriction that the network interface only injects and receives fixed-length packets.

- Cranium delivers the high throughput and low latency provided by modern scalable computer networks. To achieve low latency, it greatly reduces the overhead of message passing through the use of hardware techniques for argument checking, data movement, notification and dispatch, and protocol processing.

- Cranium is integrated with an application programmer's interface (API) that allows the programmer or compiler to take advantage of the features of the architecture directly. Both a small message protocol and a large message protocol are supported directly, so that programs can achieve high communication performance in a straightforward manner.

The organization of the rest of this dissertation is as follows. In Chapter 2 we outline the fundamental requirements for network interface design in parallel computers. We examine both the physical (hardware) layer and the logical (software) layer. We construct a taxonomy of network interface designs from the discussion. In Chapter 3 we explain the difficulty of interfacing processors to adaptive routers, through the use of a specific design example. We then specify the Cranium architecture. Chapter 4 describes the software interface for Cranium, and compares it with other message passing systems such as Intel NX [7]. Chapter 5 describes a simulation environment that was developed to evaluate the performance of Cranium, based on the Talisman

processor simulator [27] and the Chaos router [15]. Chapter 6 characterizes the performance of Cranium. We begin with an analysis of the basic latency and throughput behavior of Cranium. We measure the performance of parallel programs that run on the simulator. We introduce a methodology for comparing the Cranium network interface architecture with other architectures, by factoring out details specific to the implementation. The results of the evaluation show that Cranium meets the goal of providing a general architecture with low processor overhead, as compared with network interface styles in existing parallel computers. Chapter 7 presents a paper design for an implementation of Cranium. Because the Cranium architecture is parameterized, there is a spectrum of implementations that provide different tradeoffs between complexity and performance. The specific implementation called Teschio describes a simple single-chip design that nonetheless provides excellent performance. We conclude the chapter by exploring implementation-specific extensions to the Cranium architecture. We present our final thoughts on Cranium in Chapter 8.

Chapter 2

NETWORK INTERFACES

Order and simplification are the first steps towards the mastery of a subject; the actual enemy is the unknown.

– *Thomas Mann*

Every tiny step forward in the world was formerly made at the cost of mental and physical torture.

– *Nietzsche*

The network interface provides the glue between the processing node (consisting of a processor and a memory) and the communication network in a massively parallel or a distributed computer system. The purpose of the network interface is to move data from the processing node out to the network, and from the network into the processing node. A wide variety of communication networks have led to the development of a wide variety of network interfaces. The goal of this chapter is to examine the features of a number of network interface designs. We compare their similarities and differences in order to construct a taxonomy. We examine features of network interfaces in scalable, massively parallel (MPP) machines as well as systems based on networks of workstations (NOW) [28].

Every network interface is a combination of two interfaces:

- The *physical interface* provides a data path between the computing node and the network. The physical interface entails the location of the network interface in the computing node, its data movement style, its notification style, and its interaction between data movement and notification. It is also important to handle faults in the network that cause packets to become lost or corrupted. The physical interface itself consists of two parts: the send interface and the receive interface.

- The *logical interface* provides the programmer a set of primitives for application programs (and/or the operating system) to send messages to other nodes and to receive messages from other nodes. The three primary styles of logical interface are systolic communication, remote memory access, and send/receive communication.

In the remainder of this chapter we examine the spectrum of choices that can be taken for both the physical interface and the logical interface. The choices are compared in terms of processor overhead. Through an analysis of the choices, we determine the set of features that should be used in a network interface design to achieve the best tradeoff between high performance and ease of implementation.

2.1 The physical interface

2.1.1 Coupling the physical interface with the processing node

There are many options for the location of the physical interface of the network interface, the way in which the data path from the network connects to the computing node. Figure 2.1 describes the spectrum of choices, listed in order of increasing distance from the processor.

- A: the interface is integrated directly in with the processor. This style is also called *tightly-coupled*.
- B: the interface is connected to the processor's external cache bus.
- C: the interface is connected to the memory bus (also known as the processor-local bus).
- D: the interface is connected to an I/O bus, such as PCI or SCSI.

Tightly-coupled network interfaces

A tightly-coupled network interface brings the data path of the processor-network link directly onto the chip through a set of dedicated chip pins. The network interface becomes an extension of the processor architecture; the send and receive functionality

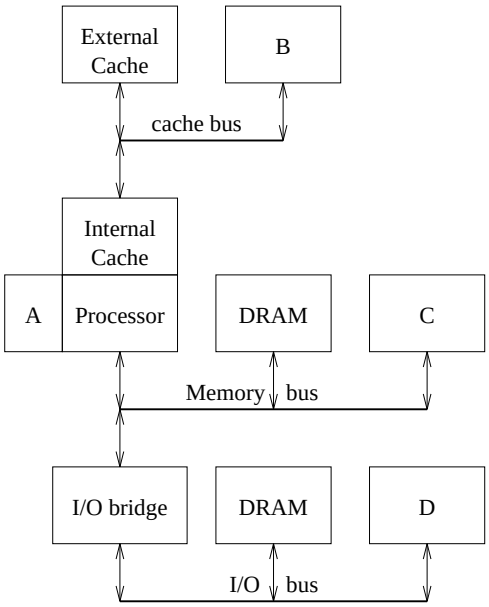


Figure 2.1: Possible locations of the network interface in the processing node of a multicomputer. Location A is tightly coupled in with the processor. Location B is connected to the external cache bus. Location C is connected to the memory bus. Location D is connected to an I/O bus.

is represented by special message registers (operands) and/or instructions (operators). Most tightly-coupled interface designs use special-purpose message instructions (e.g. a send command) in which general-purpose processor registers are the operands. Some examples include the Message Driven Processor (MDP) [22], the Caltech Mosaic [24], the Henry-Joerg network interface [29] and the Start (*T) [30] network interface. An exception is iWarp from CMU [23] whose systolic communication model is based on operands rather than operators. A send command is constructed by using a message register as the destination of an arithmetic operation; a receive command is constructed by using a message register as a source operand. Both the MDP and the Mosaic are building blocks for an entire parallel computer – each chip contains a processor core, a memory, a network interface and a network router. MDPs can be connected in a three-dimensional grid to create an instance of the J-Machine, a massively parallel architecture that scales up to thousands of nodes. The Mosaic is similar except that it is based on a 2-D rather than 3-D mesh. The Myrinet network [31] is constructed from arrays of Mosaic chips. The Henry-Joerg and *T network interfaces were integrated into the Motorola 88110, a commercial superscalar RISC processor.

The advantage of a tightly-coupled design is to be able to achieve the lowest possible latency by bringing data from the network directly into processor registers, the fastest layer in the memory hierarchy. There are however several drawbacks to a tightly-coupled network interface. First, sustaining high throughput is difficult because the set of processor registers is small. Second, it is inherently not portable to other processors and other architectures. Third, a tightly-coupled interface increases the cost of the processor chip significantly. The cost increases because the network interface increases the number of processor pins, the size of the pad frame, the amount of chip area and the power requirements of the chip. In the case of the *T project, the area required for the on-chip interface is 15 per cent of the total chip area [32]. While the extra cost of the integrated interface may not seem to be overwhelming, it is significant enough to deter its acceptance into the commodity marketplace. Recall that the commodity uniprocessor market drives the technology for multiprocessor and scalable parallel systems. The vast majority of packaged systems are uniprocessor machines like PCs in which a tightly-coupled network interface would not be useful. For these reasons, tightly-coupled network interface projects tend to end when the

processor to which it is attached becomes obsolete. However, as local area network and modem designs in PCs become standardized, we may eventually see processor manufacturers return to a tightly-coupled network interface strategy if it can reduce the overall parts count and cost in a commodity system.

Cache bus connected network interfaces

The closest network interface location that is not considered a tightly-coupled connection is through the external processor cache bus. The advantage of using the cache bus connection is that it is usually wider (providing higher throughput) and faster (lower latency) than the memory bus or the I/O bus. Unlike a tightly-coupled connection, a cache bus connected interface uses an existing data path and thereby does not increase the number of pins in the processor. However, cache bus connected designs are difficult to implement and offer very limited support for message passing primitives. As a result, very few network interfaces connect through the cache bus. One notable example is the architecture of the Kendall Square Research KSR-1 [33]. The design is based on a principle called COMA, meaning Cache-Only Memory Access (also known as ALLCACHE^{tm}). In essence, all memory is cache and there is no main memory *per se*.

Memory bus connected network interfaces

In most scalable parallel computers the network interface is located at the memory bus. Connecting through the memory bus provides greater flexibility than connecting to the cache bus, and greater performance than connecting through an I/O bus. Network interfaces in this category include the Thinking Machines CM-5 [14, 34], the Cray Research T3D and T3E [19, 20, 21], the Intel Paragon [17] and University of Washington Meerkat-1 [35].

I/O bus connected network interfaces

In parallel systems that use a local area network as its communication backbone, the preferred location of the network interface is at the I/O bus. I/O bus cards are relatively simple and inexpensive to create. They can be used with a wide variety of systems that support the I/O bus. For instance, there are many personal computers

and workstations that are based on the PCI bus standard. The drawback of connecting the network through the I/O bus is that it offers the lowest performance of the four possible locations for the network interface: the lowest bandwidth and the highest latency. The I/O bus bridge usually becomes a performance bottleneck in message passing as it increases latency and decreases the throughput. In some I/O bus based network interface designs, dynamic RAM modules are added to the interface card [36, 37, 38]. This technique is called outboard buffering. Examples of network interfaces that attach to the I/O bus are commercial Ethernet controllers, an ATM PCI interface called DART [39, 40], a PCI-to-PCI bridge called Memory Channel [41], the Myrinet network interface [31] and the Princeton SHRIMP network interface [42, 43].

2.1.2 Data movement

The fundamental purpose of the physical interface is data movement. The send interface injects packets into the network and the receive interface ejects packets from the network. The two competing styles for data movement are programmed I/O (PIO) and direct memory access (DMA). The style of data movement represents a classic tradeoff in network interface design: DMA provides higher throughput, and PIO offers lower latency. DMA is also slightly more complex, and offers a wider range of implementations. The style of data movement in tightly-coupled designs represents a hybrid of both PIO and DMA.

Programmed I/O

Programmed input/output is the simplest mechanism for moving data between the processing node and the network. The processor moves every byte of the message by explicit load or store instructions to memory or I/O mapped addresses representing the network interface. Under PIO the network interface is typically a slave-only device that does not require bus master capability.

Direct memory access

Direct memory access is a common technique in I/O subsystem design. It implies the use of a memory bus or I/O bus connected network interface. The DMA controller contains bus master capability and moves data directly between the processing node's

DRAM and the I/O device (i.e. the network). DMA is a co-processor that allows the processor to resume other processing tasks while message data are arriving or being sent. In other words, DMA makes it possible to overlap computation with communication. However, the cost of setting up a DMA transfer may be large in some systems, making DMA less effective than PIO for small messages. Unlike PIO, DMA can take advantage of burst-mode in the memory bus. DMA therefore achieves higher throughput than PIO and is faster than PIO for large messages. DMA is almost always used with network interface cards that connect at the I/O bus and contain outboard buffering. DMA improves throughput significantly over a non-outboard solution because it eliminates the need to cross the I/O bridge while message transfers are in progress. However, there is the additional latency penalty for processor access to the outboard memory through the I/O bridge after DMA has completed.

Tightly coupled

Data movement in tightly-coupled interfaces is a hybrid of DMA and PIO. Like DMA, data is moved automatically. Like PIO, access to memory is avoided entirely because the sources and sinks for message data appear directly in the processor's register set. The DMA cache coherence problem is also avoided (see below).

Cache coherence

A pitfall in using DMA is the cache coherence problem [44]. To send a message, the processor writes values to memory and then initiates the DMA. If the processor cache uses a write-back protocol, then live message data may be in the cache while stale data remains in memory; it is necessary to flush these cache lines to memory before sending the message. The problem is similar for the receive interface. When data from an arriving message are placed in memory, the corresponding cache lines in the processor cache become invalid although they may have been marked valid. Therefore these cache lines must be correctly marked as invalid after receiving a message, or they must be updated with the correct data at the same time the message is stored in memory.

There are many techniques for enforcing cache coherence in the presence of DMA. The simplest technique for maintaining cache coherence is to mark all memory pages that are used for message buffers as non-cacheable. An optimization for message

buffers that are always used as send buffers is to mark the memory pages as write-through. These techniques permit a correct but comparatively slow implementation compared with write-back caching. Techniques for cache coherence in the presence of write-back caching are possible by means of software, hardware and hybrid approaches. A software technique for maintaining cache coherence is to execute a cache-flush operation before sending a message and after receiving a message. If the processor architecture only supports a flush of the entire cache then this technique can be extremely slow, on the order of thousands of clock cycles. However, some architectures support flushing individual pages and cache lines. A hardware technique for ensuring cache coherence with a write-back cache is cache snooping. The processor watches all traffic appearing on the memory bus, and then performs updates or invalidations on its internal cache state if it discovers that addresses corresponding to active cache lines have appeared on the bus.

In general, hardware techniques for ensuring cache coherence provide better performance than using software techniques or marking the message buffers as non-cacheable. Most high-performance processor architectures provide hardware snooping capability, making this the preferred technique. See Section 3.4.2 for a related discussion on hardware cache coherence.

Gather-scatter support

Gather-scatter support is an extension to standard DMA. Simple DMA requires message buffers to be contiguous in memory. With gather-scatter capability, the sender's network interface gathers non-contiguous data and sends it as a single message. The network interface at the receiver does the inverse operation, by scattering data from a single arriving message into non-contiguous places in memory. The typical application of gather-scatter DMA is to pass array data with a constant offset (stride) between the memory locations of each array element, but implementations may generalize to allow arbitrary offsets. As with ordinary DMA to contiguous memory, care must be taken to ensure that cache and memory are kept coherent.

2.1.3 Notification and dispatch

It is important for the network interface to notify the processor so that it can react to changes in the status of the network and the network interface. After the processor is notified it is then able to dispatch a handler routine to execute the next part of the message protocol. Status information that is commonly provided by the network interface includes *network busy*, *packet present* and *network error*. *Network busy* is used by the send interface to determine if packet injection will succeed. *Packet present* is used by the receive interface to determine if a packet has arrived at the destination node. *Network error* is used by the receive interface to signal a corrupted packet in the network.

There are four styles of network interface support for processor notification and dispatch: stall, poll and interrupt, and no notification. Network interface designs often support more than one style of notification. It is common for interfaces to support both polling and interrupts, for example. Sometimes one form is used with the send interface and another with the receive interface. Here are the four types individually:

- *Stall.* The network interface stalls the processor until a state change occurs. In the send interface, the processor stalls while the network is busy. In the receive interface, the processor stalls until the packet arrives. Dispatch is implicit as the processor resumes processing. Multicomputers that use stall notification include the Cray T3E [21], DASH [45] and the Tera MTA-1 [46]. The MTA-1 is a multithreaded processor; while the thread that is waiting for the network is stalled, the processor automatically switches to another thread that is not waiting for the network. Stalling often requires special support in the memory system. In the Tera, each memory location contains its own synchronization information, called full-empty bits. Full means that the location contains valid information, and empty means that one node must write to that location before another can read from it. Attempting to read from an empty location causes the thread to stall, and the read operation is continually retried by the network interface until it eventually succeeds. Similarly, the Cray T3E contains a set of 512 special memory locations called E-registers that are used to stall and resume execution.

- *Poll*. The processor explicitly tests the status register to determine the status of the network interface. The handler is dispatched by a conditional branch based on the contents of the status register. There are many machines that use polling, including the CM-5 and the Paragon.

- *Interrupt*. The network interface interrupts the processor when the status register changes state. The handler is dispatched automatically by the interrupt vector. There are many machines that use interrupts, including the CM-5, the MDP and Meerkat-1.

- *No notification upon message arrival.* There is no explicit notification provided by the network interface when a packet arrives. A network interface in this category is the Princeton SHRIMP [42].

2.1.4 Argument checking and protection

It is mandatory that commands and arguments to the message passing system are checked by a secure, trusted entity. Checking is performed by either the operating system software or hardware that acts on behalf of the operating system. The purpose is to plug any loopholes in the communication system that could be used to snoop on or corrupt the operating system and other user programs. Two schemes for argument checking are described by Figure 2.2. In the subfigure on the left, the application program calls the operating system for each interaction with the network. In the subfigure on the right, the application program is able to access the network directly. This technique in which the network interface hardware performs argument checking on behalf of the operating system is called *direct user-level access*. Direct user-level access is necessary for reducing the overhead of message passing. Operating system calls for message passing are a primary source of processor overhead, on the order of hundreds of processor instructions. Whereas it is not always possible to eliminate every possible OS call that plays a role in message passing, direct user access provides the ability to eliminate the vast majority of these OS calls.

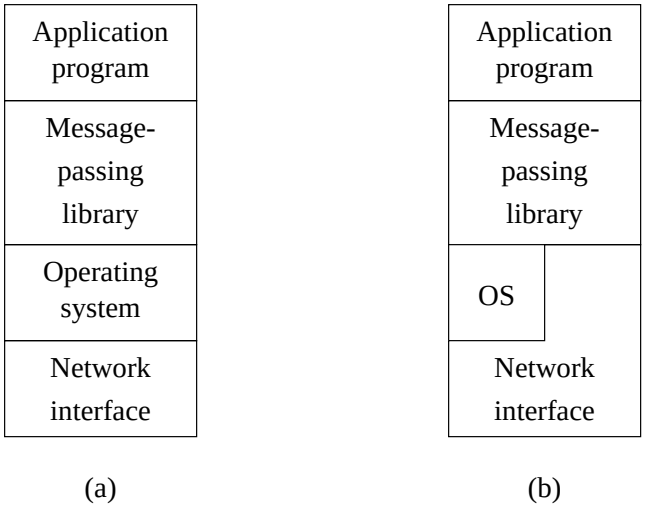


Figure 2.2: Implementation of argument checking and protection in the communication system. Subfigure (a) describes the typical system where all message passing commands are checked by the operating system. Subfigure (b) shows an optimization that allows the user program to bypass the operating system. Argument checking is performed in the network interface hardware. This capability allows direct access to the network by user programs.

Required features

Proper implementation of hardware that provides protection for direct user access requires the coordination of many layers, including the network, the network interface and the operating system. Checking arguments and denying illegal access consists of four parts: local memory protection, remote node protection, network protection and task management.

- *Local memory protection.* A user program must be prevented from reading or writing memory locations outside its protection domain. Under PIO, the memory management unit (MMU) in the processor suffices to provide this functionality by issuing a bus fault on an access to a page that is not mapped by the MMU. Under DMA, additional hardware in the network interface must exist to prevent the use of improper memory addresses.

- *Remote node protection.* The architecture must ensure that user programs send packets only to the set of other nodes that are running the same user program. This feature is necessary for partitioning the system into separate sub-units, so that each can run as an independent parallel system.

- *Task management.* The operating system should provide multiprogramming, the ability to execute more than one user program concurrently. As in a uniprocessor computer, the operating system should be able to stop running the active user program, deschedule it, schedule a different one and start running the new one.

- *Network protection.* Information in the packet header must distinguish user packets from system packets. Alternatively, a separate virtual network could be used for each. The "system bit" or virtual channel ID must be both generated and verified in hardware. This requirement prevents the user program from spoofing or intercepting system packets, which could circumvent the other protection mechanisms. A buggy or malicious user program should not consume all network bandwidth and cause the entire parallel system to thrash.

Mechanisms for implementing protection

- *Address mapping (local memory protection).* One technique for providing a safe memory mapping is for the receiver to place incoming packet data into a managed ring-queue. However, this style of buffering may be inefficient for large messages. Under unbuffered automatic-receive DMA (see Section 2.2.4), the sender directly specifies the destination location of the packet payload through either a message tag or a global virtual address (GVA) in the packet header (see Section 2.2.2). To implement protected address mapping, the network interface hardware translates the message tag or GVA into a local physical address used by the DMA engine. In most implementations the translation is performed by means of a translation lookaside buffer (TLB) using a content-addressable memory (CAM). In principle, a static RAM could be used instead of a CAM, but the input tag or address is often 32 or more bits, implying an intractably large SRAM. If the TLB maps the header of an incoming packet to an invalid local physical address, then the packet is dropped and the network interface can signal a protocol error. Address protection for gather-scatter hardware uses an extra level of indirection to generate the set of non-contiguous valid addresses, usually by means of a look-up table in memory.

- *Logical node identifiers (remote node protection).* Under the basic version, the network interface hardware simply verifies that the remote node identifier is valid and otherwise generates a protection violation. A more flexible version uses a mapping table. In this scheme, the user program provides a logical node ID, and the hardware translates it into a system-specific physical node ID. The logical-to-physical mapping feature makes partitioning and the avoidance of faulty nodes completely transparent to the user program.

- *Atomic packet injection (task management).* When the operating system intends to perform a context switch, it must ensure that restarting the user program is a reliable operation. Because packets must be injected atomically, partially injected packets and messages are a problem for task management. Atomicity is easily guaranteed by the use of DMA. If the DMA of a single packet is in progress, it is simply allowed to complete. The context of a long transfer that involves multiple packets must be saved and then restored so that

the remaining packets in the message are sent when the user process is restarted. Save-restore doesn't work under PIO because a partial packet is not guaranteed to contain sufficient routing information to be routable by the network. Two solutions to the PIO partial packet problem are cancel-retry and RAS. Under *cancel-retry* the network interface destroys the partially constructed packet and notifies the user program running on the sending node that the injection failed, and the user program must retry the operation when it resumes execution. Another technique is to use a *restartable atomic sequence* (RAS) [47]. The segment of the user code that sends or receives a packet is marked as a critical section. If a context switch occurs inside the critical section, the operating system kernel rolls the user task forward; it executes instructions on behalf of the user program until all the critical section code is executed.

- *Network-drain (network protection, task management).* A *network-drain* allows packets already injected to propagate through the network and eject, and it prevents the injection of new packets into the network. Some versions of network-drain cause packets to be ejected to a nearby processing node instead of the packet's destination. This option takes much less time than the worst case where all packets in the network are destined for a single receiver. One purpose of network-drain is to remove all user packets associated with the current user process, in preparation for a context switch. In systems like the CM-5, user packets do not contain a process identifier. These packets must be saved and later re-injected into the network when the user process is resumed. Other systems contain a process identifier in the packet header to allow packets from more than one user process to coexist in the network, so that a network-drain is not strictly necessary to prepare for a context switch.

- *Guaranteed delivery of operating system messages (network protection, task management).* It may be possible for a runaway user process to inject messages without ever attempting to receive them. In this case, the network becomes saturated with packets, and it can cause the parallel computer system to thrash. At worst it is necessary to shut down the entire machine and start over, which may be time-consuming and difficult. A preferred solution is for the operating system to detect this situation and issue a "cease and desist" message to the

offending user program. If user packets and operating system packet share the same network, then this strategy will be thwarted if these operating system packet do not get delivered ahead of the user packets. Therefore the network design must guarantee delivery of operating system packets regardless of the size of its workload. There are several techniques. One technque is to prioritize operating system packets, so that they are delivered ahead of user packets. Another technique is to use a separate physical or logical network for operating system packets, in which the OS packets are guaranteed to be delivered. An example of this technique is to have a separate network for barriers and eurekas. A message placed on the barrier-eureka network can signal to all nodes a network-drain of the main network. Once the main network is drained, it is guaranteed to deliver operating system packets in a timely fashion.

2.1.5 Fault handling

Parallel computer architectures are designed for scalability, to permit the construction of systems with thousands of processing nodes. Implementers and users of physically realized parallel systems face the problem of faulty connections, routers and processing nodes. Although individual components have a very low chance of failure, the aggregate chance of failure anywhere in a large system is potentially high. The computation aborts in the presence of a system failure, either because it cannot continue or because it is using incorrectly transmitted data. A failed router or processing node is therefore likely to cause the entire machine to stop functioning. A more pernicious problem is the case in which the computation completes but incorrect information was propagated through the network causing an improper result to be computed. Therefore the primary concern is detecting and reacting to corrupted packets. Corruption has many forms: bit errors, packet length errors, and packet duplication or loss. Bit errors and packet errors are detected by adding redundant information in the packet header in the form of checksums or error correcting codes (ECC). In principle the application program could supply and verify packet checksums but it would add a very large amount of processor overhead. For the highest communication performance, checksums are encoded in the network interface at the sender and verified in the network interface at the receiver automatically. A mismatched checksum causes the receiving node to drop the packet and signal an error to the operating system

to abort the computation. ECC could be used instead of a checksum, so that the error can be corrected on-the-fly without requiring termination of the computation. However, ECC is more expensive to implement in terms of network bandwidth and circuitry. If the error rate is very low then the overhead of an occasional corrupted packet is relatively insignificant, meaning that ECC is not necessary.

If the network delivers packets in-order, then detecting duplicated and lost packets is relatively straightforward. Hardware in the sender's network interface computes a sequence number for each packet which is inserted into the packet header. The network interface at the receiver verifies the packet header. For any particular message, sequence numbers are unique. Packets are sent in increasing serial order, so that a lost or duplicated packet is easily discovered by comparing the sequence numbers of two consecutive packets for that message.

If the network does not deliver packets in-order, it is somewhat more difficult to detect lost and duplicated packets. Single errors can be detected through the use of the packet counting scheme described in Section 2.2.4. Lost packets are detected through the use of an application-program timeout. Another technique is used in the CM-5 [14] – the global packet count in the entire system is continually computed and propagated to all nodes. The idea is like Kirchoff's Law, where the sum of everything that went in should be equal to everything that comes out. Detecting multiple errors is tricky, because the packet count is correct if one packet is lost at the same time that another packet in the same message is duplicated. To detect duplicates, the network interface maintains a bit flag for each sequence number for each message. The tradeoff is that it may increase the amount of memory needed in the network interface significantly.

2.2 The logical interface

The *logical interface* or *communication model* comprises the set of internode communication primitives directly accessible by the user program. A variety of different communication models have been designed and implemented, as there is no single model of communication that is the best for all application programs. There are three basic communication models: systolic, remote memory and send/receive. There are also a number of systems that use more than one of these models, and some are hybrid solutions.

2.2.1 Systolic communication

Systolic communication works much the same as the pipe abstraction supported by the UNIX^{tm} operating system. Communication requires two phases. First, the network and network interface layers set up a persistent virtual circuit between the sending node and the receiving node. Once the virtual circuit is established, the sender writes values to the pipe input register and the receiver reads from the pipe output register. When the virtual circuit is no longer required in the computation the application program must terminate it explicitly. The semantics require the data to appear at the receiver's end of the pipe in the same order as it is sent.

Systolic communication requires PIO for both the sender and receiver. For flexibility there is usually a hardware FIFO between the sender and receiver, so that the sender can send information before the receiver is ready to use it. The granularity of synchronization may be as small as a single value; that is, a packet is sent every time the sender stores a value into the pipe input register. For efficiency, multiple writes may be aggregated into a single packet. Under this implementation, a separate send command may be required to flush the pipe (i.e. to send a non-full packet). Systolic communication provides very efficient data movement. Once the virtual circuit is created, the application program passes values only as it does not need to pass node IDs or buffer addresses.

The iWarp system [23] provides a canonical example of systolic communication through its tightly-coupled interface. It uses a custom RISC processor core with on-chip message registers: two pipe inputs for sending and two pipe outputs for receiving. Information is sent immediately on every write to the pipe input register. Circuit switching is supported directly in the network architecture. The network ensures node protection because a sender can only send to a node where the circuit is already established. A limitation of iWarp is that it is limited to a single-user system; there is no hardware support for task management, meaning that it cannot be multiprogrammed.

2.2.2 Remote memory

The remote memory model is popular for scientific computing systems. There are many examples of multicomputers that use this communication model, including Cray Research T3D and T3E [19, 20, 21], Fujitsu AP1000 [48, 49], Stanford DASH [45],

Tera MTA-1 [46] and Kendall Square Research KSR-1 and KSR-2 [33].

Under the remote memory communication model (also known as remote load/store, put/get, shared memory or non-uniform memory access) processors access remote memory locations directly using load and store operations. Remote memory is actually two communication models: a remote load model and a remote store model. The entire shared memory of the parallel system uses a common global virtual addressing scheme that uniquely identifies every location (see Section 2.1.4). The upper bits of the address uniquely identify the remote node, known as the *home node*, that contains the corresponding memory location. When a processor issues a store instruction, a valid virtual address that does not map to a local memory location is placed on the address lines of the local processor bus. The network interface observes this address on the local processor bus and handshakes with the processor in the manner of a local memory module, then the network interface sends a packet to the home node. When this packet arrives at the home node, the receive interface translates the global virtual address in the packet header into a local physical address and completes the memory write to that location. Remote load works somewhat the same way, except that it requires two packets: a request packet and a reply packet. The processor issuing the load sends the request packet and waits for the reply. When the request packet arrives at the remote node, the network interface performs a memory fetch at that location and sends the reply packet. When the reply packet arrives at the originating node, the processor loads the value into a processor register and continues processing.

The remote memory abstraction requires several kinds of support in the network interface. It requires DMA for all data movement. It must provide protection for safe user-level access. The network interface must distinguish requests from replies, requiring either extra information in the packet header or separate logical networks. The receive interface must be automatic and therefore must provide a TLB to translate the global virtual address into its local physical address. Also, the receive interface under remote store provides no notification to the processor. At the application program level, it is still necessary to be able to figure out when a message has arrived. Since the interface doesn't notify, the application program must construct its own notification information by the use of a separate message, often called a sync message. When the sync message is detected at the receiving node by polling some mutually

agreed upon shared memory location, it implies that a previous communication has completed. Note that polling is the only option. This model for application-level synchronization requires the sync information to arrive after the data portion of the message has been delivered. One type of support is an in-order network: given any two nodes A and B in the network, if a packet A1 is sent from A, and packet A2 is subsequently sent, packet A2 is guaranteed to arrive after packet A1. Another technique is network drain: the entire network is first drained of packets, and then sync packet is sent afterward.

Remote memory systems span a wide variety of different styles, which are too numerous to completely encapsulate in this dissertation. Nevertheless, it is important to mention several important common subclasses and a few systems that provide unusual features.

- *Globally coherent remote memory.* This variant is also called cache-coherent non-uniform memory access or CC-NUMA. The idea is for the network interface to act as a cache for remote data. If the data is locally cached, it eliminates the need for a message to be passed, thereby reducing the latency of multiple accesses to the same location. There are several tradeoffs to globally coherent memory. Each node maintains a directory of its home cache lines that are cached on remote nodes. These cache directories often require a signification amount of additional memory. Maintaining global coherence generates network traffic which would often not be necessary under non-globally coherent systems. The extra directory memory and the coherence logic make global coherency much more complicated to implement compared with the basic model. Examples of globally coherent remote memory systems are Stanford DASH [45], MIT Alewife [26] and Convex Exemplar, based on the Scalable Coherent Interconnect (SCI) memory interface [50]. All of these systems are based on networks that provide in-order packet delivery.

- *Princeton SHRIMP* [42, 43]. SHRIMP is a hybrid of the systolic and remote store models. When two nodes wish to communicate, there must first be a logical link constructed between the two nodes via an operating system call. A segment of the sender's memory is mapped onto the receiver's; any write into locations within this segment causes the sender's network interface to send a

packet to the corresponding receiver. There is no remote load in SHRIMP. There are several advantages over a canonical systolic communication style. It uses a standard packet switching network, rather than the special-purpose circuit-switching network such as in iWarp. Furthermore there can be a comparatively large number of concurrent logical links, up to the number of entries in the TLB in its network interface. By contrast, iWarp's systolic communication is limited to two physical connections into and two out from each processing node. SHRIMP is also based on a network that delivers packets in-order.

- *Tera MTA-1* [46]. The MTA-1 is unusual for several reasons. Each network node contains either a memory module or a processor, but not both. Both remote load and remote store are necessary because processors do contain neither caches nor local memory. Each memory location contains a synchronization tag called a *full-empty bit*, as described in Section 2.1.3. A remote store to a memory location sets its corresponding F-E bit to the full state. An attempt to load from a location which is empty causes the processor to stall, and its network interface continually retries the operation until it succeeds. Success implies that another thread has stored a value to the location; the synchronizing thread could run on the same processor, or on any other processor in the entire system. The MTA-1's full-empty bit scheme requires notification information to be stored with every memory location, but it allows the use of an out-of-order network, unlike most other remote memory systems.

There are also a number of software-based schemes for supporting the remote memory abstraction without requiring explicit support for it in the network interface. These schemes require the compiler or the linker to automatically convert remote loads and stores in parallel programs into other communication primitives, such as send and receive primitives. Some examples are Ivy [51], Munin [52], Midway [53], Tempest and Blizzard [54], and TreadMarks [55]. These schemes help improve the portability of programs across parallel systems, regardless of the organization of the network interface. For some programs, the performance of these software schemes approach that of native hardware support for remote memory. While remote memory is an important abstraction for the programmer, it is not strictly necessary to incorporate this model directly into the network interface.

2.2.3 Send/receive communication

The send/receive model is also a popular communication model in scalable scientific computers. Examples of parallel systems that use send/receive communication include the Thinking Machines CM-1, CM-2 and CM-5, the Intel Delta and Paragon, Meerkat-1 and Caltech Mosaic. The send/receive model reflects the low-level abstraction of the underlying functionality of the network interface. In principle, bringing the programmer's model as close as possible to the hardware often provides the greatest opportunity for compiler optimizations. The disadvantage is that different optimization techniques may need to be developed for every implementation of the send/receive abstraction, because the implementation styles vary greatly. Under send/receive there is little restriction on the type of network, packet size or the built-in support provided by the network interface. The CM-5 [14] network interface provides the canonical example of a simple send/receive model under PIO. To send a packet, the user program stores the destination node ID, the packet payload and the packet length into the memory-mapped network interface registers. Arriving packets are placed into a FIFO that the user program can read directly.

2.2.4 Protocol support

A key to making the network interface as efficient as possible is support for message passing protocols directly in hardware. A message passing protocol involves the interaction of data movement, notification and dispatch and management of buffers. Figure 2.3 describes a taxonomy of strategies for the receive interface. Under PIO, notification always comes first, and then the processor moves the data. Under DMA there are two options. If DMA is *processor-initiated,* then it is like PIO: notification occurs first, then DMA is dispatched. If the DMA style is *automatic-receive*, the data movement occurs first, and the processor is optionally notified afterward.

There are two styles of protocol support: buffered and unbuffered. Under the buffered protocol, incoming packets are placed into buffers managed by the receiver, such as hardware FIFO or a ring-queue in the main memory of the processing node. Under the unbuffered protocol, data from incoming packets are passed directly into pre-allocated locations in memory, whose destinations are specified by the sender. The unbuffered protocol can be supported only under a particular type of automatic-receive DMA; all other interfaces use a buffered protocol. The Hamlyn network

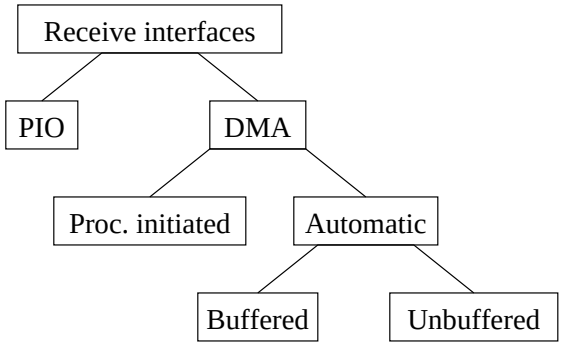


Figure 2.3: Taxonomy of the receive interface

interface from HP Labs [56, 57, 58] supports automatic, unbuffered DMA. The authors' terminology for this property is *sender-managed* communication. Hamlyn is also one of the few network interface architectures that supports a network that delivers packets out-of-order.

The principal advantage of an unbuffered protocol is the ability to filter notification information. The network interface passes a notification to the processor only when an interesting packet has been received, such as one that denotes the end of a message. The overhead of notification can be significant, on the order of tens to hundreds of processor instructions. Therefore, any notification that is not passed on to the processor reduces the overhead of communication. This reduction in overhead can be substantial when packets are small and messages are large. (Remote memory designs that do not notify the processor at all represent an extreme case of notification filtering.)

Two techniques that the receiver's network interface can use to detect the end of a message are end-of-message tagging and packet counting. Under the end-of-message tag technique there is a reserved field in the packet header set by the send interface to designate the last packet in a message. The processor at the receiver is notified only when the receiver's network interface detects the end-of-message tag. The second technique, packet counting, is similar to credit-based schemes commonly associated with asynchronous transfer mode (ATM) local and wide area networks. In the packet counting scheme, the sending node and the receiving node agree in advance on the length of the message. When the proper number of packets (credits) arrive, then the entire message has arrived. The advantage of the end-of-message tagging scheme is that it is simple, but like the remote memory model, it requires

an in-order network. The advantage of the packet counting scheme is that it can be used with either in-order or out-of-order networks.

2.3 Analysis

In this chapter we have developed a taxonomy of attributes of network interfaces, with the following categories and instances within each category:

- Physical coupling: tight, cache bus, memory bus, I/O bus.
- Network order: in-order, out-of-order.
- Data movement: systolic, PIO, DMA.
- Notification style: poll, stall, interrupt, none.
- Argument checking: direct user-level access, system call.
- Communication model: systolic, remote load, remote store, send/receive.
- Receive DMA style: processor-initiated, automatic-receive, not applicable.
- Protocol support: buffered, unbuffered.

Table 2.1 describes the attributes of eight different network interfaces: University of Washington Meerkat-1 [35], Intel Paragon, Princeton SHRIMP [42], CMU iWarp [23], Stanford DASH [45], Thinking Machines CM-5 [14], MIT MDP [22] and HP Labs Hamlyn [57, 58]. Some entries appear more than once because there are multiple attributes to many of these network interfaces. For instance, iWarp contains both systolic communication and DMA. These systems were chosen as representative examples of the wide span of possibilities that are available to the network interface designer.

In order to achieve the highest performance and lowest possible processor overhead, we compare and contrast the instances within each category. As described in Section 2.1.1, tight coupling of the network interface with the processor makes possible the highest performance and lowest overhead. Due to the difficulty of making

Table 2.1: Comparison of existing network interfaces

Name	Physical coupling	Network in-order?	Data movement	Notification	Direct user level access	Logical interface	Receive DMA style	Protocol support
Meerkat-1	Memory	Yes	DMA	Intr	No	Send/recv	Proc. initiated	Buffered
Paragon	Memory	Yes	PIO	Poll/Intr	No	Send/recv	n.a.	Buffered
SHRIMP	I/O	Yes	DMA	none	Yes	Remote store	Automatic	Unbuffered
iWarp (systolic)	Tight	Yes	Systolic	Stall	Yes	Systolic	Automatic	Buffered
iWarp (DMA)	Tight	Yes	DMA	Poll/Intr	Yes	Send/recv	Automatic	Unbuffered
DASH (rem-write)	Memory	Yes	DMA	none	Yes	Remote store	Automatic	Unbuffered
DASH (rem-read)	Memory	Yes	DMA	Stall	Yes	Remote load	Automatic	Unbuffered
CM-5	Memory	No	PIO	Poll/Intr	Yes	Send/recv	n.a.	Buffered
MDP (send)	Tight	Yes	PIO	Stall	Yes	Send/recv	n.a.	n.a.
MDP (receive)	Tight	Yes	DMA	Intr	Yes	Send/recv	Automatic	Buffered
Hamlyn	Memory	No	DMA	Poll/Intr	Yes	Send/recv	Automatic	Unbuffered

this interface style economical, the best alternative is to connect at the memory bus and use hardware to maintain cache coherence.

DMA provides the most efficient data movement for large messages. DMA allows the use of burst-mode in the memory bus, and it allows the processor to compute while a messages are arriving or being sent. DMA can also be made efficient: in remote-memory systems, a single load or store operation initiates a DMA. Send/receive interfaces can also be designed with the same property. Table 2.2 contrasts the three options for data movement: systolic (within a tightly-coupled interface), PIO and DMA. The table describes the number of memory operations needed at the receiver for each word of each message under two models: the small message model and the large message model. The large message model takes into account the need to spill message values into memory if there is not enough register space to hold the entire message at once. With the systolic interface, no load or store instructions are needed to load a word of a small message; two operations are needed if the value is spilled to memory (one store followed by one load). Under PIO, one load instruction is needed for a small message, and three memory operations are needed for a large message (load, store, load). Under DMA, no more than one load instruction is needed, because the value comes directly from memory, eliminating the spill. Systolic communication is the most efficient, but only for small messages, and it is only available in systems

Table 2.2: Number of memory operations per word per message for the three data movement types Systolic, PIO and DMA. Two types of messages are compared: small messages, in which the data value can be used immediately, and large messages, in which message values must be spilled to memory first before they can be used.

Type	Small message	Large message
Systolic	0	2
PIO	1	3
DMA	1	1

with tightly-coupled interfaces.

For large messages, the most efficient style for receive DMA is automatic-receive, using the unbuffered protocol. Message data are placed automatically and immediately into their final destination, rather than storing data in an intermediate buffer. The unbuffered protocol is best able to filter notification information, which is important for reducing the amount of processor overhead due to notification and dispatch. This is important if packets are small and messages are large. The unbuffered protocol however makes it difficult to reconstruct the sequence of events that caused the receiver's memory to be updated. For instance, say that node C receives two packets, one from A and one from B. If the information from both packets is sent directly to memory, it may be impossible for C to tell which packet arrived first. In particular, if data from both packets are written to the same location it can potentially cause a race condition. In a buffered interface, the sequence of events can be easily reconstructed by looking at the state of the ring-queue. Therefore, the buffered protocol is more flexible than the unbuffered protocol, and is a superior model for small messages.

Notification style is a tradeoff between latency and fairness. Stalling always introduces the least overhead and therefore the lowest latency – no instructions are executed while the processor is stalled, and when processing resumes, there is no overhead for restoring the state, unlike returning from an interrupt hander in which the stack and the processor registers are restored. However, consider the situation where the processor is waiting on more than one external event, e.g. a network event and a disk event. While stalled on the network event, it is unable to respond to

the disk event. Both polling and interrupts offer greater fairness than stalling at the cost of increasing the overhead. On standard RISC processors, an interrupt is more expensive than a poll. On the CM-5, which uses Sparc processors, interrupts take ten times as many processor cycles as polling [59]. Special-purpose processors have hardware support for fast interrupt handling. For example, the MDP in the J-machine can dispatch an interrupt handler to react to an incoming packet in only three clock cycles [22]. However, even if interrupts are more expensive than polling, they may introduce less overhead if they occur only in rare circumstances. For instance, for a given network architecture and application program, injection failure may be very infrequent. If the processor must poll to test if the network is available, then this cost is paid unconditionally for every packet. In the case of injecting packets on the CM-5, interrupting would be preferable to polling if the network is busy less than 10% of the time.

2.4 Summary

In this chapter we more closely examine the fundamental attributes of network interfaces that were introduced in Chapter 1: data movement, argument checking, notification and dispatch, and protocol processing. We conclude that DMA provides the most efficient data movement. Hardware support for argument checking is necessary to eliminate expensive calls into the operating system. Supporting more than one style of notification is helpful to best address a wide variety of algorithms and communication patterns. The most efficient protocol for large messages is unbuffered – it allows the network interface to shield the processor from most network events that are uninteresting to the processor. The unbuffered protocol depends on automatic-receive DMA. The buffered protocol is the most flexible for small messages.

An out-of-order network affects the styles of logical interface that can be supported directly. In-order interfaces make it much easier to support the remote memory model. If the remote memory model is supported by the network interface in a system with an out-of-order network, the processing node requires additional features, such as a synchronizing memory system like the one in the Tera MTA-1. A simpler solution is to use the send/receive logical interface instead of the remote memory model. Out-of-order networks have many advantages that make them important to support. They can provide greater throughput and lower average latency than their in-order

counterparts. They offer greater opportunity for fault tolerance due to adaptivity. Finally, they make the design of support for network drain and reinjection much simpler, wherein it may be impossible to enforce the in-order requirement. A network interface that supports out-of-order networks is universal, as it supports in-order networks as well.

We use the conclusions of this chapter to motivate the design of the Cranium network interface architecture, introduced in the next chapter.

Chapter 3

THE Cranium NETWORK INTERFACE ARCHITECTURE

Architecture is frozen music.

– *Goethe*

This chapter describes the design of the Cranium network interface architecture. The initial motivation for Cranium came from a requirement to design a high-performance companion interface to the Chaos network router [15, 60]. The Chaos router has two interesting attributes. It routes small fixed-size packets, using a payload the size of a processor cache-line (e.g. 32 bytes). It also uses adaptivity to improve its throughput and reduce its average latency; as a result, packets may overtake one another in the network, resulting in out-of-order arrival. The name Cranium comes from the acronym for Chaos Router Autonomous Network Interface for User-level Message passing. However, the Cranium approach is broadly applicable to many different network routers.

3.1 Design goals

To make it as general as possible, Cranium addresses a number of different goals:

- *Cranium provides low-latency, high-throughput communication over a wide spectrum of MPP workloads.* Traffic in networks tends to be bi-modal: most messages are small, but a substantial fraction of packets are associated with a few large messages. Cranium supports small messages efficiently through protected, direct access by user-level programs. Cranium uses DMA to support large messages efficiently and it reduces the amount of processor overhead by filtering unnecessary notifications.

- *The processor and memory in the computing node are built using commodity components.* A tightly-coupled interface between the network and the processor

was ruled out. Designs that require synchronizing memory or multi-threaded processors in an architecture like the Tera MTA-1 [46] were also ruled out. Cranium is coupled to the memory bus and takes advantage of hardware cache coherence that is built into most modern high-performance processors (see Section 3.4.2).

- *Cranium provides support for general-purpose scientific computing.* The system must run a wide variety of different compute-bound tasks. It is important to be able to run several different jobs concurrently and to provide efficient management for different user contexts (see Sections 3.3.3 and 3.4.3). This requirement rules out a special-purpose approach like that of iWarp [23], an architecture that is best suited to systolic array processing applications such as real-time vision and image processing.

3.2 The difficulty of interfacing with adaptive routers

To illustrate the difficulty associated with the design of a network interface for an out-of-order network, we present an interface based on the remote memory model. As described in Section 2.2.2, the remote memory model has a number of advantages. It filters out notification information to the processor, making it more efficient than an interface that must notify the processor for every arriving packet. Synchronization between sender and receiver is provided by an additional sync packet that occurs at the end of a message. For example, say that processing node A sends a message to node B that consists of four data packets, P1 through P4. The sync packet becomes the fifth packet, P5. Under an in-order network the arrival of P5 tells node B that all the data packets have arrived. Under an out-of-order network, the arrival of P5 does not guarantee that all the data packets have arrived. If P5 arrives too early, then it must be delayed by the network interface.

Figure 3.1 is a block diagram of a network interface architecture that reorders packet arrivals. The receiver's interface contains an outboard memory that stores packets arriving from the network. As packets arrive, they are reordered and propagated to the processing node's DRAM. Table 3.1 describes a potential packet arrival ordering from the network, and the resulting sequence of packets emitted from the outboard memory.

Table 3.1: Packet reordering example. The first column describes a time line with seven discrete timesteps T1 through T7 that are in increasing time order. The second column describes a possible arrival order of packets coming from an out-of-order network that are placed into an outboard memory. The third column describes the timing of packets that are transferred from the outboard memory into system DRAM. If the packet stream is completely in-order, arriving packets are transferred immediately. If packets arrive out-of-order, packet deliveries to the processor node are reordered, but bubbles must be inserted into the packet stream.

Time	Arriving packet	Departing packet
T1	P1	P1
T2	P3	.
T3	P4	.
T4	P2	P2
T5	P5	P3
T6	.	P4
T7	.	P5

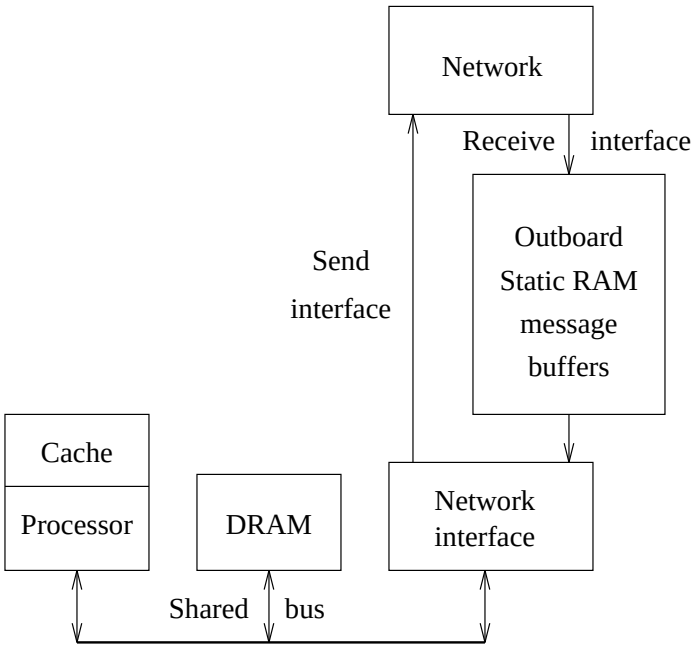


Figure 3.1: Architecture of a network interface to support remote memory with an out-of-order network

The reordering scheme described by Figure 3.1 is feasible, but it is not a good solution. Table 3.1 demonstrates the performance problem. Note that the reordering interface can only output one packet during one time slot. Out-of-order packets cause bubbles to appear in the stream. The effect is to increase the latency of a message and decrease its throughput considerably. The performance problem can be reduced but not completely solved by increasing the bandwidth between the outboard memory and system DRAM.

A second problem with this architecture is the complexity of the design. A typical implementation of a scalable system might have 256 processing nodes and a message size of up to 8K bytes (the size of a typical page frame in current processor technology). The outboard memory would need to be large enough to handle a worst-case situation involving all-to-all communication of page-sized messages. The network interface at each processing node would therefore need a 2 megabyte outboard memory. The outboard memory would need to be constructed from static RAM to achieve high performance. Such a network cache would be similar in size and performance to a processor's external cache.

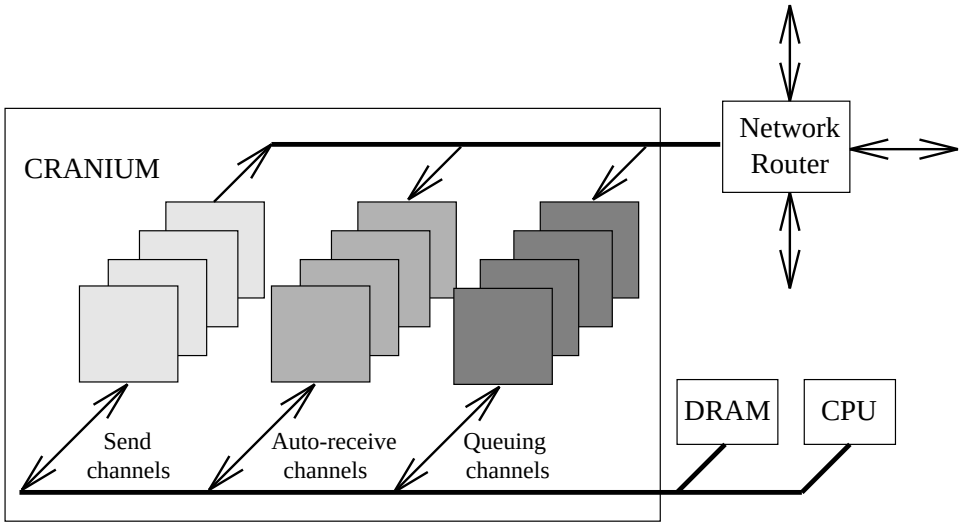


Figure 3.2: Cranium architecture

The approach described in this section is complex yet offers comparatively poor performance. The problem with the combination of the remote memory model and an out-of-order network is the cost and complexity of regenerating the necessary synchronization information. By basing Cranium on a send/receive model, much of the cost and complexity in the network interface can be eliminated. It becomes unnecessary to insist on a particular arrival order of packets – it is only necessary to count the number of packets associated with a particular message and re-assemble the message in memory. Packets arrive directly into the processing node's DRAM without temporarily residing in an outboard memory. Furthermore, the sync packet can be eliminated entirely. In Table 3.1, this means that packet P5 can be eliminated. Given the same arrival order of packets P1 through P4, the complete message arrives at time T4 instead of T7, resulting in much lower latency and achieving a higher percentage of the full bandwidth of the interconnect.

3.3 Cranium implementation-independent architecture

This section discusses the fundamental concepts of the Cranium architecture. Figure 3.2 is a block diagram showing the structure of Cranium and Table 3.2 describes its attributes. Two types of receive interface are supported: an unbuffered interface called the *auto-receive channels* and a buffered interface called the *queuing channels*.

Table 3.2: Attributes of the Cranium network interface architecture

Name	Physical coupling	Network in-order?	Data movement	Notification	Direct user level access	Logical interface	Receive DMA style	Protocol support
Cranium (auto-recv)	Memory	No	DMA	Poll, Intr	Yes	Send/recv	Auto, pkt count	Unbuffered
Cranium (queuing)	Memory	No	DMA	Poll, Intr	Yes	Send/recv	Auto, ring-queue	Buffered

The send interface contains the *send channels* and mirrors the unbuffered receive interface in form. The send and auto-receive channels support DMA transfers up to the size of an MMU page. Each queuing channel manages a separate ring-queue. Each channel represents a complete context for a message, including the physical address of the local message buffer, the remote node name, the number of packets to send or receive, and transfer completion status. The send channels convert long messages into separate packets; the auto-receive channels re-assemble these packets into messages.

Cranium is activated by channel commands issued by the processor. Cranium then autonomously schedules and executes its channel operations independent from and concurrent with processor execution. Cranium is multithreaded in the sense that multiple message commands can be in progress simultaneously. Cranium provides protected direct access to user-level programs. Channel registers are loaded and stored directly using memory mapped read and write operations. The overhead of sending and receiving messages is minimal, on the order of a few user-level instructions.

Figure 3.3 describes the format of a Cranium packet. Packets are composed of three fields: a network header, a Cranium header and the packet payload. The network header is used only by the network routers. The packet payload is the size of a cache line, typically 16, 32 or 64 bytes. The Cranium header contains the process ID, the send channel number, the auto-receive channel number, a message sequence number, some miscellaneous flags including the queue flag and the system flag, and a redundancy code to protect the Cranium header.

3.3.1 Send channels and auto-receive channels

The send channels and the auto-receive channels are symmetric; the programmer model for both sets of channels is nearly identical. At any time, a send channel or auto-receive channel is either in the idle state or in an active state where it is transmitting or receiving packets. Each activation of a channel initiates a block

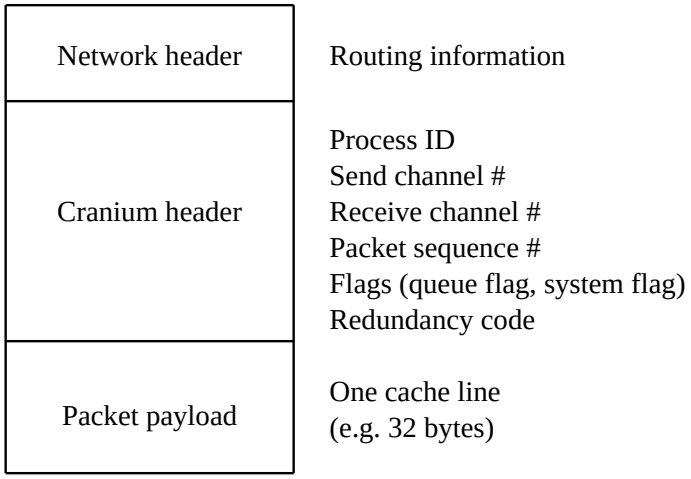


Figure 3.3: Cranium packet format

transfer of up to an MMU page. If a cache line is 32 bytes and an MMU page is 8K bytes, then there are up to 256 packets per channel command. Each send channel and auto-receive channel maintains a packet count, a node ID and a physical buffer base address. The packet counter in the send channel is copied into the sequence number field in the packet header. For each packet that is sent, the physical address of the payload data is computed by adding the physical base address to the packet counter times the size of a cache line. The auto-receive channels place incoming packet data into memory in the proper location by the same offset from the receiver's physical base address. The processor can load the packet counter value from the send and auto-receive channels to determine completion status of the transfer. Optionally, Cranium can interrupt the processor when the counter reaches zero, indicating that the transfer has completed. Note that any send channel can send to any auto-receive channel at any node. A restriction is that an auto-receive channel must be activated before the send channel starts sending packets to it. A protocol error is signaled when a packet arrives into an inactive auto-receive channel, or if one or more of the fields in the packet header do not match that expected by the auto-receive channel.

3.3.2 Queuing channels

The queuing channels implement the buffered interface. Packets in the queue are not serialized by the interface; they appear in the ring-buffer in memory in the order they are ejected from the network. Queue memory is implemented using main memory;

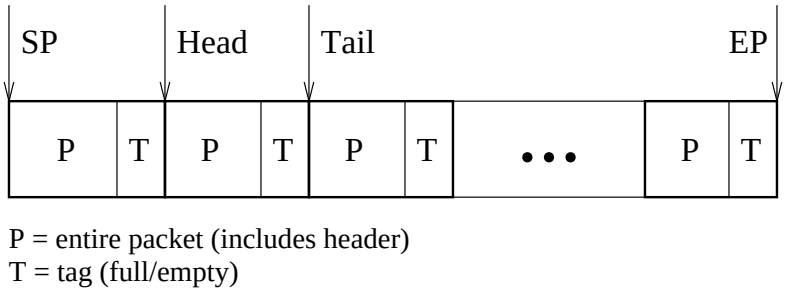


Figure 3.4: Organization of circular buffer in DRAM for queuing channel

queue buffers are locked into the physical memory map. User programs access queue memory directly using load and store operations. By using main memory to hold the ring-queues, it becomes possible to reduce the size of the hardware FIFO in the receive interface. DRAM memory is much cheaper and lower power than comparably sized SRAM needed in the FIFO. Main memory storage also allows the operating system to change the size of queue memory when the user program is started, rather than requiring the network interface hardware to be reconfigured. Like the auto-receive channels, data is moved first by DMA and the processor is notified afterward. Unlike the auto-receive channels, every packet that arrives into the queuing channels sends a notification to the processor.

There are four queuing channels: the user queue, the user error queue, the system queue and the hardware error queue. Packets that have the queue flag in the packet header enabled are routed to the user queue. The user error queue is used for packets that signal a protocol (soft) error (see Section 3.3.4). Both the user queue and the user error queue buffers are mapped into user space.

Figure 3.4 shows the organization of queuing channel memory. For each queuing channel, the channel context is a set of four pointers into main memory, called the start pointer (SP), the end pointer (EP), the head pointer and the tail pointer. Queue memory is organized as a circular buffer based on a simple producer-consumer protocol, with Cranium as producer and the user program as consumer. The network interface places incoming packet data at the tail pointer and the user program accesses packet data from the head pointer. At initialization, the start pointer, the head pointer and the tail pointer all point to the start of the queue buffer. When a packet arrives, Cranium writes the entire packet (including header) into the packet

field (P), writes a nonzero value[1] to the tag field (T) and advances the tail pointer. The user program detects the presence of a packet by polling the tag field. If a packet is present, the user program reads the packet information, processes it, then executes an Advance Queue operation to clear the tag field and advance the head pointer to release the space back to Cranium. If advancing the tail pointer causes it to become equal with the head pointer, then Cranium signals queue overflow. If an Advance Queue operation is executed when there is no packet in the queue (i.e. the head and the tail pointer are equal), the network interface signals queue underflow. A condition of either overflow or underflow will cause the network interface to interrupt the processor (see Section 3.3.4).

3.3.3 Protection

Cranium provides the protection features that were outlined in Section 2.1.4: address mapping, logical node identifiers, atomic packet injection, network drain and guaranteed delivery of operating system messages. To implement protected, safe user-level access, Cranium does not allow the user program to write the physical base address of the send or auto-receive channel directly. Similarly, the user program cannot load the node ID directly, because there may be destination nodes that the user should not be able to access. In each case there is a level of indirection provided by mapping tables, one for node IDs and one for buffer addresses. Each mapping table occupies the node's DRAM in a protected location, accessible only to the operating system and pinned into physical memory. User programs can specify only the indices into these tables. Cranium performs a table lookup in each case. If the table entry contains a valid value, then the translation succeeds. If the table entry contains an invalid value, then the translation fails and the transfer is canceled.

Figure 3.5 shows the two mapping tables: the node map and the buffer map. The network interface contains a pair of hardware registers that contain the base physical addresses for each mapping table. `Node_Map_Ptr` points to the base of the node map, and `Buf_Map_Ptr` points to the base of the buffer map. An entry in the node map is the physical identifier for a remote node, or zero to indicate an unmapped node

[1] One option is to write a 64-bit timestamp into the tag field to represent the nonzero value (the least significant bit is always 1). Timestamp information can be used by the user program for performance analysis.

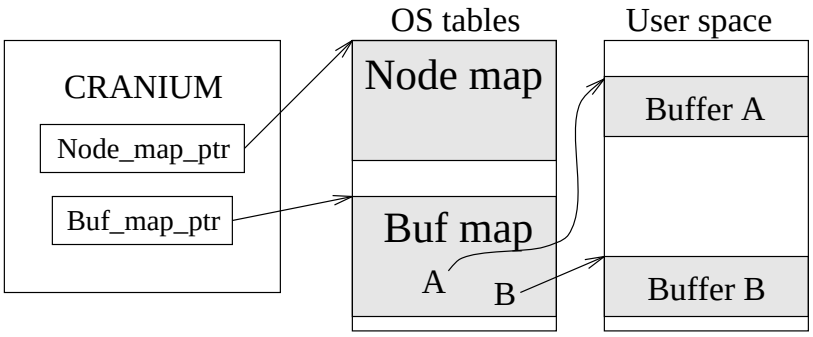


Figure 3.5: Protection for safe user-level access via mapping tables

entry. An index into the node map (i.e. the offset from the base address of the table) is called a *node handle*. An index into the buffer map is called a *buffer handle*. In order to construct entries in the node map or the buffer map, the user program must call the operating system and pass a user virtual address or node identifier as an argument. The OS performs the mapping and returns the handle to the user program. When a buffer is mapped, the OS must also pin its page into physical memory. For reasons of efficiency, it makes sense for the programmer to map all message buffers during an initialization phase of the user program, so that all subsequent network commands are performed at user level. Figure 3.5 shows two user buffers, A and B, that the user program has registered with the OS and thereby entered into the buffer map. When the user program wishes to send a message to node X using buffer A, it passes X's node handle and A's buffer handle to a Cranium send channel. The interface performs a table lookup on each handle to verify that their corresponding table entries contain valid data, and places the results of the lookup operations into the send channel's node ID and physical buffer address registers. An advantage of using handles for argument checking and mapping is that it requires the user program to pass only a small number of bits to Cranium instead of providing an entire global virtual address (GVA). The size of a GVA is 48 to 64 bits in current high performance processor architectures. If a Cranium buffer handle is 16 bits and the size of a page is 8K bytes, then the total amount of address space that can be devoted to message buffers is $2^{16+13} = 2^{29}$ bytes. It is impossible for a user program to spoof handles for either message buffers or remote nodes. An attempt to issue a send or receive command using handles that have not been granted by the OS causes the network interface to load an invalid value and abort the operation.

The operating system can control the flow of packets into and out from Cranium by setting the control flow state. The NORMAL state is used for normal operation. The FLUSH state is used to drain the network: packets are ejected but not injected. In the FREEZE state, Cranium does not allow packets to be injected or delivered. Cranium also has the ability to save the state of the active channels representing the DMA transfers in progress belonging to the current user process. The OS specifies a location in DRAM to write out this information. This feature ensures that partially injected multi-packet messages will complete when the user process is restarted.

The Chaos network does not provide a separate data network for operating system packets. In order to guarantee the delivery of system packets, the system relies on the existence of a separate control network dedicated to global operations such as barriers and eurekas. The Express Broadcast Network (EBN) [61] describes the design and implementation of one such control network that can be used in conjunction with the Chaos data network. In the presence of an overwhelming amount of congestion that may prevent operating system packets from even being injected, any node may send a broadcast message to all the other nodes to put their interfaces into the FLUSH mode. The inclusion of this kind of secondary network is necessary to create a robust system based on the Chaos router. It also provides the opportunity to improve its performance substantially. Further descriptions of the interaction between Chaos and the broadcast network are given in Section 4.3 and Section 6.1.3.

3.3.4 Error handling in Cranium

When Cranium encounters a run-time error, it sends an interrupt to the processor. There are two kinds of errors: protocol errors and hardware errors. Protocol errors are caused by an illegal access to Cranium by the user program, such as trying to initiate a new command on an already active channel, passing the handle of an unmapped node or buffer, or accessing a privileged channel. Hardware errors arise when a packet arrives from the network, but Cranium cannot process it normally, so the operating system must intervene. There are several different kinds of packet errors and corresponding error handling mechanisms.

- *Network hardware error*. Symptom: Redundancy-code mismatch on incoming packet. Action: the packet is sent to the hardware error queue.

- *Protocol error*. Symptom: either the auto-receive channel for the packet is inactive, or the packet header does not match the expected information in the channel context[2]. Action: the packet is sent to the user error queue.

- *Queue overflow*. Symptom: the tail pointer runs into the head pointer when a packet arrives (see Section 3.3.2). Action: hold packet data, wait for operating system command. To prevent live packet data from being overwritten, flow control is automatically set to the FREEZE state and is thawed when directed by the operating system. To ensure that queue overflow rarely occurs, frequently accessed queues (such as the user queue) must be made sufficiently large even in the presence of highly bursty traffic.

- *Queue underflow*. Symptom: the user program pops the user queue while it is empty. Action: do nothing (just interrupt the processor).

3.4 Cranium implementation-dependent architecture

The following subsections discuss features of Cranium that are left to the implementation: the packet scheduling algorithm, cache coherence strategy, multiple user contexts and gather-scatter support. Except for gather-scatter support, changes at this level of the architecture are invisible to the application program except that they may reduce the execution time.

3.4.1 The Cranium scheduler

If two or more send channels are active, the Cranium architecture does not specify the order in which the packets are sent. For the sake of correctness it does not matter if the packets are sent in an interleaved or a non-interleaved fashion. From a performance standpoint, different strategies have different performance implications. The simplest scheduling algorithm for the Cranium scheduler is FCFS (first-come, first-served). Send channel commands under FCFS are not pre-emptive; they simply run to completion. FCFS has the drawback that small messages can get stuck waiting behind large messages in progress. If the small messages are on the critical

[2] For instance, the source node ID in the packet header does not match the source node ID of the auto-receive channel.

path, then the large messages cause a loss of performance. Different scheduling algorithms can improve the performance of small messages. One example of an algorithm that improves performance in this setting is round-robin (RR). Under RR, the scheduler switches to a different send channel after every packet is sent, analogous to the behavior of a multithreaded processor such as the Tera MTA-1 processor. Priority-round-robin (PRR) is a simple modification to RR employing a single priority bit per queue entry to designate high or low priority. The priority bit comes from the host processor as part of the command word. If the priority bit is set for the send command at the head of the scheduler queue, the behavior is the same as in FCFS; if the bit is not set, the scheduler treats it the same as in RR.

Another packet scheduling algorithm that has been discussed in the literature is alpha scheduling [62, 63]. In alpha scheduling, the priority of a message is a function of its arrival time, the length of the message and a tuning factor called alpha (α). The priority of a message is $t_{\text{arrival}} + \alpha \times l$ where t_{arrival} is the arrival time and l is the length of the message in bytes. The lower the value, the higher the priority. If α is zero then the behavior is the same as FCFS; if α is much greater than 1, then the behavior is Shortest Message First (SMF). Like RR, SMF favors short messages over long messages. A weakness of SMF is the potential for starvation; a continual stream of commands for short messages will prevent the completion of long message commands. Selecting an appropriate value for α permits good performance while avoiding starvation. The drawback of alpha scheduling is that it is complicated to implement. The ability to insert a new command into the queue in constant time requires hardware support for a priority queue. It also requires an ALU to compute the multiply-accumulate function. Therefore, alpha scheduling is not likely to be implemented in a Cranium-compliant network interface.

3.4.2 Support for cache coherence

Cranium uses DMA for all movement of packet data; it is used in conjunction with the send channels, the auto-receive channels and the queuing channels. As discussed in Section 2.1.2, DMA capability in the network interface introduces a cache coherence problem. The preferred solution is to use the bus snooping capability that is included in most modern high performance processors. Because processor designs vary, the design of the cache coherence strategy is implementation-dependent. For

economic reasons, processors that support cache coherence need to support symmetric multiprocessing (SMP) efficiently. Because SMP and MPP systems have different requirements, it is important to identify the cache coherence capabilities targeted for SMP that can be used efficiently for an MPP node.

Cache coherence protocols are maintained on a cache-line basis. Each processor in an SMP snoops memory transactions on the shared bus and updates the state of its internal cache lines in response. States for cache lines a typical SMP system are called Invalid, Dirty-Exclusive, Clean-Exclusive and Clean-Shared. Invalid means that no processor maps the cache line. The exclusive states describe the case where exactly one processor currently maps the line in its cache. The shared state means that two or more processors currently map the cache line. In the nominal case for MPP assumed in this chapter, there is only one processor and one network interface per processing node. The network interface itself does not maintain a memory cache. With only one processor there is no sharing. This aspect can help simplify both the implementation and verification of the system by eliminating cache states that are never used.

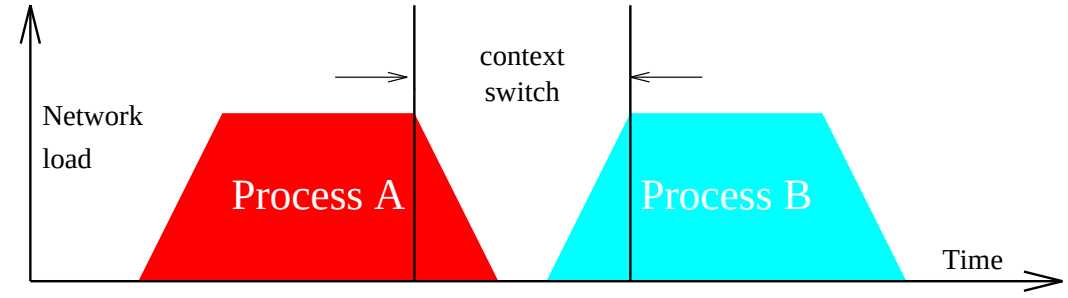
There are two cases to consider for maintaining coherence: message sending and message receiving. Sending is handled simply. If the cache line is invalid or clean, message data come from memory. If the line is dirty, then the processor overrides the memory module and places the cached data on the memory bus for the network interface to inject by DMA into the network. The line remains dirty in the cache afterward. Receiving is the more complicated case. When a packet arrives and network interface sets up a DMA-write to memory at an address that corresponds to a clean cache line, the processor can choose to either invalidate the cache line or update it using the incoming data from the memory bus. A particular strategy that works well for both MPP and SMP is multi-level, a strategy supported by the bus used in the Alpha Demonstration Unit (ADU) [64] (see Section 7.1.2). The message data are written to memory and selectively invalidated or updated in the processor cache. This model requires the processor to have a multi-level cache with the inclusion property: everything valid in level k is also valid in level $k+1$. If the line is valid in both L_k and L_{k+1}, it is invalidated in L_k and updated in L_{k+1}. If the line is only valid in L_{k+1} but not in L_k it is invalidated everywhere. The benefit of multi-level is that it adapts to the size of the message. If the processor is actively polling a section

of memory (e.g. waiting for a packet to arrive into the user queue) then the line is kept clean in first level cache, and it will be updated in the external cache when the packet arrives. Otherwise, the line is not kept in cache and the message simply goes to memory. The polling strategy helps reduce the latency for small messages, and the processor cache is not updated indiscriminately for large messages.

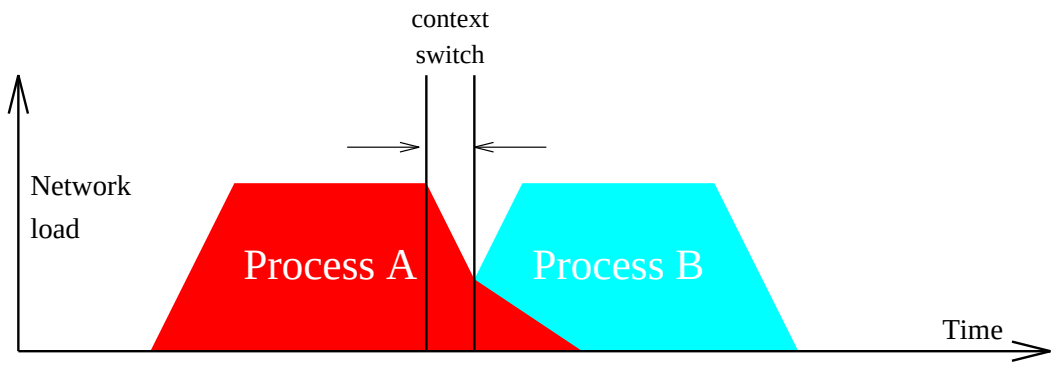
3.4.3 Multiple user contexts

The basic Cranium architecture contains only one physical set of channel registers. There are cases where it becomes desirable to include additional physical resources to support two or more user contexts efficiently. The first case is that the entire parallel system is a hybrid, consisting of an MPP in which each processing node is an SMP. Each processing node is capable of running multiple user programs concurrently. For full generality, each user program in the SMP node should be able to send and receive messages without interference from other user programs. The second case is to improve the performance of context switching in a uniprocessor node of an MPP. The usual technique for scheduling work on a MPP is to use *gang-scheduling*, in which all threads belonging to one user process run concurrently. When another user process is ready to run and the scheduling quantum expires, all the processors in the MPP switch to the new user task. All the relevant state in the network interface including the buffer map and the node map must be saved and later restored by the operating system. The performance problem of context switching in an MPP is illustrated by Figure 3.6a. The trapezoid on the left represents the packet traffic injected by user process A; the height of the trapezoid represents the saturation point of the network. At the beginning of a context switch, packets from process A are drained from the network. Once the network is empty, packets from the next runnable process (process B) are injected into the network by the operating system. When all packets for process B have been re-injected, process B is allowed to start running again. This sequence of saving packets and re-injecting them is time-consuming and difficult for the operating system designer to implement.

The solution for both the hybrid MPP-SMP system and the slow context switch problem is to include multiple sets of channel registers in the network interface. In the hybrid system, each active user context in the SMP node corresponds to a physically separate register set. To improve the performance of context switching in MPP, we



(a)



(b)

Figure 3.6: The performance implications of timesharing on a multicomputer. In subfigure (a), at any instant in time the only packets in the network belong to a single user process, either process A or process B. When the operating system grabs control and switches the context from A to B, it must first drain the network of all packets belonging to A and store them as part of A's process context. In subfigure (b), packets from process A remain in the network when the operating system switches context. Packets from both processes A and B co-exist simultaneously in the network; the process ID is denoted by a field in the packet header. If a packet for process A arrives while the processor is running process B, the network interface acts on behalf of process A to place its payload into the proper area of memory. The result is that (b) provides a faster context switch while maintaining the same degree of protection as in (a).

use two sets of channel registers per node. One set of registers is dedicated to the departing user process and the other is dedicated to the incoming user process. In this implementation it is not necessary to drain the packets of the departing process from the network (Figure 3.6b). Memory pages for the departing process are kept pinned. The result is that there is no operating system overhead for packets that arrive for the most-recently-run process. A generalization of this technique is to include physical resources for U+1 user contexts, where U is the number of processors in an MPP processing node.

There are two techniques for implementing multiple user contexts in Cranium. In an integrated approach, two physical user contexts are placed on the same chip. In a discrete approach, there is only one physical user context per chip, but it is simple to tile the chips to scale the amount of resources to the particular node architecture. The integrated approach reduces the chip count but it does not scale easily. The discrete approach is more flexible as it applies to both MPP and hybrid MPP-SMP systems. The tradeoff is that there is some additional complexity involved. The processor-network link must connect to each network interface in the SMP node, all of which are attached to the memory bus. When a packet arrives, all network interfaces observe it but exactly one network interface must handle it, to avoid packet loss or duplication.

3.4.4 Gather-scatter support

It is common in parallel application programs to move selected data from different parts of memory by means of a single data transfer. For instance, consider the case where the application program allocates an N dimensional array at each processing node. Only one of the N dimensions can be allocated in contiguous (consecutively-addressed) memory. In Section 6.2.2 we introduce a parallel benchmark application called Jacobi, whose primary data structure is a large two-dimensional array of floating-point values. Row values are passed to the north and south nearest neighbors of each processing node, while column values are passed to the east and west nearest neighbors. Either the row values or the column values are in contiguous memory, but not both at the same time. The impact is that under standard DMA, one of the dimensions cannot be sent as a single message. The programmer has two choices. The first choice is to issue a separate message for each array element. The second choice

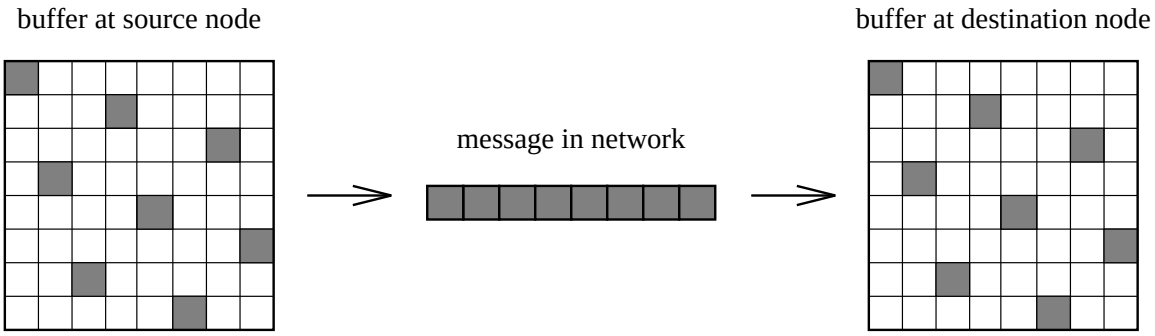


Figure 3.7: General idea of gather-scatter support in the network interface. Data not allocated contiguously (e.g. column data in a row-major-order array) are gathered into a single contiguous message that is packetized and injected into the network. When packets arrive at the receiver, the contiguous message is recreated, and the message data are scattered into discontiguous memory. In Cranium, the minimum grain of allocation is a cache line; in other network interfaces, the grain size may be as small as a single byte.

is for the sender to *marshal* all of the array elements: a contiguous block of memory is allocated, and the program copies all of the array elements into this memory block. Through DMA the entire memory block is sent as a single message. At the receiving node, the inverse operation occurs: the data arrive as a contiguous message, and the processor un-marshals the data by distributing the elements of the array to their non-contiguous locations. Both solutions cause a reduction in performance over the case where the message data are contiguous in memory, because the host processor performs expensive data copying.

Gather-scatter functionality is an extension to the standard direct-memory-access hardware. Gather-scatter hardware performs the same function as marshalling without copying the data: the data are gathered at the sender into a contiguous message, and scattered at the receiver into non-contiguous memory. There are two common forms of gather-scatter DMA hardware in I/O interfaces. The first case is for access to array data, in which it is sufficient for the sender to increment the memory address using a constant stride. However, the receiver must be more sophisticated in the case where packets arrive out-of-order, as is the case with the Chaos network. In Cranium the address must be computed using a shift-accumulate or multiply-accumulate func-

tion involving the sequence number field of the packet. The second case is completely general, where the stride is not consistent from element to element. In this case, a lookup table at the sender and the receiver can be used with the sequence number as the index into the table. The host processor loads the addresses into the lookup table at both the sending and receiving nodes. The lookup table solution is more general than the constant stride version, but implementing the table increases the amount of memory needed.

3.5 Summary

The Cranium network interface architecture provides a framework for constructing processing nodes of a scalable parallel computer system from commodity processors and memories. Cranium connects at the memory bus of the processing node, providing higher communication performance than interfaces that connect at the I/O bus. Cranium is designed to work well in the presence of bi-modal network traffic: few large messages and many small single-packet messages. Large messages are most efficiently handled by unbuffered, automatic DMA in the form of the auto-receive channels that provide packet counting. Small messages are handled by the queuing channels.

Cranium's channels permit the architecture to be scaled. A simple, low-cost implementation can be based on a small number of channels. Greater performance can be achieved by scaling up the amount of resources in the network interface. The implementation of Cranium described in Chapter 7 contains 32 send channels and 32 auto-receive channels. The amount of memory required is on the order of a few kilobytes, a tiny fraction of that required by the interface design in Section 3.1.

Cranium provides complete support for protected, safe access by user programs. It performs argument checking through the use of buffer mapping and node mapping tables. These tables are indexed by handles that are passed to the user program from the operating system. Using handles instead of physical addresses reduces the number of bits that the user program must pass to the network interface. It also eliminates the need for a content-addressable memory that is typically used to implement a translation-lookaside buffer. Cranium requires only regular system DRAM to hold the translation tables.

The buffer mapping feature of Cranium is very flexible. There is no distinction between a send buffer and a receive buffer; the same buffer can be used to collect a message from one node and then pass it to another node without copying. A single buffer can be used for *multicast*, that is, to send several messages at once to a set of nodes. Note also that the buffer map and the node map are completely orthogonal. This technique contrasts with SHRIMP [42], in which an entry in the buffer map and the node map are tied together; a given buffer can only be transferred to its associated node. This coupling between buffer and node is less flexible than Cranium's orthogonal approach. Further description of the operating system interface to Cranium is provided in Section A.3 in Appendix A. It discusses support in Cranium for atomic packet injection, network drain and guaranteed delivery of operating system packets.

The mechanisms in Cranium were designed to provide the lowest possible communication overhead to the processor, in the context of a scalable architecture that can be adapted to many different networks. The remainder of this dissertation provides the proof of the architectural concepts developed in this chapter. Chapter 4 describes the Cranium software interface. Chapter 5 describes the test environment, that was used to run message-passing programs on a simulated Cranium-based system and provide timing information. Chapter 6 describes an analysis of the mechanisms of Cranium and provides measurements of a number of parallel programs. Chapter 7 describes Teschio, a implementation of Cranium that provides an interface between the DEC Alpha processor and the Chaos network router.

Chapter 4

THE Cranium SOFTWARE INTERFACE

> *We can never, never describe* all *the features of the total situation, not only because every situation is infinitely complex, but also because the* total *situation is the universe. Fortunately, we do not have to describe any situation exhaustively, because some of its features appear to be more important than others for understanding the behavior of the various organisms within it.*
>
> – *Alan Watts*, The Book

This chapter describes the Cranium application program interface (API). The API provides the user with all of the essential services of Cranium: registering message buffers with the operating system, sending messages, receiving messages using the buffered interface, receiving messages using the unbuffered interface, detecting the completion of a message transfer and error detection. Careful attention was paid to the organization of the Cranium API. There are a number of criteria that must be considered in the design of an API for any set of services that are external to the user's application program:

- *Complexity:* the difficulty of implementing the interface from the service provider's point of view.

- *Performance:* the influence that the API has on the user's ability to extract the underlying performance out of the external service.

- *Separability:* the applicability of the API to different services or different implementations of the service. A file system API is designed to be separable as it applies to many different sizes and types of physical media. A low-level device driver is more typical of a non-separable interface.

- *Usability:* the ease of writing application programs that make use of the services accessed through the API.

For the particular instance of the Cranium API, the service provided is the ability to pass messages between different threads of control that execute on separate processing nodes. There is a tension in network interface design between deep layering for separability and usability, and thin layering for high performance and low complexity. Because the primary design goal was to produce an API with the highest possible performance, the layer between the application program and the network is as thin as possible. Although thin layering often results in an interface that is difficult to use, the Cranium API nevertheless provides good usability.

This chapter is intended as an introduction to the Cranium API, so that the reader can grasp the basic concepts without having to read code. The full API is described in depth in Appendix A; the reader is referred there for detailed explanations of all the data structures, operations and code examples. Here is the organization of this rest of this chapter. Section 4.1 explains how messages are sent using Cranium. Section 4.2.1 describes the two mechanisms for receiving a message, the auto-receive channels and the user queue. Section 4.3 motivates the use of synchronization and its interaction with send and receive. Section 4.4 covers miscellaneous topics such as interrupts and error checking. In Section 4.5, the Cranium API is compared with Intel NX and active messages (AM). Finally, Section 4.6 summarizes the chapter.

4.1 Sending a message under Cranium

The basic scenario for sending a message under Cranium is described by the flow diagram in Figure 4.1. The circle containing the S represents the thread of control that is sending the message. Each rectangular box in the flow diagram represents a step involved in sending the message; each step is small and can be implemented easily using a few lines of C code. The first step is the allocation of a DMA buffer. The next step is to store the contents of the message into this DMA buffer. A command word is then constructed that describes the entire context for the message: the receive node ID, the sender's DMA buffer handle, the destination for the packet (either the user queue or an auto-receive channel) and whether the packet causes an interrupt at the receiver. The send operation is kicked off by storing the command word into a Cranium send register. There is an array of memory-mapped send registers that correspond to the physical send channels. By polling a status bit, the program can determine when the send operation has completed at the sending node and all packets

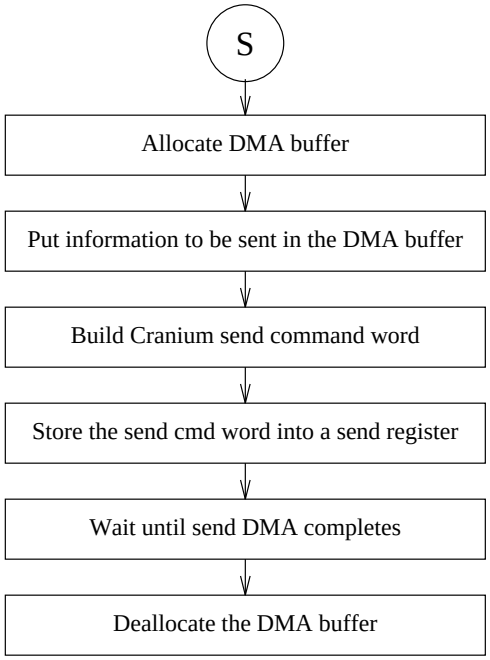


Figure 4.1: Flow diagram for sending a message

in the message have been injected into the network. The program can then safely deallocate the DMA buffer.

If the send command is used multiple times or in a loop, then the sequence of steps to send a message can be streamlined. Allocating and deallocating a DMA buffer requires an expensive operating system call. It is more efficient to allocate a DMA buffer once and then use it for many different message transfers. It is also efficient to pre-compute a set of command words that can be readily loaded into Cranium send channel registers. By allocating DMA buffers and creating command words in advance, the only operations that are necessary in the body of the loop are represented by the second, fourth and fifth boxes in Figure 4.1. This streamlining greatly reduces the processor overhead of sending messages and improves the performance of the application program.

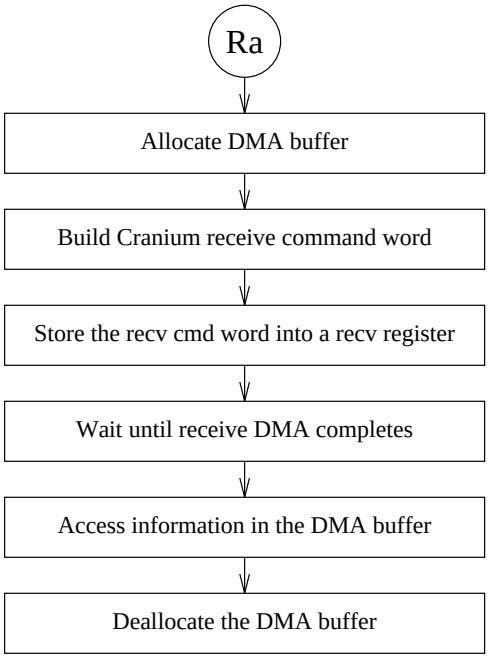


Figure 4.2: Flow diagram for receiving a message in an auto-receive channel

4.2 Receiving a message under Cranium

4.2.1 Unbuffered communication

The Cranium auto-receive channels are used in a fashion similar to the send channels. Figure 4.2 shows the flow diagram for receiving into the auto-channels; it is nearly the same as Figure 4.1, except that the information in the DMA buffer is accessed after the message arrives, rather then before it is sent in the sending case. The receive command word is constructed in the same way as the send command word, indicating the handle of the local DMA buffer and the context of the sender's environment. The receive operation is kicked off by storing the command word into a Cranium receive register. By polling a status bit, the program can determine when the receive operation has completed when all packets in the message have arrived.

4.2.2 Buffered communication

Figure 4.3 is a flow diagram describing the actions required by the program to access packets that arrive into the user queue. Queue memory is pre-allocated for the user program by the operating system. When a packet arrives it appears in the user's

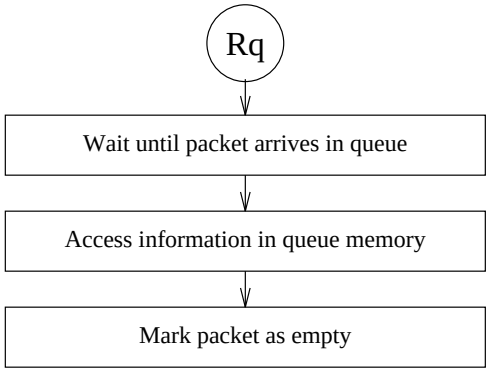


Figure 4.3: Flow diagram for receiving a message in the user queue

protection domain (address space). The user program takes whatever information it needs from the packet and then it must eventually return the buffer to the queue by marking the packet as empty. In essence, the allocation step comes at the end rather than at the beginning. While accessing the user queue seems to involve fewer steps than the auto-receive channels do, the steps in Figure 4.3 are repeated for every packet of the message. The steps outlined in Figure 4.2 suffice for receiving an entire message consisting of multiple packets, up to an MMU page in length. For a long message, the auto-receive channels are more efficient than the queue.

4.3 Synchronization

The semantics of an auto-receive channel require the receiver to be ready before any packets destined for that channel arrive. If the receiving node has not posted the receive command into its auto-receive channel register, then a protocol error is signaled for each packet that arrives for that channel. To ensure correct operation, we must guarantee that the receiver is ready before the sender sends. (This requirement is not necessary for the user queue – the queue is always in a ready state as long as queue memory is not full.) We use robust message passing protocols to ensure that the program semantics are repeatable and portable across implementations.

Figure 4.4 describes one technique for synchronization called local synchronization. In this case there are only two processing nodes involved in the synchronization step: the sending node (S) and the node receiving into its auto-receive channel (Ra). Node Ra first sets up its auto-channel. Then it sends a sync packet to the user queue

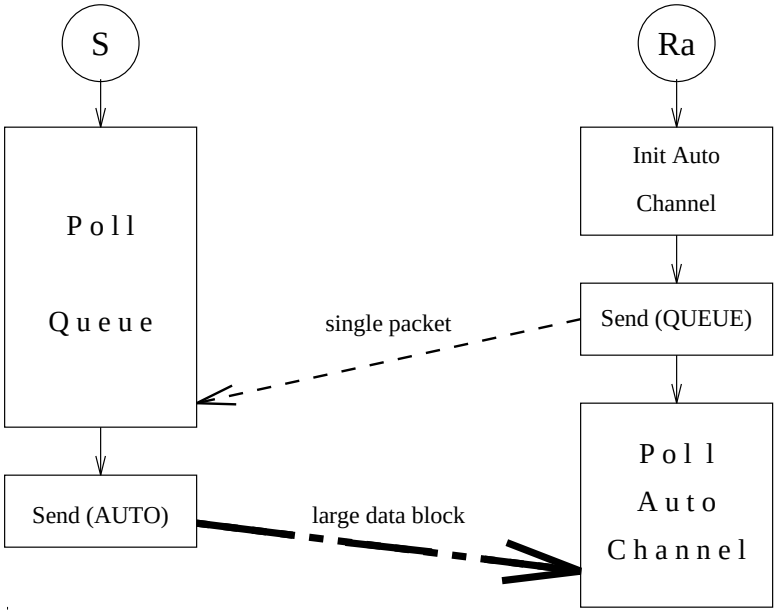


Figure 4.4: Using local synchronization to support auto-receive

of node S. When node S detects the sync packet in its queue, it starts sending data to Ra's auto-channel. Local synchronization is useful for transferring long messages (see Section 6.1.2).

Another technique used for synchronization is called barrier synchronization. The semantics of a barrier are simple: a node enters the barrier and waits until all other nodes have entered the barrier; when all nodes have entered, the barrier is complete and all nodes exit the barrier at approximately the same time. Barriers are therefore fundamentally global in scope. Also, many scalable computer systems support barrier synchronization directly in hardware [14, 49, 66]. The latency of a global barrier is often an order of magnitude less than the latency of a regular message. Figure 4.5 is a flow diagram showing the interaction between nodes S and Ra when barrier synchronization is used. The barrier must occur AFTER node Ra sets up its auto-receive channel and BEFORE node S initiates its send operation.

The greatest benefit of global synchronization occurs during a phase in the parallel program's execution where all nodes are communicating concurrently and are simultaneously acting as both senders and receivers. For instance, a program may require every node n to send a message to its neighboring nodes $n-1$ and $n+1$ and also receive messages from these nodes at the same time. Figure 4.6 illustrates how this

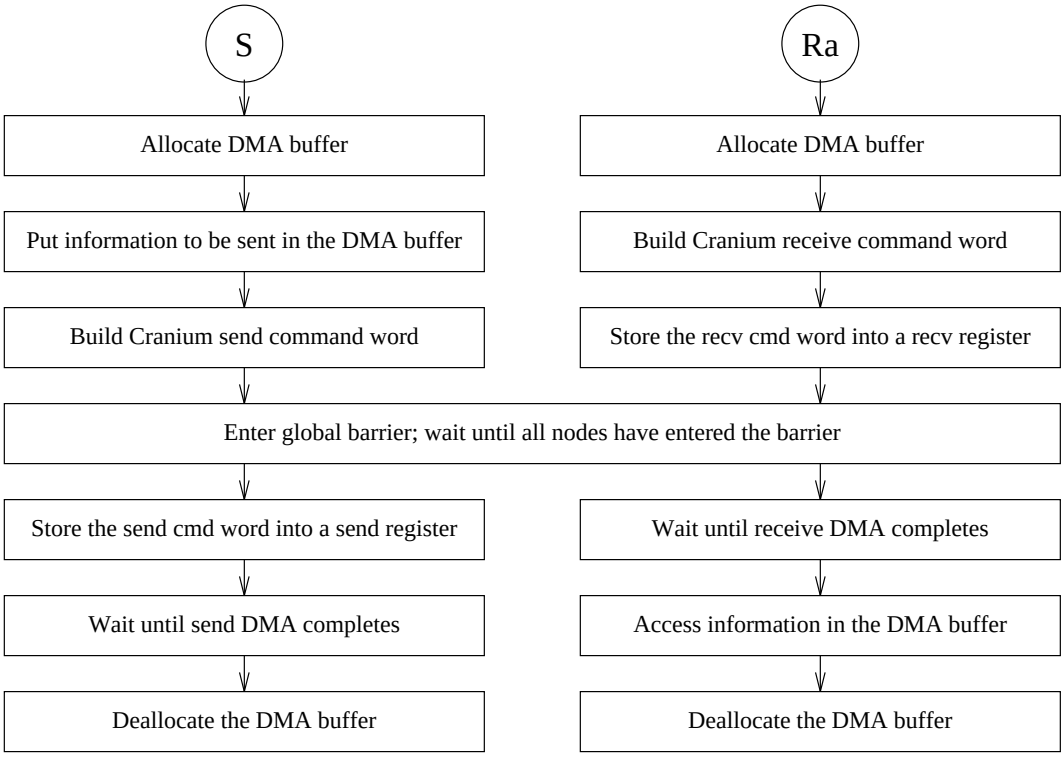


Figure 4.5: Synchronization for auto-receive using a global barrier

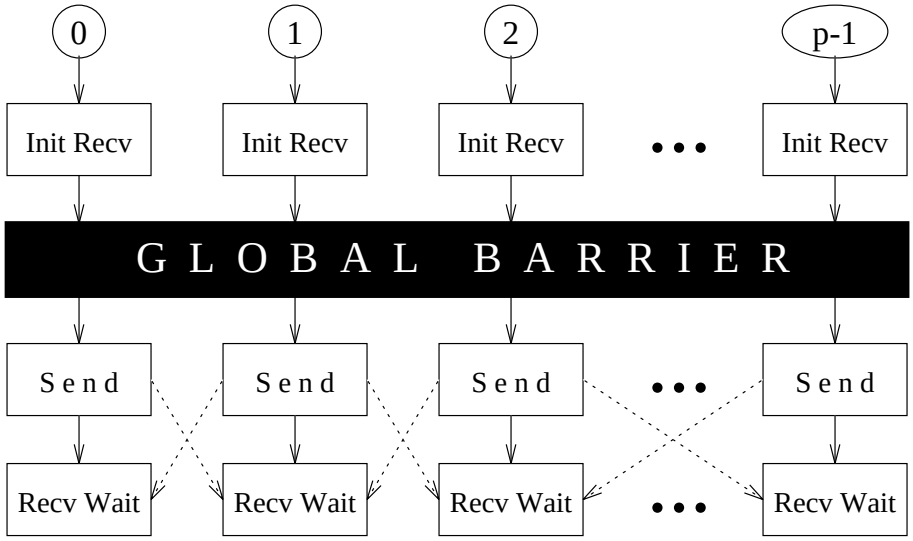


Figure 4.6: Using barrier synchronization in a communication phase where every node is both a message source and destination

communication pattern can be realized in a program using global synchronization. (The wraparound case is omitted to simplify the diagram, but in general any node can send to any other node during a communication phase.) It is common for parallel programs to exhibit this kind of communication pattern where all nodes communicate with other nodes at the same time and act as both senders and receivers. Most of the benchmark programs described in Chapter 6 have this property – see Section 6.2.3.

4.4 Interrupts and error diagnostics

When Cranium generates an interrupt, the operating system hands control off to a user-level interrupt handler. Cranium provides an interrupt status register so that the application program can determine the source of the interrupt and react accordingly. There are two classes of interrupts generated by Cranium: user-programmable interrupts and error interrupts. Cranium provides a wide variety of user-programmable interrupts. Cranium can generate an interrupt on a channel-by-channel basis for every packet or for the last packet in a channel transfer. The sending node can set specific interrupt bits in the header of a packet to cause an interrupt at the receiving node. The receiving node has the option of ignoring these interrupts, depending on how the user sets up the interrupt mask.

There are a number of error conditions that cause Cranium to interrupt the processor. The most common situation is called a protocol error, when a packet is destined for an auto-receive channel that is not ready. A second example occurs if a channel transfer is in progress (i.e. its command has not yet completed) and a new command is issued to the same channel. A third situation is when the sender's specification for the message does not agree with the receiver's. A complete breakdown of the interrupt mask and status registers is described in Section A.2.3 in Appendix A.

4.5 Comparison with other message passing interfaces

4.5.1 Intel NX

The Intel NX message passing interface is a canonical deeply-layered message passing interface for scalable parallel computing [7, 17, 35]. It is the message passing system used in every scalable parallel machine produced by Intel, from the earlier iPSC/2 and iPSC/860 machines to the later machines such as the Touchstone Delta and the

Paragon. NX has proven to be quite separable; it has been implemented on a wide variety of non-Intel systems such as Stanford DASH [45], Princeton SHRIMP [42, 43] and UW Meerkat [35]. NX is similar to PVM (Parallel Virtual Machine) and MPI (Message Passing Interface) [8, 9].

The basic subset of the NX interface is described by function calls `csend()` and `crecv()` that provide blocking communication. The parameter list for the NX functions is similar to the command structure for Cranium's send channels and auto-receive channels. The argument list for both functions includes a message tag type, a buffer address and length, the remote node and a process type. The message tag in NX is like the channel number in Cranium but more general. For instance, a message tag of -1 can match any incoming message; in contrast, Cranium requires an exact match on the channel number. Importantly, the dynamic properties of the two interfaces differ. Under NX, the call to `crecv()` at the receiving node may occur either before or after the call to `csend()` at the sender, unlike Cranium's semantics that require the receiver to be ready before the sender sends. Under NX, if the message arrives before the receiver is ready, the operating system intervenes and provides buffer space for the message; the subsequent call to `crecv()` causes the message to be copied from system buffers into the user's space. The semantics of blocking communication implemented by `csend()` and `crecv()` make it difficult to overlap communication and computation and avoid relying on the operating system for message buffering. Take the example where two nodes exchange messages: each sends a message to and receives a message from the other. The strategy under Cranium is straightforward: both nodes post their receive commands, the global barrier occurs and then the send commands are executed, followed by arbitrary computation while the messages are in transit. Under NX, if both nodes post their receive commands before executing send commands, deadlock results because each node waits indefinitely. Therefore, at least one node must send before it receives to break the deadlock. The result is that the operating system must provide buffering.

To overcome the limitations of blocking communication, NX also provides non-blocking communication via the functions `isend()` and `irecv()`. A third variation is provided by the functions `hsend()` and `hrecv()` that use interrupt handlers. These variations on the basic send/receive interface make it possible to overlap communication and computation and avoid using operating system buffers. Furthermore,

there exists a little-known variation on non-blocking communication called *force-type* messages [10]. Force-type messages provide an unbuffered message protocol to the application program. An NX message is designated as force-type if certain high-order bits in the message tag are set (resulting in a very large unsigned or negative tag value). The data of a force-type message are sent immediately through the network by the sending processor. If the receiver has posted a matching receive call before the message is sent, the receiver takes the data. Otherwise, the message is simply ignored; neither the sender nor the receiver are notified. Force-type messages allow the operating system to be bypassed completely (in principle) and represent the most streamlined mode of communication in NX.

Because NX is a higher level interface than Cranium, it is possible to implement NX on top of Cranium. An advantage of this emulation is that it allows all the programs written for NX to run directly on a Cranium-based system. A drawback of emulating NX on top of Cranium is that there would be a considerable increase in communication overhead compared with programs that use the native Cranium interface. The two primary sources of the extra overhead in NX concern buffer allocation and data alignment. Buffer allocation for message passing directly influences the resource management strategy of the operating system; in general, an expensive operating system call is needed to allocate message buffers. Under NX, message buffers are allocated every time a message is sent or received. The native Cranium interface separates the allocation of a buffer from its use; an application program can allocate a buffer once and then use it for multiple messages. Data alignment in NX is unrestricted – a message buffer can start and end anywhere; messages may be as short as a single byte in length. Data alignment in Cranium is oriented around cache lines and MMU pages – messages must be aligned to a cache line and consist of an integral number of cache lines that all occupy a single MMU page at both the sender and the receiver. These restrictions make it possible for the hardware to transmit the message at the full performance of the network and the memory system without sacrificing protection. To implement arbitrary message alignment on Cranium, the processor must copy data. In general, emulating NX on top of Cranium increases the amount of buffer allocation and data copying. Because both of these operations are expensive, NX emulation results in a significant loss of performance compared with the native Cranium interface.

The major differences between NX and Cranium can be summarized as follows:

- Buffer allocation in NX is tightly bound to the send and receive operations, unlike in Cranium where buffer allocation is separate from send and receive. Buffer allocation is an expensive operation as it requires an operating system call. Cranium makes it possible to greatly reduce the number of times buffers are allocated compared with NX.

- In NX, the operating system buffers incoming messages. This strategy requires an expensive protection boundary crossing to copy messages from system space to user space. In Cranium, messages always originate in and are delivered to user space directly, without crossing a protection boundary.

- NX permits the use of force-type messages to bypass the operating system. However, there are no diagnostics if the receiver was not initialized before the sender begins sending the message and the transfer fails. In Cranium, there is an extensive provision for error checking to inform the user of protocol errors.

- NX uses arbitrary message alignment and sizing. Cranium requires alignment along cache lines and MMU pages. Cranium's restrictions allow a simple hardware implementation that delivers the full performance of the network and the memory system.

4.5.2 Active messages

An alternative strategy to the heavily-layered approach of Intel NX is the thinly-layered technique known as *active messages* or AM [25]. Every packet in an active message contains a pointer to a handler function. When the packet arrives at the receiving node, the handler function is dispatched immediately and it runs to completion. AM is fundamentally non-blocking and offers fast recovery of message buffer space. It is ideally suited to a user-level interface that provides a buffered communication protocol, like the user queue in Cranium or the FIFO interface of the CM-5 [14]. The advantages of AM are that it is very simple and it introduces minimal overhead on top of the resources provided by the hardware. Dispatching the handler function is usually significantly faster than the tag matching approach of NX, which

requires a slow switch/case or if-then-else construct. The primary limitation of AM is that it is very low-level. Message passing protocols must be built on top of AM primitives to prevent deadlock and data loss. In the specific case of Cranium, AM can be implemented on top of the user queue very simply. AM does not apply in a straightforward fashion to the auto-receive channels because they implement an unbuffered communication protocol. The upshot is that a canonical AM implementation cannot easily take advantage of Cranium's auto-channels to achieve high performance on long messages. This characteristic is not surprising because AM was originally based on the CM-5's programmed-I/O network interface, providing buffered communication that works well with small messages but poorly with large messages. However, it is possible that a rewrite of AM for unbuffered communication may be able to address the large-message case efficiently.

4.6 Summary

The basic operations of the Cranium application programmer's interface (API) were introduced: sending, receiving, synchronization and interrupt handling. The Cranium API provides two primitives for receiving: the auto-channels for large messages and the user queue for small messages. It provides two techniques for synchronization: local synchronization and barrier synchronization. Cranium's user-programmable interrupts provide flexibility in parallel program development. Together, all of these features provide high performance for both short and long messages and a software interface that is reasonably easy to program and to implement.

The Cranium API is compared against two popular interfaces for message passing on scalable computers – the Intel NX message passing interface and active messages (AM). NX is a deeply-layered, heavyweight interface that requires a system call to send or receive a message, unlike Cranium that offers direct user access to the network. AM is antithesis of NX – it is a very lightweight, thinly-layered interface but unlike NX it pushes much of the programming burden onto the user. AM provided a significant improvement over the previous state-of-the-art for software interfacing with network hardware. However, advances in hardware technology such as Cranium that have occurred since AM was development have revealed the limitations of AM. Like AM, the Cranium API closely matches the services offered directly by the network interface hardware; however, the Cranium API provides higher performance on long messages

Table 4.1: Comparison of message passing interfaces NX, AM and Cranium

Criterion	Intel NX	Active messages	Cranium
Ease of implementation	Poor	Very good	Fair
Usability	Very good	Fair	Good
Separability	Very good	Fair	Poor
Small message performance	Poor	Very good	Very good
Large message performance	Good	Poor	Very good

than AM alone does.

Table 4.1 summarizes the comparison of the APIs discussed in this chapter. The advantages of Intel NX are its high usability and separability; its weaknesses are its complexity and its performance, especially with small messages. The advantages of AM are its simplicity and its performance with small messages; it is comparatively difficult to use because it is very low level. The Cranium API permits the network interface to achieve high performance on both small and large messages while providing good usability and it is only moderate complex to implement. Because the Cranium API is deeply embedded in the Cranium network interface hardware, its separability is poor – it would be quite difficult or inefficient to implement the Cranium API on top of non-Cranium hardware.

Now that the Cranium architecture and application programmer's interface have been introduced, the next step is to evaluate the performance. The following two chapters show the approach taken in the evaluation of Cranium. Chapter 5 describes the design and development of the simulator used for the Cranium evaluation environment. Chapter 6 provides both an analysis of Cranium's performance and empirical results using the simulator described in Chapter 5. A complete description of the Cranium API can be found in Appendix A, including all the data structures and modes of operation. Examples of sending and receiving messages are illustrated with a few lines of C code.

Chapter 5

THE TEST ENVIRONMENT

Each mind has its own method.

– *R. W. Emerson,* Essays

This chapter describes the Cranium test environment. This environment was used to test the Cranium network interface architecture both for functional correctness and for performance evaluation. Establishing functional correctness was an important initial goal because there had been no prior experience in network interface design with Cranium's properties: automatic-receive DMA, native support for an unbuffered message protocol, packet counting and notification filtering. After functional correctness was established, the environment was used to evaluate performance on a set of scientific parallel benchmarks. The evaluation of these benchmarks is discussed in Chapter 6.

The test environment models the processing node (processor and memory subsystem), the network interface and the network. Our goals for the Cranium test environment were the following:

- The environment needed to evaluate an entire parallel system. The size of the state of each modeled node and the amount of time taken per instruction modeled had to be kept small enough so that the environment could scale to a large number of nodes.

- The environment needed to provide timing information to make it possible to compare the performance of different approaches. It was important to make comparisons at many different levels of the design: the memory organization, architectural support in the network interface and implementations of algorithms used in the benchmark programs.

- The environment needed to be easy to modify and instrument. For example, it needed to provide the user with the ability to trace execution, gather timing information and other statistics, and manage execution by setting breakpoints.

- Finally, the environment had to be constructed quickly, in a time frame of a few weeks to a few months. If the environment took a year or longer to construct, then the focus of the project would have shifted to that of an implementation project rather than a research project.

The only reasonable solution for the Cranium test environment was a simulator. Building a real hardware system, especially a multiprocessor system, would have been too expensive, difficult and time-consuming. We evaluated three techniques for simulating a computing system: functional simulation, software structural simulation and hardware simulation.

- *Functional simulation* is simulating only the functionality of the hardware and offering only a coarse level of timing information, such as ensuring the proper sequentiality of global barriers. A functional simulator, also known as an emulator, works in the same basic manner as an interpreter. The functionality of the simulated hardware system is usually captured as a C program. Functional simulators are relatively easy to construct and instrument, and they run on standard platforms such as workstations and PCs. The most efficient functional simulators impose a slowdown of 10 to 100 host cycles per simulated cycle. In the literature there are a wide variety of processor simulators that provide functional simulation; an excellent survey of techniques can be found in the papers on Shade [67, 68].

- *Structural simulation* is simulating at the register-transfer level of the architecture of the underlying hardware (the network router or processor). Structures such as the cache, the memory system, the routing algorithm of the network, etc. are closely modeled on a cycle by cycle basis. The functionality of the simulated hardware system is usually captured as a Verilog or VHDL program. The advantage of structural simulators is the ability to provide cycle level accuracy. The disadvantage is that they run very slowly. Structural simulators are more difficult to construct and instrument than functional simulators are, and

require far more memory per simulated processing node to contain intermediate state. Due to the large memory requirement they usually run only on high-end workstations and servers, but not on entry-level workstations and PCs. The focus of a structural simulator is accuracy rather than host execution speed; it is common for a structural simulator to impose a slowdown of 10000 to millions of host cycles per simulated cycle.

- *Hardware simulation* is like structural simulation but it requires a special-purpose programmable hardware system. The host system is constructed from an array of programmable logic devices such as Xilinx FPGAs. Two examples of hardware simulators are the Teramac from Hewlett-Packard Labs [69] and the System Realizer M3000 from Quickturn [70]. Hardware simulators provide the same degree of fine-grain detail as structural simulators, except that execution speed is comparable to a very fast functional simulator. The primary disadvantage is cost, on the order of tens of thousands to millions of dollars.

The Cranium test environment is based on a hybrid of functional simulation and structural simulation. The processing node and memory are simulated by Talisman [27], a functional simulator augmented with a timing model. The Cranium network interface model was added to Talisman. The network is simulated by the Chaos network simulator [12, 15], a structural simulator written in C. The remainder of this chapter discusses the background of both simulators and the model for Cranium in the test environment.

5.1 Talisman

Talisman [27] is a processor simulator created by Robert Bedichek. The strength of Talisman is its fast host execution performance. Functional processor simulation requires two parts: translation from the target instruction set to the host instruction set and execution of host instructions on the host. Of the two parts, translation (decoding) is much slower than execution (dispatch). A simple but slow simulator uses the standard fetch-decode-dispatch loop: at the conclusion of a previous operation, it fetches a target instruction from the input file, translates it and executes it. To improve host execution speed, the program can be statically translated into the host's

native binary format. Static translation (essentially, recompiling from one binary format to another) avoids the decoding step entirely at run-time. This strategy was used by DEC to migrate VMS programs from the VAX to the Alpha AXP [71]. However, static translation prevents some types of programs from executing correctly, including programs that modify their instruction space on-the-fly (i.e. self-modifying or run-time compiled code). For generality, Talisman translates instructions at run-time. To optimize host execution performance, Talisman caches the decoded information. Subsequent hits to recently decoded instructions are handled at a small fraction of the cost of a full decode. This technique works well in practice because most programs exhibit good locality.

Talisman runs programs that are cross-compiled into a Motorola 88000 binary file and can simulate either a uniprocessor or a parallel processor. The native communication model in the parallel version is a subset of the Intel NX message passing run-time system. This communication model was replaced by the Cranium application programmer interface.

5.1.1 Talisman's timing model

The timing model used in Talisman is based on a black-box cost analysis of each of the major structural units in the Motorola 88100 [72]. Talisman maintains models of the memory system, the instruction cache and data cache, the translation lookaside buffer, the execution pipeline and the write buffer (a three-element FIFO). The timing model in Talisman was calibrated against the Meerkat-1 hardware prototype [35]. Through the use of a hand-tuned set of approximately 30 timing parameters, the timing of Talisman concurs with the timing measured in Meerkat-1 within a few percent. The results were verified over a wide range of benchmark programs. These timing parameters represent subtleties in the instruction pipeline and the memory system, such as the overhead of the DRAM refresh cycle.

A number of modifications were made to Talisman when it was integrated with Cranium and the network model. Changes made to the process scheduler and the memory system model are described as follows.

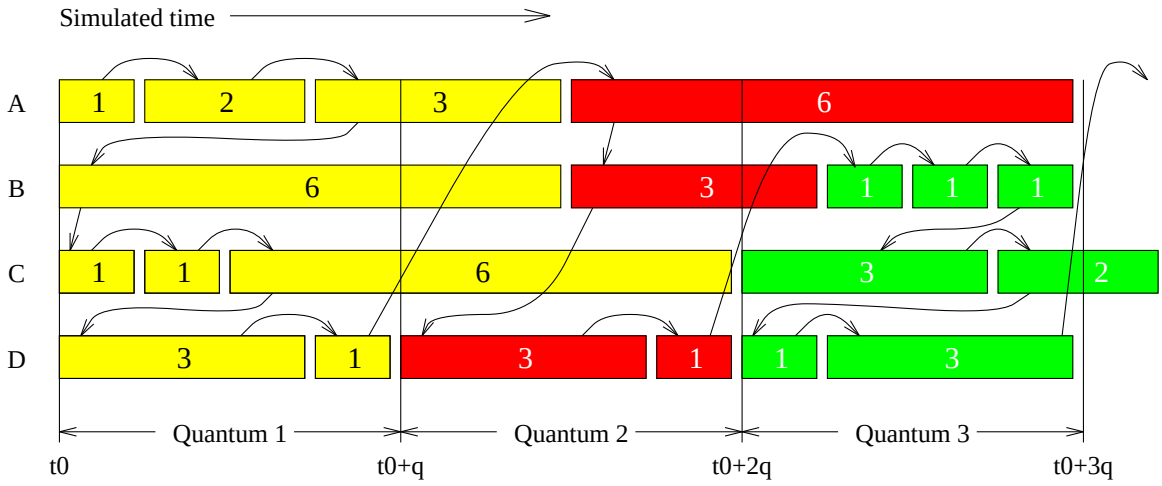


Figure 5.1: Scheduling threads in Talisman

Process scheduling in Talisman

Each simulated processor in Talisman executes as a separate lightweight thread within a single Unix process on the host. One or more instructions are executed in sequence on a single simulated processor. When the local time counter of a simulated processor is greater than or equal to the value of the global virtual time counter, then Talisman switches context to the next simulated processor using a round-robin algorithm. When all simulated processors are at or beyond global virtual time, the GVT counter is incremented by the value of the scheduling quantum. Figure 5.1 illustrates the basic idea. Four simulated nodes are represented by the four rows labeled A, B, C and D. Simulation begins at time t0. The scheduling quantum is 4 clock cycles. The arcs with arrows indicate the execution sequence performed by Talisman. Talisman starts by executing instructions on node A. The first instruction takes one clock cycle. Since this isn't beyond the quantum (t0+q), Talisman stays at node A and executes the next instruction which takes two clock cycles. Again it stays at A and executes the next instruction taking three cycles. Since this is at or over the quantum, Talisman switches to node B. This instruction takes 6 cycles. Note that the number of clocks per instruction can be greater than the size of the quantum. The light stipple pattern indicates all the instructions that are executed in the context of the first quantum period. The dark stipple pattern corresponds to the second quantum period and the medium pattern corresponds to the third period. Note that for node C there is no

instruction executed in the second period; node C is not even scheduled during this interval.

The value of the scheduling quantum has a significant impact on host execution performance. Increasing the quantum improves the host execution performance by reducing the amount of context switching. If nodes are not involved in communication then the size of the quantum has no impact on simulated timing. When nodes communicate, the scheduling algorithm can introduce jitter. Nodes early in the schedule that receive messages may not get informed until one or more quantum periods later than nodes that are late in the schedule. The larger the quantum is, the greater the error. Setting the quantum to one minimizes this jitter as it forces the scheduler to switch contexts after every instruction. The drawback is that it causes Talisman to run very slowly. There are two techniques that allow faster host execution. The simplest technique is to increase the quantum to a moderately small value (e.g. 5 clock cycles) to provide reasonable performance with only a small amount of error. The effect of the jitter on software timing is relatively small because the time it takes the processor to poll or interrupt on the arrival of a packet is much greater than the uncertainty the jitter introduces. The second technique is to adjust the quantum dynamically. When there is no activity in the network, the quantum is set to a large number (e.g. 10 or more). Whenever the network is routing one or more packets, the quantum is set to 1. This technique speeds up execution very well because most programs spend a small fraction of the total running time communicating; tens of thousands of cycles are spent in initialization routines, such as the one for the `malloc()` memory allocator. The differences in simulated time between the two techniques are very small, on average less than half a percent of the total simulated running time. The results reported in Chapter 6 and Appendices B and C reflect the use of the second technique where the quantum is changed dynamically.

The memory system

Two modifications were made to Talisman's memory system model when it was integrated with the Cranium network interface model. Talisman uses a simple reservation scheme to model the memory bus. When Cranium wishes to access the bus to perform DMA, it waits until the bus is free, and then reserves the bus for the appropriate number of clock cycles. Two memory bus models are used. The first model comes

directly from the memory model in Talisman, based on the Meerkat hardware prototype. Meerkat uses a 32 bit data bus running at 20 MHz. Burst transfers move a four-byte word of memory every clock cycle, so the peak rate is 80 MB/s. The overhead of starting a memory transfer is high; accessing the first word of DRAM costs up to 10 clock cycles. Since the bus is non-pipelined, achieving a good fraction of the peak rate requires long transfers. The problem is that the Chaos network router is based on small packets whose payload is the size of a cache line. Every time a cache line moves in or out of memory it costs 10 cycles for DRAM latency plus 8 cycles for data transfer. The effective transfer rate is therefore slightly less than half the peak rate.

Modern high-performance memory bus technology uses a pipelined or split-transaction protocol to hide DRAM latency and achieve nearly the full bandwidth of the bus under small transfers. The first modification to the memory system captures the behavior of high-performance split-transaction bus technology. This alternative models only the clock cycles in which data is transferred on the bus and omits the DRAM latency. The difference in running times of the same benchmark on the two memory models indicates the sensitivity of the benchmark to memory performance. See Section 6.2.4 for further discussion of the two memory models supported in the Talisman simulator.

5.2 Chaos network simulator

Chaotic routing is a non-minimal adaptive algorithm for routing fixed-length packets. It was invented by Magda Konstantinidou and Lawrence Snyder as an algorithm for routing in hypercube networks [73]; the algorithm was adapted by Kevin Bolding and Melanie Fulgham to apply it to mesh and torus networks [12, 74]. Konstantinidou, Bolding and Fulgham and many others[1] created a structural simulator called the Chaos simulator to evaluate the performance of the routing algorithm. In its current form, it simulates complete networks of chaotic routers organized as two-dimensional meshes or tori. The Chaos simulator achieves cycle-level accuracy at the cost of relatively slow host execution performance; it is perhaps an order of magnitude slower than Talisman (see Table 5.2 in Section 5.6). Nevertheless, it has been widely used

[1] Donald Chinn, Sung-Eun Choi, Melanie Fulgham, Neil McKenzie, Thu Nguyen and Bill Yost

to measure many different kinds of network configurations, routing algorithms and workloads.

The rest of this section is a brief overview of chaotic routing in general and the design of a prototype CMOS chip that implements the algorithm. Much more detail on chaotic routing can be found in the literature [12, 15, 13, 66, 73, 74, 75].

The chaotic routing algorithm uses fixed-length packets and a packet routing technique called virtual cut-through [76]. A packet consists of a sequence of words called phits (physical units); each phit is the width of a router link. A packet in flight in the network has two modes, similar to a slinky. When the network is lightly utilized, the packet's phits spread out over multiple links and routers, as in an extended slinky. In the presence of congestion, the packet condenses into a single routing node, like a compressed slinky. The effect is like wormhole routing when there is no resource conflict, and store-and-forward under heavier network utilization. (See Section 1.4 for an introduction to wormhole and store-and-forward routing.) Since a packet can reside fully within a single node, it requires the node to contain one or more packet buffers. Unlike in a wormhole router, packets in a cut-through router have the tail-following property: once the first phit of the packet is transmitted, subsequent phits are assumed to follow on subsequent clock cycles[2].

Adaptive routers such as Chaos make their routing decisions dynamically. The router makes its routing decision for an arriving packet using both the information in the packet header and the current state of the routing node. A packet makes progress if it is routed to a node that takes the packet closer to its destination. If possible, a packet makes progress with every hop. If it can't make progress, the Chaos router stores the packet into a node buffer. If buffer space is exhausted, packets in the buffer pool may become de-routed (sent in an un-profitable direction) to make room for subsequent arriving packets. In a lightly loaded network, packets travel in minimal paths. In a heavily loaded network, de-routing can take a packet along a non-minimal path. Since packets take different paths even in a lightly loaded network, the presence of instantaneous congestion can cause packets to overtake one another and arrive out-of-order.

Bolding's dissertation [15] describes the simulation results of chaotic routing com-

[2] It is also possible to construct asynchronous versions of chaotic routing, but for simplicity synchronous circuitry is assumed here.

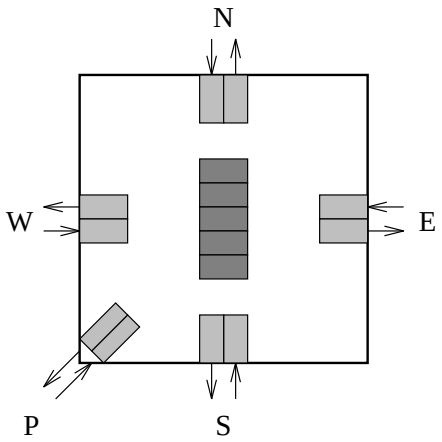


Figure 5.2: Chaotic routing chip: external interfaces and internal buffering

pared with two other routing algorithms: the standard dimension-order oblivious algorithm and deflection routing, another adaptive algorithm, under both uniform random traffic and hot-spot traffic. Bolding discovered that on mesh topologies, chaotic routing performs no better than the other algorithms, but on hypercube and torus topologies chaotic routers perform significantly better than both oblivious and deflection routers. The strength of the results provided the motivation to implement a prototype chaotic router. The router was implemented as a 132-pin CMOS chip, created by a team of graduate students supervised by Bolding and Ebeling [75].

Figure 5.2 is a diagram showing the external interfaces to the router. There are five bi-directional channels: the NEWS channels (north, east, west and south) that connect to other routing nodes and the P channel that connects to the processing node. Also shown are the packet buffers, called frames. Each frame is a FIFO for a full packet. Packets are 20 phits in length. Rectangles shown in light gray represent input and output frames. The dark gray rectangles represent frames in a separate buffer pool known as the multi-queue. The input frames, output frames and the multi-queue together yield a total of 15 frames. Figure 5.3 is data-flow diagram of the chip, organized so that all packets flow from left to right. This diagram shows the internal switching network comprising three crossbars. The multi-queue provides storage for packets that can't make immediate progress because of contention for network links. The routing algorithm continually tries to set up a connection on the crossbar from an unconnected input or multi-queue frame to an output frame in a profitable direction. If no profitable direction exists and the multi-queue is full, the

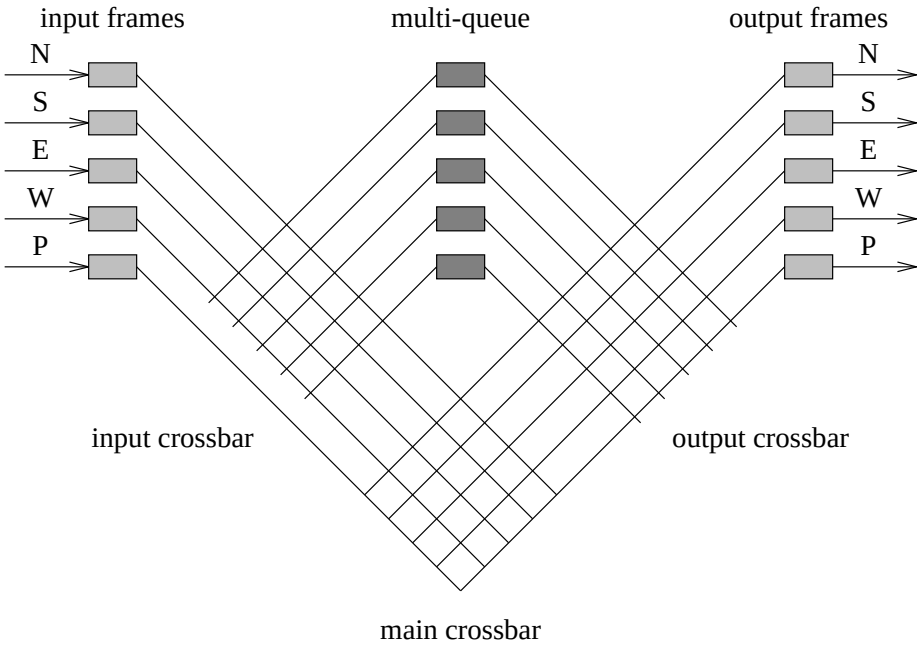


Figure 5.3: Data paths in the chaotic routing chip

router may de-route one of the packets in the multi-queue by sending the packet to an output frame of an un-profitable channel. The input crossbar is slightly asymmetric because there are no links from the injection frame into the multi-queue.

5.3 Modeling the behavior of Cranium in the simulator

In this section we describe how some of the implementation-dependent features of Cranium are modeled in the simulator: packet scheduling, number of user contexts, gather-scatter support and cache coherence (see Section 3.4). In some cases, simple approximations were used if it was too difficult to model the intended feature precisely. The packet scheduling algorithm is first-come, first-served (FCFS). Only one user context is supported; multiprogramming and its associated overhead is not modeled. Gather-scatter capability is not modeled.

The simulator supports both the write-invalidate and write-update models for cache coherence. Ideally, we would like Talisman to model the multi-level scheme for coherence discussed in Section 3.4.2. However, Talisman models only a single-level cache, because it was based on the Meerkat-1 hardware prototype. The single-level cache in Meerkat-1 is implemented by the Motorola 88200 cache-MMU chip [77]. It

was judged to be too difficult to modify Talisman to model a two-level cache (both on-chip and external). As an approximation, multi-level is modeled using write-invalidate with the auto-channels and write-update with the user queue.

5.3.1 Injection and delivery of packets

When Cranium attempts to inject a packet into a router while the router is busy, the packet goes into an injection queue. Similarly, when the router delivers a packet and the memory bus is busy, the packet goes into a delivery queue. In hardware these queues correspond to FIFOs in the router or in the incoming and outgoing links of the network interface. The default model for Chaos contains an infinite injection queue. This infinite queue model was modified to model a bounded queue by using backpressure messages from the Chaos simulator back to Talisman. The results in Chapter 6 show that for the selected benchmarks the size of the injection queue directly affects the maximum delivery queue length. Therefore, a bounded queue is implemented in the simulator only for injection; the simulator models an infinite delivery queue, but in practice the delivery queue is bounded.

5.4 Implementation: integrating the simulators

The implementation of the combined Talisman and Chaos simulator runs Talisman and Chaos in separate Unix processes, as illustrated by Figure 5.4. The two simulators were compiled and linked together to create a single host executable binary file. Upon start-up, Talisman is invoked first and then performs a fork-exec to run the Chaos simulator. After initialization, the two simulators communicate using a pair of Unix pipes called the request pipe and the reply pipe. Talisman injects simulated packet traffic by sending messages to Chaos over the request pipe. Similarly, Chaos delivers simulated packet traffic by sending messages over the reply pipe. Pipe communication transmits packet timing information and synchronizes the two simulators. Chaos can optionally open an X window and display an animation of the routing algorithm. Figure 5.5 shows a snapshot of an X window displaying the state of a simulated 12x12 network of chaotic routing nodes configured as a torus. The colors indicate the utilization of the packet frames inside each router: black indicates that all frames are idle, dark gray indicates light utilization, and light gray indicates

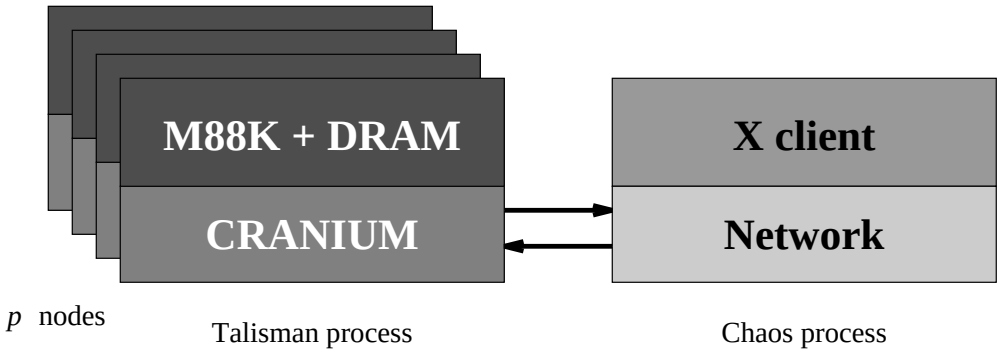


Figure 5.4: Integration of Talisman and Chaos simulators. Talisman and Chaos execute as separate Unix processes. The two communicate using a pair of Unix pipes, one for requests to the network (e.g. packet injection) and one for replies to the processor (e.g. packet delivery). The Talisman simulator is itself multithreaded; each simulated node (processor, memory and network interface) runs as a separate thread. Chaos opens an X window and updates the display after every simulated clock cycle.

heaviest utilization. Link utilization is also indicated by the same means. Only the routing nodes are shown; the processing nodes are omitted for simplicity. Animation is a very effective tool for demonstrating the routing algorithm and for debugging message-passing programs.

The structure of the combined simulator has a small effect on host execution performance. Using separate processes causes the combined simulator to run slightly more slowly than if both were run together in the same address space, due to the overhead of operating system calls to perform interprocess communication. However, the requirement for rapidly developing the environment was more important than optimal host execution performance. In particular, minimizing the effort required to bring up and debug the combined simulator was an important goal. Both simulators by themselves are large stand-alone applications. One potential source of bugs is the potential for collisions in the global variable name-space. A memory leak in either simulator might cause no problem by itself but could manifest a serious bug when the two were combined. Therefore, it was important to integrate the simulators in a way that isolated bugs due to integration to the communication link. As Section 5.6 explains, the performance of the combined simulator turned out not to be an issue.

The amount of information per message passed between Talisman and Chaos is

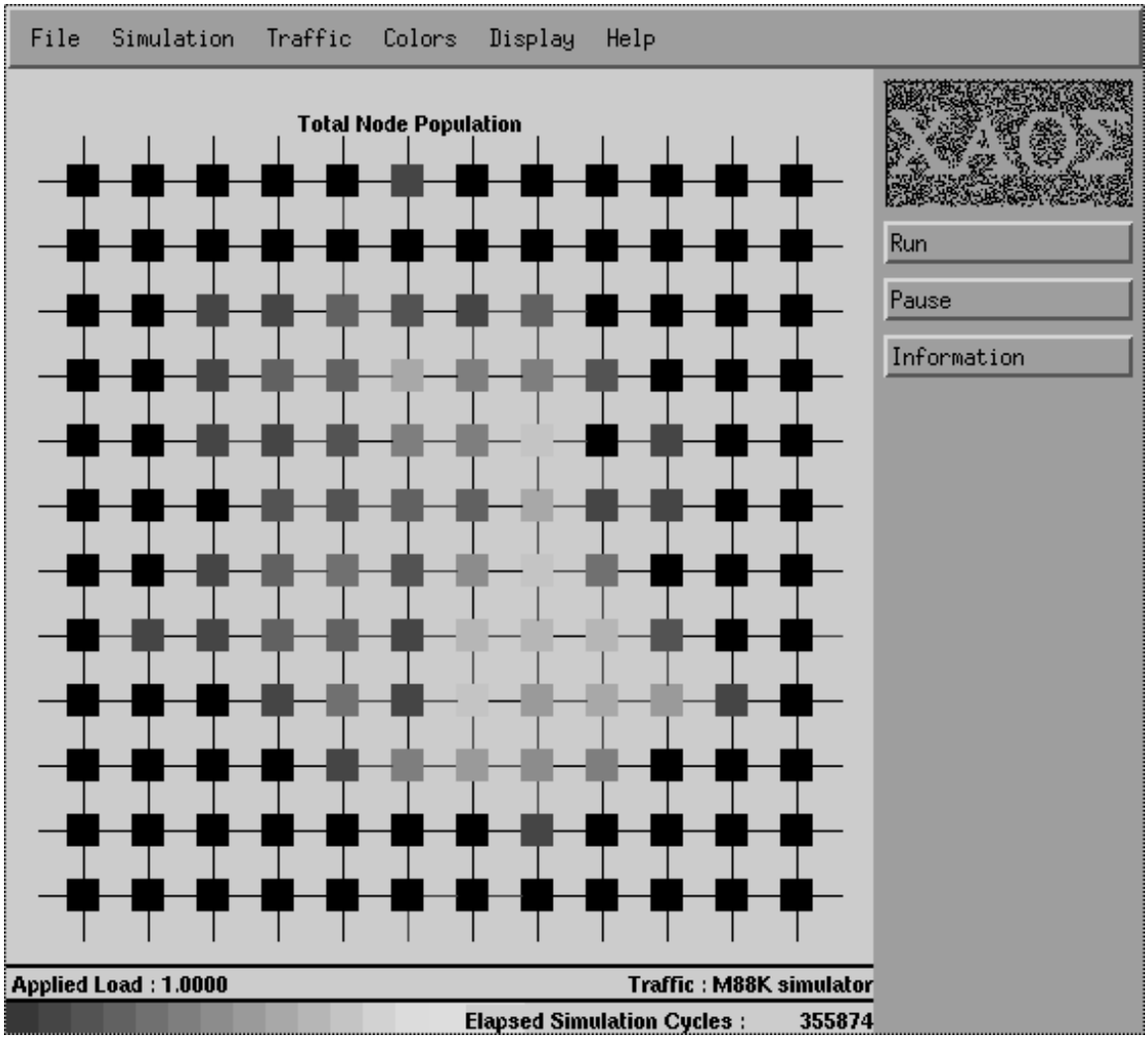


Figure 5.5: Chaos network animation. Each square represents a Chaos routing node. The node color indicates activity level; dark squares indicate nodes with no packet activity, and nodes that are lighter in color reflect correspondingly greater numbers of packets occupying internal FIFOs in the routing node. This snapshot was taken in a network with 144 nodes where the central 64 nodes in the inner 8 by 8 grid are connected to processing nodes executing a parallel sorting algorithm.

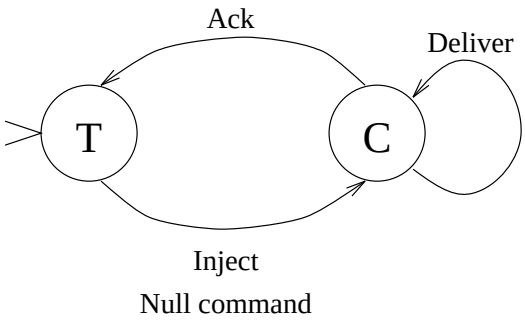


Figure 5.6: State machine for communication between Talisman and Chaos

minimal. An injection message contains only the start and end node identifiers, the cycle count at injection and a packet identifier; a delivery message contains only the cycle count at delivery and the packet ID. Talisman maintains a list of packets that are in flight in Chaos; entries in this list contain the matching ID and the contents (header and payload). Chaos does not model the contents of the packet other than its length. In essence, all Chaos provides for Talisman is timing information. Minimizing the amount of information per message makes it easy to adapt other networks or routing algorithms to Talisman.

Figure 5.6 shows the state machine describing the handshake between Talisman and Chaos. Talisman runs its simulation in state T, and Chaos runs its simulation in state C. Talisman advances its virtual clock ahead of Chaos's virtual clock. In state T, Chaos waits for input on the request pipe, either an Inject message or a Null Command message. When Talisman sends a message to Chaos the state machine makes a transition to state C. Chaos then runs until its cycle count catches up to the cycle count of Talisman. Talisman waits for any number of Deliver messages followed by an Ack message on the reply pipe. As Talisman runs it sends Null Commands to update the global cycle count and pull the Chaos simulation along. Since Talisman's virtual clock runs ahead of Chaos, the effect is to delay the arrival of packets at the simulated processors. We compensate for this delay by incorporating it in with the FIFO latency at the receiving node.

In a preliminary version of the state machine, the arc for Inject was the mirror image of the Deliver arc. However, space in the pipe is limited. In somes cases with large numbers of nodes injecting simultaneously, all space in the request pipe would fill and cause deadlock. The current solution eliminates deadlock at the cost

of requiring an Ack message for every message from Talisman.

5.5 Running the combined simulator

Talisman uses the GNU debugger `gdb` as its user interface. The executable version of Talisman is called `g88`; the name represents the combination of the front end based on `gdb` and the Motorola 88000 architecture. Parallel programs that run on the simulator are written in C and cross-compiled to Motorola 88000 binary format; this binary file is the input to the simulator. All processing nodes run the same static executable program, but may have different dynamic behavior as the program can compare and branch on the value of the node's identifier. Nodes communicate by using the Cranium application programmer interface to send and receive messages. Cranium appears in the user's address space as a set of memory-mapped registers. Executable programs for the simulator contain the suffix `.88k` to distinguish them from host executable files. To run the simulator on the program *prog*.`88k`, the user enters a string of the following form to the Unix shell:

`%` `g88_`*config* *prog*`.88k` *arglistT* `--` *arglistC*

The configuration of Talisman is determined entirely at run-time. The user selects the number of nodes as a starting configuration, which must be a square number between 1 and 256, inclusive. Individual nodes can be disabled after start-up, in order to run with a non-square number of nodes. By contrast, the physical configuration of Chaos including topology and number of nodes is determined at compile-time. Therefore it is necessary to recompile to run with a different network. The *config* suffix denotes the network configuration; e.g. the 64-node version is called `g88_64`. Talisman checks the number of processing nodes selected; it must be equal to or less than the number of routing nodes. If there are fewer processing nodes than routing nodes, the processing nodes are connected to routing nodes in the central square of the network. This is illustrated in Figure 5.5 where there are 144 routing nodes and 64 processing nodes.

The user specifies options to both Talisman and Chaos on the command line. The Talisman argument list *arglistT* appears first and is separated from the Chaos argument list *arglistC* by the double dash. Talisman uses all the standard command

Table 5.1: Host configuration data

Parameter	Description
Platform	Sun Microsystems SPARCstation 10/61
Processor	Ross Technologies hyperSPARC 90 MHz (4)
Operating system	SunOS 4.1.3_U1
Window system	X version 11 release 5 with Motif 1.2.2
Native compiler	GNU C compiler 2.4.0
Cross compiler	GNU C compiler 2.2.2

line flags of `gdb`. Several important options for Chaos are `-A` to suppress animation, `-I` to model an infinitely fast network and `-V` to save a trace of all packet traffic. The trace indicates all routing decisions made for every packet, including de-routing decisions. Tracing is useful for replaying a particular animation sequence off-line.

5.6 Host execution performance of the combined simulator

Table 5.1 displays the programming environment for the simulator. The host system, a SPARCstation 10/61, is itself a multiprocessor containing four 90-MHz hyperSPARC processors, each rated at 100 SPECint92. The extra processors contribute little to improving host execution latency but make it possible to run several independent simulations concurrently. The simulated target architecture is based on a 20-MHz Motorola 88100 processor [72] and a pair of 88200 cache/memory units [77]. One 88200 is used as an instruction cache and another as a data cache; they complement the 88100's separate instruction and data busses. (This separation is also known as a Harvard architecture.) The 88200 contains 16K bytes of data organized as 16 byte lines with four-way set associativity. Pages are 4K bytes. Each 88200 contains a hardware translation lookaside buffer (TLB) with 56 entries for the most recently used page translations. A physical analog of the simulated system called the Hypermodule consisted of a plug-in card that contains the 88100 and two 88200s. The performance of the Hypermodule as measured using the Meerkat-1 hardware prototype is about 15 SPECint92.

Table 5.2 shows typical values for host execution performance of the combined simulator, evaluated using a Fast Fourier Transform benchmark running with 16

Table 5.2: Slowdown of combined simulator

Option	Chaos flags	Wall-clock time (sec)	Slowdown
Animation, visible	*none*	606	285
Animation, iconified	*none*	422	198
Generate trace file	`-A -V` *tracefile*	208	98
Base rate	`-A`	176	83
Infinitely fast network	`-I`	72	34

simulated processors and a 4x4 network. The simulated execution time is 0.133 seconds. Figures in the slowdown column are calculated by dividing the wall-clock time of the simulation run by 2.128, the simulated execution time times the number of simulated processors.

The simulator was run under five different options, selected through command-line arguments to Chaos. The first option is to run the simulator with animation enabled and visible on the desktop. The second option is similar except that the animation window is iconified (not visible on the desktop). Under the third option, animation is disabled and Chaos saves a trace of packet traffic to a disk file. The fourth option is the base rate with neither animation nor traffic tracing selected. The fifth option selects the infinitely fast network model, in which the network delivers packets immediately.

A number of conclusions can be drawn from the timings. Slowdowns are significantly worse when the animation is running, even when the X server has relatively little work to do. By contrast, tracing increases the slowdown only slightly over the base rate. The infinitely fast network model shows that Chaos takes about two to three times as many host CPU cycles as Talisman over the run of the entire program. When both Chaos and Talisman are active, Chaos takes four to ten times as much host CPU time as Talisman. However, the simulated network is only active a fraction of the entire running time. While Chaos imparts a significant slowdown, it is not prohibitive. The aggregate simulated performance of the system is approximately the same as a machine with a SPECint92 rating of 1.2 (assuming the base version). Large configurations of the combined simulator run proportionally slower than the 16-node system described in Table 5.2. Given the same base slowdown rate, one simulated

second on 256 processing nodes requires about 25,000 host seconds or about 7 hours, which is within the scope of an overnight batch run.

Overall, the host execution performance of the combined simulator has been acceptable. Should the need arise in the future, host execution performance may be improved significantly by focusing on the following aspects:

- *Packet ID matching.* When a packet is ejected from Chaos, Talisman looks up the ID in the table of packets in flight. Currently a linear search is used. A data structure such as balanced tree or hash table would improve performance over the linear list when the table becomes large.

- *Event driven simulation.* Chaos uses a node-based iterator to walk through every node in the network on every simulated cycle. Converting the iteration loop from node-based to event-driven (i.e. packet-based) would significantly reduce the number of host cycles per simulated cycle when there is little traffic in the network. It is unknown how much faster Chaos would run after this conversion. However, it should be significant; traffic in networks tends to be cyclical, alternating between periods of high and low activity. Currently, the only optimization in Chaos is to test if the network is empty and return immediately if so.

- *Print functions.* It is useful for a parallel program running on the simulator to call `printf()` to display the intermediate results at each processing node. All nodes print directly to the Unix terminal session where `g88` was invoked. Barriers are used to sequentialize printing and prevent interleaving the output from the simulated nodes at the ASCII character level. Pseudo-code for a typical print function in a parallel benchmark looks like the following:

```
/*
 *  N is the total number of nodes.
 *  All N nodes execute the same code together.
 *  Node IDs range from 0 to N-1.
 *  'my_node' identifies the node ID running this
 *  invocation of the code.
 */
```

```
for i = 0 to (N-1) {
  if (i == my_node) {
      print node i's information
  }
  global_barrier(N);
}
```

Node 0 prints first, followed by node 1, up to node N-1. From a host execution standpoint, the problem is that all nodes execute simulated cycles but only one node performs useful work; all nodes except one simply spin and wait for barriers to complete. The slowdown becomes quite pronounced when a large number of nodes are in use. It is usually not important to time the execution of print functions. One solution is to improve the intelligence of the Talisman processor scheduler specifically to support printing. For instance, say that when a designated print function is entered, only node 0 is allowed to execute. Talisman keeps executing instructions on a single node until it is specifically directed to deschedule that node and then schedule the successor node. Thus, only the node that is actively printing is allowed to execute. The result is that instructions are executed on the node performing useful work and no simulated node spends time uselessly spin-waiting for barriers.

The infinitely fast network was developed to provide faster host execution performance, to help bring up and debug a benchmark quickly before it is run with the Chaos simulator. However, it has many side benefits. It provides a lower bound for the benchmark's simulated running time; the difference in execution time between the infinite network and Chaos provides a rough estimate of the cost of communication in the benchmark. It provides an alternative set of network timings, to help debug programs that might be incorrectly based on delays or timing loops instead of robust message passing protocols. It also helps isolate simulator bugs to either Chaos or Talisman.

5.7 Summary

A test environment was constructed for evaluating Cranium. A software environment was used because it was easier to construct, more flexible and less expensive

than a hardware prototype or a hardware simulator. The test environment was constructed from two readily-available software simulation tools: the Talisman processor simulator and the Chaos network simulator. The test environment allows testing of simulated parallel systems up to 256 nodes. Host execution performance is reasonable, with a slowdown on the order of 100 host instructions per simulated instruction per processor. The logical separation between the processor and the network made the environment easy to create and debug, and it permits the use of different models for the network, the memory system and other attributes. The combined simulator was used to generate all the test results in Chapter 6 and Appendices B and C.

The combined simulator turned out to be an excellent vehicle in general for parallel program development. Since the environment is self-contained and runs on a standard workstation, it greatly reduces the cost of parallel program development compared with developing directly on a massively parallel computer. The debugging and animation capabilities provide much more of a hands-on feel than a native system provides. The detailed timing information is very useful for helping the programmer locate the bottlenecks in the parallel algorithm. Beyond the scope of this dissertation, the combined simulator has the potential to find use in education, such as in an introductory course in parallel programming.

Chapter 6

EVALUATION OF Cranium

Comparisons are odious.

– *Christopher Marlowe*

All unhappiness is caused by comparison.

– *D. A. Burns & Sons, Inc. (Seattle carpet cleaner store)*

In this chapter we evaluate Cranium both analytically and empirically. First, we provide an analysis of the underlying latency and throughput that can be achieved by an implementation of Cranium. We follow with two types of empirical studies: one that measures the communication performance of a set of parallel benchmark application programs and the speedups of these benchmarks, and one that compares Cranium with other network interfaces. The analysis of latency and throughput provides bounds on performance that apply to all programs and all message traffic patterns. If messages are long and the communication patterns are simple, then the resulting performance of the communication system approaches that of the upper bound on throughput. However, in many cases the traffic patterns are complex and/or the messages are small. These situations increase the software overhead and reduce the measured performance of the communication system. The empirical studies complement the analysis by characterizing the performance of the communication system achieved by real parallel programs.

A goal of the evaluation of Cranium is to compare and contrast its performance with that of other network interfaces. However, such a comparison is difficult to perform directly. Network interfaces designed for different scalable parallel systems are seldom interchangeable. For instance, the CM-5 network interface is tailored to the CM-5 network consisting of two data networks and a control network. The SHRIMP-I and SHRIMP-II interfaces require the network to deliver packets in order. None of these interfaces can be plugged directly the chaotic routing network used to evaluate Cranium. Furthermore, there are too many other variables that affect performance: processor architecture, memory bus and memory module design, and the implemen-

tation technology. Our approach to comparing Cranium with other network interface styles is to use the same processor, memory and network organization, and change only the network interface. We extract the underlying abstractions upon which the other network interfaces are based, and modify Cranium in a way that provides the best approximates these abstractions. Then the performance of the systems using the modified versions of Cranium are evaluated and compared.

The rest of this chapter is organized as follows. Section 6.1 presents a performance analysis of Cranium to determine lower bounds on latency and upper bounds on throughput. Both point-to-point and broadcast messages are analyzed. Section 6.2 discusses the experiments used to evaluate Cranium empirically. The benchmark suite consists of five parallel programs that were selected by three criteria: relevance, simplicity, and requiring significant communication. The benchmark programs execute on the Talisman simulator described in Chapter 5. The timing information provided by Talisman is used to calculate the throughput of the communication system. The message traffic patterns used in the benchmarks are broader in scope than the simple point-to-point and broadcast patterns assumed in the analysis. Cranium allows the overlap of communication and computation and thereby increases the effective communication performance. Section 6.3 abstracts the logical interface of three other network interfaces (CM-5, SHRIMP-I and SHRIMP-II) into modifications to Cranium and evaluates them. Section 6.4 describes related work that concerns the comparison of different network interface styles.

6.1 Performance analysis

Performance analysis results are given in terms of the following parameters based on the timing models used in the combined Talisman/Chaos simulator (see Chapter 5) and the implementation of Cranium (see Chapter 7). The relevant parameters are the network latency, the width of the processor-network link, the packet format, the software overhead of slave accesses to the network interface and the performance of the memory subsystem. We use cycle counts rather than absolute time (e.g. microseconds). As the underlying implementation technology continues to improve over time, the relative performance of processors and routers is expected to remain constant in terms of clock cycles [45]. Even though DRAM access latency is not improving at

the same rate as processors and networks are, the throughput of memory subsystems are keeping pace by means of pipelining and widening the memory bus.

- *Network latency.* The network is a square two-dimensional torus mesh. The routing decision at each node requires four cycles. The minimum latency for a packet taking j network hops is $4 \cdot (j + 1)$ cycles plus the length of the packet. In particular a single-hop path involving two nearest-neighbor routers requires 8 cycles plus the packet length. These assumptions are based on the routing algorithm used in the Chaos router [15].

- *Processor-network link.* The processor-network link is four bytes wide; the maximum throughput into or out from the processing node is four bytes per cycle. Data movement on the link is bi-directional and half-duplex. The direction can be switched at the beginning of a new packet. (See Section 7.1.3 and Section D.1 in Appendix D.)

- *Packet format.* A packet consists of 44 bytes. In both the processor-network link and in the links between routers, the packet is structured as eleven consecutive 32-bit physical digits (phits[1]). The first three phits are packet header information; the last 8 phits are packet payload (application program data).

- *Network interface.* It takes 10 cycles to transfer command or status information takes between the processor and network interface. The network interface registers are mapped into the user program's address space. The user program stores values to these registers to initiate send and receive commands; the program loads these registers to retrieve status information from the network interface. These timing assumptions are based on the Motorola 88100 processor [72].

- *Memory performance.* It takes 10 cycles to load or store 32 bytes (the size of the packet payload, also the size of a cache line). The bus is synchronous; it is

[1] There is a distinction between a physical digit (phit) and a flow-control unit (flit). A phit is the amount of information that is transferred in parallel in a single clock cycle. A flit is the amount of contiguous information that can be halted by backpressure due to congestion in the network. In a wormhole router a phit and a flit are often the same size; in the J-machine a flit is 36 bits consisting of two 18-bit phits [22]. In a store-and-forward or virtual-cut-through router such as a chaotic router, the entire packet can be considered a single flit.

Table 6.1: Latency of a single packet message

Network interface	Latency cost				Total	
	Send SW	Send HW	Recv HW	Recv SW	One hop	8 hops
Ideal, no mem	10	0	0	10	28	56
Ideal, w/mem	10	5	11	10	44	72
Cranium auto, warm	10	7	17	13	55	83
Cranium queue, warm	10	7	22	13	60	88
Cranium auto, cold	10	22	17	13	70	98
Cranium queue, cold	10	22	22	13	75	103

8 bytes (64 bits) wide and it multiplexes address and data. One address cycle is followed by four data cycles. There is a five cycle delay between the address cycle and its corresponding data cycles. The bus is pipelined and allows the network interface to read or write memory at the rate of two packet payloads every ten cycles. These timing assumptions are based on the system bus used in the Alpha Demonstration Unit (ADU) [64].

6.1.1 Latency of a single packet

Table 6.1 breaks down the latency of a single packet into four components: the software overhead at the sending node (Send SW), the network interface hardware latency at the sending node (Send HW), the network interface hardware latency at the receiving node (Recv HW) and the software overhead at the receiving node (Recv SW). The figures for the sender indicate the latency of inserting the head of the packet into the network; the figures for the receiver include the latency due to the length of the packet (11 cycles). The two columns at the right side of Table 6.1 display the total cycle counts for two different network distances. Values under the column labeled "One hop" represents the total cycle count based on a nearest-neighbor network distance of one hop which requires eight cycles to cross. Values under the column labeled "8 hops" represents the total cycle count based on the network distance of 8 hops which requires 36 cycles to cross. The latter figure is derived from the average distance on a 16×16 torus.

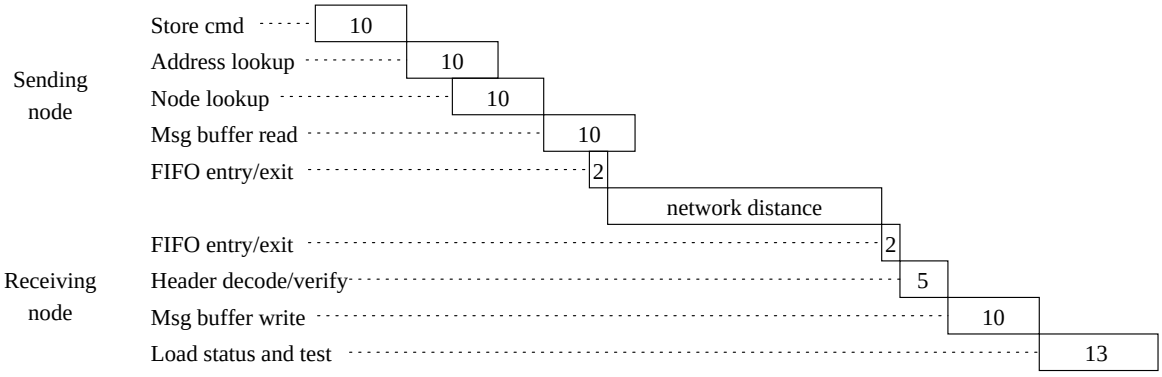


Figure 6.1: Latency of a single packet under the model "Cranium auto, cold"

The table contains six rows describing the cycle counts for six different cases. The first two rows represent lower bounds that a real network interface cannot exceed. The first row, labeled "Ideal, no mem," represents an interface that does not transfer information to or from the memory subsystem; it assumes that all packet data are transferred directly to and from network interface registers. The second row, labeled "Ideal, mem," includes the DRAM access latency at both the sender and the receiver. The cycle counts for the two rows representing the ideal network interface reflect only the command and status accesses by the processor in the non-memory case, plus the cost of the memory accesses for the packet payload in the memory case. The cost of FIFO entry/exit, header decode and the receiving node's test and branch are omitted. The cost of this test and branch can be eliminated when the processor architecture uses a synchronization mechanism like the full-empty bits in the Tera MTA-1 [46].

The cycle counts contained in the four following rows are based on the simulator described in Chapter 5 and the timing analysis of the Teschio implementation of Cranium presented in Section7.4. The four cases come from two situations at the sender (cold channel, warm channel) and two at the receiver (automatic receive channel, queue channel). Recall that in the cold channel case, the physical node identifier and the physical buffer address must be fetched by the network interface from main memory. In the warm channel case, the physical node ID and the physical buffer address have been accessed recently and are cached in the network interface, thereby avoiding this lookup step.

Figure 6.1 shows an in-depth breakdown of the components of latency under the case "Cranium auto, cold". Each rectangle in the figure represents a different

component; total time accumulates from left to right. In some circumstances, the rectangles are overlapped to represent internal pipelining. The first component is a store command from the processor. Since the send channel is cold, both the buffer address and the node identifier must be looked up in memory. These lookups are pipelined and take a total of 15 cycles; this cost is zero when the send channel is warm. The next component is reading the packet payload out of memory. This operation takes a total of 10 cycles but the first phit of the payload is ready at the sixth cycle. The packet goes into a FIFO and from the FIFO into the network; it takes one cycle to enter the FIFO and one to enter the network. When the packet is delivered from the network it again takes two cycles to enter and exit the FIFO at the receiver. It takes four cycles for the receiver to decode and verify the header. Then it takes 10 cycles to write the packet payload into memory. The processor at the receiving node takes 13 cycles to load the status information from the interface, then test the status and branch.

6.1.2 Throughput

The per-node throughput is the effective rate at which the network interface injects or delivers a message, calculated as the total number of bytes divided by the total number of clock cycles. The throughput of the processor-network link is a function of the implementation parameters w, y, h, v_{msg}, v_{packet} and g. w is the raw bandwidth the processor-network link. y is the length of the packet payload in bytes, the portion of the packet that is usable by application programs. h is the length of the packet header in bytes. v_{msg} and v_{packet} represent software costs: v_{msg} describes the number of cycles lost to software overhead per message and v_{packet} is the number of overhead cycles per packet. g is number of cycles it takes to send a single packet in a multiple packet message, equal to the length of a whole packet (the header plus the payload) plus w times v_{packet}. Equation 6.1 calculates the peak throughput TP_{peak} given w, y, h and v_{packet}.

$$\text{TP}_{\text{peak}} = y/g = \frac{w \cdot y}{y + h + w \cdot v_{\text{packet}}} \tag{6.1}$$

v_{packet} can be reduced to zero if the network interface uses DMA (see Section 2.1.2), but not if it is based on programmed I/O. In the implementation of

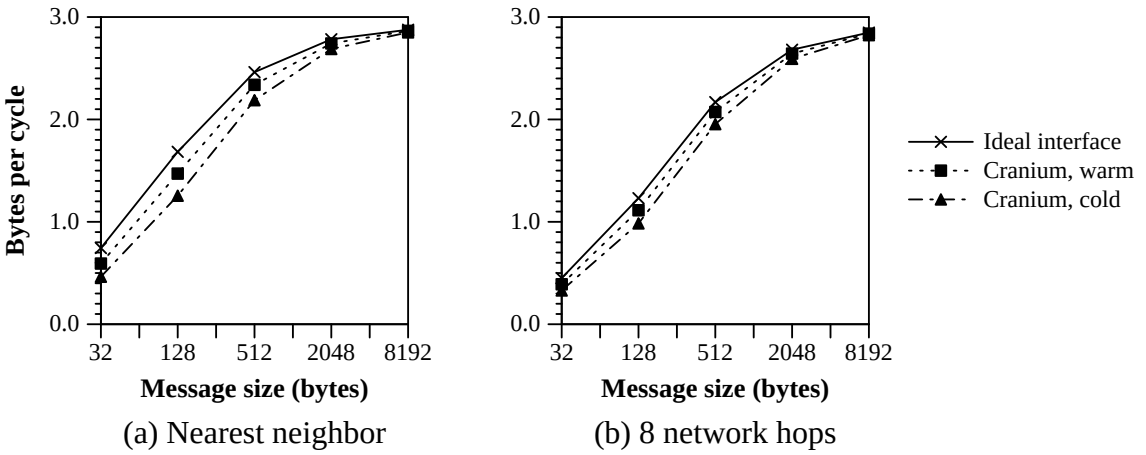


Figure 6.2: Throughput of a point-to-point message

Cranium in Chapter 7, w is four bytes per cycle, y is 32 bytes, h is 12 bytes and v_{packet} is zero. Therefore TP_{peak} is $4 \times 32/44 = 2.91$ bytes per cycle.

The actual throughput of a multiple-packet message is somewhat less than TP_{peak} due to the startup cost l of sending the first packet, i.e. the latency due to the distance the packet travels through the network plus the length of the packet. Pipelining in the network and the network interface hides this latency for subsequent packets. Equation 6.2 calculates $\text{TP}(k)$, the throughput of a k-packet message:

$$\text{TP}(k) = \frac{k \cdot w \cdot y}{w \cdot (l + v_{\text{msg}}) + k \cdot (y + h + w \cdot v_{\text{packet}})} \tag{6.2}$$

Figure 6.2 displays a pair of semi-log graphs of $\text{TP}(k)$ describing the throughput of a point-to-point message. Each graph contains three curves representing two cases for Cranium (auto-channel cold and auto-channel warm) and the ideal interface. We assume that messages must start and end in memory, so the non-memory ideal case from Table 6.1 is not included. Figure 6.2a represents the throughput when sending to a nearest-neighbor node and Figure 6.2b represents the throughput when sending to a node eight network hops away from the sender. Higher curves in the graphs represent higher performance. As the length of the message increases, the difference between any two of the three graphs becomes increasingly small. For a message size of 2048 bytes, Cranium yields 95% of the throughput that is achievable using the ideal network interface. The worst case for Cranium is for a cold channel with a message length of a single packet and a network distance of a single hop; the throughput in this

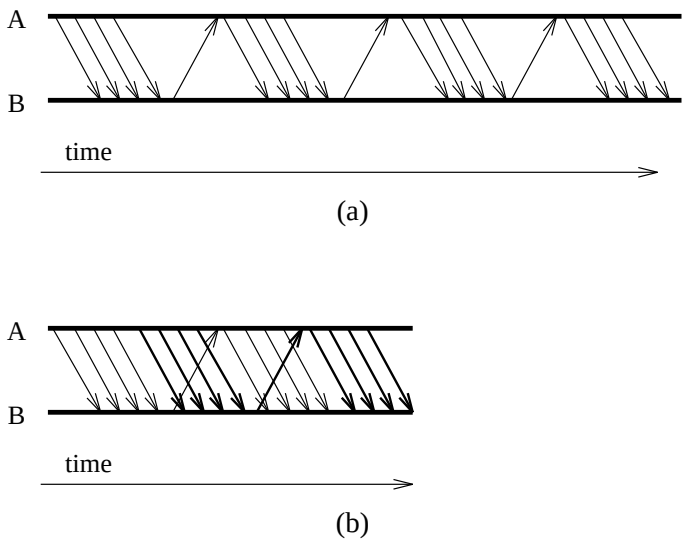


Figure 6.3: Sliding window protocol

case is only 60% of the achievable throughput using the ideal interface. Replacing the cold channel with a warm channel and keeping the other parameters fixed improves the percentage of throughput achieved from 60% to 80%.

Very long messages

Cranium cannot directly transmit a message longer than an MMU page directly using a single channel. Very long messages are handled by using a software layer on top of Cranium that partitions the transfer across multiple channels. A message whose length is only a few pages can be transmitted by activating several channels concurrently. However, in general the length of the message in pages may be larger than the total number of channels. Therefore a software protocol manages the reuse of channels for a very long transfer. One such protocol is known as the sliding-window protocol. Figure 6.3a illustrates how the protocol works. Node B waits to receive a page of data from node A; a page of data is represented in the figure by each group of four consecutive downward diagonal lines. When all packets in the page arrive, node B sends a synchronization packet back to node A; the sync packets are represented by the upward diagonal lines immediately to the right of each group of data packets. The sync packet indicates that the receiver is ready to receive the next page of data from the sender. After sending the page, node A waits until it receives the sync packet from

B before continuing. In Figure 6.3a there is only one channel used for sending. The problem with using a single channel is that the bandwidth of the network link is not utilized efficiently; node A does not transmit while it is waiting for the sync packet to arrive. To increase the network link utilization and decrease the cycle count, two channels are used concurrently (Figure 6.3b). One channel actively transmits while the other waits to synchronize.

Equation 6.3 describes $\mathrm{TP}(k, m)$, the throughput of very long transfers (i.e. greater than two MMU pages in length) using the scheme described above. k is the number of data packets transferred and m is the number of packets per page.

$$\mathrm{TP}(k, m) = \frac{k \cdot w \cdot y}{w \cdot (l + v_{\mathrm{msg}}) + (k + \lceil k/m \rceil - 2) \cdot (y + h + w \cdot v_{\mathrm{packet}})} \tag{6.3}$$

The subexpression $\lceil k/m \rceil - 2$ in Equation 6.3 represents the number of sync packets needed when two channels are used. In essence, the only extra cost incurred for transmitting very long messages is the bandwidth penalty of the additional sync packets. There is no additional latency penalty assessed for the sync packets. The processor and the network interface operate independently; Cranium allows send or receive commands to be accepted while DMA is in progress. The extra processor instructions required to react to the sync packet and start the next send channel transfer are overlapped with the DMA in progress.

For message sizes of 16K bytes and above, Cranium achieves 98% or greater of the peak throughput of the channel. This result comes from using Equation 6.3 with $k = 512$, $l = 50$ and $m = 128$. A plot of $\mathrm{TP}(k, m)$ as k increases beyond 512 for the three cases used in Figure 6.2 is practically flat with the three curves closely approaching the asymptotic value of 2.91. Indeed, software support for very large messages is efficient enough to eliminate the need for supporting this feature directly in hardware.

6.1.3 Broadcast

A broadcast operation distributes information from a single node (known as the root node) to all the other nodes. Broadcasting is a common operation in many parallel application programs. An example is the Gaussian elimination program described in Section 6.2.2, in which the pivot row located at the root node is broadcast to all

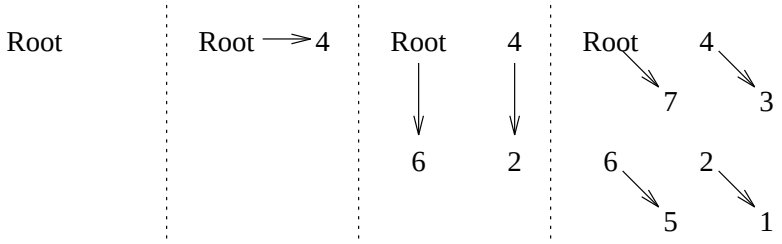


Figure 6.4: A hypercube topology for tree broadcast

other nodes. Two operations related to broadcasting are barriers and global combines. Barriers and global combines execute in two phases: a reverse broadcast (fan-in) followed by a broadcast (fan-out). The difference between a barrier and a global combine is that a barrier carries synchronization information only. In a global combine, values are sent to the root node and combined using a binary arithmetic operation that is associative and commutative. Common combining operations are addition, multiplication and maximum. The root node serializes the execution of both types of operations; all nodes must enter the barrier or global combine before any node observes the end of the barrier or the result of the global combine.

There are a variety of methods for implementing a broadcast on a point-to-point network [78]. The most obvious approach is for the root node to send single messages directly to all the other nodes. In a system with p nodes, the root node sends $p-1$ messages. This approach works well only if p is small. The time it takes for the last node to receive the message is the single message latency plus the time it takes the root node to inject the previous $p-2$ messages. As the number of nodes increases, the bandwidth bottleneck at the root node dominates the total time. A more efficient technique for large p is a tree broadcast. One type of tree broadcast is based on hypercubes of increasing dimension [4]. The root node (node 0) sends to nodes $p/2$, $3p/4$, $7p/8$ and so on. Node $p/2$ sends to node $p/4$, $3p/8$, and so on, recursively. In all there are $\log p$ phases. Figure 6.4 illustrates the case for $p = 8$. If the receiving nodes could retransmit at the same instant that the sender stopped sending then the total time would be exactly $kg \log p$, where k is the number of packets in the message; this expression is a lower bound on the latency of tree broadcast. By similar reasoning an upper bound on throughput of a tree broadcast of a long message is $\mathrm{TP}_{\mathrm{peak}}/\log p$. Under realistic assumptions of network traversal time and the overhead at the receiver, the messages comprising a phase of the broadcast are not perfectly synchronized.

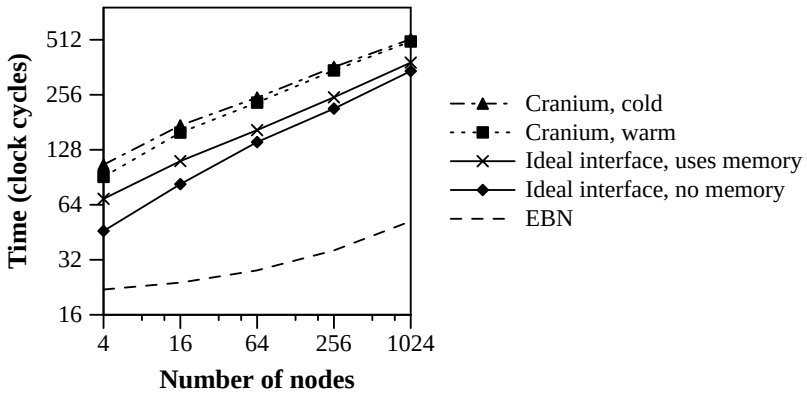


Figure 6.5: Latency of optimal tree broadcast a single packet message

For example, in Figure 6.4, the packet from the root node to node 6 would arrive sooner than the packet from node 4 to node 2 due to the receive overhead at node 4. When these assumptions are taken into account, it turns out that the hypercube scheme is not always the optimal pattern for tree broadcast. Culler et al. describe the construction of the optimal pattern for broadcast on a point-to-point network [79, 80, 81]. The pattern for optimal tree broadcast is a function of the relative values of the g, the network distance and the software overhead. In general a faster broadcast pattern is constructed by increasing the branching factor at each sending node to decrease the height of the tree compared with the hypercube scheme.

The bounds on latency and throughput of tree broadcast determine the bounds on latency and throughput for barriers and global combines based on trees. The optimal pattern for the fan-in phase of the global combine is exactly the reverse of the optimal pattern for broadcast [80]. The lower bound on execution time for barriers and global combines is therefore twice the lower bound for broadcast. Global combines are slightly slower than barriers due to the extra time needed to execute the combining operation. A lower bound on latency of barriers and global combines is $2kg \log p$. An upper bound on throughput of global combines is $\mathrm{TP}_{\mathrm{peak}}/\log p$, the same as the throughput bound on broadcast, because twice the number of bytes are communicated in twice the time. Thus, it is simple to apply the analysis of the performance of tree broadcast to determine bounds on the performance of barriers and global combines based on trees.

To improve the performance of application programs that use broadcast, some

multicomputers use dedicated hardware and/or a separate broadcast network to reduce the latency of broadcasting. A well known example in a commercial system is the CM-5 control network [14]. For the torus network, an elegant solution is provided by the Express Broadcast Network (EBN) [61]. EBN is a low-cost extension to mesh and torus data networks for supporting low-latency broadcast of control messages. EBN increases the width of each link in the existing network by one extra wire. The broadcast pattern in EBN on the torus is a wavefront that expands by one hop per clock cycle. The total broadcast time is $\sqrt{p}$, the diameter of the network, plus the software overhead at the sender and receiver. The latency of broadcasting under EBN grows according to the square root instead of the log of the number of nodes, but its constant factor is much lower than that of the point-to-point network broadcasts. For a network whose size is a few thousand nodes or less, EBN provides a much more efficient mechanism than point-to-point broadcast.

Figure 6.5 is a log-log graph that plots the latency in clock cycles of an optimal broadcast of a single packet versus network size. Lower curves indicate better performance. Assuming the standard point-to-point network is used for broadcasting, four kinds of network interfaces are compared: the two Cranium auto-channel models (warm and cold) and both the with-memory and non-memory versions of the ideal interface. For comparison, a fifth curve displays the latency of EBN. The curves for Cranium do not converge with the curves for the ideal interface; the difference between the two pairs of curves is slightly less than a factor of two. The reason is that broadcasting a single packet is not as efficient as sending a long point-to-point message under Cranium. Each packet in the broadcast requires a separate send command; in a point-to-point message up to a page in length, one send command suffices. The software overhead of one send command per packet increases the gap between packets from 11 to 20 cycles. In the ideal interface the gap between packets in a single packet broadcast is 11 cycles, the same as in a long message.

Figure 6.6 is a log-log graph that shows the throughput of a broadcast of a 1K byte message in bytes per cycle. For comparison, the curve for the upper bound described by $\mathrm{TP}_{\mathrm{peak}}/\log p$ is included. In this case, Cranium achieves the large message gap of 11 cycles rather than 20 cycles in the single packet broadcast case. The three curves describing Cranium auto-channel warm, Cranium auto-channel cold and the ideal interface are nearly coincident.

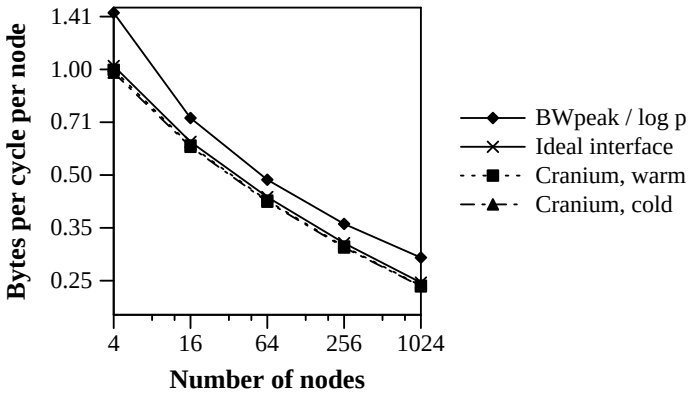


Figure 6.6: Throughput of broadcasting a 1 Kbyte message

The performance results from this study show that Cranium is an effective interface for broadcasting data values. Cranium provides performance that is within a factor of two of the ideal interface for the broadcast of minimum sized messages, and virtually identical performance for tree broadcast of long messages. However, the broadcast of control information (i.e. support for barrier synchronization) is better suited to a faster mechanism provided by the network if possible. The Express Broadcast Network is one such facility that works directly with torus networks. The latency of broadcast under EBN is a tiny fraction of the latency of tree broadcast over the standard point-to-point network.

6.2 Empirical evaluation of Cranium

A suite of parallel benchmark programs was created to evaluate Cranium empirically. This evaluation technique complements the analytic approach by providing a more realistic determination of the performance of the communication system. The benchmarks involve a mix of large and small messages using a variety of communication traffic patterns; they are computation-intensive and the interaction between computation and communication is complex. Execution time is minimized by overlapping communication and computation; wherever possible, the programs communicate one set of data while simultaneously operating on a different set of data.

The rest of this section is organized as follows. Section 6.2.1 introduces the goals used in creating the benchmark suite. Section 6.2.2 motivates and describes each of

the five benchmarks in the suite. Section 6.2.3 discusses the implementation of the benchmarks. Section 6.2.4 lists the quantities measured in each run of the simulator. Section 6.2.5 shows the computation of the maximum performance of the benchmarks if the communication cost is ignored. Section 6.2.6 describes the significance of the communication cost; Section 6.2.7 shows the actual performance (speedup) of each benchmark. Section 6.2.8 compares the empirical performance of the communication system with the analysis from Section 6.1. Section 6.2.9 summarizes the ideas presented in this section.

6.2.1 Goals

A good parallel benchmark has three qualities: relevance, simplicity and significant communication. The parallel programs in the benchmark suite are based on relevant problems in computer science or one of the physical sciences such as chemistry. Simplicity provides several benefits for a benchmark: quick construction, ease of analysis and ease of source-level optimization. Simple programs can fit entirely within the instruction cache and thereby reduce cache effects on the running time.

The time spent communicating should comprise a significant fraction of the total running time of the parallel program. The idea is to avoid so-called embarrassingly parallel applications that perform very little communication. These applications often deliver good speedups using even a low-bandwidth communication subsystem such as Ethernet. Since there is little stress on the communication system, embarrassingly parallel applications provide no compelling reason to improve the performance of the network interface. On the other hand it makes no practical sense to insert unnecessary communication into a benchmark. Therefore it is equally important is to minimize the computation time by making the benchmark as efficient as possible. Otherwise, inefficiencies in the computation may mask the role of communication as a factor in the overall performance of the benchmark.

There are two contrasting styles of parallel program execution: balanced and unbalanced. Communication patterns in a balanced program tend to be regular. All processors share the workload equally; no processor spends considerably more time waiting than any of the others. Unbalanced programs on the other hand tend to be irregular. The communication patterns are often dependent on the input data and cannot be statically optimized. Some processors are idle while others are busy,

resulting in less than optimal speedups. Load imbalance often dominates the total running time of the parallel application program. A well-balanced program is more likely to isolate the network interface as the performance bottleneck than an unbalanced program is. Problems that have sparse data structures, such as particle-in-cell simulations, tend to be difficult to load-balance [82]. While these types of computations are important and would need to work well with Cranium in principle, they are less suitable as benchmarks for detailed comparison of network interface features.

The most popular choice for an efficient implementation language style is the sequential imperative style. Usually a C or Fortran program is augmented with a message-passing library such as the Intel NX library [7]. This technique for parallel program development is known as hand-crafted code, where each communication operation is stated explicitly in the program. The competing approach is to write programs in a high-level language that is designed for parallel programming. The HLL program is translated into a message-passing program by a source-to-source compiler. While research prototypes of parallel HLL implementations are improving [83], the message-passing programs produced by commercially available parallel HLL compilers do not yet provide the efficiency of hand-crafted message-passing code.

6.2.2 Suite of parallel benchmark programs

The following subsections describe the benchmarks that comprise the suite used to evaluate Cranium: Fourier transform, bucket sort, Jacobi successive-over-relaxation, Gaussian elimination and dense matrix multiply.

Fast Fourier Transform

The discrete Fourier Transform (DFT) [84] is a standard computation in digital signal processing for converting a vector of k complex points sampled in the time domain into a vector of k complex points in the frequency domain. DFT generalizes to multiple dimensions; for simplicity the benchmark used in this chapter uses one-dimensional input and output data sets. A straightforward coding of DFT is a matrix-vector product that requires $O(k^2)$ complex multiplications. Fast Fourier Transform [85] is a variant of DFT that is more computationally efficient; the FFT algorithm executes $O(k \log k)$ complex multiplications.

FFT executes in $\log k$ phases of computation. The first $\log p$ phases require both communication and computation, where p is the number of processing nodes. Each node in each phase communicates with the conjugate node specific to the phase. The ID of the conjugate node is the ID of the given node with one of the bits complemented. In the first phase, this bit is the most significant bit of the ID. On each successive phase the next least significant bit is complemented, on down to the least significant bit. The final $(\log k - \log p)$ computation phases require no interprocessor communication[2]. The computation in FFT consists of the k complex multiplications per phase plus some bookkeeping such as computing the complex roots of unity needed for the phase. Each complex multiplication requires four scalar multiplication operations. Communication is overlapped with the bookkeeping operations but not the multiplications. Nevertheless, FFT parallelizes very well even when the input data set is relatively small compared with the number of processing nodes.

Bucket sort

Sorting is a ubiquitous computer application. A sizable fraction of the world's computing cycles go to sorting database records, bank transactions and so on. The sorting benchmark given here is bucket sort, similar to radix sort. For simplicity only the keys are sorted. Each processing node begins with an initial set of keys and sorts them locally, then partitions them into buckets. Each bucket is sent to its designated destination processor, where a final merge is performed. In general, load balancing may be a problem for the final merge if the number of keys in each bucket varies greatly. One strategy to improve load balancing is to perform a two-pass algorithm. In the first pass the input data set is sampled to determine the boundaries of each bucket dynamically [86], so that all buckets will contain roughly the same number of keys when the sorting operation is complete. To simplify this implementation of the sorting benchmark, the bucket boundaries are determined statically. The keys are generated using a uniform random distribution. During most runs of the program, the variation in bucket size tends to be small; the overhead due to load imbalance does

[2] There is a final communication phase in FFT called descrambling that places the output data in the same order as the input data set. In descrambling, each node communicates with its conjugate node whose ID is the ID of the original node with the bits in reverse order. For simplicity, descrambling is not implemented in this FFT benchmark.

not dominate the running time. The communication pattern in the parallel bucket sort algorithm is all-to-all. There is no overlap between sorting and communication, but communication overlaps with other communication. Since the number of keys per bucket varies across different runs of the program, the length of each message is determined at run-time.

Jacobi successive-over-relaxation

The Jacobi algorithm provides a solution to a finite-element problem such as a heat-transfer problem. Its output is a discrete approximation rather than an exact analytic solution. The simplest non-trivial version is in two dimensions. The data set consists of a rectangular 2-D array of real numbers. Values on the edges of the array are constant through the entire execution. The interior values of the array change on each iteration and converge to their final values. For each iteration, each value is computed as the average of its four nearest neighbor values to the north, south, east and west. The difference in the value in the same location on two successive generations is called the error term. The largest error over the entire array is computed on every iteration; when this maximal error is smaller than a user-specified constant value, the algorithm terminates.

To parallelize the sequential version of Jacobi, the array is divided into rectangular tiles. Each processor updates the values in its tile on each iteration. Internode communication is necessary for computing values on the edges and corners of each tile but not for values in the interior of a tile. Processing nodes that share a north-south tile border pass row values, and nodes that share an east-west border pass column values. Because column values are not contiguous in memory, they must be gathered into a contiguous block for DMA transfer before sending, and scattered from a contiguous block into columns at the receiver. The global error term is collected and distributed via a global combine operation. Communication is overlapped with computation in two ways: computation of the tile interior values is overlapped with nearest neighbor communication, and the local copying operations to implement gather and scatter are overlapped with the global combine. As a result, Jacobi achieves good speedups when the data set size is sufficiently large. If the data set size is small, the global combine is the critical delay in the benchmark.

Two versions of Jacobi were developed: one that performs a global combine to test the termination condition for every iteration, and another that performs a fixed number of iterations and terminates. Because the error term decreases monotonically, the termination test could occur less frequently (e.g. once every four iterations) and the resulting program would still return the correct result. The tradeoff is that the program will execute a number of extra iterations and may cause the program to run more slowly. Determining the optimal number of iterations to skip seemed to be a difficult problem; the simplest solution was to eliminate the global combine entirely. The latter version more closely reflects the maximum throughput that the benchmark can achieve from the network interface.

Gaussian elimination

Gaussian elimination (or Gauss for short) is a standard algorithm for inverting an $n \times n$ matrix to solve a system of linear equations [87]. Here is the implementation used in this chapter. The matrix is partitioned into rows, with one or more rows allocated to each processing node. The algorithm has n phases. In phase k, the values in every row of the matrix are divided by the value in the kth column of that row (if it is nonzero). By subtracting row k (the pivot row) from all the other rows, the values in column k of the other rows are zeroed (eliminated). By repeating this process, nonzero elements are left only along the diagonal of the matrix. Because the same value is used as the divisor for every value in the row, a simple optimization is to take the reciprocal of the divisor and multiply every value in the row by this reciprocal. The computation runs faster because floating-point multiplication is about six times as fast as floating-point division on the 88100. The algorithm performs $n - k$ multiplications per phase per row, for a total of $n^2/2$ multiplications per row per processor. The communication pattern is a broadcast from the pivot row processor to all the other processors on each phase. The size of the message that is broadcast is the entire length of the row during the first iteration; after each iteration there is one fewer value that needs be transferred, so the size of the message declines linearly until the final iteration where the length of the message to broadcast is a single packet. Gauss parallelizes well with a sufficiently large input data set because it is able to overlap its communication with the multiplication operations.

Dense matrix multiply

Computing the product of two matrices is a common operation in many scientific programs. There are two common cases: *dense*, where the input matrices contain mostly non-zero elements, and *sparse*, where the input matrices contain mostly zero elements. In an optimal computation of a sparse matrix product, most of the multiplication operations can be eliminated. However the remaining multiplication operations may be unevenly distributed across the computing nodes, creating a load balancing problem. By contrast, all the partial products in a dense matrix product must be computed, and it is simple to distribute them evenly across all the computing nodes. Therefore the dense matrix product solver has good load balance and is a better benchmark for evaluating network interfaces than sparse matrix product is.

The standard algorithm for dense matrix multiplication (DMM) is called Cannon's algorithm [88]. The algorithm is very regular and well suited to special-purpose systolic cellular automata hardware [89]. It has been implemented on general-purpose multicomputers using a variety of parallel high-level languages including C* [90], Spot [91] and Orca C [92]. Here is a brief description of the algorithm. For simplicity, the two input matrices $\mathbf{A}$ and $\mathbf{B}$ are square and each contains $n \times n$ elements. The desired product $\mathbf{C}$ is $\mathbf{AB}$ and is also an $n \times n$ matrix. The initialization step in Cannon's algorithm requires the array elements in the rows of $\mathbf{A}$ and the columns of $\mathbf{B}$ to be rotated around (skewed) as follows:

$$\mathbf{A}_0[i,\ j] = \mathbf{A}_{\text{input}}[i,\ (i+j) \bmod n] \tag{6.4}$$

$$\mathbf{B}_0[i,\ j] = \mathbf{B}_{\text{input}}[(i+j) \bmod n,\ j] \tag{6.5}$$

The algorithm proceeds as a series of n iterations. In the first iteration, the skewed arrays $\mathbf{A}$ and $\mathbf{B}$ are overlaid and the scalar product at each array position is stored into $\mathbf{C}$. For each subsequent iteration, the rows of $\mathbf{A}$ are rotated one position left and the columns of $\mathbf{B}$ are rotated one position up. Again the scalar products are taken at each array position and summed into $\mathbf{C}$. Another way to state the algorithm is through the recurrences:

$$\mathbf{C}_0[i,\ j] = \mathbf{A}_0[i,\ j] \cdot \mathbf{B}_0[i,\ j] \tag{6.6}$$

$$\mathbf{A}_{\mathrm{k}}[i,\ j] = \mathbf{A}_{\mathrm{k\text{-}1}}[i,\ (j+1) \bmod n], \quad \text{for } k = 1 \text{ to } \ n-1 \tag{6.7}$$

$$\mathbf{B}_{\mathrm{k}}[i,\ j] = \mathbf{B}_{\mathrm{k\text{-}1}}[(i+1) \bmod n,\ j], \quad \text{for } k = 1 \text{ to } \ n-1 \tag{6.8}$$

$$\mathbf{C}_{\mathrm{k}}[i,\ j] = \mathbf{C}_{\mathrm{k\text{-}1}}[i,\ j] + \mathbf{A}_{\mathrm{k}}[i,\ j] \cdot \mathbf{B}_{\mathrm{k}}[i,\ j], \quad \text{for } k = 1 \text{ to } \ n-1 \tag{6.9}$$

In each iteration there are n^2 multiplications and n^2 additions. For the complete algorithm there are a total of n^3 multiplications and $n^3 - n^2$ additions.

Cannon's algorithm parallelizes very intuitively. The array data are distributed equally across the computing nodes. Rotating the input arrays requires only nearest-neighbor communication. The standard approach alternates communication and computation phases without overlap, but a simple transformation makes it possible to overlap communication and computation.

There are two approaches to distributing the array data across the processing nodes [93]. The two-dimensional approach is to divide the arrays into rectangular tiles as in Jacobi. The one-dimensional approach is to distribute rows of the input arrays across the processing nodes as in Gauss. In the two-dimensional approach, row information from **A** is sent to the neighbor on the left and received from the neighbor on the right. Likewise, column information from **B** is sent to the neighbor above and received from the neighbor below. If gather-scatter hardware is available it can be used to aggregate the values in each row into one DMA operation and the values in each column into a second DMA operation. Otherwise there must be a separate message for each row and each column in the tile managed by the processing node.

The communication pattern is simpler under the one-dimensional partitioning than under the two-dimensional approach. In the one-dimensional approach, each processor contains all the row information from **A** it needs at initialization. Only column information from **B** is communicated at each iteration. Furthermore the column information from **B** becomes a single contiguous DMA transfer and thereby eliminates the need for gather-scatter operations. A single long DMA is more efficient than a series of short DMA transfers. The impact is that the one-dimensional data partitioning improves the performance of the communication system substantially over the two-dimensional partitioning and results in a higher performance implemen-

tation of the benchmark. Therefore the implementation of DMM in the benchmark suite uses a one-dimensional partitioning.

6.2.3 Benchmark implementation

The benchmarks used in this chapter were written in C and make direct use of the Cranium message-passing primitives. Every effort was made to ensure a high quality implementation of each benchmark. Where appropriate, optimizations were performed in the source code. Loops were unrolled to take advantage of the fixed size of the input set. The programs were then compiled by GCC version 2.2.2 with full optimization enabled.

In order to measure everything needed to evaluate the network interface, three different versions of each benchmark were created:

- A uniprocessor version that provides the basis for computing speedups.
- A message-passing version that has alternating phases of communication and computation that do not overlap, so that each can be measured separately.
- A message-passing version that overlaps communication and computation in order to minimize the total execution time.

The implementation of the benchmarks requires the number of processors p to either be a power of 2 or a square number. The values chosen for p for the measurements of the parallel versions of the benchmarks were 4, 8 or 9, 16, 32 or 36, and 64.

The input data sets for each benchmark are described in Table 6.2. All the benchmarks use a constant size input data set across all configurations of the simulator. The impact of a constant size input set is that the relative cost of communication becomes increasingly large as the number of nodes increases. The goal is to make communication significant rather than to determine the most efficient problem size for a given number of nodes. An alternative strategy is to scale the input data set with the number of nodes, and as a result the relative cost of communication may or may not increase as the number of nodes increases.

Table 6.2: Input data set sizes

Benchmark	Input data set
DMM	8192 (2x64x64) double-precision values
FFT	2048 (2x1024) double-precision values
Gauss	4096 (64x64) double-precision values
Jacobi	4096 (64x64) single-precision values
Sort	8192 integer keys

6.2.4 Benchmark measurements

Using the three versions of the benchmarks, the following quantities are measured with respect to p:

- $I_{comp}(1)$ is the number of instructions executed in the uniprocessor version of the benchmark. Integer and floating-point instructions are weighted equally.

- $C_{comp}(1)$ is the number of clock cycles executed in the uniprocessor version of the benchmark.

- $C_{ol}(p)$ and $C_{nol}(p)$ are the numbers of clock cycles it takes to execute the overlapping and the non-overlapping parallel versions of the benchmark, respectively. In programs that are able to overlap communication and computation, $C_{ol}(p) < C_{nol}(p)$. In benchmarks such as Sort that do not overlap communication and computation, $C_{ol}(p) = C_{nol}(p)$.

- $C_{comp}(p)$ and $C_{comm}(p)$ are the numbers of clock cycles it takes to execute only the computation part and only the communication part of the non-overlapping version of the benchmark, respectively. By definition, $C_{nol}(p) = C_{comp}(p) + C_{comm}(p)$. Computation and communication occur in alternating phases. Each phase is concluded using a global synchronization operation; the cycle count in each phase represents the maximum time taken across all nodes to complete the phase. The number of clock cycles lost to load imbalance in either computation or communication is not measured. The communication measurement includes

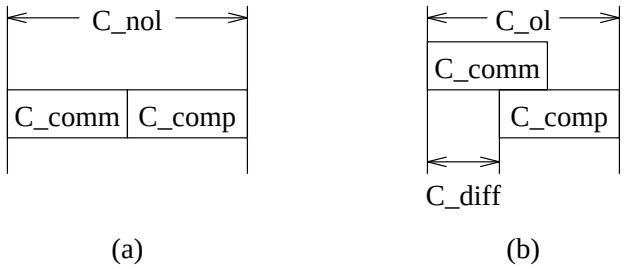


Figure 6.7: Non-overlapping communication vs. overlapping communication

all the software overhead associated with message passing: setting up send and receive buffers, computing addresses and indices, polling, etc.

- The derived quantity $C_{diff}(p)$ is equal to $C_{ol}(p) - C_{comp}(p)$. It is the effective cost of communication, the total cost of communication minus the part that is overlapped with computation. See Figure 6.7.

- $B_{comm}(p)$ is the total number of bytes communicated in the benchmark per node averaged over all nodes. Because packets are both sent and received, each byte is counted twice: once in the sender's interface and once in the receiver's.

Two separate measurements are taken for $C_{ol}(p)$, $C_{comm}(p)$ and $B_{comm}(p)$ in order to reflect two models of the memory system that are simulated. The measurements provided by the two models bound the performance of the communication system. The upper bound (pessimistic) measurement is based on the default memory timing model used in Talisman. In this model, only one memory transaction may be in progress at a time, and the DRAM access latency is 10 clock cycles. Pipelined or split-transaction memory busses have become commonplace in 1995 for connecting high-performance RISC processors with memory; the default DRAM timing model in Talisman does not reflect this trend. Therefore a second memory bus model is used to approximate this modern high performance memory bus technology. In this model, the DRAM access latency for the network interface is zero. However when the processor misses in the cache and accesses main memory, the DRAM latency is kept at 10 cycles in order to keep the computation cost the same under both models. The zero-latency memory model provides a lower bound on the communication cost and it better approximates the performance of long messages, because the latency of nearly

every DRAM access can be hidden (overlapped). This assumption is optimistic in the case of short messages because the DRAM latency is less likely to be hidden. The difference between the two measurements indicates the sensitivity of the benchmark to the performance of the memory system. The impact of the optimistic model is to remove the memory system bottleneck from the communication system, so that messages can utilize the full bandwidth of the network link.

Both memory models that are simulated are different than the memory model used in the implementation of Cranium in Chapter 7. The memory model described in Section 6.1, based on the Alpha Demonstration Unit's system bus, is the model used in the implementation chapter. The implementation's data bus is twice as wide as the simulated data bus (64 bits vs. 32); the former has higher throughput than the latter (a cache line every five cycles vs. a cache line every eight or eighteen cycles). The network is the same in all three versions. The impact of using a fast memory bus is that it improves the performance of the processor by servicing requests to the external cache more rapidly. However, the network is the performance bottleneck for both the implementation and the optimistic simulated memory model; consequently the peak throughput of the interface is the same for these two cases.

Sections 6.2.5 through 6.2.9 present the results of the experiments based on the five parallel benchmark programs. The raw measurements and the derived quantities based on these measurements for the selected values of p are displayed in Appendix B.

6.2.5 Determining maximum speedup and efficiency

The maximum speedup of a parallel benchmark is the speedup that could be achieved if the cost of communication were zero; it is the amount of speedup that the implementation is guaranteed not to exceed. Equation 6.10 is used to determine the maximum speedup:

$$\mathrm{SU}_{\max}(p) = \mathrm{C}_{\mathrm{comp}}(1)/\mathrm{C}_{\mathrm{comp}}(p) \tag{6.10}$$

Figure 6.8a represents a typical format for displaying speedup curves. Both axes of the graph are logarithmic to make better use of the area of the graph; with linear axes most of the points are bunched at the lower left corner of graph. All five of the benchmark programs exhibit good speedups as they lie on or near the diagonal representing linear speedup. Since it becomes difficult to distinguish points from

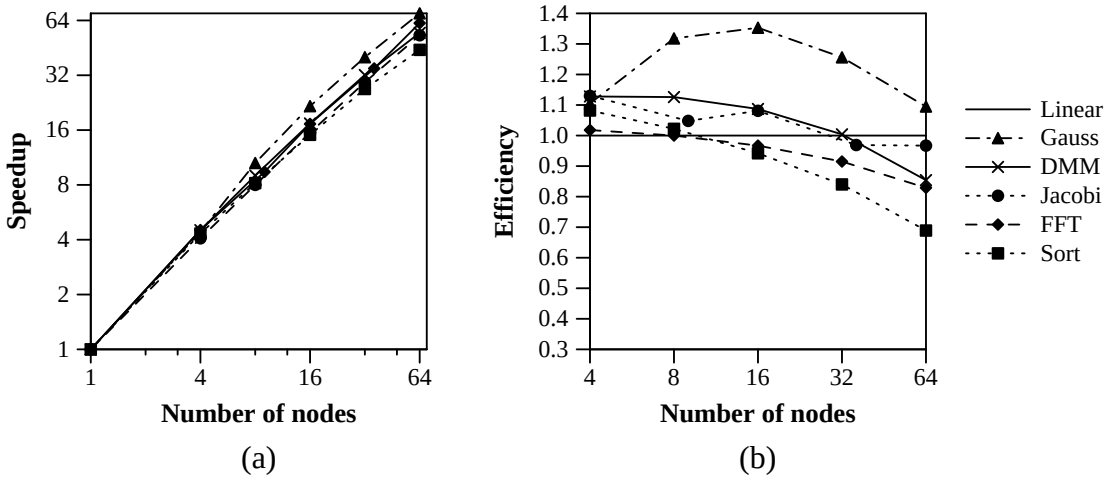


Figure 6.8: Maximum speedup and efficiency of benchmark programs

different curves because they lie nearly on top of one another, Figure 6.8b plots *efficiency* instead of speedup. The efficiency is the speedup divided by the number of nodes. The maximum efficiency $\mathrm{Eff}_{\max}(p)$ is calculated using Equation 6.11:

$$\mathrm{Eff}_{\max}(p) = \mathrm{C}_{\mathrm{comp}}(1)/(p \cdot \mathrm{C}_{\mathrm{comp}}(p)) \tag{6.11}$$

Figure 6.8b is a semi-log graph of maximum efficiency as a function of the number of nodes. Linear speedup is represented by the solid horizontal line at 1.0. The speedup is linear or better than linear in the number of nodes (superlinear) for all applications with eight or fewer nodes. Superlinear speedup happens because p processors contain p times the amount of cache memory. In the parallel versions of the benchmark programs the larger aggregate amount of cache reduces the systemwide number of capacity misses. The benefit of the extra cache to is the most pronounced with a small number of processors. Load imbalance and the overhead of synchronization decrease the speedups of the benchmarks as the number of nodes increases. The speedup curve for Sort drops off the most rapidly because Sort is the most susceptible to these factors.

A metric related to speedup is $\mathrm{IPC}(p)$, the aggregate number of instructions the parallel system executes per clock cycle. Achieving a linear increase in execution rate means that $\mathrm{IPC}(p) = p \cdot \mathrm{IPC}(1)$. Because the underlying architecture is a RISC that executes up to one instruction per cycle, a team of p processors can execute up to p instructions per cycle. In practice the IPC of the Motorola 88100 is as little as

Table 6.3: Maximum aggregate instructions per cycle executed when $p = 16$

Benchmark	$\mathrm{IPC}_{\max}(16)$
Sort	10.1
Gauss	6.08
Jacobi	6.21
FFT	4.85
DMM	2.96

one instruction every eight cycles, due to memory latency and the latency of floating-point arithmetic. $\mathrm{IPC}_{\max}(p)$ is the maximum rate of instruction execution that could be achieved if the cost of communication were zero; it is calculated using Equation 6.12:

$$\mathrm{IPC}_{\max}(p) = \mathrm{I}_{\mathrm{comp}}(1)/\mathrm{C}_{\mathrm{comp}}(p) \tag{6.12}$$

Table 6.3 displays the values of $\mathrm{IPC}_{\max}(\mathrm{p})$ for the five computation-intensive benchmarks using $p = 16$ as the representative number of processing nodes. Note that $\mathrm{IPC}_{\max}(p)$ varies from about 3 to 10 with 16 processors. Sort achieves the highest IPC because it uses no floating point arithmetic. DMM has the lowest IPC but the highest speedup as it makes extensive use of the floating point unit. FFT, Gauss and Jacobi achieve IPC ratings greater DMM but less than Sort. FFT and Gauss use a mix of floating point and integer arithmetic; Jacobi uses single-precision instead of double-precision floating point arithmetic. The benchmarks that achieve the lowest IPC shown in Table 6.3 are most likely to achieve a substantial improvement in IPC when run on today's high-performance multi-scalar processor architectures. As IPC increases, the relative cost of computation decreases, and consequently the relative cost of communication increases. The impact on performance studies of network interfaces is that the benchmark sizes may need to be scaled up so that the cost of communication does not completely dominate the running time of the benchmark. The relative costs of communication and computation are discussed in greater detail below.

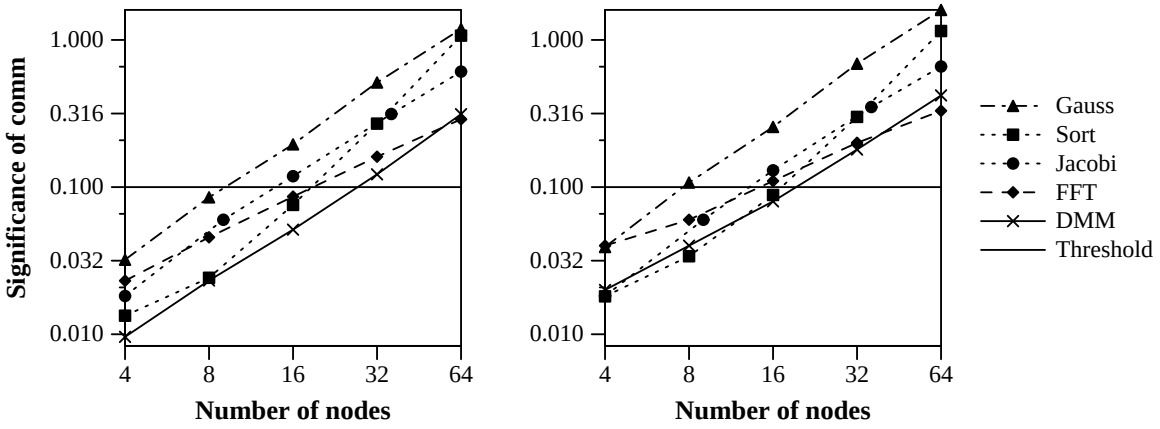
6.2.6 Determining the significance of the communication cost

In a parallel implementation of a benchmark designed to evaluate the network interface, how much time should be spent on communication? On one hand, the cost of communication cannot be trivial, because the point of the experiment is to have a significant amount of communication. On the other hand, if the cost of communication is very large then the resulting speedup will be poor. Ideally the communication cost is within a range that is both high enough to be significant and low enough to permit good speedup. Assume that the execution time of the serial version ($C_{comp}(1)$) is known. If the maximum speedup of a parallel implementation is approximately linear, then the cost of computation for p nodes is approximately $C_{comp}(1)/p$. For the purposes of this chapter, the cost of communication ought to be less than $C_{comp}(1)/p$ but not more than an order of magnitude or two less. Equation 6.13 defines $S_{comm}(p)$ as *the significance of the communication cost*, the ratio of the communication time to the computation cost predicted by the serial version of the computation:

$$S_{comm}(p) = C_{comm}(p)/(C_{comp}(1)/p) = p \cdot C_{comm}(p)/C_{comp}(1) \tag{6.13}$$

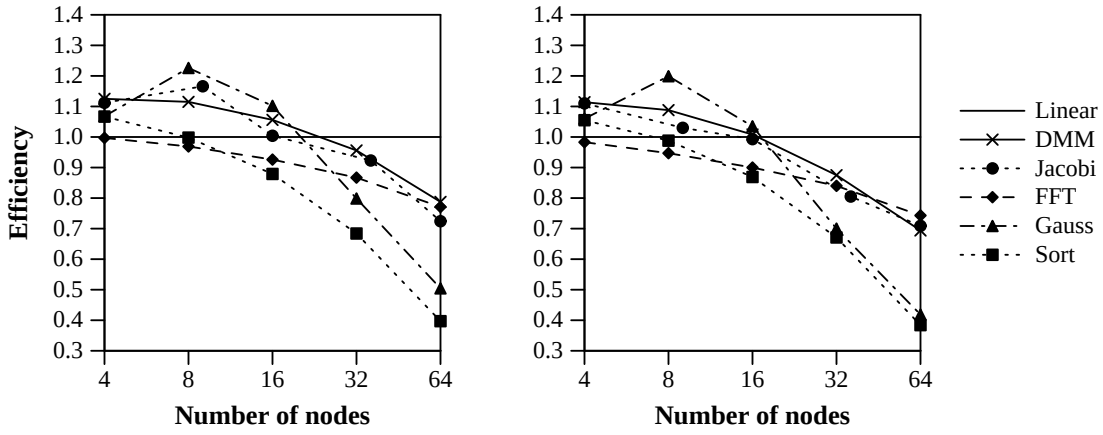
Defining the significance in this way makes it independent of the parallel implementation of the computation. For applications that parallelize well, $C_{comp}(p)$ is approximately the same as $C_{comp}(1)/p$. However, the computation alone may not parallelize well, so it may be misleading to assume that $C_{comp}(p) \approx C_{comp}(1)/p$.

Figure 6.9 displays a pair of log-log graphs that plot $S_{comm}(p)$ versus p for each benchmark. The graph on the left uses the optimistic memory model (DRAM access latency = 0 cycles) and the graph on the right uses the pessimistic model (DRAM access latency = 10 cycles). The slope of each curve shows how fast the communication cost grows as the number of nodes increases. The horizontal line shows the 10% threshold of significance, where the communication cost is an order of magnitude less than $C_{comp}(1)/p$. In all the benchmarks the significance of the communication cost is greater than 10% when 32 or more nodes are used; it is greater than 1% (two orders of magnitude less than $C_{comp}(1)/p$) in all cases except DMM running on four processors. These graphs show that the amount of communication is indeed significant for the given problem sizes. The applications that are most sensitive to the performance of the memory system are DMM and Gauss. DMM sends long fixed-size messages;



(a) 0 cycle DRAM latency (b) 10 cycle DRAM latency

Figure 6.9: Significance of communication cost



(a) 0 cycle DRAM latency (b) 10 cycle DRAM latency

Figure 6.10: Efficiency of the benchmark suite when the cost of communication is considered

the broadcast pattern in Gauss makes the communication time sensitive to the inter-packet gap (and hence the memory performance) at the sending node. Jacobi sends shorter messages and its communication performance is limited by software overhead rather than the communications hardware. FFT and Sort show aspects of both cases: the long message case when the number of nodes is small (16 or fewer) and the small message case with 32 or more nodes.

6.2.7 Putting it all together

By combining the efficiency of the computation with the significance of the communication, a complete picture emerges for the actual efficiency of each benchmark. The actual efficiency $\mathrm{Eff}_{\mathrm{act}}(p)$ is calculated using Equation 6.14:

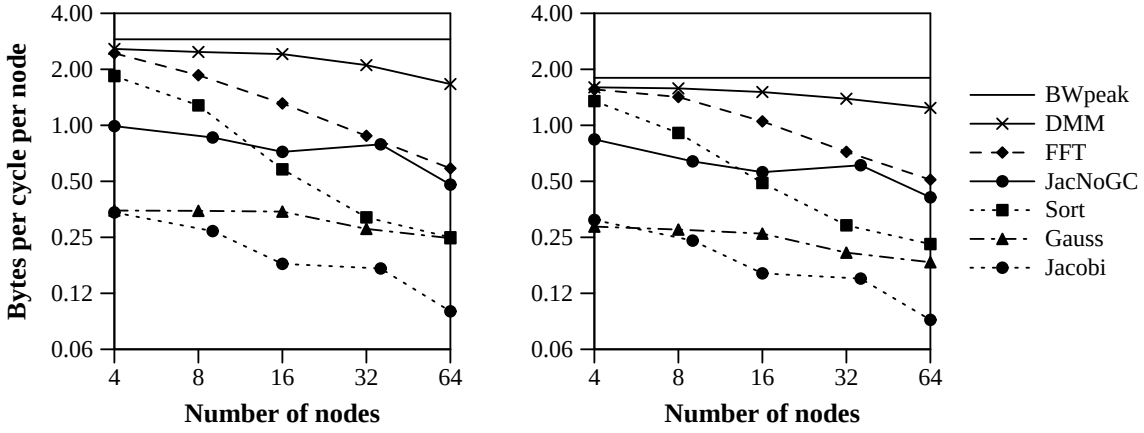
$$\mathrm{Eff}_{\mathrm{act}}(p) = \mathrm{SU}_{\mathrm{act}}(p)/p = (\mathrm{C}_{\mathrm{comp}}(1)/\mathrm{C}_{\mathrm{ol}}(p))/p \tag{6.14}$$

Figure 6.10 displays a pair of semi-log graphs that plot $\mathrm{Eff}_{\mathrm{act}}(p)$ for each benchmark under the two models of the memory system. It is instructive to compare Figure 6.10 with Figure 6.8b. All three graphs use the same scale in order to simplify the comparison. With 16 or fewer nodes, all the benchmarks achieve an actual efficiency rating of 85% or better. The actual efficiencies of both Sort and Gauss drop off quickly with 32 or more nodes. Sort has three strikes against it: its computation does not balance perfectly, it does not overlap communication and computation, and it passes an increasing number of smaller messages as the number of nodes increases. In Gauss, the communication pattern plays a role in making its performance difficult to scale up. The performance of Gauss would scale better if the broadcast of long messages were supported directly in the network.

6.2.8 Performance of the communication system

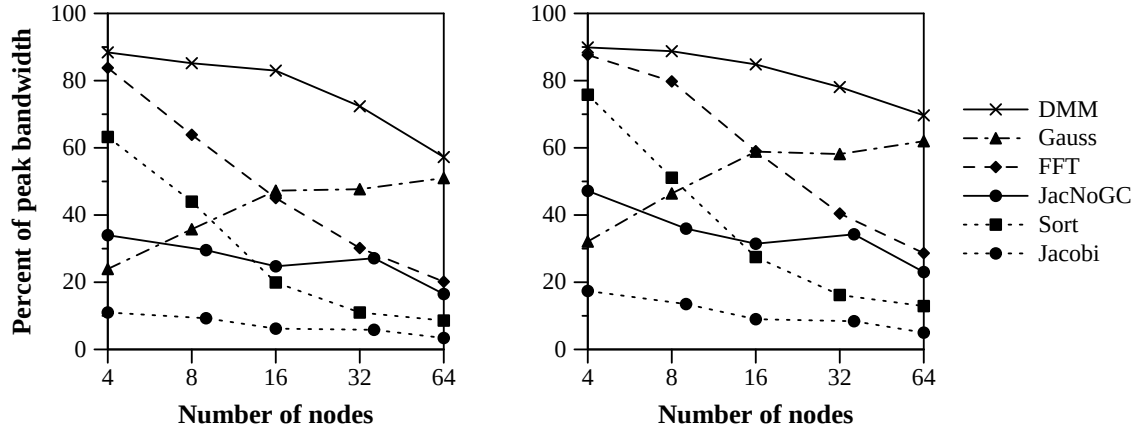
The performance of the communication system is the number of bytes it moves in a given number of clock cycles. $\mathrm{TP}(p)$ is the number of bytes moved per node per clock cycle; its equation is

$$\mathrm{TP}(p) = \mathrm{B}_{\mathrm{comm}}(p)/\mathrm{C}_{\mathrm{comm}}(p) \tag{6.15}$$



(a) 0 cycle DRAM latency (b) 10 cycle DRAM latency

Figure 6.11: Raw performance of the communication system in bytes per cycle



(a) 0 cycle DRAM latency (b) 10 cycle DRAM latency

Figure 6.12: Performance normalized to the maximum achievable throughput

The maximum throughput is $\mathrm{TP}_{\mathrm{peak}}$, introduced in Equation 6.1. Under the optimistic memory model, the processor-network link is the limiting factor, so it takes 11 cycles to inject or deliver a packet with its payload of 32 bytes. Under the pessimistic memory model, the memory system is the limiting factor; it takes $8 + 10 = 18$ cycles to store the 32 bytes into memory. The values for $\mathrm{TP}_{\mathrm{peak}}$ under the two memory models are:

$$\mathrm{TP}_{\mathrm{peak,opt}} = 32/11 = 2.91 \text{ bytes per cycle} \tag{6.16}$$

$$\mathrm{TP}_{\mathrm{peak,pess}} = 32/18 = 1.78 \text{ bytes per cycle} \tag{6.17}$$

The network itself never becomes a performance bottleneck in either the point-to-point messages or in any of the selected benchmark programs with the given data set sizes. If the data set sizes were larger and subsequently the network needed to handle a large number of large messages simultaneously, then internal congestion in the network can cause a performance bottleneck. The impact of the network on communication performance has been the focus of other research projects [12, 15, 74].

Figure 6.11 contains a pair of log-log graphs that plot $\mathrm{TP}(p)$ versus p. They demonstrate that the communication performance achieved by these benchmarks is spread over a wide range. The horizontal solid lines in the two graphs represent $\mathrm{TP}_{\mathrm{peak}}$ for the two cases. DMM achieves the highest communication performance as it has the best fit between hardware capability and software requirements – one message per communications phase per node, a relatively large number of packets per message and a fixed communication pattern that is the same for every phase. FFT bears some similarities to DMM: one message per communications phase per node and a fixed communication pattern. However in FFT the size of each message falls off by a factor of two for every factor of two increase in the number of nodes. FFT also incurs the software overhead of calculating the destination node ID which is different in every phase. For these reasons, the communication performance of FFT falls off more rapidly than that of DMM as the number of nodes increases. Jacobi is displayed using two curves: the original application (Jacobi) and a modified version with no global combine operation (JacNoGC). Global combine makes use of only a small fraction of the potential throughput. Without global combine, Jacobi achieves much a greater fraction of the peak throughput. However, Jacobi must copy data to perform gather and scatter operations for communicating column data between

east-west neighbors, resulting in lower communication performance than DMM and FFT. The communication performance of Sort drops off rapidly as the number of nodes grows. As message sizes become shorter, the communication cost becomes dominated by software overhead and the cost of synchronization.

A limitation of Figure 6.11 it that it is difficult to determine visually the percentage of peak throughput that Cranium achieves. To address this concern, Figure 6.12 provides the same information as Figure 6.11 using a percentage format. Figure 6.12 contains a pair of semi-log graphs that plot $\mathrm{TP}_{\mathrm{pct}}(p)$, the percentage of maximum throughput achieved by the benchmarks for the two memory models. In the general case, Equation 6.18 is used to compute $\mathrm{TP}_{\mathrm{pct}}(p)$:

$$\mathrm{TP}_{\mathrm{pct,general}}(p) = 100 \cdot \mathrm{TP}(p)/\mathrm{TP}_{\mathrm{peak}} \tag{6.18}$$

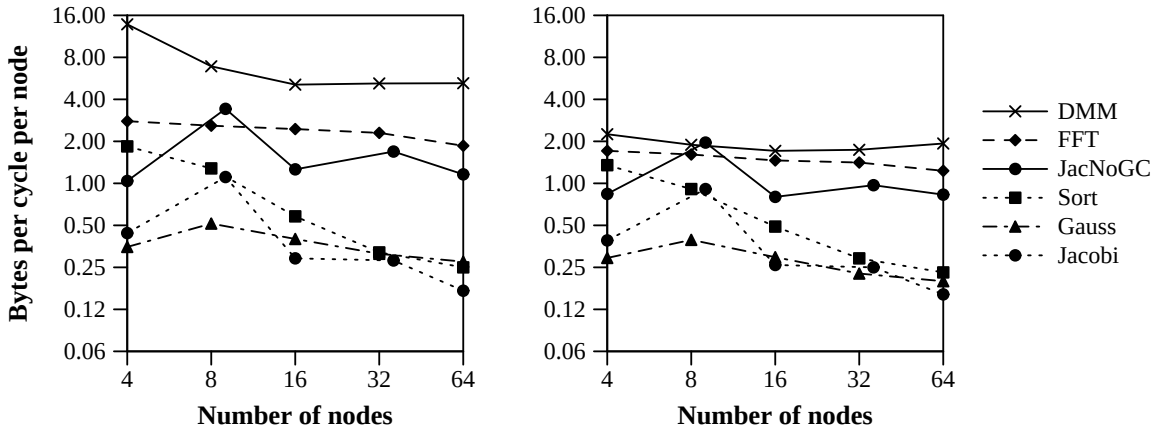
Gauss is treated as a special case because the only traffic pattern it uses is a tree broadcast. As shown in Figure 6.6, the throughput of a tree broadcast operation is bounded by $\mathrm{TP}_{\mathrm{peak}}/\log p$. Equation 6.19 computes the peak achievable throughput of a broadcast:

$$\mathrm{TP}_{\mathrm{pct,bcast}}(p) = 100 \cdot \log p \cdot \mathrm{TP}(p)/\mathrm{TP}_{\mathrm{peak}} \tag{6.19}$$

Figure 6.12 demonstrates that Cranium is an effective interface for a number of benchmarks, notably for DMM and FFT, and also for Sort with a small number of nodes and Gauss with a large number of nodes. It also highlights the cost of copying data: even when the global combine is omitted, the achieved throughput in Jacobi is less than 50% of the maximum.

In the case where the application program overlaps communication and computation, it is interesting to calculate the effective performance of the communication system. The effective communication cost in the overlapped version of the benchmark is $\mathrm{C}_{\mathrm{diff}}(p)$, defined in Section 6.2.4. The effective performance of the communication system is given by Equation 6.20:

$$\mathrm{TP}_{\mathrm{eff}}(p) = \mathrm{B}_{\mathrm{comm}}(p)/\mathrm{C}_{\mathrm{diff}}(p) = \mathrm{B}_{\mathrm{comm}}(p)/(\mathrm{C}_{\mathrm{ol}}(p) - \mathrm{C}_{\mathrm{comp}}(p)) \tag{6.20}$$



(a) 0 cycle DRAM latency (b) 10 cycle DRAM latency

Figure 6.13: Effective performance of the communication system in bytes per cycle using Equation 6.20

Figure 6.13 displays a pair of log-log graphs that plot $TP_{eff}(p)$ versus p. The graphs show that the effective communication performance is boosted above the corresponding base communication performance in Figure 6.11. All the benchmarks except Sort achieve a moderate to high amount of overlap between communication and computation. The effective throughput of DMM is greater than TP_{peak} across all configurations; also, it is quite sensitive to the performance of the memory system, shown by the difference in the height of the curves for DMM in Figure 6.13a and Figure 6.13b. FFT and Gauss exhibit a good degree of overlap between communication and computation. Both Jacobi and JacNoGC achieve a high degree of overlap with nine nodes. The reason for the spike at nine nodes is an artifact of the implementation. The other instances of Jacobi and JacNoGC send less data per node or spend more time in synchronization and thereby reduce the effective communication performance.

6.2.9 Summary

Methodology

A high-level benchmark suite was created for evaluating Cranium, consisting of five application programs: dense matrix multiply, fast Fourier transform, Gaussian elimi-

nation, Jacobi iteration and bucket sort. Three different versions of each benchmark were created: a serial version, a parallel version with separate communication and computation phases, and a parallel version that overlaps communication and computation for the shortest overall execution time. The measurements from the serial version were used as the basis for computing speedup and efficiency. The maximum speedup and efficiency are determined for each benchmark. The difference between the ideal speedup and the actual speedup demonstrates the impact of the cost of communication on the overall performance of the benchmarks. The measurements also provided the necessary information to calculate a number of interesting derived values: the significance of communication, the throughput, the percentage of achievable throughput and the effective throughput due to the overlap between communication and computation.

Two different memory models are used to bound the performance of the communication system. The optimistic (lower bound) model assumes the latency of accessing DRAM to be zero cycles; this assumption models the use of a split-transaction or overlapping memory bus to hide DRAM latency. The pessimistic (upper bound) model assumes that the DRAM access latency is 10 cycles and represents the default timing used in the simulator. The difference between the two measurements indicates the sensitivity of the benchmark to the performance of the memory system. The applications that show the most sensitivity are the ones that send large messages and achieve the highest percentage of peak throughput. The difference between the two bounds is small if the benchmark sends small messages or its communication performance is otherwise limited by software overhead. Since the difference is small in the small message case, and the optimistic model is more accurate than the pessimistic model in the long message case, the optimistic model is overall the more accurate indicator of performance.

Results

The selected parallel benchmarks demonstrate that Cranium is capable delivering more than 50% of the peak throughput that is achievable on very long point-to-point messages. Cranium achieves this level of communication performance despite the overhead of allocating buffers and synchronization, the use of complex communication patterns and the use of small input data set sizes. One benchmark, dense matrix

multiply, achieves more than 70% of the peak throughput when run on a system with 4 to 32 processing nodes (Figures 6.11 and 6.12). High throughput in the communication system does not come at the expense of parallel program performance: the benchmarks achieve excellent speedup figures despite the small input data set sizes (Figure 6.10). The ability of Cranium to overlap communication with computation improves the effective performance of the communication system (Figure 6.13). In the following section we show that Cranium's high communication performance is achieved by the combination of its low latency logical interface (the queue channels) and the high throughput logical interface (the auto-channels). The omission of either style of communication primitive causes a substantial reduction in both communication performance and the speedup of the parallel benchmarks.

6.3 Comparing Cranium against other network interface styles

The Cranium programming model allows the user to access the three principal structures in the network interface hardware directly: the send channels, the automatic-receive channels and the queue channels. The send channels allow a node to send a multiple-packet message with single software command, and allow more than one multiple-packet message send to be in progress. The automatic-receive channels complement the send channels and permit the reception of one or more multiple-packet messages from one or more source nodes. The automatic-receive channels implement the non-buffered communication style directly. Non-buffered means that there is no auxiliary buffer for temporary storage of incoming packet data; the processor never needs to transfer data from auxiliary buffers to the user program's address space. The restriction with non-buffered communication is that the incoming message must be anticipated – the source node and the size of the message have to be known in advance. In case the message's source and size are not knowable in advance, a buffered communication style is preferable. Cranium's receive queue implements buffered communication directly. The auxiliary buffer memory is organized as a circular queue in the user's address space; when it is necessary to copy data from the queue to a buffer allocated by the application program, the potential cost of crossing a protection boundary is avoided.

6.3.1 Modifying Cranium to emulate other network interfaces

A hypothesis of this dissertation is that the full Cranium programming model provides application programs with a minimum set of features needed to extract the full performance out of the underlying communications hardware. The hypothesis can be tested by modifying the Cranium programming model to emulate competing designs. The effects of these modifications can be evaluated by observing the change in maximum throughput achievable by the interface and the change in the cost of communication for application programs. The competing network interfaces of interest are the interfaces in the CM-5, SHRIMP-I and SHRIMP-II [14, 42, 43]. The following list of modifications M1 through M4 transform Cranium into interfaces that bear a close resemblance to one of the three competing network interface styles. In general, these modifications describe interfaces that have less hardware complexity than Cranium and potentially reduce the cost of implementation.

- **M1**: Each send command results in the injection of one packet into the network. Software is responsible for breaking long messages into separate packets and staging their injection into the network. The implementation is simpler than the Cranium send channel architecture that automatically packetizes and injects messages up to an MMU page in length.

- **M2**: Each incoming packet causes the processor to be notified; packet data are volatile and must be consumed by the processor or copied to a user memory buffer before the next incoming packet is processed. This modification simplifies Cranium by omitting the automatic-receive channels.

- **M3**: There is a one-to-one mapping between a physical message buffer on the sending node and a physical message buffer on the receiving node. Distributing a message to multiple destinations requires the sending node to copy the contents of the message buffer locally to other message buffers in the sending node's memory. This modification may simplify the implementation by reducing the number of bits passed from the processor to the network interface in the send command. The name of the buffer provides sufficient information to the network interface to select the destination node.

Table 6.4: Throughput of a long message under the modifications to Cranium

Model	g (cycles)	TP_{peak} (bytes/cycle)
Cranium unmodified	11	2.91
M1	18 to 31	1.03 to 1.78
M2	31 to 113	0.28 to 1.03
M3	29 to 111	0.29 to 1.10
M4	11	2.91

- **M4**: Only the unbuffered style of communication is supported. This modification to Cranium simplifies the implementation by omitting the queuing channels.

The modifications to Cranium that make it the most similar to the CM-5 are M1 and M2. The CM-5 uses only programmed I/O; the CM-5 processor creates, injects and retrieves each packet separately. Modification M3 is based on the SHRIMP-I interface. Modification M4 applies to both SHRIMP-I and SHRIMP-II.

6.3.2 Analytical evaluation of the modifications

There is very little difference in the minimum latency of packets under all four modifications. However, the throughput of large multi-packet messages decreases dramatically. The modifications greatly increase the value g from Equation 6.1 in Section 6.1.2. g is the minimum gap between successive injections or deliveries of packets. Table 6.4 summarizes the effect of each modification on g and the throughput of long messages. Under modifications M2 and M3, g is increased by the cost of copying the packet payload. Based on the timing information in the Talisman simulator, the cost of copying 32 bytes between two different memory buffers on the same processing node is 21 cycles if every load and store is a cache hit and 103 cycles neither the source nor the destination is resident in the cache when the copy is initiated. Under M2 this cost is paid at the receiving node and under M3 it is incurred at the sending node. The following analysis explains how the figures in Table 6.4 were derived for each of the four cases.

- **M1**: the sender is the bottleneck. The gap betwen packets is limited by the send command (10 cycles) and reading packet payload from DRAM (8 cycles). If the sender tests for network-busy status before injecting a packet, there are another 13 cycles needed for the test and branch.

- **M2**: the receiver is the bottleneck. The total cost is the cost of copying plus 10 cycles to increment the queue pointer.

- **M3**: the sender is the bottleneck. In principle, M3 requires the sender to copy data when data from the source buffer are sent to two or more nodes. This case turns out to common to four of the five benchmark programs. In FFT, data from a single buffer are sent to all the conjugate nodes; in Sort, data from the sorted list are distributed to all other nodes. Gauss uses broadcast to distribute the pivot row. In Jacobi the values at the corners of a tile must be sent to both the horizontal and the vertical neighbors. DMM provides the only exception in which each node sends data to exactly one other node; the traffic pattern is a ring. Therefore the numbers for M3 include the cost of copying to reflect the common case, plus another 8 cycles to fetch the packet payload from memory.

- **M4**: on a long message the queue would not be used, so the result is the same as Cranium unmodified.

6.3.3 Empirical evaluation of the modifications

The results of the performance analysis above were confirmed empirically using Gauss as a representative example from the benchmark suite. Modifications M1, M2 and M3 were implemented in the simulator and in the communication routines in Gauss. The cost of communication was measured for each modification. Figure 6.14 plots the cost of communication in Gauss for five different versions: Cranium unmodified, M1, M2, M3 and the combination M1+M2, the latter being the closest approximation of the CM-5 interface. The optimistic memory model was assumed in all cases. Values in the graph are in thousands of clock cycles; lower values indicate better performance. These measured values are also displayed numerically in Table C.1 in Appendix C. As expected, both M3 and the pseudo CM-5 interface (M1+M2) yielded the largest communication costs, approximately a factor of four greater than

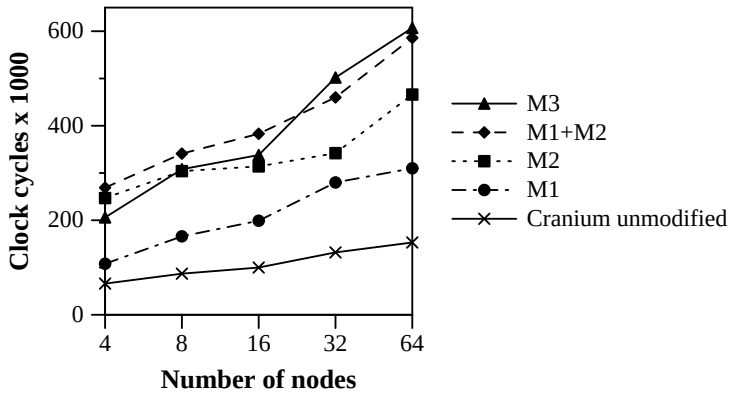


Figure 6.14: Cost of communication in the Gauss benchmark for unmodified Cranium and the modifications M1, M2, M3 and M1+M2. Plotted values represent thousands of clock cycles.

Cranium unmodified. As the number of nodes increases, the cost of communication due to M3 increases the most because the gap between packets at the sender becomes the critical bottleneck under a broadcast. Normally, the send cost and the receive cost can be overlapped; the cost of communication is usually the maximum of the send and receive cost. However if the traffic pattern is a broadcast under the pseudo CM-5 interface (M1+M2), the effects of both costs are cumulative.

Together, Table 6.4 and Figure 6.14 confirm the assertion that modifications M1, M2 and M3 cause a a significant loss of performance both analytically and empirically. Modification M4 has little impact on the maximum throughput of the interface. However there are three important reasons for not eliminating the queuing channels from Cranium.

- *Polling overhead.* Cranium's user queue funnels the arrival of unexpected packets into a centralized location. Polling the queue for an arrival of any packet involves a single access to memory. Without the queue, the receiver must allocate a separate buffer for each potential sender and then test the contents of the buffer to discover when a packet has arrived. If there are p potential source nodes then the cost of a poll is p memory accesses. In Table 6.1 the cost of polling at the receiving node ranges from one eighth to one quarter of the total latency of a single packet message. In a system with 64 nodes, the latency of a

single packet message under an application program that expects a packet from any node increases by a factor of 8 to 16 in the worst case. The queue makes it much easier for the programmer to avoid this kind of worst-case behavior.

- *Scalability.* In many application programs, as the number of nodes increases the number of channels or virtual connections between nodes also increases. The queue is universal in that the only limitation on the number of virtual connections it supports is the size of the buffer memory. It is simple to increase the size of the buffer memory by adding DRAM chips and changing a parameter in the operating system. By contrast, the number of DMA channels in the network interface is hardwired. The network interfaces in SHRIMP-I and SHRIMP-II require a content addressable memory to support the virtual connections between source and destination node. In particular, SHRIMP-II uses a CAM called the Network Interface Page Table. The NIPT is indexed with 15 bits to map 32K distinct remote pages [43]. The number of simultaneous mappings is not specified. The scalability of the size, cost and latency of the NIPT may be a concern in the implementation of the SHRIMP-II network interface.

- *Ease of use.* The queue provides an interface that is familiar to users of programmed I/O interfaces such as the network interface of the CM-5. The queue serializes packet arrivals and provides fairness. Unless queue memory is full, packets may arrive at any time. In many cases it is possible to eliminate barrier synchronization operations that are necessary to support Cranium's automatic-receive channels (see Section 4.2).

6.3.4 Summary

The full Cranium model consists of three high-level features: the send channels, the automatic-receive channels and the queuing channels. The automatic-receive channels implement unbuffered communication; the queuing channels implement buffered communication. All of these features are necessary for providing high performance communication. Attempts to simplify the feature set result in a loss of performance or a loss of flexibility. A set of four simplifying modifications M1 through M4 were proposed and evaluated both analytically and empirically. Modifications M1 and M2 together create an interface that is very similar to the CM-5 interface; modifications

M3 and M4 create an interface very similar to the SHRIMP-I interface. Modifications M1 through M3 result in an increase in the cost of communication ranging between 50% to 300% greater than Cranium unmodified.

The principal problem in the SHRIMP-I interface is that it often requires the processor at the sending node to copy data. The SHRIMP-II interface is a significant improvement over SHRIMP-I because it does not have this limitation; the SHRIMP-II interface appears to be very similar in performance to Cranium for a variety of application programs. SHRIMP-II however lacks the flexibility of Cranium's queuing channels. Also, SHRIMP-II does not work with networks that deliver packets out of order; these adaptive networks have been demonstrated to have higher performance than oblivious networks constructed from the same technology.

6.4 Related work

To date there does not appear to be a definitive study and comparison of network interfaces in the literature. However there are several papers that have been published that address some of the same issues that are presented here. Network interfaces that were previously studied included those in the Thinking Machines CM-5, the Intel Paragon, the MIT J-machine, the Motorola Star-T, the Intel Touchstone Delta and the UW Meerkat.

6.4.1 Study #1: CM-5 vs. Paragon

This study by Kwan, Totty and Reed [94] focused on the measurements gathered from the CM-5 and the Paragon. Through the use of simple throughput and latency benchmarks, the authors demonstrated that the CM-5 achieves a throughput rating of 8 MB/sec out of 20 MB/sec maximum and the Paragon achieves 20 MB/s out of 200 MB/sec maximum. From their analysis they concluded that software overhead limits the peak achievable throughput of both systems. The weakness of this study was that there is little insight into how the structure of the network interface forces the software overhead to be much larger than necessary.

6.4.2 Study #2: CM-5 vs. J-machine vs. Star-T

This study by Spertus et al [59] provided much more insight into how the structure of the network interface affects the overall performance of scalable systems. In particular, the CM-5 and the J-machine were compared. A related study by Papadopoulos, Boughton, Greiner and Beckerle from MIT and Motorola included a comparison of the *T (Star-T) interface [30]. In both studies the abstracted versions of each system were evaluated and compared. The abstracted version of the CM-5 was called CM-5' and likewise the J-machine was abstracted into a system called J'. The CM-5' had two significant improvements over the CM-5: the maximum packet size was increased from five 32-bit words to sixteen 32-bit words, and it could poll both data networks in a single operation rather than requiring a separate poll for each data network. The actual J-machine contains neither a floating-point unit nor a cache; these missing components were added to the J' processor to make it equivalent to the Sparc processor in the CM-5. With these improvements, J' was determined to be more efficient than CM-5' over a variety of computation-intensive benchmark programs. The strength of the study by Spertus et al was that it identified a number of weaknesses in the CM-5'. In particular, sending is inefficient because the processor must always poll the interface before injecting a packet. Polling accounts for 33% of the send cost. The weakness of this study was that it used an unconventional language and run-time system to perform the evaluation. The benchmarks were written in the dataflow language Id90; the run-time system is called TAM (Threaded Abstract Machine) [95]. An artifact of TAM is that it only uses small messages of no more than sixteen 32-bit words. While this study was interesting in the case of small messages, it provided little insight into the study of systems and benchmarks that perform well using large messages. As the paper stated, "A complete quantitative comparison in this regime is very difficult because there are so many variables that can influence performance and there is little consensus on what constitutes a representative workload." Arguably, a representative workload should include programs written in a conventional language such as C or Fortran and contain a wide range of message sizes.

6.4.3 Study #3: Meerkat vs. Delta

This study was performed by Bedichek for his PhD dissertation [35]. Bedichek designed and implemented a four-node hardware prototype of Meerkat and also devel-

oped the software simulator described in the previous chapter of this dissertation. A set of parallel benchmarks were written in C and made use of the Intel NX message-passing library. They were compiled and executed on both Meerkat and the Intel Touchstone Delta. Both a simple set of latency and throughput tests and a suite of computation-intensive benchmarks were measured. Bedichek discovered that Meerkat achieves better speedups than the Delta does on three different computation-intensive benchmarks: FFT, red-black successive-over-relaxation (similar to Jacobi) and SIMPLE, a hydrodynamics benchmark. The strength of this study was that unlike the Spertus and Papadopoulos studies, it used a conventional language and run-time system for benchmarking. The weakness was that Meerkat differs from the Delta in almost every facet: the processor, network and network interface are all different, making it difficult to pinpoint where the advantage lies. In the computation-intensive benchmarks there is no indication of what fraction of execution time is spent in communication.

The Cranium test environment and the Meerkat project are based on the same node architecture (processor, cache and memory). Despite the similarity, the two systems nevertheless differ in many ways. Meerkat uses a grid of busses for its network, whereas Cranium uses a point-to-point torus network with chaotic routing. The network interfaces are different in four significant ways. First, the Meerkat logical interface is based directly on the Intel NX model, unlike Cranium which has its own custom set of primitives. Second, the processor at the receiving node under Meerkat must flush the cache explicitly for each message. The cache flush operation itself represents about 30% of the cost of communication when using messages with sizes between 128 bytes and 1K bytes [35]. Under Cranium, cache and memory are kept locally coherent. Third, the processors in Meerkat stall during communication. This feature precludes the possibility of overlapping communication and computation. Fourth, Meerkat's interface mandates interrupting the processor at the receiving node for every packet; there is no automatic-receive DMA as in Cranium.

6.5 Summary

Our analysis showed that Cranium delivers both low latency and high throughput on both point-to-point messages and broadcast messages. In Section 6.1 we explained that the end-to-end latency of a one-way, single-packet message is approximately

60 to 100 clock cycles. Cranium achieves 90% of the maximum possible sustained throughput with messages as short as 2048 bytes, and 96% of this maximum with 8K byte messages. The performance of broadcast is within a factor of two of the bound for any tree-based broadcast algorithm. The performance of broadcast can be improved substantially by using a network that supports broadcast directly.

We created a test suite for Cranium consisting of five parallel benchmark programs, described in Section 6.2. All of the benchmarks are written using hand-crafted message-passing code. These programs are executed on the combined Talisman/Chaos simulator described in Chapter 5. We observed that these benchmarks yield excellent speedups as well as high communication performance, even though the input data set sizes were very small. The selected parallel benchmarks demonstrate that Cranium is capable delivering more than 50% of the peak throughput. The dense matrix multiply benchmark achieves more than 70% of the peak throughput with up to 32 nodes.

In Section 6.3, we compared the logical interface (programmer's model) of Cranium against that of other network interfaces. Evaluation is performed by modifying Cranium to more closely model the abstractions of these other network interface architectures. Three different modifications to Cranium independently increase the communication cost to be 50% to 300% greater than Cranium unmodified. A fourth modification does not affect throughput directly, but it increases the latency of small messages, reduces flexibility and increases the difficulty of writing message passing programs. In Section 6.4, we examined the literature to research other methodologies used in comparison studies of network interfaces. The lack of other comparison studies demonstrates the opportunity for further work in this area, and that we have introduced a new approach for these kinds of comparisons.

The performance analysis and empirical studies presented in this chapter confirm that the Cranium architecture provides an efficient, powerful and flexible network interface. The comparison with other network interface styles demonstrates that all of the features in Cranium are necessary to deliver the full performance of the communication system. The following chapter presents the Teschio implementation of Cranium which was used to provide the timing information assumed in the analysis and the simulation studies presented here.

Chapter 7

TESCHIO: A VLSI CHIP IMPLEMENTATION OF Cranium

Ah, to build, to build! That is the noblest of all the arts.
– *Longfellow*

Teschio is a paper design of a chip based on the Cranium network interface architecture that was introduced in Section 3.3. It is configured as a single ASIC that can be fabricated using standard CMOS VLSI technology. Teschio combines these features:

- It connects directly to the multicomputer network. The network used in this implementation is based on the Chaos router, described in Chapter 5.2. The Chaos network uses a two dimensional torus mesh topology.

- It connects directly to the processor-memory bus in the computing nodes of the multicomputer. The processor-memory bus used in this implementation is a split-transaction bus similar to the one designed for the Alpha Demonstration Unit [64]. Connecting at the processor-memory bus yields higher performance than connecting through an I/O bus such as PCI (see Section 2.1.1).

- It requires implementation technology that is well within the limits of today's CMOS fab lines. We estimate that the circuit can be implemented using fewer than 400,000 gates. The number of external signals is approximately 150, which can be supported easily using ball grid array (BGA) packaging technology. The projected clock rate is 100 MHz (see Sections 7.1.2, 7.1.3 and 7.5).

Figure 7.1 shows a simple block diagram of Teschio comprising four primary structural modules: the bus interface, the core, the inbox and the outbox. The *bus interface* allows two types of access: slave access by the host processor and bus master (DMA)

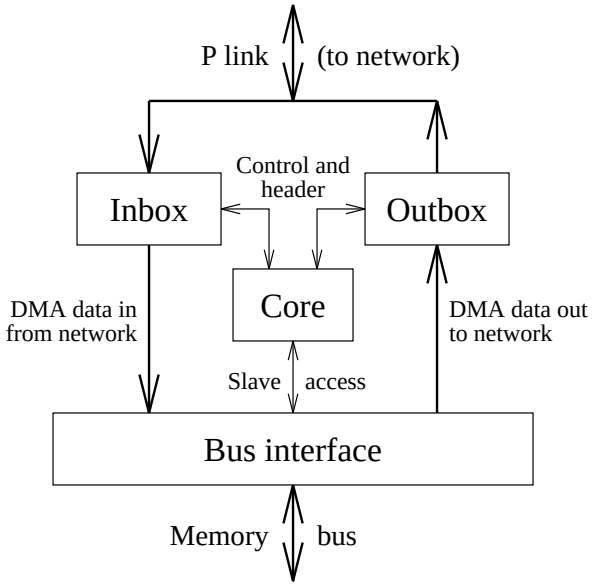


Figure 7.1: Simple block diagram of Teschio

access to the DRAM-based memory modules. The *core module* contains all the channel information visible to the application program. It schedules packets to be sent, constructs and validates packet headers and sets up DMA addresses for the bus interface. The *inbox* and *outbox* are complementary modules that connect directly to the processor-network link (P link). The inbox contains FIFO[1] buffering for packets delivered from the network waiting to be processed; the outbox likewise contains FIFO buffering for packets waiting to be injected into the network. More detail on the internal organization of Teschio is shown in the block diagram in Figure 7.5 located in Section 7.2.

The rest of this chapter is organized as follows. Section 7.1 describes the environment external to Teschio. The internal structure of Teschio introduced above is described in greater detail in Section 7.2. We describe the interactions among Teschio's internal modules in Section 7.3. We discuss the timing of Teschio in Section 7.4. We estimate the area and pin requirements of a single-chip implementation of Teschio

[1] While the terms FIFO and queue are often used interchangeably, there is a subtle but important distinction in the use of these terms in this chapter. We refer to a FIFO as a dedicated hardware structure, and a queue as a general concept that could be realized in hardware or as a combination of hardware and software (e.g. the queue channels).

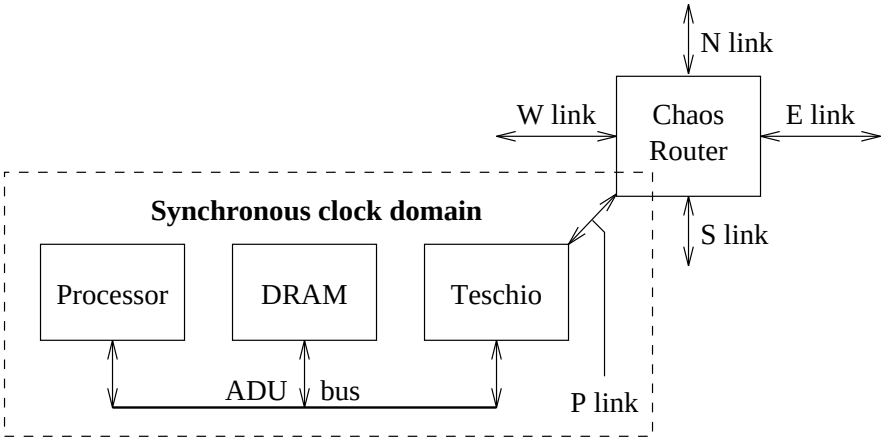


Figure 7.2: One node of a multicomputer system based on Teschio

in Section 7.5. Extensions to Teschio are explained in Section 7.6. We summarize our findings in Section 7.7.

7.1 Teschio system environment

This section describes the external environment that surrounds Teschio. We discuss both the environment of a single node and the environment of the whole system.

7.1.1 Environment of a single node

Figure 7.2 describes the organization of a single node of the multicomputer. Each node contains a processor, a DRAM-based memory module and the Teschio interface chip connected together via the processor-memory bus. The other side of Teschio connects to the router chip via the processor link (P link). The entire node is a synchronous clock domain encapsulating the processor, memory, Teschio chip and P link. Because it may be difficult to construct a large multicomputer that is globally synchronous, we allow each node to be placed into its own separate synchronous clock domain. The whole multicomputer system is a globally asynchronous collection of these synchronous domains. The linkage between the clock domains is implemented in the router chip using an asynchronous design methodology such as self-tuning [96] or self-timing [97].

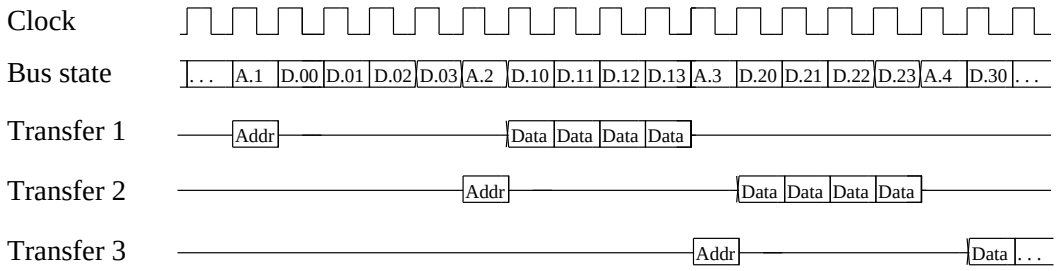


Figure 7.3: ADU bus timing

7.1.2 The ADU bus

The processor-memory bus is a representative design based on the Alpha Demonstration Unit [64], known in this dissertation as the *ADU bus*. The ADU bus is synchronous and it multiplexes address and data onto the same 64 wires. In principle the address bus is 64 bits wide, but for practical reasons only the lower 48 lines are used, to yield an addressable memory space of 2^{48} bytes (64 terabytes). All transactions on the ADU bus are the size of a cache line (32 bytes). Since the bus is eight bytes wide, it takes four cycles to transfer a cache line. A complete transfer requires five cycles on the bus: one for address and four for data. The lowest 5 bits of the 48 bit address field are not needed because data are always transferred in 32 byte blocks that are aligned on cache-line boundaries.

The ADU bus uses a split-transaction protocol to overlap the DRAM access penalty of one bank of memory with the time to transfer a cache line into or out from a second bank of memory. Figure 7.3 is a timing diagram that describes the basic idea. The three traces labeled Transfer 1, Transfer 2 and Transfer 3 indicate three overlapping memory transactions. The ADU bus itself reflects the superposition of these transactions over the same wires, but it is broken out this way in the diagram for clarity. The trace labeled Bus State is used to infer the type of information that is placed on the bus. Bus masters place address information on the ADU bus only during bus states labeled with an A (e.g. A.1). The data fields corresponding to the address field at state A.n occupy states D.n0 through D.n3. This organization allows the DRAM to delay its response by five bus states. Memory is interleaved into two or more banks, and each bank is itself four-way interleaved to allow four 64-bit words to be transferred on four consecutive cycles.

The ADU bus allows the processor to take advantage of cache coherence in hardware. The ADU bus protocol uses five states: each cache line can be marked as exclusive-clean, exclusive-dirty, shared-clean, shared-dirty or invalid. Cache coherency issues are not discussed further in this chapter. The interested reader may refer to Sections 2.1.2 and 3.4.2 in this dissertation and the article by Thacker, Conroy and Stewart [64] for more detail.

Our implementation of the ADU bus is clocked at 100 MHz and the cycle time is 10 nsec. This clock rate was chosen to match the access delay of standard DRAM technology, which is roughly 50 nsec. DRAM latencies are not improving at the same rate that processor clock speeds are increasing. To increase the throughput of split-transaction busses, it becomes necessary to increase the depth of pipelining from two to three or more overlapping memory accesses as processor clock rates continue to outpace the DRAM access delay.

7.1.3 The P link

The P link is the interface between Teschio and the network router chip. Like the ADU bus, the P link is synchronous and runs at 100 MHz. The data path is 32 bits wide, half the width of the ADU bus. The data path in the P link is bi-directional and half-duplex. A packet is injected into the network or delivered from the network via the same wires, so that both cannot happen simultaneously. This strategy contrasts with full-duplex communication in which two uni-directional data busses are used instead of a single bi-directional data bus. Full-duplex allows simultaneous packet injection and delivery. The advantage of half-duplex over full-duplex is that the average throughput is higher. For a fair comparison between the two, it is assumed that the total number of pins in the data path is the same for both full-duplex and half-duplex. At high levels of utilization, the throughput is the same because all wires are continuously in use. At low levels of utilization, there are occasions in which only one packet needs to move in or out at a time. With a full-duplex link only half the number of wires are used; the throughput is half that of the half-duplex solution.

A detailed description of the handshaking signals used in the P link are described in Section D.1 in Appendix D.

7.1.4 Node mapping

Each processing node in a multicomputer must have a unique, global identifier to make the system programmable. The identity of a node is usually provided in hardware in the form of a binary signature, usually coming from a field programmable logic device. An FPLD simplifies the manufacturing step: if the signature were hard-wired into the printed circuit board, then a unique circuit board layout would be needed for every processing node. One standard for these binary signatures is called the Universal Logical Address, a 48-bit code defined by IEEE specification 802.3 [98]. While this code guarantees uniqueness it may contain no information about the topology of the network.

Application programs do not directly use the naming scheme described above. A common naming scheme for remote nodes for user programs is one that starts with 0 and goes up to $p-1$ where p is the total number of nodes involved in the computation. The naming scheme known to the application program is known as the *logical* node map. Mapping the node names in this way allows two copies of the same program to run simultaneously in different logical partitions without colliding. It also permits the system to avoid allocating faulty processing nodes to the application program.

There is a third name space for processing nodes – the naming scheme used directly by the network router hardware. In many network routers, including the Chaos network router, nodes are addressed by their relative position, rather than using absolute addressing. We call this node naming scheme the *physical* node mapping. Teschio performs the translation from logical node names used by the application program to physical node names used by the routers, by means of a lookup table. This mapping scheme implements the node protection requirement described in Section 2.1.4 and Section 3.3.3. See Section D.2 in Appendix D for an example of node mapping in Teschio specific to the Chaos network.

7.1.5 Data redundancy

Data integrity is a concern in all multicomputer designs: it is the ability to detect and correct errors due to noise on the busses and network links. Teschio implements redundant information in two ways. First, all network links (including the P link) use one bit of parity per byte. Second, it is particularly important to include redundant information for the packet header; the sixth word of the packet header is

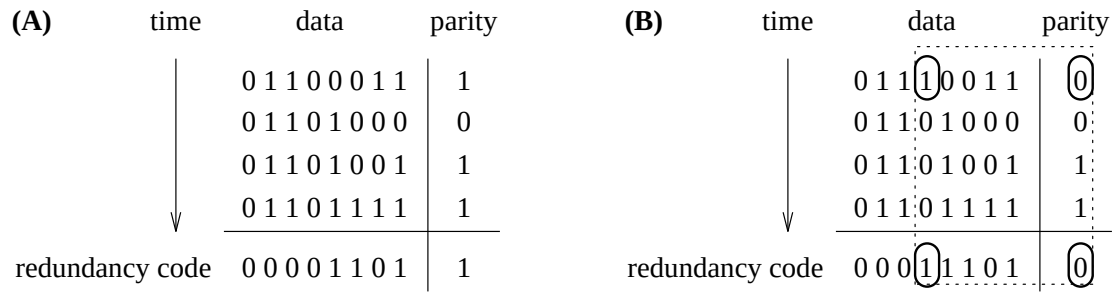


Figure 7.4: Using two-dimensional parity as a substitute for the Cyclic Redundancy Code. Odd parity is used horizontally and even parity is used vertically. Subfigure *a* shows the header packet as it was originally sent. Subfigure *b* shows an example where the packet header contains four single-bit errors. This is the minimum number of errors needed to bypass the parity check at the receiver. Such a situation is extremely unlikely to happen, because the erroneous bits must also form a rectangle.

reserved expressly for this purpose. The solution provided in Teschio is bit-serial (vertical) parity for the packet header. (The first 16-bit field of the header is always 0 under normal circumstances when the packet is ejected from the network.) The effect is that parity bits protect the header in both the usual (horizontal) direction as well as the orthogonal (vertical) direction (Figure 7.4a). For additional protection, complementary forms of parity are used: odd parity is used horizontally and even parity is used vertically. The reason is to detect erroneous packets that consist of all ones or all zeros, a situation that might arise early in the debugging stage. An odd parity error is detected for the all-zero packet and an even parity error is detected for a packet consisting entirely of ones. Two-dimensional parity is a vast improvement upon parity in only one dimension. In one-dimensional parity, two single-bit errors in the same byte will cancel each other out, potentially allowing the passage of incorrect data. With two-dimensional parity, it takes a minimum of four single-bit errors to pass undetected. Furthermore the erroneous bits must form the corners of a rectangle to be undetectable (Figure 7.4b).

An alternative to Teschio's bit-serial parity is the standard cyclic redundancy code (CRC) [99, 100]. A 16-bit CRC code can be encoded serially using 16 bit shifts per word using very simple hardware, but this implementation is very slow. Implementations of a parallel CRC encoder and decoder tend to be complicated

[100]. By contrast, the vertical parity circuitry is trivial to implement as it requires only an XOR gate per bit.

7.2 Teschio internal structure

Figure 7.5 is a block diagram of Teschio that describes its structure at an additional level of detail beyond that shown in Figure 7.1. The top part of the diagram (bus interface and core) describe the control structures; the bottom part of the diagram (inbox and outbox) shows the flow of data from left to right. The P link appears at both the lower left and right sides of the diagram. At the lower left it is a source of information for packets arriving from the network, and at the lower right it is a sink for packets that are injected into the network. Likewise, the symbol for the ADU bus appears twice – once at the top of the diagram to represent slave access by the host processor, and at the bottom to represent DMA access for packet payloads.

Finite state control in Teschio is implemented in eleven communicating finite state machines (FSMs). Three are depicted explicitly in Figure 7.5: the core FSM, the bus interface FSM and the packet interface FSM. Each of the five FIFO submodules contains its own FSM. The other three are not shown: the inbox FSM, the outbox FSM and the P link FSM. The collective purpose of the first three FSMs and the scheduler FIFO is to allocate the ADU bus among its tasks, in order of increasing priority: reading packet payload data from DRAM and writing the data to the outbox FIFOs, reading packet payload data from the inbox FIFOs and writing the data to DRAM, and supporting slave mode access for transfer of command and status information to the host processor. The inbox FIFO coordinates the activities of the inbox header FIFO and the inbox payload FIFO; it splits the packet into header and payload. The function of the outbox FIFO is similar; it joins the header and payload portions of the packet into a whole packet. The P link FSM handles the arbitration and handshake of the P link between Teschio and the network router. The purpose of the FIFOs in the inbox and outbox modules is to decouple the ADU bus from the P link. This decoupling helps manage the complexity of the design. The P link FSM can be developed without knowledge of the processor bus timing and architecture; the core and bus interface can be developed without dependence on the timing and architecture of the P link.

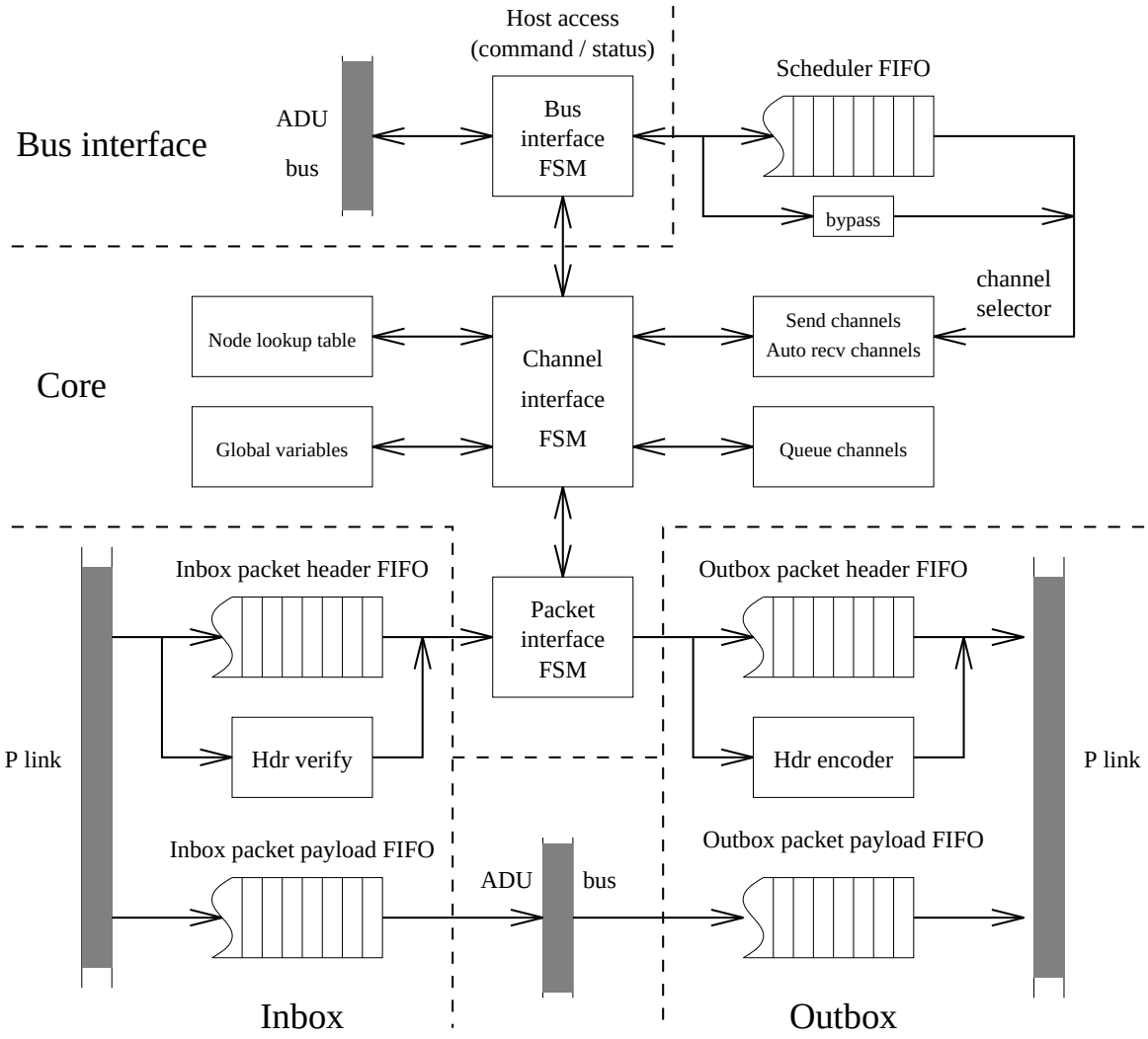


Figure 7.5: Structural diagram of Teschio

7.2.1 Core module

The core module is the most complex module in Teschio. It consists of seven submodules as shown in Figure 7.5: the core finite state machine (CFSM), the scheduler FIFO, the node lookup table, a set of persistent global variables, the send and automatic-receive channel array, the queue channel array and the packet interface FSM (PIFSM). All of these submodules are constructed using static RAM. For the CFSM, the RAM is used as its writable control store. This capability is needed only for debugging the first prototype run. Production runs could be built using mask-programmed ROM.

The scheduler FIFO contains the active list of pending channel commands. When the core FSM is in a state where it is ready to execute a command (such as sending a packet) it uses the value at the head of the scheduler FIFO to select the channel context for the command. If the scheduler is empty and Teschio is not in the process of actively completing a command, the core FSM assumes an idle state. New commands arrive into the scheduler FIFO from the host processor. During the handshake with the bus interface, the scheduler is bypassed so that state information is deposited directly into the channel array. Upon confirmation that the command is valid, the channel number is entered into the scheduler. Typically, three subsequent actions occur:

- The address handle is translated into a physical address by looking up the value in DRAM.

- The node handle is translated into a physical node address by looking up the value in the node lookup table.

- If the head of the scheduler FIFO contains an auto-receive channel and both translations have completed, the channel number is popped. (Auto-receive channels are scheduled automatically when a packet arrives from the network.) Otherwise, if the head of the scheduler FIFO contains a send channel, and all packets for that channel have been sent, then the channel number is popped.

The node lookup table is a 256-element SRAM indexed by logical node number whose contents contain physical node numbers. The on-chip lookup-table eliminates

Table 7.1: Description of fields in one channel of the channel array

Field name	Width (bits)	Field name	Width (bits)
Node handle	12	Physical node ID	16
Address handle	15	Physical address	38
Number of packets	12	Sequence number	12
Remote channel	8	Flags	6

Table 7.2: Description of fields in one channel of the queue channel array

Field name	Width (bits)	Field name	Width (bits)
Queue buffer start	42	Head pointer	42
Queue buffer end	42	Tail pointer	42

the need to go to off-chip DRAM to perform the translation for logical nodes 0 to 255. It is worth noting that the majority of physically-realized scalable multicomputers contain 256 or fewer nodes. Also, the simulator described in Chapter 5 simulates a maximum of 256 nodes.

The set of global variables includes a number of different fields. Many of these fields are privileged to the operating system and are therefore unavailable to application programs directly. One example is the field that describes the MMU page size currently in use by the processor. A second example is the application program's process identifier. A third example is the FREEZE mask and status information (see Section A.3 of Appendix A). The FREEZE mask information tells Teschio to allow or deny packet injection and/or delivery. This capability is useful during special circumstances such as switching the user context or assisting the network in performing an error recovery operation. Application programs can read and write certain fields directly, such as the interrupt mask and status fields that govern the situations in which the send and automatic-receive channels generate interrupts.

The channel array and the queue channel array embody the primary data structures in Teschio that reflect its high-level programming model. Table 7.1 is a description of the fields in the channel array and Table 7.2 describes the fields in the

queue channel array. The channel array consists of an equal number of send channels and automatic-receive channels. The channel number field in the packet header addresses up to 256 send channels and 256 auto-receive channels in the channel array. In this implementation there are 32 send channels and 32 auto-receive channels. The queue channel array contains four queue channel entries to represent the user queue, the user error queue, the system queue and the system error queue. For each queue channel, the fields Queue Buffer Start and Queue Buffer End are static pointers into DRAM that delimit the start and end of queue memory; they are changed only during initialization. The fields Head Pointer and Tail Pointer are updated continually during normal operation. Teschio inserts packet information into the tail and the host processor removes packet information from the head. Teschio updates the hardware head and tail pointers as a consequence of insertion and removal of packet data.

The width of the pointer fields in the queue channels is 42 bits, whereas the width of the physical address field in the send and auto channels is 38 bits. The reason for 42 bits in the former case is that two cache lines are stored for every queue entry; two caches lines = 64 bytes requiring $\log(64) = 6$ bits of address; $6 + 42 = 48$. The reason for 38 bits in the latter case is that it only needs to address MMU page boundaries. The minimum page size supported is 1K (10 bits); i.e. $38 + 10 = 48$. The largest page size is 128K (17 bits), for which only its topmost 31 bits physical address field are used (i.e. $31 + 17 = 48$).

7.2.2 Inbox and outbox

The outbox and the inbox are mirror images of each other, both in function and in form. The outbox provides four services. It accepts packet payload information from the bus interface. It accepts packet header information from the core module. It appends a redundancy code to the end of the packet header. It merges the header and the payload, and ships the whole packet out to the network. The inbox performs the inverse operations. It accepts a full packet from the network and splits the packet into header and payload fields. It verifies the header using its redundancy code. It presents the packet header to the core module for post-processing. It ships the packet payload to the bus interface.

The inbox and the outbox decouple the ADU bus side of Teschio (the core and bus interface) from the P link side. While Teschio is a synchronous circuit overall, there

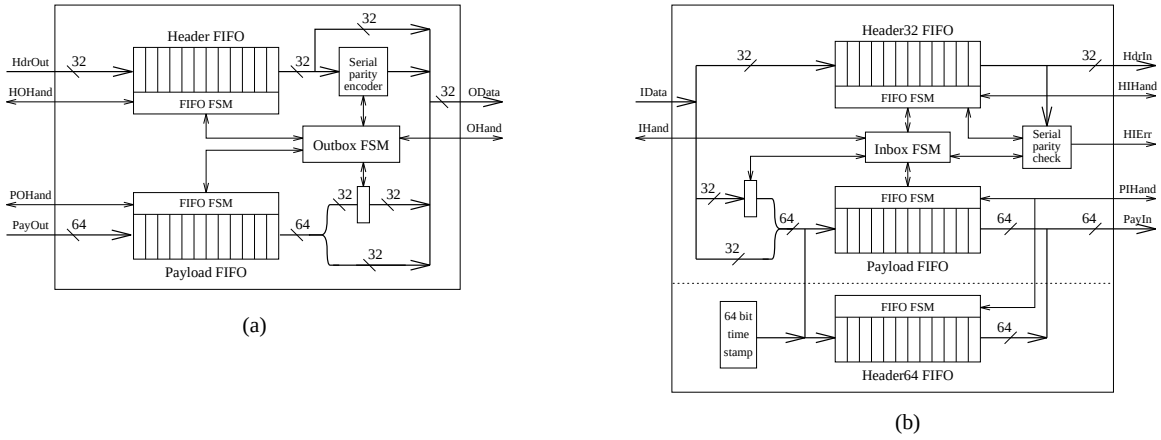


Figure 7.6: Block diagrams of the outbox and the inbox. Subfigure *a* illustrates the outbox; subfigure *b* is a diagram of the inbox.

exists a significant "impedance mismatch" between the ADU bus and P link. The ADU bus is 64 bits wide and uses a split-transaction protocol; its maximum effective data rate is 640 MB/sec. The P link is 32 bits wide and uses a single-transaction protocol; its maximum effective data rate is 291 MB/sec. The ADU bus has higher bandwidth than the P link because the host processor uses a significant fraction of its bandwidth to transfer cache lines to and from DRAM. The difference in bus width and timing makes it necessary to hold packets temporarily in the interface, in the FIFO queues of the inbox and the outbox. By making the pipelining as efficient as possible, it is possible for a packet to encounter only a small number of cycles of delay when the interface is lightly loaded. At heavy loads, Teschio delivers the full bandwidth of both the ADU bus and the P link. Thus, the inbox and the outbox provide a smooth impedance match between the ADU bus and the P link.

Figure 7.6 displays block diagrams of both the outbox in Figure 7.6a and the inbox in Figure 7.6b. Each module contains a pair of FIFO queues, one for storing header information and the other for storing payload information. Separate FIFOs are used for header and for payload for two reasons. The first reason is that data going to or coming from the ADU bus are presented 64 bits wide; it makes sense for the payload FIFOs to be the same width as the processor bus. Packet header information is more conveniently presented 32 bits at a time, the same as the width of the P link. A second reason is that it is convenient to be able to construct or verify the header concurrently with the flow of payload data.

Figure 7.6 shows both the data path and the finite state control used in the outbox and the inbox. To pass data from the outbox payload FIFO to the P link, the 64-bit wide data path is converted to a 32-bit wide data path by using a 32-bit wide register. The wide FIFO is popped every other cycle; the 32-bit path takes the lower or upper bits on alternate cycles. The packet header is passed through the serial encoder; its output is also merged into the 32-bit wide stream destined for the P link. The outbox FSM (OFSM) coordinates the activities of both FIFOs and performs handshaking with the P link FSM (PLFSM). Similarly, the payload FIFO in the inbox composes 64-bit wide words from a pair of 32-bit wide words on alternating cycles. The serial parity decoder verifies the packet header; an error signal (`HIErr`) is passed to the core module.

The inbox must save a copy of the packet header that is stored in memory if and only if the packet is destined for a queue channel. In the queue channels, the packet payload is stored into one cache line, and the packet header, a 64-bit timestamp and a presence flag are stored into the subsequent cache line. If the packet is stored into an auto-channel, the header information is not stored into memory. Since the core module performs the header decode, the inbox must assume that the packet is for a queue channel; the header is discarded if it turns out not to be needed. Figure 7.6b shows two logically distinct FIFOs for holding 64-bit data: the payload FIFO and a second header FIFO called Header64. The handshaking signals between the core and the Header64 FIFO (collectively known as `PIHand`) include the signal (`PIAuto`) from the core module; if this signal is true, the packet header in the Header64 FIFO is dropped. While it is conceptually useful to make the payload and the Header64 FIFOs logically distinct, it is possible to combine them into a single larger FIFO in the implementation.

Handshaking signals are used at all the interfaces between the inbox and the outbox and environment that surrounds them (Figure 7.7). A separate finite state machine called the P link FSM (PLFSM) coordinates the activities of the inbox, outbox and the P link. The PLFSM implements the structure and behavior described in Figure 7.1.3, including the tiebreaker information for negotiating the data direction of the P link. The handshaking signals used internally follow the same name and function as the signals used at the P link itself. For instance, the P link uses handshake signals RTS, CTS, RTR and CTR. The interface between the outbox and the PLFSM

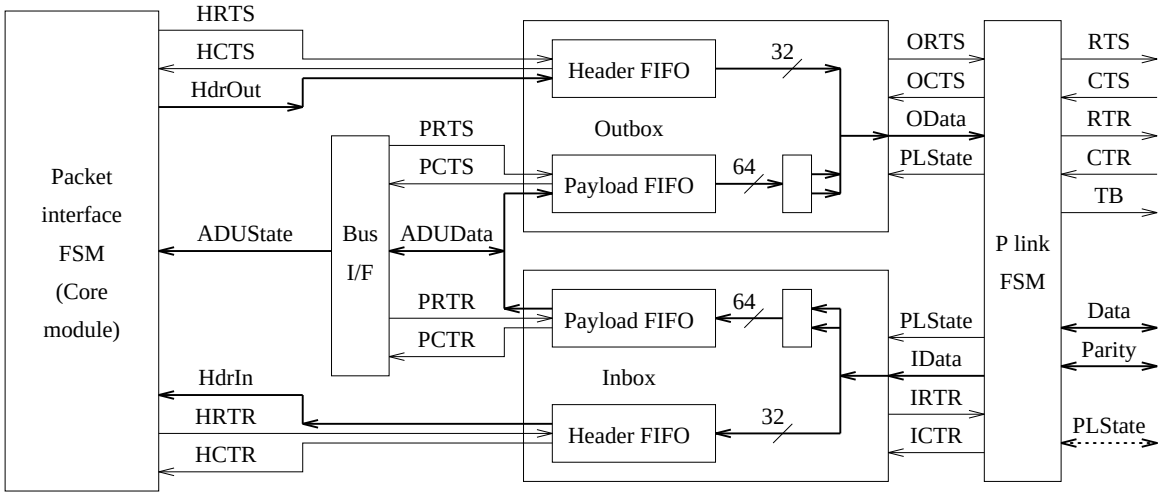


Figure 7.7: Handshaking between inbox, outbox and surrounding environment. All the internal handshaking signals are similar in name and function to the handshaking signals used at the P link (external interface between Teschio and the network router chip).

uses handshake signals ORTS and OCTS; the interface between the inbox and the PLFSM uses IRTR and ICTR. For consistency the interfaces between the core and the outbox, the bus interface and the outbox, the inbox and the core, and the inbox and the bus interface all bear the same naming convention and format.

7.3 Internal micro-operations of Teschio

So far in this chapter we have described the external environment to Teschio and provided an introduction to its internal block structure. What follows now are detailed explanations of the internal micro-operations of Teschio, the interactions between pairs of modules that describe how Teschio operates internally. These explanations use diagrams to explain the behavior and do not precisely describe the structure. Nonetheless the correspondence between behavior and structure is straightforward; these diagrams could be easily converted into their structural counterparts.

There are three uses for a given ADU bus cycle that involves Teschio, in order of decreasing priority:

- Command and status interface with the host processor
- Writing the payload into DRAM of a packet received from the network
- Reading the payload from DRAM of a packet ready for injection into the network

One and only one of these actions can occur on any given bus cycle. If more than one of these actions is ready to happen, the highest priority action takes place. This mutual exclusion greatly simplifies the design, because the state machines for the three activities can be single-threaded. However, the pipelined nature of the ADU bus means that there may be several bus cycles in progress at the same. We require Teschio to make maximal use of the ADU bus whenever possible. Teschio is capable of any combination of activities on consecutive ADU bus cycles: injecting two packets into the outbox, storing two packets from the inbox into memory, and injecting a packet into the outbox and storing a packet from the inbox, in either order.

We now describe the various micro-operations: command and status requests from the host processor, performing table lookup functions, composing a packet header for sending, decoding a packet header for receiving, placing a packet into a queue channel, placing a packet into an automatic-receive channel, and updating data structures as a consequence of sending or receiving a packet.

7.3.1 Command and status interface with the host processor

Figure 7.8 is a dataflow diagram describing the acceptance of a channel command from the host processor. The bus interface continually decodes the ADU bus to detect slave accesses from the host processor. When the physical address matching Teschio appears on the bus, the signal `SelectNI` becomes active (see in Figure D.3 in Section D.3 in Appendix D). The corresponding command word is then latched by the bus interface module and presented to the core module. The core module indexes into the channel array using the channel number in the field `ChannelID` directly, by bypassing the scheduler. Since the ADU bus is busy transferring the command word from the host processor, the bus interface is momentarily prevented from accessing DRAM to load or store a packet payload. This means that the core module can

be dedicated to the processing of the command word, and there is no other module contending for the channel array.

Acceptance of the incoming command depends on the state of the Reset bit from the command word and the Busy bit from the channel structure. There are three cases:

- If Reset is true, then the channel becomes inactive regardless of the previous state of the channel.

- If Reset is false and Busy is true, then there is a previous command loaded into the channel that has not yet completed. In this case the new command is ignored, and Teschio generates an interrupt to the host processor.

- If both Reset and Busy are false, then the command is accepted.

When a new command is accepted from the host, two structures are updated immediately: the scheduler and the selected channel structure from the channel array (Figure 7.8). The channel ID field is placed into the scheduler. Most of the fields in the channel structure are initialized directly by copying the information from the command word into these fields. The exceptions are the reset logic as noted above, and the physical node ID field and the physical buffer address field. The latter two fields are filled in later as the result of the table lookup operations (Section 7.3.2). However, an optimization is possible. If the new value of the buffer handle matches the previous value of the buffer handle, and the previous value of the physical buffer address is a valid address, then the lookup operation will load the identical value of the buffer address. In this case, the lookup operation is redundant; the optimization eliminates the lookup in this case to save the cost of an ADU bus transaction. The same optimization applies to the node handle and physical node ID lookup.

7.3.2 Performing the table lookup functions

Figure 7.9 is a dataflow diagram that describes the table lookup functions. The channel ID at the head of the scheduler FIFO indexes into the channel array. Two fields are examined: the physical node ID and the physical buffer address. If either one of these fields is not valid, then its corresponding table lookup function is performed.

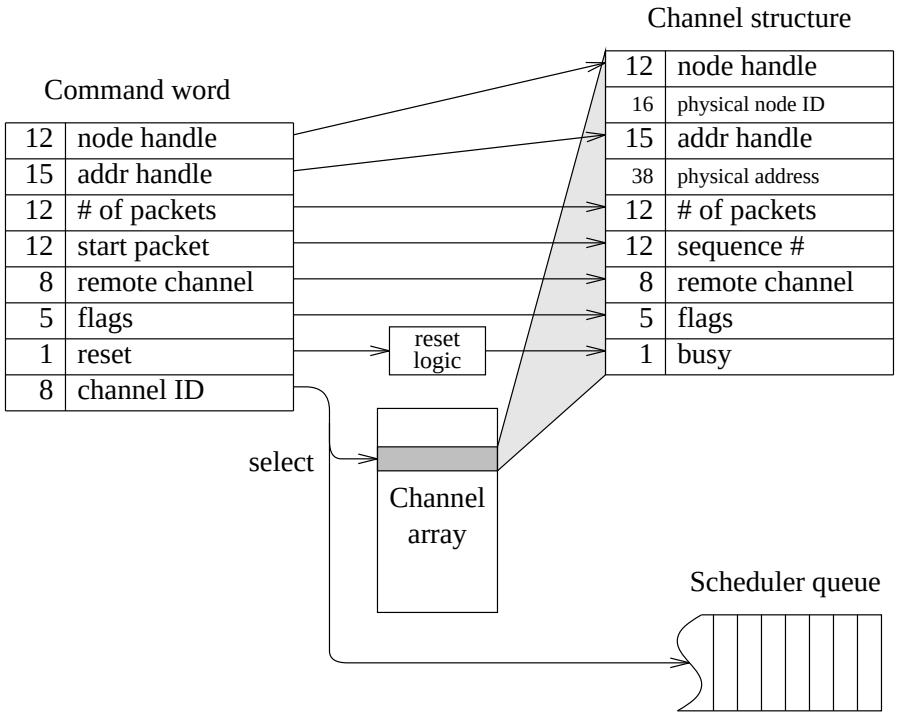


Figure 7.8: Behavioral description of the handler in the core module for accepting a command word from the host processor. A channel structure is selected using the channel ID field of the command word. Most of the fields in this channel structure are initialized directly by copying from corresponding fields in the command word. The channel ID field is also entered into the scheduler FIFO.

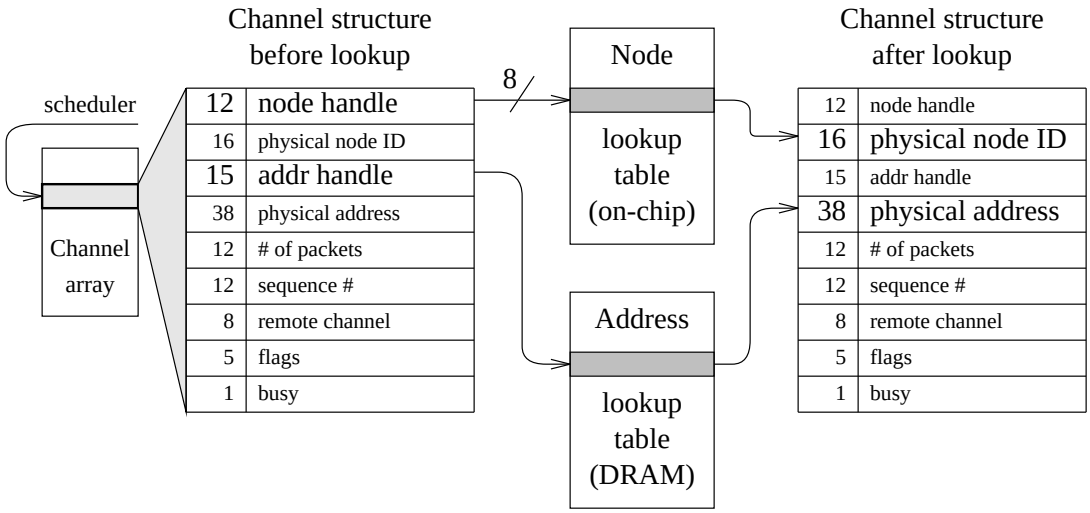


Figure 7.9: Behavioral description of the table lookup functions. The node lookup table is in SRAM within the Teschio chip. The buffer address lookup table is located in DRAM.

The on-chip node lookup table contains 256 entries; in the common case, the node ID lookup stays on-chip. The address lookup table is in DRAM and requires an ADU bus transaction to access it. In the case that both lookup functions are required, it is very likely that both are performed in parallel. At the conclusion of the lookup operation, the channel ID at the head of the scheduler is re-examined. If the channel is a send channel, it remains in the scheduler; if the channel is a receive channel then it is popped from the queue.

7.3.3 Sending a packet

Figure 7.10 describes the construction of packet headers using the information in the send channel structure, selected by the channel number field at the head of the scheduler FIFO. The first field of the packet header, Dest Node ID, comes directly from the physical node ID field of the channel structure and consists of the two eight-bit fields [Y,X] (see Sections 7.1.1 and D.2). The second field of the packet header, Source Node ID, contains the two's complement of both eight-bit fields in Dest Node ID [−Y,−X]. The third field, Process ID, is copied from its corresponding global variable. The fourth field, Dest Channel, comes from the remote channel field of the send channel. The fifth field, Source Channel, is a copy of the channel number at

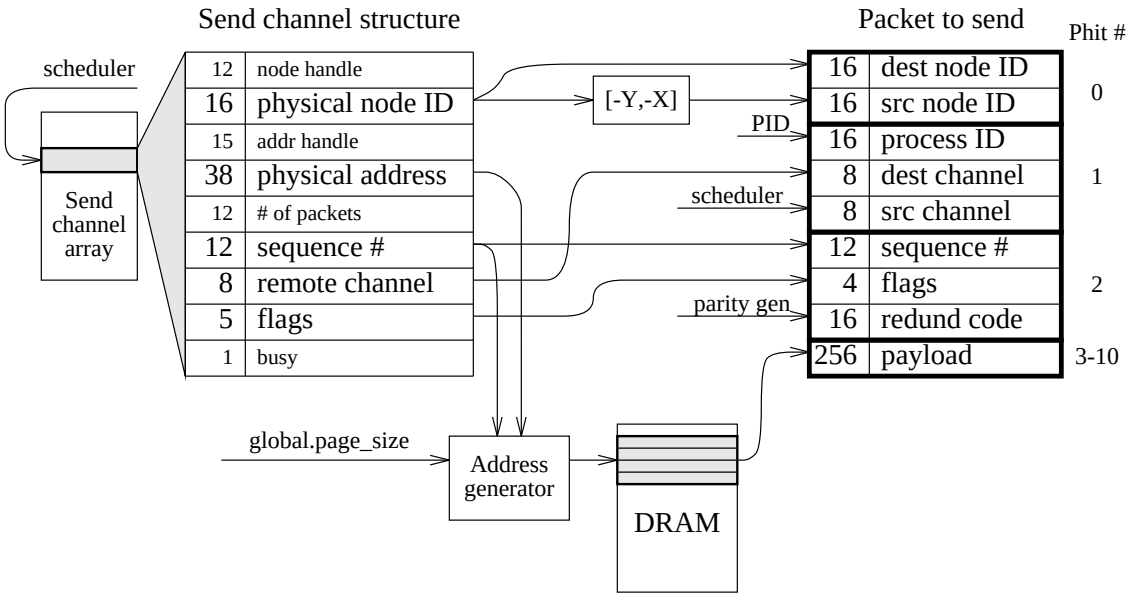


Figure 7.10: Behavioral description of the core module handler for injecting a packet into the outbox. The DMA address for reading the packet payload from DRAM comes from the MMU page in the physical address register plus the offset, the sequence number multiplied by the size of a cache line (32 bytes). Many different page sizes are supported, consisting of all powers of two between 1K to 128K bytes, inclusive.

the head of the scheduler FIFO. The sixth field, Sequence Number, is copied from the Sequence Number field in the send channel. The seventh field, Flags, is a copy of the flags field in the send channel. The eighth and final field of the packet header, Redundancy Code, is constructed by the outbox. The outbox also merges the packet header with the packet payload coming from DRAM.

The address of the packet payload is computed from the Physical Address field and the Sequence Number field in the channel structure. The address generator takes into account the page size to determine how many bits of the Sequence Number field to use. The lowest five bits of the address are always 0. If the page size is p then the number of bits from the Sequence Number field is $p - 5$. The number of bits contributed from the Physical Address field is $48 - \log p$. The size of the physical address used by the bus interface module is 48 bits.

7.3.4 Receiving a packet

Receiving a packet is handled as two operations. The first operation is to decode and validate the packet header. Once the packet header is decoded, it is then handled using either an automatic receive channel or a queue channel.

Decoding and validation of the header are outlined in Figure 7.11. There are four bit flags that are computed in the process: the system flag, the system error flag, the user flag and the user error flag. Some of these flags can be triggered in two or more ways. For instance, there are four different ways to cause the user error flag to be set. All the individual triggers for a flag variable are combined through an OR gate. To simplify the diagram the other OR gates are not drawn. If any of the four bit flags are set then the packet is sent to a queue channel; otherwise the packet is sent to an automatic-receive channel.

The state machine for decoding and validation uses the following sequence. When phit 0 is received into the core, the Destination Node ID field is compared with zero; if it is non-zero, the system error flag is activated. The Source Node ID field is saved for a later comparison. When phit 1 arrives, the third, fourth and fifth fields are presented as operands. The Process ID field in the packet header is compared with the Process ID field in the core module's global variable structure; if they contain different values, the system flag becomes active. The Destination Channel field selects the automatic-receive channel from the channel array. (This path was omitted from

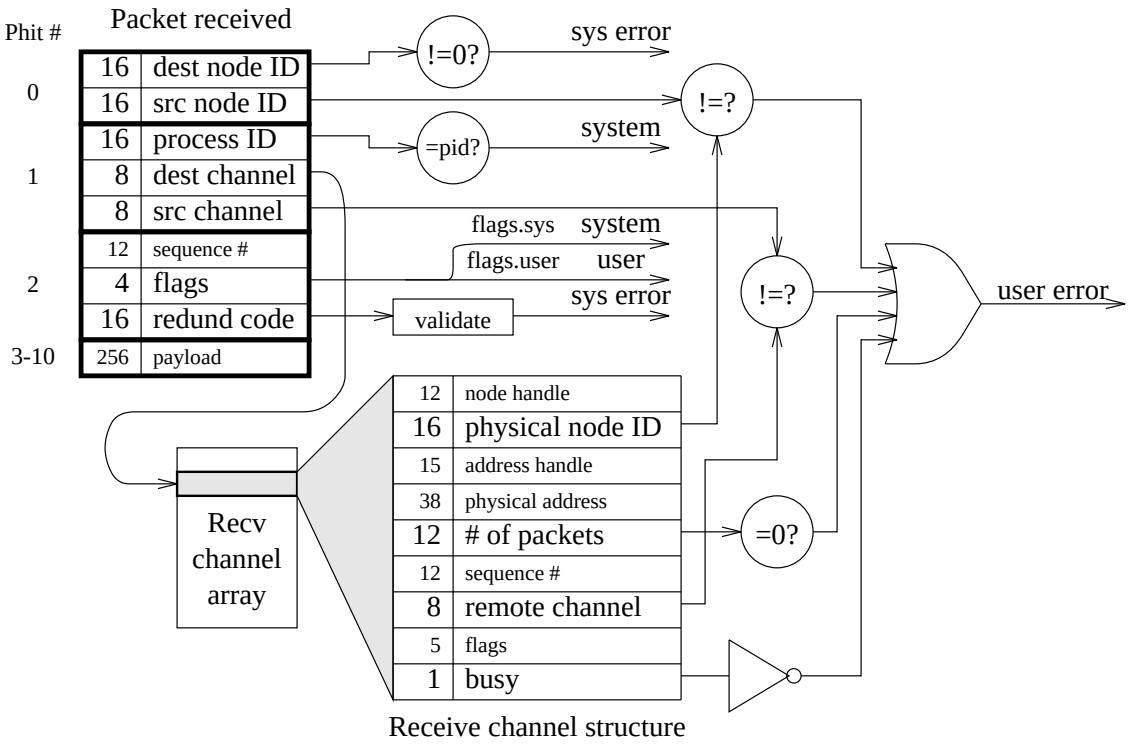


Figure 7.11: Behavioral description of the actions taken by the core module to decode and validate the header of a packet arriving from the inbox. The execution of this behavior determines the subsequent handler for the packet payload, which is either the handler for the queue channel handler (Figure 7.12) or the handler for the automatic-receive channel (Figure 7.13).

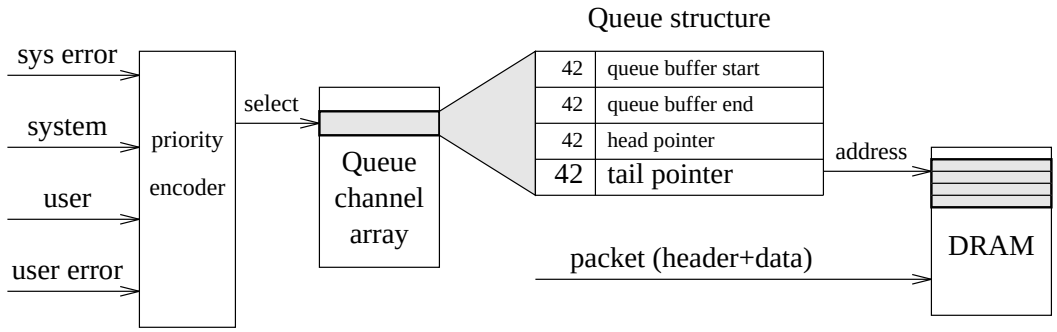


Figure 7.12: Behavioral description for determining the proper queue channel for handling the arriving packet. The tail pointer is read from the queue channel structure and is used as the DMA address for the ensuing write to DRAM over the ADU bus.

Figure 7.5 to simplify the diagram). The Physical Node ID field in the channel structure is compared with the Source Node ID field in the packet header. Likewise the Remote Channel Field in the channel structure and the Source Channel field in the packet header are compared. If a mismatch occurs in either case, the user error flag is set. Note that at this point, the packet could be destined for the user queue and no actual error occurs. The third phit of the packet header contains the Sequence Number, Flags and the Redundancy Code. The system and user flags are selected from the flags field. The inbox validates the Redundancy Code. If it is not valid then the system error flag is set. Two other conditions are tested: the packet count field and the busy bit in the auto-receive channel structure. If either of these are zero, the user error flag becomes active.

The four flags described above are inputs to a priority encoder (Figure 7.12). If at least one of these flags is set, the flag with the highest priority selects one of the four queue channels. The highest priority flag is the system error flag, followed by the system flag, the user flag and the user error flag, with the lowest priority. A user error comes only as the result of a failed access to an automatic-receive channel; the user queue flag in the packet header takes precedence over the user error flag. The output of the priority encoder is a selector that selects a queue structure from the queue channel array. The tail pointer is extracted from the selected queue structure (see Table 7.2). The packet payload is stored first, followed by the packet header, into two consecutive cache lines in DRAM starting at the address in the tail pointer field.

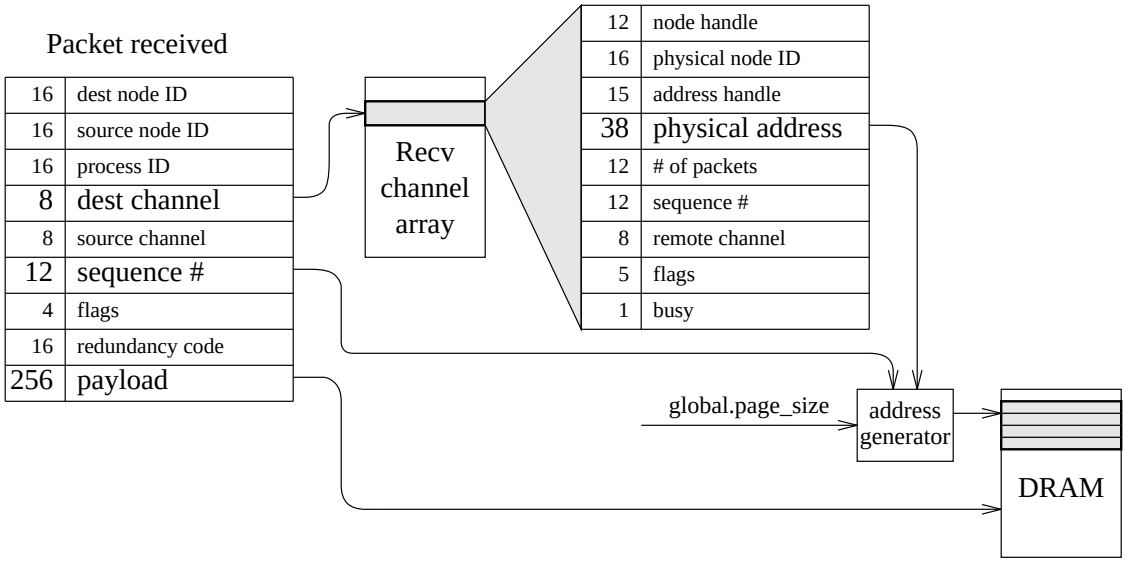


Figure 7.13: Behavioral description for determining the proper automatic-receive channel for handling the arriving packet. The generation of the DMA address for writing the packet payload into memory uses the same technique as in Figure 7.10.

If none of the bit flags described above are active, then the packet is sent to the automatic-receive channel selected by the Destination Channel field. Figure 7.13 describes the situation where the packet is destined for an automatic-receive channel. The physical address from the receive channel structure is combined with the Sequence Number field in the packet header, using the same sizing technique described in Section 7.3.3.

7.3.5 Updating the internal data structures

After Teschio sends a packet, receives a packet into a queue or receives a packet into an automatic-receive channel, it updates its internal data structures. Figure 7.14 shows the behavior required for updating the queue channel structures. The actions in Figure 7.14a occur when a packet is deposited into queue memory. Teschio deposits packets into memory into the tail of the queue and the application program removes packets from the head of the queue. There are two conditions that may occur when a packet arrives: wraparound and overflow. Wraparound means that pointer is beyond the end of its allocated memory, so it is wrapped back to the beginning. The wraparound condition is detected by comparing the tail pointer with Queue Buf End.

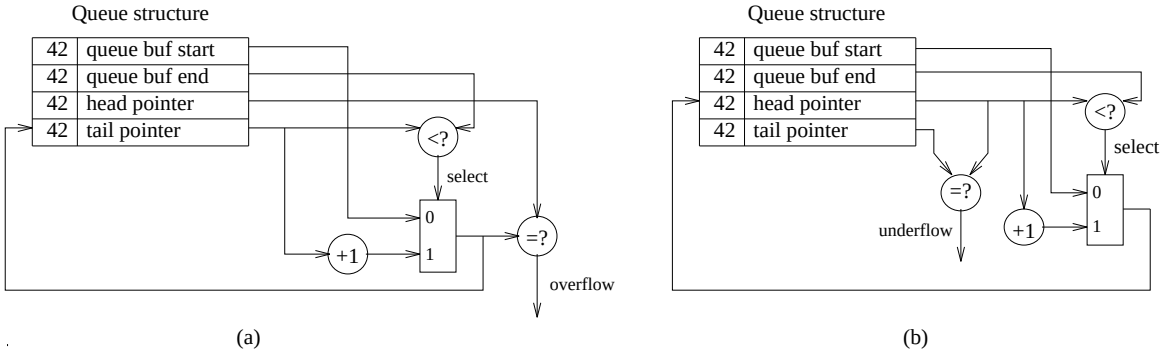


Figure 7.14: Behavioral diagram for updating the queue structure. Subfigure *a* shows the actions taken when a packet is deposited in the queue; subfigure *b* shows the actions taken when the application program advances the head pointer. In both cases, the wraparound condition is detected and used to compute the successor of the head or tail pointer. Error conditions are also detected: overflow in *a* and underflow in *b*.

If the result is less-than, then no wraparound occurs and the tail pointer is simply incremented. Otherwise the new value for the tail pointer is copied from Queue Buf Start. The overflow condition is calculated by comparing the result of the tail pointer computation with the head pointer; if they are equal then the tail pointer has run into the head pointer. Note that there is no immediate loss of information, but a subsequent packet arrival for that channel would overwrite data from a previous message that has not been handled by the application program. The complementary operation is advancing the head pointer (Figure 7.14b) which occurs when the application program running on the host processor sends an Advance Head Pointer command to Teschio. As in advancing the tail pointer, there are two conditions to consider in advancing the head pointer: wraparound and underflow. The wraparound test and calculation in advancing the head pointer is exactly the same as in advancing the tail pointer. The test for underflow differs from the test for overflow in one small detail: the underflow test occurs before the new head pointer is calculated, whereas overflow test occurs after the new tail pointer is calculated. Both the overflow and underflow conditions can be used to generate an interrupt or freeze the interface, depending on the state of the interrupt mask and the freeze mask in the core's global variable structure.

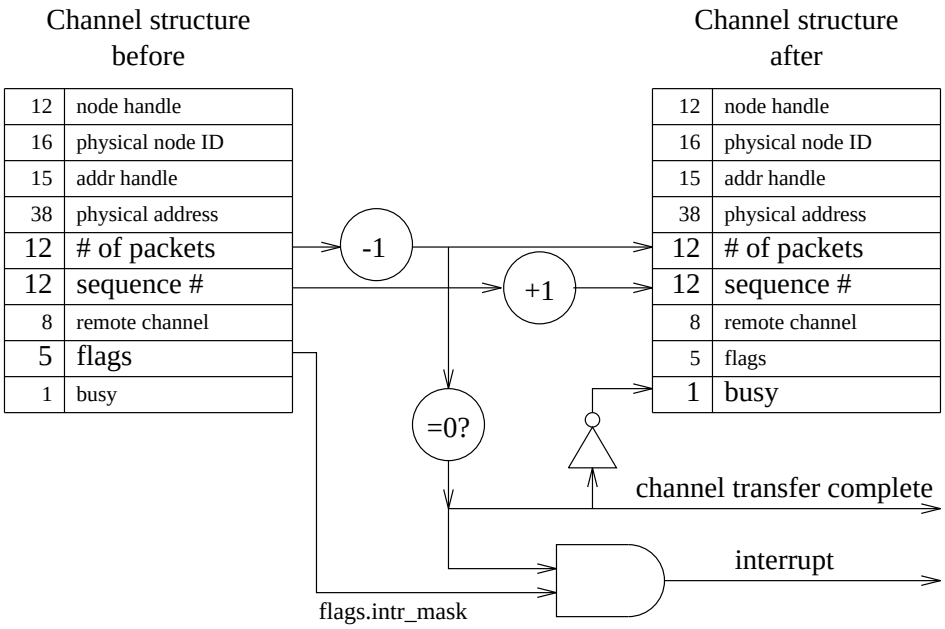


Figure 7.15: Behavioral model for updating the send channel and auto-receive channel information in the channel array. The channel structure is selected from the channel array using the channel ID at the head of the scheduler FIFO.

Figure 7.15 describes the behavioral model for updating send channel and auto-receive channel information in the channel array. After a packet is sent or received the number of remaining packets in the channel transfer operation is decremented. For the send channels, the next packet to send is indicated by incrementing the sequence number. Because packets arrive out-of-order in the Chaos network, it is not meaningful to compare the sequence number of the packet with the sequence number in its auto-receive channel. However, the situation would be different if the network architecture guarantees in-order delivery (such as an oblivious network or ATM). In that case, comparing the sequence number in the channel with that in the packet header could be included in the list of sanity checks in Figure 7.11. The interrupt bit is set according to the state of the interrupt status register, either after every packet is sent or received, or when the channel transfer completes (the packet counter field becomes 0).

The scheduler FIFO is updated whenever a new command arrives from the host processor or a command in progress completes. The update occurs when the last packet in the transfer has been sent or has been received. When one send channel transfer completes, the next send channel transfer begins on the next available ADU bus cycle.

7.4 Timing analysis

In this section we provide a timing analysis of Teschio. We start by stating the guidelines for the design that allow a smooth transition between the Cranium architecture and the Teschio implementation. We then derive the timing behavior of Teschio by counting the number of clock cycles it takes to send or receive a packet. This timing behavior is constrained by both the timing of the external environment and the sequence of the internal micro-operations. We focus on the internal timing behavior in this section; a detailed description of the external timing behavior can be found in Section D.3 in Appendix D. We follow the latency subsection with a discussion of the throughput requirements and the impact on the sizes of the FIFOs in the inbox and the outbox.

7.4.1 Design guidelines

The design guidelines consist of four main points:

- All of the logic internal to Teschio is synchronous and edge-triggered using the rising edge of the clock. For finite state control, Teschio uses Moore machines rather than Mealy machines. In other words, outputs come directly from state variables to ensure that they do not glitch.

- The pin drivers for Teschio use registers for both incoming and outgoing signals. The impact is that signals going on or off chip are delayed by one clock cycle.

- Internal FIFOs impose a one clock cycle delay. The architecture of the FIFOs is based on a scheme using a dual-port SRAM and two counters representing the head and tail pointers [15]. The dual-port property permits the FIFO to accept an incoming value and propagate an outgoing value concurrently at separate memory locations.

- All ALU functions are relatively simple and execute within a single clock cycle. Teschio requires only a small set of standard ALU operations: add, subtract, AND, OR and XOR. Other operations include table lookup and priority encoding. The number of inputs to the priority encoder is small, with 4 or fewer inputs in all cases.

One ramification of these assumptions is that it takes at least three clock cycles for any input signal at the chip's pins to influence the output signals of the chip. It takes one cycle to bring the signal on-chip, one cycle to pass the signal through a FIFO and perform simple arithmetic on the signal, and one cycle to send the signal off-chip. A more aggressive architecture might specify that information is allowed to propagate across the chip in as few as two clock cycles or even a single clock cycle. By restricting the timing to be less aggressive, an implementation can be developed more easily because fewer bypass paths are required.

7.4.2 Latency

We evaluate the latency and throughput results for Teschio. For latency there are three cases of interest: sending a single packet message, receiving a single packet

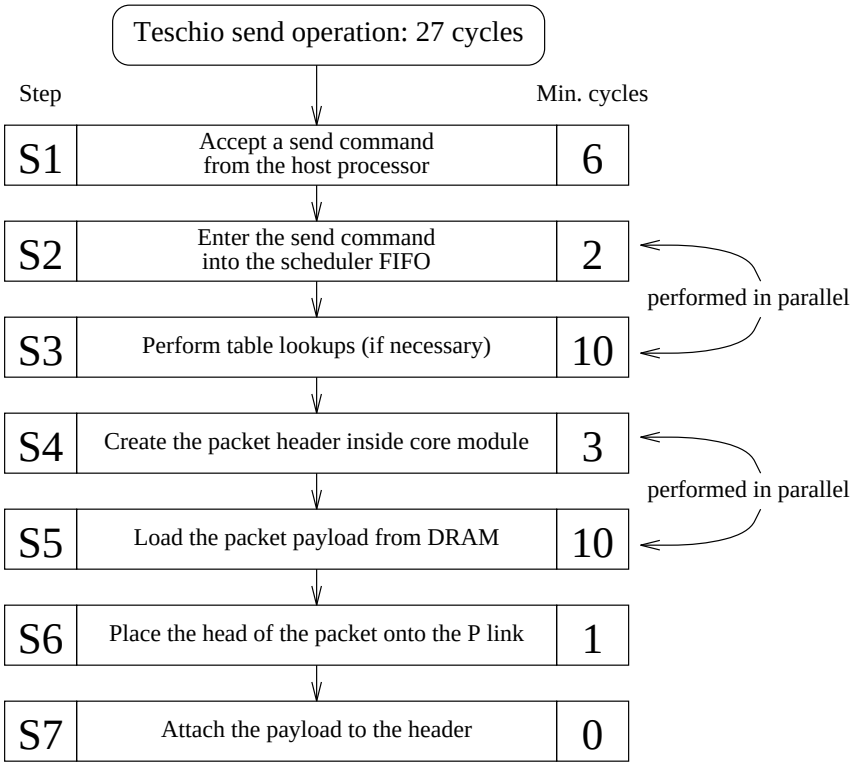


Figure 7.16: Sequence of micro-operations involved in a packet send operation and the minimum number of cycles taken at each step.

message into an automatic-receive channel and receiving into a queue channel. All of the latency figures assume the context of an otherwise idle system, as they represent the minimal amount of delay imposed by Teschio on message passing.

Figure 7.16 describes the minimum-latency timing of sending a packet in Teschio, in which this operation is the only activity present in the network interface. Sending a packet consists of the steps S1 through S7. The total latency is computed from the latencies of each step subject to the dependencies between steps. We consider only the time it takes to place the head of the packet onto the P link. (We account for the length of the packet in the time it takes to receive a packet.) Step S1, the arrival of a send command, takes six cycles. The limiting factor is the ADU bus. In principle, under a non-multiplexed bus, the time to execute S1 could be reduced to as little as one cycle. S2 follows S1 and adds two cycles: one to retrieve the information from the bus interface, and one to place it into the scheduler FIFO. S3

follows S2. We assume that there is exactly one table lookup that involves the ADU bus. Typically the physical node ID lookup is performed using the on-chip node lookup table, whereas the buffer address lookup requires an access to DRAM. The next available address bus state in the ADU bus occurs in two cycles; Teschio places a DMA address for table lookup in this clock cycle. One clock cycle is taken by the bus interface, leaving one cycle to compute the DMA address. Computing the DMA address is illustrated in Figure 7.10. It involves indexing into the channel array using the channel number at the head of the scheduler FIFO, retrieving two fields from the selected channel structure and a shift-and-combine operation to complete the computation. Since there are two RAM accesses plus an arithmetic operation, it may take more than one clock cycle to complete. The case where the send command is executed immediately after the command is accepted from the host processor is handled specially and involves a bypass path to eliminate one of the RAM accesses so that it completes in one cycle. The data accessed over the ADU bus is returned after another six cycles. Steps S4 and S5 follow S3 and both proceed in parallel. Of the two operations, S5 is the limiting factor because it involves an ADU bus access that adds ten cycles. Assembly of the header can be performed in as few as three cycles (Figure 7.10), so it is not on the critical path. The header cannot arrive too soon because the entire packet must be contiguous when it is handed off to the P link. The header is placed into the header FIFO in the outbox and waits until the payload is placed into the payload FIFO in the outbox, whereupon the two are combined. One cycle is taken by S6, in which the P link FSM places the first flit of the packet header onto the P link. S7 is omitted from this analysis because we are only concerned with the time to place the first packet onto the P link. The total minimum latency is 27 cycles.

To simplify the implementation we are concerned with the timing in only two cases: the packet arrives into its selected automatic-receive channel, or the packet arrives into the user queue. It is much less likely for packets to arrive into the other three queue channels, and we do not worry about those cases being handled more slowly.

Evaluating the latency of receiving a packet is complicated by the timing of the ADU bus. Recall that the ADU bus is on a fixed schedule of one address field followed by four data fields. Data that flows from the P link to the ADU bus may therefore

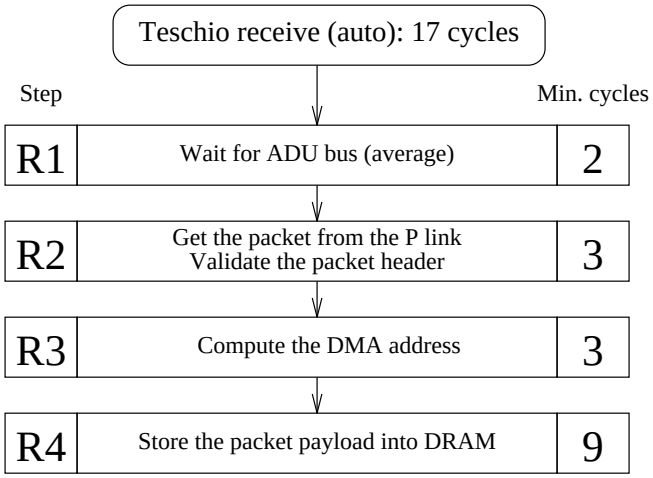


Figure 7.17: Sequence of micro-operations involved in a packet receive operation and the minimum number of cycles taken at each step.

encounter up to four extra cycles of delay. We make the assumption that the P link and the ADU bus are independent, meaning that any of the five states of the ADU bus are equally likely. The equally weighted average of the arithmetic sequence 0 to 4 is two, meaning that two extra cycles of delay are added to packets received from the P link and written to DRAM. Conversely, there is no extra delay needed to evaluate the latency of sending a packet, because the P link is not subject to the same fixed schedule as the ADU bus. When the P link is idle, it can become active on the very next cycle.

Receiving a packet into an auto-receive channel consists of the steps R1 through R4 in Figure 7.16. We assume that the command to initialize the channel has been issued far enough in advance so that its lookup table operations have completed and the channel is ready to accept a packet. Refer back to Figure 7.11 for a description of the behavior involved in R2. The most important deadline to meet is computing the DMA address for storing the packet into DRAM. By the guidelines set in Section 7.4.1, there are a minimum of three clock cycles from the receipt of the third flit (the final flit of the packet header) to when the DMA address is placed on the ADU bus during an address bus state. For the case of the user queue, the address of the tail pointer in the user queue structure can be read directly into a temporary variable. For the case of the automatic-receive channels, the address of the selected channel is determined

based on the channel ID in the second flit of the packet header. The user queue address or the auto-channel address is selected based on the value of the user queue flag in the third flit of the header.

The total number of states for handling the auto-channel case is seventeen: an average of two wait states imposed by the ADU bus, three to validate the packet header, three to place the DMA address onto the ADU bus, and nine to write the entire payload to memory. For the queue case there are five more states associated with writing the packet header to memory, for a total of 22. In the event that neither of these cases applies, the DMA access is quashed and an error handler for the packet is invoked. This error handler places the DMA address on the ADU bus five clock cycles later during the next available memory address bus state.

A more detailed description of the transactions that occur on the ADU bus for sending and receiving a packet are covered in Section D.3 in Appendix D.

7.4.3 Throughput

In addition to supporting low latency, Teschio must also support high throughput. Teschio must be able to sustain throughput at the rate of the P link: one packet may be sent or received every 11 clock cycles. Teschio must also be able to handle burst throughput at the rate of the ADU bus: two packets may be sent or received every 10 cycles, thus one every five cycles. Since the throughput requirement of the ADU bus is more stringent than that of the P link, we will focus our discussion on the timing of the ADU bus.

Teschio must perform the update tasks listed in Section 7.3.5 at the rate of one every five cycles. When a packet is sent or received, an entry in the send/receive channel array or the queue array is updated. When a send channel or an automatic-receive channel command completes, the scheduler FIFO is updated. Comparatively little computation is needed to perform these updates. Updating the queue channels involves two subtractions and one increment operation using 42-bit operands (Figure 7.14). Updating the send/receive channels involves an increment, a decrement and a wide OR gate on 12-bit operands (Figure 7.15). Updating the scheduler FIFO involves popping one value from the FIFO and potentially pushing one value onto the FIFO. The most obvious implementation is to transform the dataflow diagrams directly into structural elements. It is possible to reduce the amount of silicon needed

to implement these update tasks by multiplexing the arithmetic units, which is possible because there are enough clock cycles available. However there is little motivation for doing so because the amount of area saved is tiny compared with the area needed to implement the large structures of the chip such as the send/receive channel array.

Teschio's inbox and outbox contain FIFO buffering in order to provide an "impedance match" between the ADU bus and the P link. It is important to quantify the sizes of the FIFOs and the impact on the performance. The sizes of the FIFOs determine the amount of burstiness Teschio is able to handle. Increasing the sizes of the FIFOs increases the performance by reducing the number of bubbles in the packet stream. The incremental improvement in performance drops off sharply when the FIFO size is larger than a particular threshold. The goal is to make the FIFOs as small as possible to minimize the amount of chip area needed, while providing performance that is very nearly optimal. Note that the sizes of the inbox FIFOs and the outbox FIFOs may be different. In this context, the outbox FIFO is the combination of the outbox header FIFO and the outbox payload FIFO; similarly the inbox FIFO is the sum of the inbox header FIFO and the inbox payload FIFO. The inbox payload FIFO contains more space per entry than the outbox payload FIFO due to the special handling of packets destined for the queue channels (see Section 7.2.2), but our concern here is the total number of packets per FIFO rather than the details of its implementation.

Our estimates for the sizes for the FIFOs in the inbox and outbox are based on the following analysis. We assume that Teschio is used in the context of the Chaos network. We used the combined Talisman/Chaos network simulator to quantify the performance with both large and small FIFOs. In our experiment we varied three simulation parameters: the size of the outbox FIFO, the memory model and the message-passing program. The size of the inbox FIFO was left unbounded. Three quantities were measured: the sustained message throughput of the program, the maximum measured length of the send (outbox) FIFO, and the maximum measured length of the receive (inbox) FIFO. We ran two different message-passing programs that involved an exchange of page-length messages. We assumed a page size of 4K bytes. In the single message program, node A sends a 4K byte message to node B; when B receives the entire message it repeats the message back to A. The dual message program has two 4K byte messages in flight at once: node A sends to node B

Table 7.3: Impact of FIFO size on throughput (4K byte messages)

Test program	Outbox FIFO size	0 cycle DRAM latency			10 cycle DRAM latency		
		T'put % of peak	QLen Send	QLen Recv	T'put % of peak	QLen Send	QLen Recv
Single message	4	91	4	5	83	4	6
	100	93	20	5	83	6	7
Dual message	4	98	4	4	88	4	4
	100	98	36	4	88	4	4

and B sends to A simultaneously. The two memory models used in the experiments in Chapter 6 were used: the fast memory model (0-cycle DRAM delay) and the slow memory model (10-cycle DRAM delay). We chose two sizes for the outbox FIFO: small (4 packets) and large (100 packets).

Table 7.3 displays the results of the experiment. The three input axes are the test program, the size of the outbox FIFO and the memory model. The results are the sustained throughput (T'put) and the maximum measured lengths of the outbox queue (QLen Send) and the inbox queue (QLen Recv). The receiver code in each test program uses only the automatic-receive channels. For 0-cycle DRAM delay model, the peak throughput is 2.91 bytes per clock cycle; for the 10-cycle delay model, the peak throughput is 1.78 bytes per cycle. With the large outbox FIFO under the fast memory model, the sustained throughput is 2.65 bytes/cycle (91%) with the single message test and 2.71 bytes/cycle (93%) on the dual message test. Under the slow memory model the throughputs are 1.48 bytes/cycle (83%) for the single message test and 1.57 bytes/cycle (88%) for the dual message test. When the size of the outbox FIFO is reduced to 4 packets, the performance is practically identical in three of the four tests, and is diminished by less than 2% in the fourth test. The longest queue length measured in the inbox FIFO over all four tests is seven packets. This behavior was typical of all the test programs used to measure the impact of queue length on network throughput. All of the computation-intensive programs showed negligible difference in running time when outbox queue size was varied from 4 to 100. From these experiments, we conclude that an outbox FIFO size of four packets and an inbox FIFO size of eight packets is sufficient.

The experiments that were used to determine the size of the inbox and outbox FIFOs in Teschio were run only in the context of the Chaos network. Each Chaos router contains buffer space for up to 15 packets. Placing buffering in the network node is a well-recognized technique for improving throughput. Many commercially-available network routers (such as ATM switches) contain a moderate to large amount of buffering for this reason. However, there are networks that have very little internal buffering, such as wormhole networks that use oblivious routing (see Section 1.4). In a system where neither the network nor the network interface has much FIFO buffering, performance is expected to be relatively poor. Such a system can be improved by increasing the amount of buffering in either the network interface or in the network routers. The amount of extra buffering needed in Teschio to help overcome the limitations of a wormhole network is unknown and worthy of future study.

7.4.4 Summary

In Section 7.4.1 we stated a set of guidelines for transforming the behavioral model of Teschio into a structural model, from which Teschio's timing information is derived. These guidelines keep the implementation relatively simple and provide a realistic expectation of the number of clock cycles per each micro-operation. In Section 7.4.2 we provided an analysis of the latency of sending a packet and receiving a packet. This analysis was used to construct Table 6.1 in Chapter 6 and provides essential timing information that was used in the simulator (see Chapter 5). In Section 7.4.3 we determined the minimal configuration of the input and output FIFOs that would yield virtually optimal performance when Teschio is used in conjunction with the Chaos network. The FIFO depths were determined to be four packets for the output (send) FIFO and eight packets for the input (receive) FIFO. The results might differ if Teschio is used with a different network architecture and routing algorithm.

7.5 Fabrication parameters for Teschio

The description of Teschio presented in this chapter is a paper design; a physical chip has not been fabricated. To ensure that our paper design can be implemented in silicon successfully, it must satisfy a number of physical constraints. In this section we focus on three issues: clock frequency, pin count and gate count (area).

Clock frequency

A representative ASIC vendor is Mitsubishi Electric's Electronic Device Group (EDG) [101]. EDG supports many different ASIC chip processes, including the 0.6, 0.5 and 0.35 μm fabs. All three support clock rates of 100 MHz or more. Chips built using the 0.35 μm technology runs at speeds up to 200 MHz internally, and can support external clocking at this rate if differential signaling is used.

Pin count

The number of pins required for Teschio is computed from the sum of number of signals used its two external linkages – the ADU bus and the P link – plus some extra pins for "glue." The ADU bus contains 102 signals [64]. The P link contains 41 signals (see Section D.1). We assume that there are another 12 signals devoted to clocks, built-in self-test and other auxiliary functions. The total number of signals is $102 + 41 + 12 = 155$. If there is one pin per signal plus another 80 pins for power and ground, the total number of pins in the Teschio package is 235. This number of pins is easily addressed by using a ball grid array (BGA) package offered with EDG's processes. Standard molded BGAs contain up to 456 pins, and cavity-type BGAs can contain up to 672 pins. This packaging capability makes it possible to use differential signaling for every signal in Teschio to ensure reliability, at the cost of requiring two pins per signal rather than one per signal.

Area

We estimate the area of Teschio by counting the number of bits of memory that are implemented on-chip. In many custom VLSI chips the amount of static RAM is a good predictor of the total area. Most large ASICs contain a large amount of SRAM, such as the on-chip caches in modern high-performance processors. This SRAM usually accounts for more than 50% of the total chip area. Table 7.4 is a summary of the amount of memory within the submodules of Teschio. The bus interface does not contain an appreciable amount of memory, so it is not listed in the table. The memory sizes are listed in decreasing order. The second field in the table (Dimensions) lists two or three figures that describe the dimensionality of the memory array.

The largest amount of memory in Teschio is used by the send and automatic-

Table 7.4: Total count of the number of bits of memory in Teschio

Field name	Dimensions	Memory size (K bits)
Core: S/AR channel array	64×120	< 8
Core: Node lookup table	256×16	4
Inbox: Payload FIFO	$8 \times 4 \times 64$	2
Inbox: Header FIFO	$8 \times 6 \times 32$	< 2
Outbox: Payload FIFO	$4 \times 4 \times 64$	1
Core: Queue channel array	$4 \times 4 \times 42$	< 1
Core: Scheduler	64×6	< 0.5
Outbox: Header FIFO	$4 \times 4 \times 32$	< 0.5
Total		< 20

receive channel array in the core module. Since there are 32 send channels and 32 auto-receive channels, the total number of channels is 64. Each channel contains roughly 120 bits, yielding a total of just under 8K bits. The node lookup table contains 256 entries of 16 bits (4K bits). The inbox payload FIFO holds up to eight cache lines each consisting of four 64-bit words (2K bits). The inbox header FIFO contains two copies of eight packet headers (see Section 7.2.2). Each packet header contains three 32-bit words. The total number of bits for the inbox header FIFO is $2 \cdot 8 \cdot 3 \cdot 32$ (less than 1K bits). The queue channel array consists of four groups of four 42-bit addresses (see Table 7.2). The outbox payload FIFO contains space for four cache lines (1K bits). The scheduler FIFO in the core module contains 64 six-bit channel identifiers – one for each send channel and each automatic-receive channel (less than 0.5K bits). Finally, the header FIFO in the outbox contains storage for four packet headers requiring three 32-bit phits per header (less than 0.5K bits). Altogether, Teschio contains approximately 20K bits, less than 3K bytes of memory.

We construct a conservative estimate of gate count via the following line of reasoning. It takes 10 gates to implement one bit of SRAM memory that includes self-test capability [101]. The total number of gates devoted to Teschio's memory is therefore approximately 200,000. To calculate an upper bound on chip area, we use the assumption that SRAM occupies 50% or more of the total area. Therefore we add another 200,000 gates to account for the bus interface, datapath buffers and multi-

plexors, finite state machines and control store memory. Therefore our estimate for the number of gates in the entire chip is fewer than 400,000. All three of the EDG fab processes listed above (0.6, 0.5 and 0.35 μm) allow 400,000 or more gates per chip; the 0.35 μm process supports up to 2 million gates.

We conclude that all three implementation requirements of Teschio (clock frequency, pin count and area) can be achieved using today's high performance fabrication technology.

7.6 Extensions to Teschio

The Teschio implementation of Cranium can be extended in many different ways. We comment on three possible extensions: packet scheduling, support for fast context switching and gather-scatter support.

7.6.1 Packet scheduling and traffic shaping

In Section 3.4.1 we discussed the tradeoffs involved with scheduling packets coming from competing send channels. The base case is FCFS (first come, first served). The cost of implementing round-robin scheduling in Teschio is an extra data path to bring the output of the scheduler back to the input. After a packet is sent, the channel ID at the head of the queue is popped and then pushed onto the tail of the same queue.

The packet injection side of Teschio does not perform ATM-style traffic shaping. That is, the scheduler assumes that the full bandwidth of the P link is always consumed (except when the network approaches saturation, in which the router prevents packets from injection). A typical example of traffic shaping in an ATM network interface is leaky bucket traffic shaping. A packet is sent every R'th cycle, where R is greater than L, the number of flits in a packet [40]. The advantage of traffic shaping is that it prevents the network from saturation and provides throughput guarantees. In the multicomputer domain, traffic shaping is performed in the application program, if at all. However, there is trend towards lowering the cost of parallel computing by using networks of inexpensive workstations instead of tightly-coupled multicomputers. This trend may in turn spur a merging of the designs of network interfaces for both tightly-coupled multicomputers and LAN-connected workstations.

7.6.2 Fast context switching

In Section 3.4.3 we introduced the performance problem due to the need to support multiple user contexts. The solution was to provide physical channel resources for U+1 user contexts where U is the number of processors per node of the MPP system. Two approaches to implementing multiple user contexts were to place two complete sets of Cranium registers (e.g. a second core module) on a single Teschio chip, or to allow multiple Teschio chips to be tiled together externally. Deciding to use multiple cores on a chip versus external tiling depends on economic factors. A dual-core version is more difficult to build than a single-core version as it requires more area to hold the extra blocks of static RAM and more development effort to test and verify it. However, it also helps reduce the chip count in systems that require the highest performance.

Here is how an externally-tiled version of Teschio might work. Figure 7.18 describes a hybrid MPP-SMP system with two processors and two Teschio network interface chips per MPP node. The purpose of the second Teschio chip is only to replicate the internal channel structures in a single chip. The throughput of the P link does not increase; only one Teschio chip can use the P link at a time. Such a multiple-Teschio configuration requires some additional control signals in the P link. An additional signal is needed to arbitrate when both network interfaces are ready to inject a packet. Another set of signals is needed so that when a packet arrives, one and only one network interface chip handles the packet and stores its payload into memory. If the process ID of the packet matches the current context of one of the network interface chips, it signals a match and handles the packet. If the packet's process ID does not match any of the current contexts, one chip must be designated as its handler.

Evaluation and implementation of the multiple-user-context versions of Teschio are left to future work. Evaluating even a two context version is much more difficult than the already difficult task of evaluating the single context version. The difficulty stems from the combinatorial explosion in trying to benchmark a multitasking time-sharing workload. If there are N separate benchmark programs, then there are roughly N^2 possible workloads that can be constructed by choosing two of these benchmarks. The number of workloads employing all N benchmarks in succession is the factorial of N (N!). To our knowledge there is no universal standard for the evaluation of time-

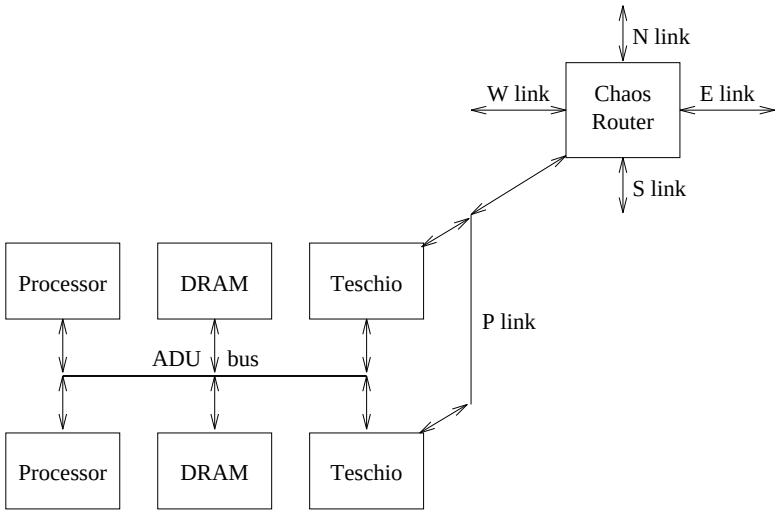


Figure 7.18: An architecture for a hybrid MPP-SMP system that uses multiple Teschio chips per processing node

sharing workloads on multicomputers. Another difficulty comes from the handling of exceptional conditions under a multitasking workload. For instance, if the arrival of a packet signals an interrupt (such as an end-of-channel-transfer interrupt) then the interrupt should be serviced only if the packet belongs to the user process that is currently executing. Interrupts for user processes that are not running must be saved and then executed when the process runs again. The techniques for handing these cases and the measurement of their impact on performance have not yet been explored.

7.6.3 Gather-scatter support

The basic strategy for incorporating gather-scatter capability in Cranium was introduced in Section 3.4.4. Adding gather-scatter functionality to Teschio is straightforward. In Teschio, we assume the limitation that the minimum size access to memory is a cache line. If the stride size is approximately the size of a cache line or smaller, then it is necessary to send the entire array if it is desirable to make it a single message. However, if the stride size is much larger than a cache line, but on the same MMU page, the following scheme could be used. The packet sequence number is used as the index into a gather or scatter lookup table, to look up the address offset from the base MMU page. Figure 7.19 describes an implementation of this scheme on

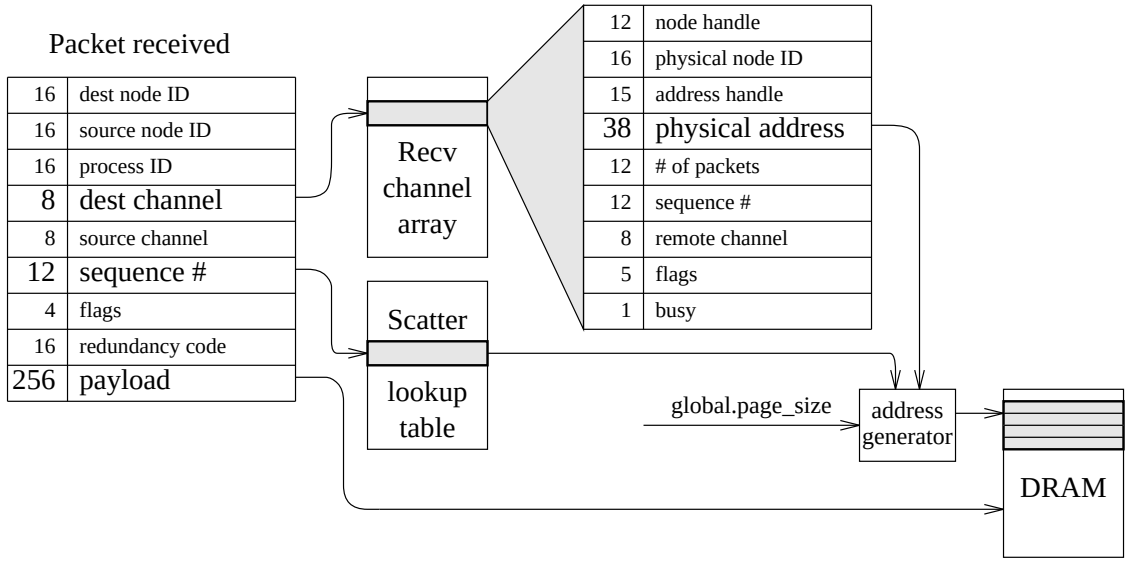


Figure 7.19: One technique for supporting the scatter operation in Teschio. The sequence number of the packet performs a lookup into a scatter table located in on-chip static RAM for low latency.

the receive (scatter) side, as an extension to the standard automatic-receive channel. Both the sending node and the receiving node require lookup tables. For highest performance, these additional lookup tables are located on-chip. Each table is an SRAM that uses 12 bits of address and 12 bits of data. Each SRAM contains 12×2^{12} bits or 6K bytes, for a total of 12K additional bytes of storage. Since the access stays on the same MMU page, the access protection mechanism stays exactly the same.

There are many possible variations on the basic scheme outlined above to overcome the restriction that the gathered or scattered data fit in a single MMU page. For instance, each slot in the lookup tables can store a complete physical memory address, or include both a page offset and an address handle. In addition to increasing the size of on-chip memory needed to contain this additional information, the design must ensure that the same level of memory protection is provided, to prevent a malicious user program from overwriting the operating system, for instance. The simplest version is to restrict access to lookup table memory to the operating system, instead of letting the application program access it directly as in the basic scheme.

The efficacy of a gather-scatter scheme depends on its ability to amortize the overhead of initializing the lookup tables. If the number of elements to transfer is

small, then marshalling the data might be more efficient than gather-scatter support. The improvement in performance gained by gather-scatter hardware depends strongly on the set of benchmarks used to evaluate it. Performing this evaluation in the context of Teschio is left to future work.

7.7 Summary

This chapter presents a paper design of Teschio, an instance of the Cranium network interface architecture. Teschio is a synchronous single-chip VLSI ASIC that connects directly to both the processor-memory bus in the processing node and the processor-network link of the network router. In Section 7.1 we defined the system environment for Teschio, including the ADU (memory) bus, the P link and the node mapping strategy. Teschio uses two-dimensional parity for improving data integrity with very little circuitry.

We described the internal organization of Teschio in Section 7.2. There are four primary structural modules within Teschio: the bus interface, the core module, the inbox and the outbox. In Section 7.3 we explained how these modules function as a team. The bus interface provides both slave and bus master access to the ADU bus. The core module schedules packet departures. It also provides table lookup functionality for node and message buffer address mapping to implement fast, secure argument checking that acts on behalf of the operating-system. Packet headers are assembled in the core module. Packet payload data comes directly from memory via bus master DMA. The outbox joins the packet header with the payload and provides buffering for packets to be injected into the network. Similarly, the inbox provides buffering for packets ejected from the network. The inbox separates the packet header from the payload; the header goes to the core and the payload goes to the ADU bus.

In Section 7.4 we provided cycle-by-cycle timing for Teschio from the timing of the external environment (the ADU bus and the P link) and the interactions of the internal modules. We assumed a set of design guidelines that help make the chip easy to implement by reducing the number of long paths between registers. We measured the impact of internal FIFO size on the performance of Teschio. We found that a transmit FIFO that contains as few as four packets and a receive FIFO that contains as few as eight packets performs equally well as an interface with space for 100 packets in both directions. Our belief is that through the use of a high-performance network

such as one based on the Chaos router, it is necessary to include only a small amount of buffering in the network interface hardware to achieve the same throughput as an interface with a large on-chip FIFOs. Another way to interpret this finding is that the entire Chaos network behaves as a large FIFO. A different result might be found if a network architecture such as the wormhole-oblivious router is used instead of Chaos, because that style of network contains only a very tiny amount of internal buffering. In that case it may be necessary to greatly increase the amount of buffering in the network interface in order to compensate.

In Section 7.5 we argued that today's high performance fabrication technology is more than adequate to fabricate Teschio in a single VLSI ASIC chip. The internal circuitry of Teschio, the ADU bus and processor-network link (P link) form a single clock domain that runs at speeds up to 100 MHz. Since the performance figures from Chapter 6 indicate that the peak throughput delivered to the application program is 2.91 bytes per cycle, the peak throughput of Teschio is $2.91 \times 100 = 291$ MB/sec.

We conclude that the Teschio implementation is feasible. It takes an order of magnitude less chip area than a companion high-performance processor. It is relatively simple to implement: it requires nothing more than the standard set of structural elements such as SRAM, adders, comparators and data paths. There are no content-addressable memories in Teschio, whose performance does not scale well with size. Despite the simplicity of the implementation, Teschio delivers the high performance capability shown by the simulation studies in the previous chapter. Nevertheless, Teschio is only the first implementation of the Cranium architecture. In Section 7.6 we examined three potential areas for future work: traffic shaping, support for fast context switching and gather-scatter capability.

Chapter 8

CONCLUSIONS

We must have new solutions to new problems.
– *J. Kennedy Toole,* A Confederacy of Dunces

In this dissertation we introduced the Cranium architecture as a solution to the network interface problem in massively parallel processing (MPP) computers. Cranium differs from previous network interface designs by providing hardware support for three important features in a single architecture: support for adaptive networks, user-level, bus-master DMA for both low latency and high throughput, and support for both the buffered and unbuffered message protocols. Cranium reduces the cost of communication by providing parallel application programs with efficient mechanisms for both small messages and large messages. User programs have direct access to Cranium's communication primitives through its integrated application program interface, which greatly reduces software overhead compared to a heavily-layered message passing library such as the Intel NX library. Our performance results confirm that the Cranium approach achieves low latency on small messages and asymptotically optimal throughput as message size increases beyond a few thousand bytes.

The remainder of this chapter is organized as follows. We summarize the Cranium architecture, its application program interface (API), the test environment, our performance results and our paper design for a VLSI chip implementation of Cranium. We list the principal contributions of this dissertation and cover two interesting areas of future work. We close with some thoughts on network interface design in general.

8.1 Cranium architecture

Cranium is located at the memory bus of the processing node and it accepts message passing commands from the host processor via memory-mapped load and store commands. This organization permits the use of bus-master DMA but with higher performance than connecting at an I/O bus. It also isolates the network interface

architecture from the processor architecture, unlike designs that involve a tight coupling between network interface and processor. To adapt Cranium to work with future processor and network designs, it is necessary to change only Cranium's external connections: its memory bus interface and its network link interface. Future implementations of the Cranium architecture retain the same low-level programming interface, and we expect efficient message-passing programs to be ported to new generations of hardware with a minimum of effort.

Cranium provides direct user-level access to application programs. Direct user access allows application programs to send and receive messages with no overhead from the operating system in the common case. The only mode of message passing that involves operating system overhead is the allocation of DMA buffers. Buffer allocation can be localized to the program's initialization phase so that all subsequent message passing commands are performed at user level.

Cranium handles both large and small messages efficiently using separate mechanisms. Over a wide range of parallel benchmark programs, most messages are small but a sizable fraction of the number of bytes in network traffic come from large messages. It is important to support both cases independently, because no single mechanism has been demonstrated that supports both cases well. Cranium's user queue provides low latency on small messages and its automatic-receive channels provide high throughput on large messages. In both mechanisms, the network interface writes packet payload data directly into message buffers in DRAM-based memory whose pages are pinned and mapped into the user's address space. The user queue implements the buffered message protocol; queue space is pre-allocated by the operating system, but directly available at user level. The auto-channels implement the unbuffered message protocol; the application program pre-allocates buffer space before packets are sent to that channel. Message data are reassembled in-place without copying or requiring processor intervention, and the processor is notified only at the completion of a message transfer, rather than every time a packet (i.e. a cache line) arrives. Thus, the auto-channels deliver the highest performance for large messages.

Cranium's automatic channels provide compatibility with adaptive packet-routing networks such as the Chaos network, by counting packets to determine the complete delivery of a message. It is important to support adaptive routing directly in the network interface to reduce processor overhead. Adaptive routers are important be-

cause they deliver a higher percentage of the raw bandwidth of the network than an oblivious router does, thereby providing higher effective throughput and latency under high network loads. Adaptive routers therefore are excellent candidates to be used as the backbone of an MPP interconnection network.

8.2 Cranium application program interface

A primary design goal of the Cranium project was to provide a clean, powerful and robust programming interface for interprocessor communication. The Cranium API is a very thin layer on top of the underlying hardware architecture. It takes only a single store operation to a memory-mapped network interface register to initiate sending or receiving a message up to the size of an MMU page. Cranium accommodates MMU pages sizes ranging from 2K to 64K bytes. These send and receive operations are asynchronous and non-blocking. Because Cranium uses bus-master DMA to move data between DRAM memory and the network, messages can be sent or received while the processor is performing computation. Implementing blocking communication on top of the asynchronous message primitives is as simple as waiting for an interrupt or polling a status register. The Cranium API provides the efficiency of a lightweight message passing kernel such as active messages with nearly all the generality of a heavyweight message passing interface such as Intel NX or MPI.

8.3 Test environment

The test environment for Cranium was created from a combination of simulation strategies. The processor, memory and network interface were simulated using an extended version of Talisman, a functional simulator augmented with a statistically accurate timing model. We used a structural simulator that provides exact cycle-by-cycle timing to simulate the Chaos network. The Chaos simulator can display a graphical animation of packet traffic. Animation was helpful for debugging and demonstration purposes. The combined simulator simulates a parallel computer with up to 256 processing nodes. Host execution performance of the simulator was reasonable; it was fast enough to run a wide variety of experiments and thereby help refine the design iteratively. The slowdown factor per simulated processing node was approximately 100 without animation or about 300 when animation was activated.

8.4 Performance analysis

Our performance analysis in Section 6.1 demonstrated that Cranium is capable of both low latency and high throughput. The end-to-end latency of a one-way, single-packet message is approximately 60 to 100 clock cycles, i.e. between 0.6 and 1.0 microsecond if the clock frequency is 100 MHz. End-to-end latency is the sum of the following cycle counts: 17 to 32 cycles at the sender's network interface, 8 to 36 cycles in the network and 30 to 35 cycles at the receiver's network interface. These figures incorporate the time charged to processor overhead and to the bandwidth and latency penalty for accessing DRAM. The latency figures for Cranium are comparable to those in tightly-coupled network interface designs. Cranium achieves 90% of the maximum possible sustained throughput with messages as short as 2048 bytes, and 96% of this maximum with 8K byte messages.

8.5 Empirical results

Proving the effectiveness of a network interface design requires more than just an analysis of latency and throughput. It is necessary to demonstrate that parallel programs are capable of achieving linear speedup when used in conjunction with the network interface. Parallel programs expose the software overhead of the interface and thereby provide a more realistic determination of the communication performance than the simple latency and throughput measures do.

In Section 6.2 we outlined our suite of parallel benchmark programs: fast Fourier transform, bucket sort, Jacobi, Gaussian elimination and dense matrix multiply. All of the benchmark programs were run with small data set sizes that were kept fixed while the number of processors was increased from 4 to 64 in power-of-two increments. The purpose was to emphasize the communication component of total execution time. Despite the small input data set sizes, the selected parallel benchmark programs achieved both high communication performance and excellent speedups. In particular, the dense matrix multiply benchmark achieved nearly perfect overlap of computation and communication and sustained network throughput that was up to 90% of the maximum possible.

In Section 6.3 we compared Cranium to other network interface styles. Our approach was to omit features of Cranium that were not available with these other

network interfaces. The resulting modifications to the network interface increased the time spent in communication on the Gauss benchmark by factors of 1.5 to 4. We conclude that Cranium provides a more complete set of communication primitives than the other approaches do, so that Cranium provides efficient, high-performance communication over a wider range of programs and traffic patterns than the other interface approaches.

8.6 Teschio

In Chapter 7, we described Teschio, a paper design for a VLSI chip implementation of Cranium. Teschio is composed of four modules: the bus interface, the core module, the inbox and the outbox. The core module contains all of the data structures that reflect Cranium's high-level programming model including the send and auto-receive channel structures and the queue for scheduling send operations. The inbox contains FIFO buffering for arriving packets and the outbox contains buffering for packets to be injected into the network. Simulations showed that when Teschio is used with the Chaos network, an FIFO size of eight cells (packets) in the inbox and four cells in the outbox was sufficient. Teschio does not use a TLB; rather, all that is required to ensure protection for remote node IDs and local physical message buffers is a lookup table implemented simply by using a static RAM. The total number of memory bits needed to construct Teschio is less than 20,000, i.e. less than 3K bytes. We estimated that Teschio can be implemented using fewer than 400,000 gates and support a clock frequency of 100 MHz. We conclude that Teschio is a feasible design that is relatively simple to construct.

8.7 Contributions of this dissertation

This dissertation advances the science of computer architecture by providing a taxonomy of network interface designs in Chapter 2. To our knowledge, no previous taxonomy covers as broad a range of issues in network interface design. We use the taxonomy to characterize a set of selected network interfaces in Table 2.1. We showed that previous work tends to focus on a single design issue. For example, tightly-coupled designs focus on achieving the lowest possible latency on small messages, and do not achieve high throughput on large messages. Furthermore, there has

been little previous work on network interfaces that work efficiently with adaptive networks (the Hamlyn project from HP Labs is a notable exception). It is necessary to identify all of the important design issues to ensure that a network interface design achieves sufficient coverage to be effective.

The Cranium architecture was developed to provide an example of a minimally complex network interface that contains all of the necessary functionality identified in the taxonomy. Despite its simplicity, Cranium achieves both low latency and high throughput. The process of developing the simulation environment to evaluate Cranium produced a testbed for evaluating new ideas in network interface design in general. The environment simulates a large number of processing nodes yet achieves sufficiently fast host execution speed to allow interactive execution and debugging on real applications. This makes the environment amenable to future studies in network interface development.

8.8 Future work

8.8.1 The Chaos-LAN project

A team of researchers and undergraduate students at the University of Washington are actively working on the *Chaos-LAN* project. The purpose of this project is to investigate the use of adaptive routing in a local area network, through the construction of a high performance network of workstations (NOW) [28]. The proposed configuration of Chaos-LAN is described in Figure 8.1. The environment consists of a collection of workstations from Digital Equipment Corporation based on the Alpha processor. They are connected to a central hub containing a Chaos network consisting of 16 Chaos routers connected as a two-dimensional torus. Up to 16 workstations can be used. The link between each workstation and the hub is based on the Fibrechannel physical layer. The network interface is the DEC PCI Pamette card [102], a PCI card that sits between the Alpha and the Fibrechannel encoder-decoder. The Pamette contains several Xilinx field programmable gate arrays (FPGAs) that can be programmed, tested and debugged using a variety of circuits. The development environment for the Pamette makes use of the Verilog hardware description language and a set of logic synthesis tools.

To complete the construction of the hardware environment, there are two primary

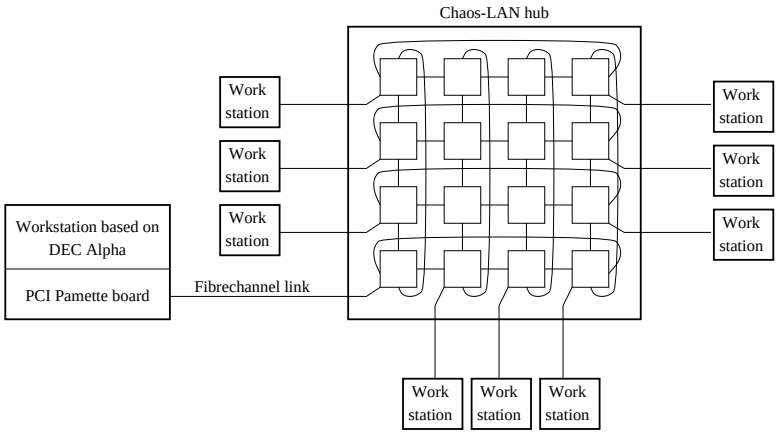


Figure 8.1: Overview of the Chaos-LAN research project

remaining tasks. The first task is to tape out a new version of the Chaos router; Kevin Bolding is expected to complete this aspect of the project by summer 1997. The second task is to develop the network interface circuit in the FPGAs in the Pamette board, guided by the principles of the Cranium network interface architecture. Since a full Cranium implementation is too large for the Pamette, the challenge will be to identify the minimal set of features that will nevertheless yield the underlying performance of the Chaos-LAN network.

To evaluate the effectiveness of Chaos-LAN and the Pamette-based network interface, we plan to run a software environment called the Global Memory System (GMS) [103, 104] on the NOW. GMS is based on the idea of paging over the network to idle workstations instead of to local disk. As network bandwidth is becoming comparable to disk bandwidth, paging over the network becomes the faster solution once the latency of the network interface becomes sufficiently low. A network interface based on Cranium is expected to work well with GMS because the unit of transfer in GMS is an MMU page, the same as the maximum unit of transfer for a Cranium auto-channel. Cranium also supports sub-page transfers efficiently. The user queue can receive the first cache line of a page transfer and then subpages can be subsequently transferred using one or more auto-channels.

The goals of the Chaos-LAN project are similar to that of the Myrinet NOW [31]. At its conclusion, we plan to demonstrate that Chaos-LAN delivers superior performance for comparable cost with Myrinet, or equivalent performance at lower cost.

8.8.2 Additional performance studies

In addition to the Chaos-LAN hardware implementation project, there are additional performance studies that can be performed. These studies are better suited to running on the simulator than on the Pamette because they are in the initial evaluation phase.

- *Fast context switching.* Presently, the performance of context switching is not measured in the simulator. As explained in Section 3.4.3, the performance of context switching is expected to be improved by implementing a second set of channel registers in the network interface circuit, or ganging together two network interface chips (see Section 7.6.2)).

- *Gather-scatter support.* The goal is to explore and quantify the tradeoffs in different approaches to supporting gather-scatter under Cranium (see Section 7.6.3). The impact of gather-scatter support would be measured using a set of applications that are likely to benefit from this feature.

8.9 Closing thoughts

8.9.1 Message passing vs. shared memory

This dissertation was undertaken with the intuition that message passing provides a superior communication model to that of distributed shared memory in a scalable parallel computer. It became obvious early on that the message passing (send/receive) model was simpler to construct and simpler to interface with adaptive routers than its shared memory counterpart. Support for this point of view was confirmed by the two following observations.

- The granularity of transfer in Cranium is a cache line. This unit of transfer turned out to be quite efficient, especially when used in conjunction with a memory bus architecture that works directly with cache line units of data, such as the ADU bus [64] (see Section 7.1.2). A message passing system or shared memory system that implements with a smaller unit of transfer directly is likely to increase the complexity of implementation significantly without increasing its performance. To transfer a sub-cache-line unit of data under Cranium, the receiving node receives a cache line into its user queue and then the application

program copies the required data. The cost of copying is small in this case because the size of the data object itself is small. If possible the data object is simply integrated into the ongoing computation (e.g. through its active message handler) and does not need to be stored in memory as an intermediate step.

- Since the size of packet used by Cranium is small, the size of the packet header directly affects the maximum throughput of the network seen by application programs. The Cranium packet header does not contain a global virtual address (GVA); instead it uses a channel number that identifies how the packet is handled at the receiver. Channel numbers are small (16 bits or less) compared with GVAs (48 to 64 bits). Therefore, Cranium's smaller packet headers reduce the amount of bandwidth lost, hence yielding higher effective throughput. Furthermore, a small channel number can take advantage of directly indexing into SRAM instead of using a content-addressable memory. This means that under the Cranium architecture, the network interface at the receiver can be made simpler and its latency is lower than in a system that uses a GVA in the packet header.

8.9.2 The road ahead

We do not expect that Cranium is the final word in network interface architecture. There are a myriad of trends that will affect the design and implementation of future scalable parallel computers. The greatest impact comes from the opposite ends of the spectrum: the trends in VLSI technology and the trends in system level design. VLSI technology continues to follow Moore's Law [1], but there are signs that the historical exponential growth in chip density and performance will finally level off in the next decade or two. The trend in system level design is away from large monolithic systems and is heading towards networks of workstations. The combination of these trends will modify the tradeoffs in the network interface design and where the interesting implementation points will lie in the design spectrum. Nevertheless the underlying principles of Cranium will remain important: support for both large and small messages, support for adaptive networks, and direct access by the application program to the most frequently used features of interprocessor communication so that they incur the smallest possible amount of processor overhead.

Bibliography

[1] David A. Patterson. Microprocessors in 2020. *Scientific American*, September 1995, pp. 48-51.

[2] W. Daniel Hillis. *The Connection Machine.* MIT Press, Cambridge MA, 1985.

[3] Charles L. Seitz. The Cosmic Cube. *Communications of the ACM* 28(1), January 1985, pp. 22-33.

[4] William C. Athas and Charles L. Seitz. Multicomputers: message-passing concurrent computers. *IEEE Computer* 21(8), August 1988, pp. 9-24.

[5] Kevin Bolding and Lawrence Snyder, eds. *Proc. of Parallel Computer Routing and Communication Workshop*, Seattle WA, May 1994, Springer-Verlag.

[6] William J. Dally. Wire-efficient VLSI multiprocessor communication networks. *Advanced Research in VLSI: Proc. of the 1987 Stanford Conference,* MIT Press, 1987, pp. 391-415.

[7] Paul Pierce. The NX message passing interface. *Parallel Computing* 20(4), April 1994, pp. 463-80.

[8] J. J. Dongarra, R. Hempel, A. J. G. Hey and D. W. Walker. A draft standard for message passing in a distributed memory environment. *Proceedings of the Fifth ECMWF Workshop on the Use of Parallel Processors in Meteorology: Parallel Supercomputing in Atmospheric Science,* Reading, UK, Nov. 1992, pp. 465-81.

[9] J. Bruck, D. Dolev, Ching Tien Ho, M. C. Rosu and R. Strong. Efficient message passing interface (MPI) for parallel computing on clusters of workstations. *Proc of 7th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '95)*, Santa Barbara, CA, July 1995, pp. 64-73.

[10] Edward W. Felten. Protocol compilation: high-performance communication for parallel programs. PhD dissertation, University of Washington, Dept. of CSE, Sept. 1993, UW-CSE-TR 93-09-09.

[11] John Y. Ngai and Charles L. Seitz. A framework for adaptive routing in multicomputer networks. *Proc. of the Symposium on Parallel Architectures and Algorithms,* May 1989.

[12] Kevin Bolding and Lawrence Snyder. Mesh and torus chaotic routing. *Advanced Research in VLSI and Parallel Systems; Proc. of the 1992 Brown/MIT Conference,* March 1992, pp. 333-347.

[13] Kevin Bolding and William Yost. Design of a router for fault-tolerant networks. *Proc. of Parallel Computer Routing and Communication Workshop*, Seattle WA, May 1994, Springer-Verlag, pp. 226-240.

[14] Charles Leiserson, Z. S. Abuhamdeh, D. Douglas, C. Feynmann, M. Ganmuki, J. Hill, W. D. Hillis, B. Kuszmaul, M. St. Pierre, D. Wells, M. Wong, S.-W. Yang and R. Zak. The network architecture of the CM-5. *Symposium on Parallel Algorithms and Architectures*, 1992, pp. 272-285.

[15] Kevin Bolding. *Chaotic routing: design and implementation of an adaptive multicomputer network router.* PhD dissertation, University of Washington, Dept. of CSE, Seattle WA, July 1993.

[16] Alexander C. Klaiber. Architectural support for compiler-generated data-parallel programs. PhD dissertation, University of Washington, Dept. of CSE, Sept. 1994, UW-CSE-TR 94-09-09.

[17] Paul Pierce and Greg Regnier. The Paragon implementation of the NX message passing interface. *Proc. of the Scalable High Performance Computing Conference (SHPCC94)*, May 1994, pp. 184-190.

[18] Joseph Carbonaro and Frank Verhoorn. Cavallino: the Teraflops router and NIC. *Proc. of Hot Interconnects IV*, Stanford University, Palo Alto CA, August 1996, pp. 157-160.

[19] Steve Scott and Greg Thorson. Optimized routing in the Cray T3D. *Proc. of Parallel Computer Routing and Communication Workshop*, Seattle WA, May 1994, Springer-Verlag, pp. 281-294.

[20] Steve Scott and Greg Thorson. The Cray T3E network: adaptive routing in a high performance 3-d torus. *Proc. of Hot Interconnects IV*, Stanford University, Palo Alto CA, August 1996, pp. 147-156.

[21] Steve Scott. Synchronization and communication in the T3E multiprocessor. *Proc. of ASPLOS VII*, Cambridge MA, October 1996, pp. 26-36.

[22] William J. Dally, J. A. S. Fiske, John S. Keen, Richard A. Lethin, Michael D. Noakes, Peter R. Nuth, Roy E. Davison and Gregory A. Fyler. The Message-Driven Processor: a multicomputer processing node with efficient mechanisms. *IEEE Micro,* April 1992, pp. 23-39.

[23] Shekhar Borkar, R. Cohn, G. Cox, T. Gross, H. T. Kung, M. Lam, M. Levine, B. Moore, W. Moore, C. Peterson, J. Sussman, J. Sutton, J. Urbanski and J. Webb. Supporting systolic and memory communication in iWarp. *Proc. of the 17th Annual International Symposium on Computer Architecture,* Seattle WA, May 1990, pp. 70-81.

[24] Charles L. Seitz and Wen-King Su. The design of the Caltech Mosaic C Multicomputer. *Proc. of the 1993 Symposium on Integrated Systems,* Seattle WA, April 1993, pp. 1-22.

[25] Thorsten von Eicken, David E. Culler, Seth C. Goldstein and Klaus E. Schauser. Active messages: a mechanism for integrated communication and computation. *19th Annual International Symposium on Computer Architecture,* May 1992, pp. 256-266.

[26] Anant Agarwal et al. The MIT Alewife machine: a large-scale distributed-memory multiprocessor. *Proc. of Workshop on Scalable Shared Memory Multiprocessors,* Kluwer Academic Publishers, 1991.

[27] Robert Bedichek. Talisman: fast and accurate multicomputer simulation. *Proceedings of ACM SIGMETRICS '95,* Ottawa, Ontario, Canada, May 1995.

[28] Thomas E. Anderson, David E. Culler and David A. Patterson. A case for networks of workstations (NOW). *IEEE Micro*, Feb. 1995.

[29] Dana S. Henry and Christopher F. Joerg. A tightly coupled processor-network interface. *Proc. of the 5th ASPLOS*, October 1992, pp. 111-122.

[30] Greg M. Papadopoulos, G. A. Boughton, R. Greiner and M. J. Beckerle. *T: integrated building blocks for parallel computing. *Proc. of Supercomputing '93,* Portland OR, November 1993, pp. 624-635.

[31] Nanette J. Boden et al. Myrinet: a Gigabit-per-Second Local Area Network. *IEEE Micro*, February 1995. Also available on the World Wide Web through `http://www.myri.com/research/index.html` .

[32] Greg M. Papadopoulos. Personal communication. Supercomputing '93, Portland OR, November 1993.

[33] Gordon Bell. Ultracomputers: a Teraflop Before Its Time. *Communications of the ACM* 35(8), August 1992, pp. 26-47.

[34] John Palmer and Guy L. Steele Jr. Connection Machine Model CM-5 System Overview. *IEEE 4th Symposium on the Frontiers of Massively Parallel Computation,* 1992, pp. 474-483.

[35] Robert Bedichek. The Meerkat multicomputer: tradeoffs in multicomputer architecture. PhD dissertation, University of Washington, Dept. of CSE, Seattle WA, June 1994, UW-CSE-TR 93-09-05.

[36] Peter Steenkiste. A systematic approach to host interface design for high-speed networks. *IEEE Network*, July 1993, pp. 8-17.

[37] Peter Druschel, Mark Abbott, Michael Pagels and Larry Peterson. Network subsystem design. *IEEE Network*, July 1993, pp. 8-17.

[38] Chris Dalton, Greg Watson, David Banks, Costas Calamvokis, Aled Edwards and John Lumley. Afterburner: a network-independent card provides architectural support for high-performance protocols. *IEEE Network*, July 1993, pp. 36-43.

[39] Randy Osborne. A hybrid deposit model for low overhead communication in high speed LANs. Technical report TR 94-02, MERL — A Mitsubishi Electric Research Laboratory, June 1994.

[40] Randy Osborne, Qin Zheng, John Howard, Ross Casley, Doug Hahn and Takeo Nakabayashi. DART: a low overhead ATM network interface chip. *Proc. of Hot Interconnects '96*, Stanford University, August 1996, pp. 175-186.

[41] Richard Gillett and Richard Kaufmann. Experience using the first-generation Memory Channel for PCI network. *Proc. of Hot Interconnects '96*, Stanford University, August 1996, pp. 205-214.

[42] Mattias A. Blumrich, Kai Li, R. Alpert, Cezary Dubnicki, Edward W. Felten and J. Sandberg. A virtual memory-mapped network interface for the SHRIMP multicomputer. *Proc. of the 21st International Symposium on Computer Architecture*, Chicago IL, April 1994, pp. 142-153.

[43] Mattias A. Blumrich, Cezary Dubnicki, Edward W. Felten and Kai Li. Protected, user-level DMA for the SHRIMP network interface. *Proc. of High-Performance Computer Architecture 2,* San Jose CA, February 1996.

[44] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan-Kaufmann, 1990, chapter 9.7, pp. 535-537.

[45] Daniel Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz and M. S. Lam. The Stanford DASH multiprocessor. *IEEE Computer* 25(3), March 1992, pp. 63-79.

[46] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield and B. Smith. The Tera computer system. *Proc. of 1990 International Conference on Supercomputing,* Amsterdam, Netherlands, June 1990, pp. 1-6.

[47] Brian N. Bershad, David D. Redell and John R. Ellis. Fast mutual exclusion for uniprocessors. *Proc. of the 5th Annual International Conference on Architectural Support for Programming Languages and Operating Systems,* October 1992, pp. 223-233.

[48] H. Ishihata, T. Shimizu, M. Ikesaka, S. Inano and M. Ikesaka. Architecture of [the] highly parallel AP1000 computer. *Systems and Computers in Japan,* 24(7), 1993, pp. 69-77.

[49] K. Hayashi, T. Doi, T. Horie, Y. Koyanagi, O. Shiraki, N. Imamura, H. Ishihata and T. Shindo. AP1000+: architectural support of PUT/GET interface for parallelizing compiler. *SIGPLAN Notices*, 29(11), Nov. 1994, pp. 196-207.

[50] D. V. James et al. Distributed directory scheme: Scalable Coherent Interface. *IEEE Computer* 23(6), June 1990, pp. 74-77.

[51] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems* 7(4), November 1989, pp. 321-359.

[52] John B. Carter, John K. Bennett and Willy Zwaenepoel. Implementation and Performance of Munin. *Proc. of 13th ACM Symposium on Operating Systems Principles*, October 1991, pp. 92-103.

[53] B. N. Bershad, M. J. Zekauskas and W. A. Sawdon. Midway distributed shared memory system. *Proc. of COMPCON*, 1993, pp. 528-537.

[54] Mark D. Hill, James R. Larus and David A. Wood. Tempest: a substrate for portable parallel programs. *Proc. of COMPCON*, Mar. 1995.

[55] Christiana Amza et al. TreadMarks: shared memory computing on networks of workstations. *IEEE Computer*, February 1996, pp. 18-28.

[56] John Wilkes. Hamlyn: an interface for sender-based communications. Technical report HPL-OSR-92-13, Hewlett-Packard Company, HP Labs, Operating System Research Dept., November 1992.

[57] Greg Buzzard, David Jacobson, Scott Marovich and John Wilkes. Hamlyn: an high-performance network interface for sender-based memory management. *Proc. of Hot Interconnects III Symposium,* Stanford University, Palo Alto, CA, August 1995. Also available as technical report HPL-95-86, Hewlett-Packard Company, HP Labs, Computer Systems Laboratory, July 1995.

[58] Greg Buzzard, David Jacobson, Milton Mackey, Scott Marovich and John Wilkes. An implementation of the Hamlyn sender-managed interface architecture. *Proc. of the 2nd Symposium on Operating Systems Design and Implementation,* Seattle WA, October 1996.

[59] Ellen Spertus, S. C. Goldstein, K. E. Schauser, T. von Eicken, D. E. Culler and W. J. Dally. Evaluation of mechanisms for fine-grained parallel programs in the J-machine and the CM-5. *Proc. of the 20th Annual International Symposium on Computer Architecture,* May 1993, pp. 302-313.

[60] Neil McKenzie, Kevin Bolding, Carl Ebeling and Lawrence Snyder. Cranium: an interface for message passing on adaptive packet routing networks. *Proc. of Parallel Computer Routing and Communication Workshop*, Seattle WA, May 1994, Springer-Verlag, pp. 266-280.

[61] William Yost. Cost effective fault tolerance for network routing. Technical Report UW-CSE-95-03-03, University of Washington, Dept. of CSE, March 1995.

[62] Ludmila Cherkasova and Tomas Rokicki. Alpha message scheduling for optimizing communication latency in distributed systems. *Proc. of 13th IFAC Workshop on Distributed Computer Control Systems*, 1995.

[63] Ludmila Cherkasova, Vadim Kotov and Tomas Rokicki. The impact of message scheduling for packet switching interconnect fabrics. *Proc. of 29th Hawaii International Conference on System Sciences,* 1996.

[64] Charles P. Thacker, David G. Conroy and Lawrence C. Stewart. The Alpha Demonstration Unit: a high-performance multiprocessor. *Communications of the ACM* 36(2), February 1993, pp. 55-67.

[65] ChangYun Park. Predicting deterministic execution times of real-time programs. PhD dissertation, University of Washington, Department of CSE, Summer 1992, UW-CSE-TR 92-08-02.

[66] Kevin Bolding and William Yost. The Express Broadcast Network: a network for low-latency broadcast of control messages. *Proc. of 1995 Intl. Conf. on Algorithms and Architectures for Parallel Processing*, April 1995.

[67] Robert F. Cmelik and David Keppel. Shade: a fast instruction-set simulator for execution profiling. Technical report UW-CSE 93-06-06, June 1993. Also available as Sun Microsystems Laboratories technical report SMLI 93-12.

[68] Robert F. Cmelik and David Keppel. Shade: a fast instruction-set simulator for execution profiling. *Proceedings of ACM SIGMETRICS '94,* Nashville TN, May 1994, pp. 128-137.

[69] W. Culbertson, R. Amerson, R. Carter, P. Kuekes and G. Snider. The Teramac configurable custom computer. *Proc. of the International Society of Optical Engineering (SPIE) Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing*, Philadelphia, PA, Oct. 1995, pp. 201-209.

[70] Quickturn Design Systems, Inc. System Realizer Family page on the World Wide Web, accessed on October 17, 1996. URL: `http://www.quickturn.com-/prod/realizer/sysreal.htm`

[71] Nancy Kronenberg, Thomas R. Benson, Wayne M. Cardoza, R. Jagannathan and Benjamin J. Thomas. Porting OpenVMS from VAX to Alpha AXP. *Communications of the ACM* 36(2), February 1993, pp. 45-53.

[72] Motorola Inc. *The MC88100 RISC Microprocessor User's Manual,* second edition. Prentice-Hall, Englewood Cliffs NJ, 1990.

[73] Smaragda Konstantinidou and Lawrence Snyder. The Chaos router. *IEEE Transactions on Computers,* Dec. 1994.

[74] Melanie Fulgham and Lawrence Snyder. A study of chaotic routing with non-uniform traffic. Technical Report UW-CSE-93-06-01, University of Washington, Dept. of CSE, June 1993.

[75] Kevin Bolding, Sen-Ching Cheung, Sung-Eun Choi, Carl Ebeling, Soha Hassoun, Ton Ngo and Robert Wille. The Chaos router chip: design and implementation of an adaptive router. *Proceedings of IFIP Conf. on VLSI,* Sept. 1993.

[76] Parviz Kermani and Leonard Kleinrock. Virtual cut-through: a new computer communication switching technique. *Computer Networks* 3, 1979, pp. 267-286.

[77] Motorola Inc. *The MC88200 Cache/Memory Management Unit User's Manual,* second edition. Prentice-Hall, Englewood Cliffs NJ, 1990.

[78] M. Barnett, R. Littlefield, D. G. Payne and R. van de Geijn. Global combine algorithms on mesh architectures with wormhole routing. *Proc. of 7th IPPS*, Newport Beach CA, April 1993.

[79] David Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian and T. von Eicken. LogP: towards a realistic model of parallel computation. *Proc. of 4th Principles and Practices of Parallel Processing,* 1993, pp. 1-12.

[80] R. M. Karp, A. Sahay, E. Santos and K. E. Schauser. Optimal broadcast and summation in the LogP model. *5th Symp. on Parallel Algorithms and Architectures*, June 1993.

[81] Albert Alexandrov, Mihai Ionescu, Klaus E. Schauser and Chris Scheiman. LogGP: incorporating long messages into the LogP model: one step closer towards a realistic model for parallel computation. *7th Annual Symposium on Parallel Algorithms and Architectures* (SPAA'95), July 1995.

[82] S. S. Mukherjee, S. D. Sharma, Mark D. Hill, James R. Larus, Anne Rogers and Joel Saltz. Efficient support for irregular applications on distributed-memory machines. *Proc. of the 5th ACM SIGPLAN PPoPP,* July 1995.

[83] Manish Gupta, Sam Midkiff, Edith Schonberg, Ven Seshadri, David Shields, Ko-Yang Wang, Wai-Mee Ching and Ton Ngo. An HPF Compiler for the IBM SP2. *Proc. of Supercomputing '95*, San Diego CA, December 1995.

[84] E. Oran Brigham. *The Fast Fourier Transform.* Copyright 1974 Prentice-Hall, Englewood Cliffs NJ. ISBN 0-13-307496-X.

[85] J. W. Cooley and J. W. Tukey. An algorithm for the machine computation of complex Fourier series. *Mathematics of Computation* 19(90), 1965, pp. 297-301.

[86] Guy E. Blelloch, Charles Leiserson, Bruce M. Maggs, C. Greg Plaxton, Stephen J. Smith and Marco Zagha. A comparison of sorting algorithms for the Connection Machine CM-2. *Proc. of 3rd Annual ACM SPAA,* Hilton Head SC, 1991, pp. 3-16.

[87] Gilbert Strang. *Linear Algebra and Its Applications,* second edition. Academic Press, New York NY, 1980, ISBN 0-12-673660-X.

[88] L. F. Cannon. A cellular computer to implement the Kalman filter algorithm. PhD dissertation, Montana State University, 1969.

[89] H. T. Kung and C. E. Leiserson. Systolic arrays. In *Introduction to VLSI Systems*, C. A. Mead and L. A. Conway, Addison-Wesley, 1980, section 8.3, pp. 271-292.

[90] W. Daniel Hillis and Guy L. Steele, Jr. Data parallel algorithms. *Communications of the ACM* 29(12), December 1986, pp. 1170-1183.

[91] David G. Socha. Supporting fine-grain computation on distributed-memory parallel computers. PhD dissertation, University of Washington, Department of CSE, June 1991, UW-CSE-TR 91-07-01.

[92] Calvin Lin. The portability of parallel programs across MIMD computers. PhD dissertation, University of Washington, Department of CSE, December 1992, UW-CSE-TR 92-12-04.

[93] Calvin Lin and Lawrence Snyder. Accommodating polymorphic data decompositions in explicitly parallel programs. *Proceedings of the 8th International Parallel Processing Symposium*, April 1994, pp. 68-74.

[94] Thomas T. Kwan, Brian K. Totty and Daniel A. Reed. Communication and computation performance of the CM-5. *Proc. of Supercomputing 93*, Portland OR, November 1993, pp. 192-201.

[95] David E. Culler, Seth C. Goldstein, Klaus E. Schauser and Thorsten von Eicken. TAM: a compiler controlled threaded abstract machine. *J. of Parallel and Distributed Computing* 18(3), July 1993, pp. 347-370.

[96] Ted Kehl, Steve Burns and Chris Fisher. Self-tuned clocks and crystal clocks. Technical report TR 94-05-03, Dept. of CSE, University of Washington, May 1994.

[97] Scott Hauck. Asynchronous design methodologies: an overview. *Proceedings of the IEEE* 83(1), pp. 69-93, January 1995. Also available as University of Washington Dept. of CSE TR 93-05-07, May 1993.

[98] Greg Chesson. HIPPI-6400 overview. *Proc. of Hot Interconnects '96*, Stanford University, August 1996, pp. 121-128.

[99] Tenaski V. Ramabadran and Sunil S. Gaitonde. A tutorial on CRC computations. *IEEE Micro* 8(4), Aug. 1988, pp. 62-75.

[100] Guido Albertengo and Riccardo Sisto. Parallel CRC generation. *IEEE Micro* 10(5), Oct. 1990, pp. 63-71.

[101] Walter A. Hiatt. *Mitsubishi Electric Semiconductor Products.* Marketing literature from the Electronic Device Group (EDG) division of Mitsubishi Electric America (MEA). Presentation given on January 9, 1997 at MERL — A Mitsubishi Electric Research Laboratory, Cambridge MA.

[102] Mark Shand et al. The DEC PCI Pamette V1. World Wide Web site, `http://www.research.digital.com:80/SRC/pamette/`.

[103] M.J. Feeley, W.E. Morgan, F.H. Pighin, A.R. Karlin, H.M. Levy, and C.A. Thekkath. Implementing global memory management in a workstation cluster. *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[104] H.A. Jamrozik, M.J. Feeley, G.M. Voelker, J. Evans II, A.R. Karlin, H.M. Levy, and M.K. Vernon. Reducing network latency using subpages in a global memory environment. *Proceedings of the Seventh ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, October 1996.

Appendix A

THE Cranium APPLICATION PROGRAMMER'S INTERFACE

The Cranium application programmer's interface (API) is depicted in Figure A.1 by the dashed line between the application program and the entities with which it interacts directly. Message passing systems have three modes of operation: buffer allocation, data movement and synchronization. In many message passing environments, all three modes require a system call per operation (Figure A.1a). Under Cranium, two of these modes are optimized – the user program bypasses the operating system and interacts directly with the network when performing data movement and synchronization (Figure A.1b).

The application programmer interface consists of a set of data structures and their associated operations. Table A.1 shows a list of the data structures relevant to message passing under the Cranium interface. The first four structures in the list are the *microstructures* and the last five structures are the *macrostructures*. A microstructure is a set of bit fields packed into a 32-bit or 64-bit word. A macrostructure describes the organization at a higher level than the microstructures do. For instance, a microstructure may describe a single network interface register, whereas a macrostructure may describe a set of network interface registers.

For clarity and readability, all of the data structures use a consistent naming scheme. All of the structure names use the word "cranium" as a prefix, as shown in the first column of Table A.1. All of the elements within a structure use the same prefix. The naming convention for the prefix is "NIα" where α is a capital letter, as shown in the second column of the table. (NI stands for network interface.) For instance, all of the elements of the microstructure `cranium_command_word` use the prefix NIC in the names of the elements. The third column indicates the type of the structure, micro or macro. The fourth column is a terse description of the purpose of the structure. The following sections discuss the purpose of each structure in greater depth.

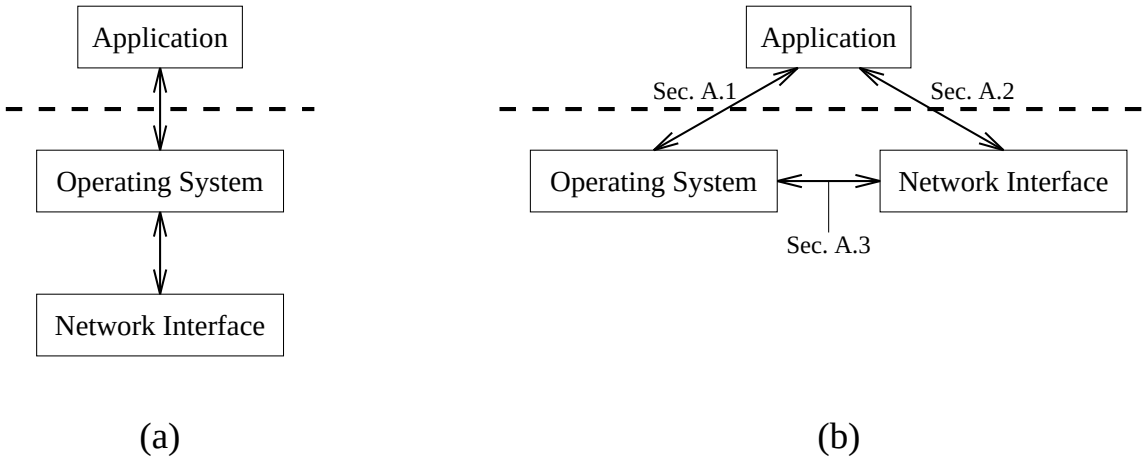


Figure A.1: Schematic view of the interactions among the user application program, the operating system and the network interface. The dashed line represents the application programmer interface. Subfigure (a) represents the traditional layered approach in which the network is a service provided by the operating system. Subfigure (b) represents the Cranium approach that allows the user program to access the network directly. Each arc is labeled with the number of the section in this chapter that describes the interactions between the two entities connected by the arc.

Table A.1: List of data structures to support message passing under Cranium

Structure name	Prefix	Type	Section	Description
`cranium_command_word`	NIC	micro	A.2.2	command word
`cranium_packet_header`	NIH	micro	A.2.1	packet header
`cranium_gen_intr_mask`	NII	micro	A.2.3	interrupt mask
`cranium_queue_ptr`	NIQ	micro	A.2.4	queue pointers
`cranium_buf_alloc`	NIB	macro	A.1.2	DMA buffer allocation
`cranium_init_info`	NIN	macro	A.1.1	initialization information
`cranium_OS_reg_map`	NIO	macro	A.3	operating system interface
`cranium_queue_packet`	NIP	macro	A.2.4	format of packet in user queue
`cranium_register`	NIR	macro	A.2.5	command register array

```
typedef struct cranium_init_info_s {
    unsigned  NIN_num_nodes;          /* total number of nodes p       */
    unsigned  NIN_my_node;            /* linear node ID of this node   */
    unsigned  NIN_rows;               /* height of node array          */
    unsigned  NIN_columns;            /* width of node array           */
      void *  NIN_reg_map_base;       /* base of Cranium register map  */
      void *  NIN_user_queue_base;    /* start of user queue           */
    unsigned  NIN_user_queue_length;  /* size of user queue in bytes   */
      void *  NIN_err_queue_base;     /* start of error queue          */
    unsigned  NIN_err_queue_length;   /* size of error queue in bytes  */
    unsigned  NIN_phys_node_map[p];   /* table of physical node IDs    */
} cranium_init_info_t;
```

Figure A.2: C structure describing the Cranium initialization information

A.1 Application program interface to the OS for message passing

A.1.1 Initialization information

The operating system call `cranium_get_init_info()` provides the user program with all the information it needs to access Cranium. The user program passes a pointer to block of user-space memory, and then the operating system fills in this memory with all the fields of the structure `cranium_init_info` displayed in Figure A.2. The first two fields of this structure, `NIN_num_nodes` and `NIN_my_node`, provide the total number of nodes in the system (hereafter also referred to as p) and the linear identifier of the node that the code is currently executing upon (also called the absolute or virtual node identifier). Nodes are numbered serially between 0 and $p-1$, inclusive; the value of `NIN_my_node` is guaranteed to be in this range.

The fields `NIN_rows` and `NIN_columns` provide topological information to the user about the layout of processes onto processing nodes. The product of these two fields equals `NIN_num_nodes`. This organization is used assuming that the network topology is a rectangle (two-dimensional); the extension to higher dimensions is straightforward.

The pointer `NIN_reg_map_base` is the user virtual base address of the register map in the network interface. The user program casts this pointer to the type `cranium_register_t *` (see Section A.2.5) and then performs structure accesses to read and write Cranium registers directly.

The pointers `NIN_user_queue_base` and `NIN_err_queue_base` point to the base address of the user queue and user error queue, respectively. They are cast to the type `cranium_queue_ptr_t *` and thereafter are used to access packets arriving in the queue (see Section A.2.4). The values `NIN_user_queue_length` and `NIN_err_queue_length` denote the sizes of the queues in bytes. Memory allocated to the queues by the operating system is guaranteed to be page-aligned and contain an integral number of MMU pages.

The array `NIN_phys_node_map_id` permits the user look up a physical node ID given the linear node ID. When packets are placed into the user queue, the header field contains the physical source ID; the table makes it possible to perform pattern matching to determine the linear node ID of the node that sent the packet. See Section A.2.1.

A.1.2 Allocation and deallocation of DMA buffers

Before the send channel can send any message or the automatic-receive channels can receive any message, the user program must allocate one or more message buffers for DMA by the network interface. These buffers are used with both the send channels and the automatic-receive channels; one buffer can be used as a source, a destination or both. A message buffer is the size of an MMU page; the semantics of Cranium requires that these message buffers are pinned into physical page frames by the operating system. The system call `alloc_cranium_buffer()` encompasses all of these operations: a buffer is allocated, its page is pinned in memory, the physical address of the frame is entered into the buffer map, and both the user virtual address and the handle of the buffer are returned to the user program. The operating system places the allocated pages onto contiguous user virtual page addresses; the effect is similar to standard contiguous memory allocators such as `malloc()` or `sbreak()`. Note that the physical DRAM backing the virtual store for the DMA buffers does not need to be allocated into contiguous physical frames.

`alloc_cranium_buffer()` takes two arguments. The first argument is a pointer to a block of user-space memory, which the operating system treats as an array of structs of type `cranium_buf_alloc` (Figure A.3). The size of the array is equal to the requested number of pages to allocate, given in the second argument. `alloc_cranium_buffer()` returns a boolean value representing success or failure. Al-

```
typedef struct cranium_buf_alloc_s {
      void *   NIB_base;        /* User virtual addr of pinned page   */
    unsigned   NIB_handle;      /* Cranium buffer handle              */
} cranium_buf_alloc_t;
```

Figure A.3: C structure for DMA buffer allocation for Cranium

location failure results from either an improper input parameter or a shortage of physical page frames. If the system call succeeds, each struct in the array of structs consists of a user virtual address pointer (`NIB_base`) and a handle (`NIB_handle`) for each DMA buffer allocated.

Deallocation of Cranium buffers is performed by the system call `free_cranium_buffer()`. The user program passes the handle of the buffer to be deallocated as the single input parameter. The operating system must verify that the buffer handle already has a valid mapping and has no outstanding message passing commands associated with it. Then the buffer can be safely deallocated.

A.1.3 Interrupt handler

The operating system call `set_user_intr_handler()` passes the address of a user-level function to the operating system. When an interrupt occurs that was caused by the network interface, the operating system passes control to the function at this address. If the user program does not initialize the user-level handler address, a Cranium-generated interrupt is treated as an abort signal (e.g. SIGABRT under Unix) and causes the user program to terminate.

A.2 Application program interface to Cranium

A.2.1 Packet header

The packet header (Figure A.4) is a fundamental data structure of the Cranium architecture. It contains the packet's location information for both its source and destination nodes (buffer address and offset), the context (user process ID) and redundancy information for error detection. The packet header is visible directly to

```
typedef struct cranium_packet_header_s {
    unsigned NIH_dest_ph_id:   16;  /* dest node id (physical)      */
    unsigned NIH_src_ph_id:    16;  /* source node id (physical)    */
    unsigned NIH_proc_id:      16;  /* user process id              */
    unsigned NIH_dest_chan:     8;  /* dest node recv channel       */
    unsigned NIH_src_chan:      8;  /* source node send channel     */
    unsigned NIH_sequence:     12;  /* seq number (memory offset)   */
    unsigned NIH_intr_every:    1;  /* interrupt flags              */
    unsigned NIH_intr_final:    1;  /* interrupt flags              */
    unsigned NIH_q_flag:        1;  /* user queue flag              */
    unsigned NIH_sys_flag:      1;  /* system queue flag (OS only)  */
    unsigned NIH_crc:          16;  /* redundancy code              */
} cranium_packet_header_t;
```

Figure A.4: C structure describing the Cranium packet header

the user program in the case of packets that arrive in the user queue. The size of the packet header is 96 bits (six 16-bit words).

The first two fields, `NIH_dest_ph_id` and `NIH_src_ph_id`, are 16-bit fields containing the physical identifiers of the destination and source nodes in the network. The values used in these fields are dependent on the implementation. For concreteness it is assumed that the network is a two-dimensional torus using chaotic routing. Physical node identification under chaotic routing is relative (differential). Both 16-bit fields are subdivided into two 8-bit fields describing the horizontal and vertical differences as signed two's complement quantities. From the point of view of the network, only the first 16-bit field `NIH_dest_ph_id` is meaningful for routing the packet. Normally when the packet is ejected, `NIH_dest_ph_id` is 0 to indicate that the packet has arrived at its destination. During testing, error detection and recovery, this field may be non-zero to indicate that the packet has not been delivered to its proper destination. (In some cases the receiving node is obligated to save the packet for later re-injection). It is often useful to the user program to determine the originating nodes of the packets in the user queue. Application programs use linear node identifiers for Cranium channel operations rather than the physical IDs in the packet header. The operating system provides a look-up table to the user program to allow the user to convert `NIH_src_ph_id` into a linear node ID (see `NIN_phys_node_map` in Section A.1.1).

NIH_proc_id is the process ID of the user program. The network interface does a simple comparison to verify that the process ID field in the packet matches the process ID of the NI context. Normally a match is found; a mismatch causes the NI to handle the packet specially. (See Section 8.2 for discussion of multiprogramming support in Cranium.)

NIH_dest_chan and NIH_src_chan identify the receive channel at the destination node and the send channel at the source node, respectively. Both fields are 8 bits to address up to 256 send channels and 256 receive channels; a typical implementation is likely to use smaller numbers than these, such as 64 send channels and 64 receive channels.

NIH_sequence is the sequence number of the packet. Since we are assuming a cache line size of 32 bytes, the offset in memory is determined by multiplying the sequence number by 32, i.e. shifting left by 5 bit positions. This 12-bit field allows a page size as large as 2^{17} or 128K bytes. For a page size of 8K, only 8 bits of the field are used. Normally the offset applies to both the source address and the destination address. In a variation on this scheme, NIH_sequence is used to index a table in the receiver's memory to look up the destination address. This variant scheme forms the basis for the gather-scatter extension – see Section 3.4.4 for a discussion of scatter-gather support in Cranium.

The remaining fields are a set of boolean flags and the CRC field. There are two maskable interrupt flags: NIH_intr_every and NIH_intr_final. When set, NIH_intr_every causes the receiving node to interrupt if its corresponding interrupt mask bit is set. When set, NIH_intr_final causes an interrupt at the receiving node if the packet is received by an auto-channel, the packet turns out to be the final packet of the transfer, and the corresponding mask bit for the channel is enabled. NIH_q_flag tells the receiving node to enter the packet into the user queue. NIH_sys_flag tells the receiving node to place the packet into the system queue.

A.2.2 Command word

A message is sent using the send channels or received using the automatic-receive channels by storing a 64 bit command word into a network interface register. The command word is a bit field described by the C language structure shown in Figure A.5.

```
typedef struct cranium_command_s {
    unsigned NIC_channel_reset:    1;  /* reset the channel           */
    unsigned NIC_buffer_handle:   15;  /* handle of local buffer      */
    unsigned NIC_remote_node_id:  12;  /* handle of remote node       */
    unsigned NIC_remote_channel:   8;  /* channel ID on remote node   */
    unsigned NIC_num_packets:     12;  /* number packets to send/rcv  */
    unsigned NIC_start_packet:    12;  /* first packet (send only)    */
    unsigned NIC_queue_flag:       1;  /* queue flag (send only)      */
    unsigned NIC_chan_intr_mask:   3;  /* channel interrupt mask      */
} cranium_command_t;
```

Figure A.5: C structure for the Cranium command word

The first field is the boolean flag `NIC_channel_reset`. When this bit is set, the channel is cleared (placed into an inactive state); all the other fields in the command are ignored.

The field `NIC_buffer_handle` is a buffer handle passed back to the user from the system call `alloc_cranium_buffer()`. This field is a maximum of 15 bits; up to 2^{15} or 64K distinct handles may be allocated. The handle indicates to the network interface to send or receive data in or out of its corresponding user buffer. The message passing semantics used by the send channels and the auto-receive channels is called unbuffered; no auxiliary buffering is involved in the transfer. If a send command is issued, followed by an immediate store operation to the same buffer memory, the value transmitted may be the old value or the new value. It is simple to avoid such race conditions by testing the status of the channel. In the context of the given example, the status of the channel performing the send operation can be tested to ensure that the store operation waits until after the send completes.

The field `NIC_remote_node_id` is the linear node identifier of the remote node. This field is a maximum of 12 bits; up to 2^{12} or 4K distinct remote nodes can be identified. For a send channel the field indicates the ID of the receiving node. The sender's network interface uses this linear node ID as an index into the node ID translation table. The value at this slot in the table is the corresponding physical ID to be copied into the packet header field `NIH_src_ph_id` of every packet in the transfer. For a receive command the field indicates the linear node ID of the sending

node. As in the sending case, this value is looked up in the node ID translation table using the linear node ID as the table index. The physical node ID from the table is matched against the source node field of each packet destined for this channel. Normally the IDs match and the packet is accepted. If a mismatch occurs then the NI signals a protocol error and interrupts the processor.

The field `NIC_remote_channel` identifies the remote channel at the remote node. For a send operation, this field is copied into the packet format field `NIH_dest_chan` for every packet in the transfer. For a receive operation, this field is compared against `NIH_dest_chan` in the header of incoming packets. Again, a mismatch in the channel numbers indicates a protocol error.

The field `NIC_num_packets` is used by both the sender and the receiver to specify the number of packets to send in a channel transfer. An additional field, `NIC_start_packet`, is used only with send channels to identify the sequence number of the first packet sent in the transfer. Ordinarily the first packet sent has a sequence number of zero. The packet number counter is incremented after each packet is sent; if the packet number is equal to the number of packets per page (e.g. PAGESIZE/32) then the count rolls over to zero. For example, if `NIC_num_packets` is 9, `NIC_start_packet` is 60 and the number of packets per page is 64, then the sequence numbers of packets to be sent are 60, 61, 62, 63, 0, 1, 2, 3 and 4, in that order.

The boolean field `NIC_queue_flag` is used only by the send channels; this bit is copied into the field `NIH_q_flag` in the header of every packet in the transfer, to steer packets to the user queue at the receiving node.

The three-bit field `NIC_chan_intr_mask` is used differently for sending and receiving. When used with the send channels, the first two bits are copied into the fields `NIH_intr_every` and `NIH_intr_final` in the header of every packet in the transfer. The third bit is the send channel's interrupt mask; if it is set, then an interrupt is raised when the send operation completes. When receiving, `NIC_chan_intr_mask` defines the interrupt mask for the selected auto-receive channel. There are two types of interrupt: interrupt on every packet received (Every) and interrupt on the final packet of the transfer (Final). Each auto-receive channel is in one of three states for each type of interrupt: ALWAYS, SOMETIMES and NEVER. ALWAYS means that the interrupt is always taken, regardless of the state of the corresponding interrupt flag

Table A.2: Interrupt mask state for a Cranium auto-receive channel

NIC_chan_intr_mask	type = Every	type = Final
000	NEVER	NEVER
001	NEVER	SOMETIMES
010	NEVER	ALWAYS
011	SOMETIMES	NEVER
100	SOMETIMES	SOMETIMES
101	SOMETIMES	ALWAYS
11x	ALWAYS	ALWAYS

in the packet header. NEVER means that this interrupt type is masked and ignored. SOMETIMES means that the interrupt is taken if and only if the corresponding bit in the packet header is set. With two types of interrupt (Every and Final) and three states for each type, the number of possible states of the receive channel interrupt mask is 3^2 or 9, but only 7 states are logically distinct – if every packet causes an interrupt (i.e. Every = ALWAYS) then the final packet of a transfer must also cause an interrupt (i.e. Final = ALWAYS is implied). These seven states are encoded by the three bits of `NIC_chan_intr_mask`. See Table A.2.

A.2.3 Cranium general interrupt mask

The general interrupt mask complements the set of auto-channel interrupt masks. It is general because it applies to all interrupt activity in the network interface, but it is local to a particular node and not globally exported to all nodes. The general interrupt mask has two parts: a read mask and a write mask. The read mask (i.e. the interrupt status register) contains all the information required to identify the event that caused the interrupt; the user program may perform different actions depending on the event that caused the interrupt. The write mask is set by the user program to tell the network interface what events are allowed to cause interrupts. The general interrupt mask supercedes the auto-channel write mask state; i.e. the appropriate fields of the general interrupt mask must be enabled before selected auto-channel interrupt events take place. However, certain types of events (in particular, error events) cannot be masked.

```
typedef struct cranium_gen_intr_mask_s {
    unsigned NII_intr_type:      2;  /* interrupt type          (r/o) */
    unsigned NII_error_mask:     3;  /* interrupt error mask    (r/o) */
    unsigned NII_channel_id:     8;  /* pointer to channel      (r/o) */
    unsigned NII_multiple:       1;  /* multiple interrupts     (r/o) */
    unsigned NII_send_complete:  1;  /* send chan xfer done     (r/w) */
    unsigned NII_recv_single:    1;  /* recv chan one pkt       (r/w) */
    unsigned NII_recv_complete:  1;  /* recv chan xfer done     (r/w) */
    unsigned NII_recv_user_q:    2;  /* recv user queue packet  (r/w) */
    unsigned NII_barrier_done:   1;  /* barrier complete        (r/w) */
} cranium_gen_intr_mask_t;
```

Figure A.6: C structure for the Cranium general interrupt mask

Table A.6 shows the C structure for the Cranium general interrupt mask. All fields labeled `r/o` are read-only and cannot be masked by the user program. All other fields (labeled `r/w`) refer to interrupt types that are maskable. The first field `NII_intr_type` identifies the type of interrupt – a send event, an auto-channel event, a queue event or a miscellaneous event. If the event was a send or auto-channel event, the field `NII_channel_id` identifies the associated channel number. The field `NII_error_mask` encodes seven kinds of error status:

- Channel was busy when another non-reset command was issued for that channel
- Packet arrived for an inactive auto-receive channel
- Auto-channel protocol error, if the remote node or remote channel declared in the receive command does not match the remote node or remote channel in the packet header
- Buffer handle or node handle in channel command references an uninitialized (NIL) physical entry in the look-up table
- User queue overflow, if a packet arrives when the queue is full

- User queue underflow, if the user program advances the head pointer when the queue is empty

- Illegal barrier synchronization operation

The field `NII_multiple` indicates that there was more than one event that caused the interrupt. The way that multiple interrupts are handled by the interface is left to the implementation. In one possible implementation, only the first event is noted in the interrupt mask; subsequent events are discarded, requiring the program to figure out the other events by some other means. In a second version, the events are queued; reading the mask multiple times reveals the entire sequence. Queuing interrupt events requires a more sophisticated hardware implementation, but it simplifies the software support required.

The final set of fields represents all the maskable interrupt types. `NII_send_complete` indicates that the send channel completed its operation. `NII_recv_single` indicates that a single packet was received by an auto-receive channel. `NII_recv_complete` indicates that an auto-receive channel completed its operation. `NII_recv_user_q` indicates that a packet was received by the user queue. The write mask requires this field to contain two bits to encode the states NEVER, SOMETIMES and ALWAYS; it is analogous with the write mask for the auto-receive channels. `NII_barrier_done` indicates that a global barrier synchronization operation completed.

A.2.4 Queue interface

The two queues that are directly accessible by the user program are the user queue and the user error queue. Packets arrive in the user queue when the boolean field `NIH_q_flag` is set in the packet header. A packet arrives in the user error queue as the result of a channel protocol error. In both cases, packets that arrive from the network are placed at the tail of the queue and the user program consumes packets from the head of the queue. At a high level, the user's interface to the queue consists of three parts: testing the presence of a packet in the queue, accessing the packet and discarding the packet. It is necessary for the user program to discard packets from the queue in order to free up space and permit the safe arrival of subsequent packets.

```
typedef struct cranium_queue_s {
    unsigned   NIQ_head_ptr:  32;  /* pointer to head of queue  */
    unsigned   NIQ_tail_ptr:  32;  /* pointer to tail of queue  */
} cranium_queue_t;
```

Figure A.7: C structure for the Cranium queue pointer interface

The implementation of the queues is a combination of hardware and software. Physically, queue memory is a circular buffer in DRAM that is pinned and mapped into the user's address space. The head and tail pointers to the queue are realized in the network interface hardware; these pointers can be copied into software by loading the network interface register containing the queue pointers (Figure A.7). When a packet arrives, the network interface writes the packet payload into one cache line and its auxiliary information into a second cache line, consisting of the packet header, a timestamp for performance analysis and a presence flag, set to a non-zero value to indicate that the packet has arrived.

Once the user program has loaded the head pointer, it can poll the presence field in queue memory to detect packet arrivals. A logically equivalent test for packet arrivals is to load the queue pointers and compare them; if the head and queue pointers are not equal, the queue is not empty. However, the presence field technique is more efficient than loading the queue pointers. Every access to a network interface register generates traffic on the memory bus. The presence field is just a memory location that can be cached reduce bus traffic. After the first access to queue memory, testing the presence field generates no activity on the memory bus because the line is loaded into the cache. The arrival of a packet causes an invalidation or update of the line to keep cache and DRAM coherent (see Sections 2.1.2 and 3.4.2).

After the presence of a packet has been detected, the user program accesses the packet directly in memory. Figure A.8 shows the format of packets placed into the queues by the network interface. (The type `uint32` is a 32-bit unsigned integer; the type `uint64` is a 64-bit unsigned integer, equivalent to `unsigned long long` in many C compilers.) Because queue memory is just standard cacheable DRAM, the user program is free to read or modify any of the fields in the queue packet structure. In particular, there are twelve bytes in the structure that are used only to pad the

```
typedef struct cranium_queue_packet_s {
    uint64                  NIP_payload[4];  /* 32 bytes                    */
    uint32                  NIP_flag;        /*  4 bytes (presence flag)    */
    uint32                  NIP_pad[3];      /* 12 bytes (app defined)      */
    uint32                  NIP_timestamp;   /*  4 bytes (arrival time)     */
    cranium_packet_header_t NIP_header;      /* 12 bytes                    */
} cranium_queue_packet_t;
```

Figure A.8: C structure for the format of packets in the queue

structure to a total of 64 bytes (two cache lines); this extra space can be used by the application as scratch memory. The user program sets the presence field NIP_flag to zero to indicate that the packet is discarded. The network interface never reads this memory location, but it writes a non-zero value to it to indicate the presence of a subsequent packet. (To ensure proper operation in the presence of an optimizing compiler, every field in the structure should be declared `volatile`).

Two actions are required for the user program to discard a packet from the queue. First, it clears the presence field in queue memory. Second, it advances the head pointer in the network interface. The head pointer is advanced by reading from a second network interface register for the queue. As a side effect, the new value of the head pointer is returned. Since the buffer is circular, the new value may be either 64 bytes beyond the previous head pointer or the base address of the queue for the wraparound case. Computing this value in hardware is more efficient than in software, as the latter would require a test for the wrap condition and a branch.

There are two types of exceptional conditions for the queue: underflow and overflow. The network interface hardware detects both types of exceptions and thereupon signals the processor with an interrupt. The network interface compares the head and tail pointers for every packet arrival and head pointer advance. Underflow results when the head pointer and tail pointer were equal (i.e. the queue is empty) and the user advances the head pointer. Overflow results when a packet arrives and advancing the tail pointer causes it to become equal with the head pointer.

```
typedef struct cranium_register_s {
    cranium_command_t    NIR_send_channel[64];  /* read/write  */
    cranium_command_t    NIR_recv_channel[64];  /* read/write  */
    uint64_t             NIR_send_chan_busy;    /* read only   */
    uint64_t             NIR_recv_chan_busy;    /* read only   */
    cranium_queue_ptr_t  NIR_user_queue[2];     /* read only   */
    cranium_queue_ptr_t  NIR_error_queue[2];    /* read only   */
    uint64_t             NIR_barrier;           /* read/write  */
} cranium_register_t;
```

Figure A.9: C structure for the Cranium register map

A.2.5 Cranium register map

The Cranium register map consists of all the network interface registers that the user program is permitted to access directly (Figure A.9). There are 135 registers in all. For compatibility with busses that require all memory accesses to be cache-line aligned, all registers are placed on 32-byte boundaries. 64 registers access the send channels and an additional 64 access the automatic-receive channels; the nth register corresponds to the nth channel. All the send and auto-receive registers are read/write. As write registers, they take a command word as described in Section A.2.2. As read registers, they return the number of packets remaining in the transfer; a zero value means that the operation has completed. The read-only registers `NIR_send_chan_busy` and `NIR_recv_chan_busy` are bit vectors; if bit n is set then channel n is busy. The bit vectors reduce the number of network interface accesses when the user program is waiting for more than one channel operation to complete.

There are two read-only registers apiece for the user queue and the user error queue. Reading the first register returns the queue pointer microstructure; reading the second register has the side effect of incrementing the head pointer before returning the pointer information (see Section A.2.4).

The final register supports global barrier synchronization. Global barrier synchronization is a useful abstraction in parallel programming, particularly in the context of data-parallel programs. The basic idea is to guarantee that all threads of executiong have reached a particular statement in the program. In the case of message passing using unbuffered communication, the barrier ensures that all receivers have

Table A.3: Barrier state transition table. The states are I (idle), W (wait), C (complete) and A (abstain). The events that can cause a state change are generated by one of three sources – the processor (write 0 or write 1), the network or a timer. The first column represents the input state. Values in the other columns represent the next state. The pseudo-state *e* represents an error condition.

Input state	Event			
	write 0	write 1	net event	timer event
I	A	W		
W	*e*	*e*	C	
C	*e*	*e*		I
A	*e*	W		

been initialized before any node performs a send operation.

A node can be in one of four states with respect to barrier synchronization (Table A.3). The initial state is idle (I). Entering the barrier causes the node to move to one of two states: barrier-wait (W) where it waits for the barrier to complete, or barrier-complete (C) if the node was the last node to enter the barrier. If a node is in the barrier-wait state, it remains there until all nodes have entered the barrier, whereupon it moves to the barrier-complete state. After a short time delay, the node moves from barrier-complete to idle. A fourth state is called abstain (A), in which the node opts to not participate in barrier processing. A node can move from idle to abstain, and from abstain to barrier-wait by entering the barrier.

See Section 6.1.3 for a related discussion on the implementation of fast barrier synchronization.

A.3 Interface between Cranium and the operating system

The operating system interface to Cranium provides all the necessary information for the Cranium device driver developer. This level of detail is hidden from application programs and is not necessary knowledge for parallel program development. However it is necessary for the design and development of the network interface itself. This part of the interface between the network interface hardware and the software is

```
typedef struct cranium_OS_reg_map_s {
    cranium_register_t  NIO_user_reg_map;      /* OS copy of user reg map      */
              unsigned  NIO_process_ID;        /* OS process ID for the user   */
              unsigned  NIO_flow_control;      /* packet flow in/out of NI     */
                void *  NIO_node_map_ptr;      /* user node map pointer        */
                void *  NIO_buffer_map_ptr;    /* user buffer map pointer      */
                void *  NIO_user_q_ptr[4];     /* user queue pointers          */
                void *  NIO_error_q_ptr[4];    /* user error queue pointers    */
                void *  NIO_sys_q_ptr[4];      /* sys queue pointers           */
                void *  NIO_syserr_q_ptr[4];   /* sys error queue pointers     */
                void *  NIO_save_context;      /* save user context            */
                void *  NIO_restore_context;   /* restore user context         */
              unsigned  NIO_blt_status;        /* block transfer status        */
} cranium_OS_reg_map_t;
```

Figure A.10: C structure describing the Cranium interface to the operating system

important for making a system that is both secure and usable. The operating system must be able to perform three operations: initialize a context for a newly running user program, terminate a user program context and switch user program contexts. Performing a context switch requires the internal state of the network interface to be saved and a saved user state to be restored.

The structure of the operating system interface is described by Figure A.10. At boot time, the operating system is given the physical base address of Cranium; registers are accessed by reading and writing the elements of this structure. The OS uses a separate window into the Cranium user message passing registers, in addition to any dedicated send registers dedicated to the OS that are provided by the particular hardware implementation. When the OS send a message using this separate window into the Cranium register set, the packet header flag `NIH_sys_flag` is set by default (see Section A.2.1). An important control that the operating system has over the network is the control of packet flow in or out of the processing node. Writing to the register `NIO_flow_control` sets the flow control: NORMAL (packets flow in and out), FLUSH (packets flow in but not out) and FREEZE (no packets flow in or out). The other registers are used to initialize, save and restore the user context, and are explained below.

A.3.1 Initializing and terminating a user message-passing context

The operating system must set up the message passing context for the user program and then load the pointer registers inside Cranium to point to the proper memory locations. The message passing context includes the buffer map, the node map, the user queue and the user error queue and all the associated pointers inside Cranium. First the OS sets the interface's flow control to the FREEZE state. The OS clears the buffer map by setting all entries in the buffer map to the value NIL. The node map is initialized with all the topology information to map the linear sequence 0 through $(p-1)$ into the physical node IDs. If the network uses relative physical node ID mapping (as is the case with chaotic routing) then each node receives a custom version of this map. Entries in the node map with index p or greater are also set to NIL. This map is also exported to the user program via the system call `cranium_get_init_info()` (see Section A.1.1). There are four queue contexts: the user queue, the user error queue, the system queue and the system error queue. Each queue context requires four pointers in the network interface hardware: the beginning of queue memory, the end of queue memory, the head pointer and the tail pointer. When a queue is initialized, the head and tail pointer are set equal to the beginning of that queue's memory. The remaining steps of initializing the network interface for a new user context is setting the process ID register `NIO_process_ID` and setting all send channels and all auto-receive channels to an idle state.

A user message-passing context is terminated by first flushing the network to deliver all the packets intended for that user process, then clearing all the channel operations, then freezing the interface. At this point a new user context can be started up or a ready context can be made runnable, as shown in the following section under context switching.

A.3.2 Context switch

A context switch in Cranium consists of the following sequence of operations:

- The operating system sets the flow control to FREEZE.

- The OS writes a memory address to the NI register `NIO_save_context`. The network interface responds by executing a block write of all its internal user

state beginning at this address. (The size and format of this information is left up to the implementation.) The OS tests the completion of the block transfer by polling the register `NIO_blt_status`.

- The OS writes a memory address to the NI register `NIO_restore_context`. This command causes the network interface to perform a block read from memory to restore a previously saved user state. Again, `NIO_blt_status` provides the completion status of the block transfer.

- The operating system sets the flow control to NORMAL.

Note that part of the message passing state is in user-space memory (such as the user queue). Since user memory is saved in the normal course of a context switch (e.g. user-owned memory becomes paged or swapped out), there is no special command or mode needed in the network interface to support this part of saving and restoring the message-passing state.

There are other details not noted here that are left to the implementation. See Sections 3.4.3 and 7.6.2 for more discussion on hardware support in Cranium for multiple user contexts.

A.4 Examples of message passing using the Cranium API

Figures A.11 through A.14 present a series of code modules that access the Cranium API; all the modules are linked together to form a complete application program. Figure A.11 contains the top-level function `main()`; it in turn calls the subfunctions `send_message()`, `receive_message_queue()` and `receive_message_auto()` shown in Figures A.12, A.13 and A.14, respectively. All the code examples are in ANSI C. Code fragments such as pointer casts and function prototype declarations are omitted for clarity. The purpose of these figures is to illustrate the basic functionality of the message passing system, rather than provide literal examples of code that would appear in an implementation tuned for the highest performance.

A.4.1 Initialization

When a parallel application program starts up, it begins executing a separate thread of execution on every processing node in its partition. In the example in Figure A.11, all nodes start executing by entering the function `main()` at approximately the same time. Two local functions are called: `set_cranium_gen_intr_mask()` to set the general interrupt mask and `set_cranium_user_intr_handler()` to set up the user's interrupt handler[1]. At each node the application program executes the system call `cranium_get_init_info()` to return the initialization information. Fields within this structure identify the node number of each thread of execution. This information allows the code running on a particular node to identify itself as the source or destination of a message. In the block of code labeled "Buffered communication," every node running a copy of the program performs one of three actions. The code running on node 0 calls the function `send_message()`. The code running on node 3 calls the function `receive_message_queue()`. All other threads take the default action, which is to do nothing. The effect is that a message is sent from node 0 to node 3. In the following code block labeled "Unbuffered communication" the same division of labor occurs: node 0 sends a message to node 3, but the automatic-receive channels are used instead of the user queue. In this code block a barrier synchronization is performed. The default nodes (i.e. nodes other than nodes 0 and 3) enter the barrier by calling the function `barrier_synch()`; each node spin-waits until all nodes have entered the barrier. Both `send_message()` and `receive_message_auto()` contain calls to `barrier_synch()`, so that all nodes will eventually enter the barrier.

A.4.2 Sending a message

Figure A.12 displays the function `send_message()`. This function sends a message consisting of a single packet to the node whose linear ID is `dest`. The flag `qflag` determines the destination of the packet when it arrives at the receiving node (either an auto-channel or the user queue). The first action taken in the function is the allocation of a message buffer; if this allocation step fails then the entire function returns FAILURE. The next action is to place information into the message buffer.

---

[1] The message passing examples presented here do not use interrupts. For brevity, the code implementing these two functions is omitted.

```
/*
**   A separate copy of this code runs on each processing node
*/
#include <craniumAPI.h>
void main(void) {
    cranium_init_info_t info;

    set_cranium_gen_intr_mask();
    set_cranium_user_intr_handler();

    cranium_get_init_info(&info);
    if (info.NIN_num_nodes < 4) {
       error("Sorry, not enough nodes");
       exit();
    }

/*  Buffered communication */
    switch (info.NIN_my_node) {
    case  0:
       send_message(&info, 3, QUEUE);
       break;
    case  3:
       receive_message_queue(&info, 0);
       break;
    default:
       ; /* do nothing */
    }

/*  Unbuffered communication */
    switch (info.NIN_my_node) {
    case  0:
       send_message(&info, 3, AUTO);   /* contains barrier synch */
       break;
    case  3:
       receive_message_auto(&info, 0); /* contains barrier synch */
       break;
    default:
       barrier_synch();
    }
}
```

Figure A.11: Example code for initialization and the two communication examples

```
/*
**  Simple message sending example
*/
int send_message(cranium_init_info *P_info, int dest, int qflag) {
    cranium_command_t       cmd;
    cranium_buf_alloc_t     sbuf;
    cranium_register_t      *cranium;

    if (!alloc_cranium_buffer(&sbuf, 1))
       return FAILURE;

    strcpy(sbuf.NIB_base, "Hello from Node 0\n");

    cmd.NIC_buffer_handle   = sbuf.NIB_handle;
    cmd.NIC_channel_reset   = 0;     /* don't reset the channel          */
    cmd.NIC_remote_node_id  = dest;  /* send to node 'dest'              */
    cmd.NIC_remote_channel  = 6;     /* send to rcvr's channel 6         */
    cmd.NIC_num_packets     = 1;     /* send one packet                  */
    cmd.NIC_start_packet    = 0;     /* first cache line in buffer       */
    cmd.NIC_queue_flag      = (qflag == QUEUE); /* set rcvr type         */
    cmd.NIC_chan_intr_mask  = 0;     /* don't use interrupt flags        */

    if (qflag != QUEUE)              /* Only if sending to auto-channel  */
       barrier_synch();              /* Synch occurs BEFORE send cmd     */
    cranium = P_info->NIN_reg_map_base;
    cranium->NIR_send_channel[2] = cmd;              /* Execute send cmd  */
    while (cranium->NIR_send_channel[2])
       ;                             /* wait until send cmd completes    */
    dealloc_cranium_buffer(sbuf.NIB_handle);
    return SUCCESS;
}
```

Figure A.12: Example code to send a message

In this simple example, the text string `"Hello from Node 0\n"` is copied into the message buffer. Since this string is less than 32 characters (bytes) in length, it is sent to the destination node in a single packet.

The next eight lines of code describe how the command word is built by setting up the buffer handle, remote node ID, remote channel, number of packets, start packet, queue flag and send channel interrupt mask. This code sends the message using send channel 2. The message is received into the user queue if `qflag` is set to the constant `QUEUE`, otherwise the message is received into auto-receive channel 6 at the destination node. The choice of channel is arbitrary in this case – any send channel and any receive channel could be used as long as the transfer information at the receiver matches that of the sender. If `qflag` is set to AUTO, then the barrier synchronization function is called. The call could come anywhere in the function as long as it occurs *before* executing the send command.

After both the command word and the message buffer are initialized, the message is ready to send. The user virtual address attached to the physical base address for the Cranium command register comes from the field `NIN_reg_map_base` in the initialization struct. The line with the comment "Execute send cmd" causes the command word to be written into Cranium, thereby initiating the send command. The two subsequent lines of code test the completion status by loading the channel's read register indicating the remaining number of packets to be sent, and looping back if the value is not zero. When the send command has completed, the message buffer can be safely deallocated.

A.4.3 Receiving a message

Figure A.13 shows the implementation of the function `recv_message_queue()`. The semantics of this function are to wait for a single packet to arrive, and if the node ID of the sender of the packet matches the parameter `src`, then the contents of the packet are printed as a text string. The node ID comparison is included only for illustration because it is unnecessary – there is only one source node sending packets in this example.

The block of code labeled "Initialization" shows the initialization of the queue head pointer `P_user_q`; this code is executed no more than once at each node every time the program runs. The queue pointer variable must be either be static or global

```
/*
**  Simple message receiving example using the queue
**  Wait for a packet to arrive; if it's from 'src', print it as text
*/
int recv_message_queue(cranium_init_info *P_info, int src) {
    int                             retflag = FAILURE;
    static cranium_queue_packet_t   *P_user_q;
    static int                      initflag = TRUE;

    /* Initialization */
    if (initflag == TRUE) {         /* init pointer to user queue        */
       P_user_q = P_info->NIN_user_queue_base;
       initflag = FALSE;
    }

    while (P_user_q->NIP_flag == 0)  /* Wait for a packet to arrive       */
       ;
/*
**  Compare node ID in packet header against 'src' ID
*/
    if (P_user_q->NIP_header.NIH_src_ph_id == P_info->NIN_phys_node_map[src]) {
       retflag = SUCCESS;
       printf(&(P_user_q->NIP_payload[0]));
    }

    P_user_q->NIP_flag = 0;          /* mark packet for recycling         */
    P_user_q = P_info->NIN_reg_map_base->NIR_user_queue[1].NIQ_head_ptr;
                                     /* advance user queue pointer        */
    return retflag;                  /* return SUCCESS or FAILURE         */
}
```

Figure A.13: Example code to receive a message into the user queue

so that its value persists beyond the function's scope.

The presence of a packet in the user queue is determined by polling the flag field in queue memory. After determining that a packet has arrived, the program compares the node ID in the packet header against the expected node ID passed in the parameter `src`. The translation table `P_info->NIN_phys_node_map[]` is used to convert the linear ID in `src` into a physical ID for comparison against the physical ID in the packet header. If the match succeeds then `printf()` is called the address of the packet payload to print the packet as a text string. After the last use of the payload occurs, the program marks the packet for recycling by writing a zero into the flag and advancing the queue head pointer. The queue head pointer is updated to prepare for subsequent calls to `recv_message_queue()`.

Figure A.14 shows the implementation of the function `recv_message_auto()`. Its implementation is quite similar that of `send_message()` – a message buffer is allocated, the channel command is composed, the command is executed and the program waits until the message arrives before returning from the function. Note that the barrier synchronization function call must occur AFTER the receive command executes.

A.4.4 Discussion

This section displays a series of figures describing examples of message passing in parallel application programs using the Cranium API. These functions are intended only for illustration and do not achieve the highest performance. The basic strategy for performance optimization is to eliminate operations that increase the overhead unnecessarily. For instance, the implementations presented here allocate and deallocate message buffers every time a message is sent or is received with an automatic-receive channel. A more streamlined implementation separates message buffer allocation from the send and receive code. Another topic that was omitted for simplicity concerns error detection and recovery. In both code examples for message receive, if no packet ever arrives then the program waits forever. An implementation including a timeout or a maximum loop count would terminate the function gracefully and yield a parallel application program that is easier to debug.

```
/*
**  Simple message receiving example using auto channels
*/
int recv_message_auto(cranium_init_info *P_info, int src) {
    cranium_command_t       cmd;
    cranium_buf_alloc_t     rbuf;
    cranium_register_t      *cranium;

    if (!alloc_cranium_buffer(&rbuf, 1))
       return FAILURE;

    cmd.NIC_buffer_handle   = rbuf.NIB_handle;
    cmd.NIC_channel_reset   = 0;    /* don't reset the channel      */
    cmd.NIC_remote_node_id  = src;  /* recv from node 'src'         */
    cmd.NIC_remote_channel  = 2;    /* sender sends from channel 2  */
    cmd.NIC_num_packets     = 1;    /* recv one packet              */
    cmd.NIC_start_packet    = 0;    /* first cache line in buffer   */
    cmd.NIC_chan_intr_mask  = 0;    /* don't use interrupt flags    */

    cranium = P_info->NIN_reg_map_base;
    cranium->NIR_recv_channel[6] = cmd; /* recv into channel 6      */
    barrier_synch();                /* occurs AFTER recv command    */
    while (cranium->NIR_recv_channel[6])
       ;                          /* wait until recv cmd completes  */

    printf(rbuf.NIB_base);

    dealloc_cranium_buffer(rbuf.NIB_handle);
    return SUCCESS;
}
```

Figure A.14: Example code to receive a message into an automatic channel

Appendix B

MEASUREMENTS OF PROGRAMS IN THE PARALLEL BENCHMARK SUITE

This appendix contains tables of figures that were produced by running the combined Talisman/Chaos simulator (see Chapter 5) on the benchmark programs described in Section 6.2: dense matrix multiplication (DMM), fast Fourier transform (FFT), Gaussian elimination (Gauss), Jacobi successive over-relaxation (Jacobi) and parallel bucket sort (Sort). Measurements of DMM are in Table B.1 and the corresponding derived values are in Tables B.2 and B.3. Measurements of FFT are in Table B.4 and the corresponding derived values are in Tables B.5 and B.6. Measurements of Gauss are in Table B.7 and the corresponding derived values are in Tables B.8 and B.9. Measurements of Jacobi are in Tables B.10 and B.13 and the corresponding derived values are in Tables B.11, B.12, B.14 and B.15. Measurements of Sort are in Table B.16 and the corresponding derived values are in Tables B.17 and B.18. Section 6.2.4 provides the background for the tables of measurements; Sections 6.2.5 through 6.2.8 provide the background for the tables of derived figures. The following list reviews the parameters, measurements and derived values that appear in the tables.

- p (parameter): the number of nodes used in the execution of the simulator.
- DRAM latency (parameter): the model for the memory system used in the execution of the simulator. The optimistic model assumes zero-cycle DRAM access latency; the pessimistic model assumes 10 cycle DRAM access latency. See Section 6.2.4.
- Zero communication cost (parameter): assumed for computing the hypothetical maximum IPC, speedup and efficiency.
- $\mathrm{I}_{\mathrm{comp}}(1)$ (measurement): the number of instructions executed in the uniprocessor version of the benchmark.
- $\mathrm{B}_{\mathrm{comm}}(p)$ (measurement): the total number of bytes communicated in the benchmark per node averaged over all nodes.
- $\mathrm{C}_{\mathrm{comp}}(p)$ (measurement): the number of clock cycles it takes to execute only the computation part of the non-overlapping version of the benchmark.

- $C_{comm}(p)$ (measurement): the number of clock cycles it takes to execute only the communication part of the non-overlapping version of the benchmark.
- $C_{ol}(p)$ (measurement): the number of clock cycles it takes to execute the overlapping parallel version of the benchmark.
- $IPC_{max}(p)$ (derived value): the aggregate number of instructions executed per cycle if the cost of communication were zero.
- $SU_{max}(p)$ (derived value): the relative speedup compared to the serial version of the benchmark if the cost of communication were zero.
- $Eff_{max}(p)$ (derived value): equal to $SU_{max}(p)/p$.
- $SU_{act}(p)$ (derived value): the actual relative speedup compared to the serial version of the benchmark.
- $Eff_{act}(p)$ (derived value): equal to $SU_{act}(p)/p$.
- $S_{comm}(p)$ (derived value): the significance of the communication cost, equal to $p \cdot C_{comm}(p)/C_{comp}(1)$.
- $TP(p)$ (derived value): simple throughput, equal to $B_{comm}(p)/C_{comm}(p)$.
- $TP_{pct}(p)$ (derived value): simple throughput times 100 divided by the upper bound on throughput.
- $TP_{eff}(p)$ (derived value): effective throughput, equal to $B_{comm}(p)/C_{diff}(p) = B_{comm}(p)/(C_{ol}(p) - C_{comp}(p))$.

Measurements in *slanted* type were measured using the serial version of the benchmark. Measurements in normal type were measured using the non-overlapping parallel version of the benchmark. Measurements in **bold** type were measured using the overlapping parallel version of the benchmark.

The suffix "(p)" is omitted in the tables to conserve horizontal space.

Table B.1: Measurements of dense matrix multiplication (DMM).

p	$I_{comp}(1)$	B_{comm}	DRAM latency = 0			DRAM latency = 10		
			C_{comp}	C_{comm}	C_{ol}	C_{comp}	C_{comm}	C_{ol}
1	*1344023*	0	*7904960*	0	*7904960*	*7904960*	0	*7904960*
4	*1344023*	49152	1751842	19091	**1755403**	1752018	30755	**1773885**
8	*1344023*	57344	877756	23129	**886058**	877730	36261	**908013**
16	*1344023*	61440	454449	25441	**466555**	454334	40700	**490342**
32	*1344023*	63488	245952	30134	**258173**	245837	45672	**282422**
64	*1344023*	64512	144550	38706	**156904**	144485	52228	**177890**

Table B.2: Calculated instructions per cycle, speedup and efficiency of DMM.

p	Zero communication cost			DRAM latency = 0		DRAM latency = 10	
	IPC_{max}	SU_{max}	Eff_{max}	SU_{act}	Eff_{act}	SU_{act}	Eff_{act}
1	0.170	1.00	1.00	1.00	1.00	1.00	1.00
4	0.767	4.51	1.13	4.50	1.13	4.46	1.11
8	1.53	9.01	1.13	8.92	1.12	8.71	1.09
16	2.96	17.4	1.09	16.9	1.06	16.1	1.01
32	5.47	32.1	1.00	30.6	0.957	28.0	0.875
64	9.30	54.7	0.854	50.4	0.787	44.4	0.694

Table B.3: Significance of communication, actual throughput, percentage of ideal throughput and effective throughput of DMM.

p	DRAM latency = 0				DRAM latency = 10			
	S_{comm}	TP	TP_{pct}	TP_{eff}	S_{comm}	TP	TP_{pct}	TP_{eff}
4	0.010	2.58	88.5	13.8	0.016	1.60	89.9	2.25
8	0.023	2.48	85.2	6.91	0.037	1.58	89.0	1.89
16	0.051	2.42	83.0	5.08	0.082	1.51	84.9	1.71
32	0.122	2.11	72.4	5.20	0.185	1.39	78.2	1.74
64	0.313	1.67	57.3	5.22	0.423	1.24	69.5	1.93

Table B.4: Measurements of fast Fourier transform (FFT).

p	$I_{comp}(1)$	B_{comm}	DRAM latency = 0			DRAM latency = 10		
			C_{comp}	C_{comm}	C_{ol}	C_{comp}	C_{comm}	C_{ol}
1	*181714*	0	*579402*	0	*579402*	*579402*	0	*579402*
4	*181714*	8192	142344	3358	**145275**	142526	5265	**147308**
8	*181714*	6144	72395	3305	**74767**	72625	4313	**76444**
16	*181714*	4096	37438	3124	**39112**	37430	3901	**40228**
32	*181714*	2560	19780	2915	**20895**	19751	3574	**21561**
64	*181714*	1536	10922	2618	**11747**	10936	3028	**12185**

Table B.5: Calculated instructions per cycle, speedup and efficiency of FFT.

p	Zero communication cost			DRAM latency = 0		DRAM latency = 10	
	IPC_{max}	SU_{max}	Eff_{max}	SU_{act}	Eff_{act}	SU_{act}	Eff_{act}
1	0.314	1.00	1.00	1.00	1.00	1.00	1.00
4	1.28	4.07	1.02	3.99	0.997	3.93	0.983
8	2.51	8.00	1.00	7.75	0.969	7.58	0.947
16	4.85	15.5	0.967	14.8	0.926	14.4	0.900
32	9.19	29.3	0.919	27.7	0.867	26.9	0.840
64	16.6	53.1	0.829	49.3	0.771	47.6	0.743

Table B.6: Significance of communication, actual throughput, percentage of ideal throughput and effective throughput of FFT.

p	DRAM latency = 0				DRAM latency = 10			
	S_{comm}	TP	TP_{pct}	TP_{eff}	S_{comm}	TP	TP_{pct}	TP_{eff}
4	0.023	2.44	83.8	2.79	0.036	1.56	87.6	1.71
8	0.046	1.86	63.9	2.59	0.060	1.42	79.8	1.61
16	0.086	1.31	45.1	2.45	0.108	1.05	59.0	1.46
32	0.161	0.878	30.2	2.30	0.197	0.716	40.4	1.41
64	0.289	0.587	20.2	1.86	0.334	0.507	28.7	1.23

Table B.7: Measurements of Gaussian elimination (Gauss).

p	$I_{comp}(1)$	B_{comm}	DRAM latency = 0			DRAM latency = 10		
			C_{comp}	C_{comm}	C_{ol}	C_{comp}	C_{comm}	C_{ol}
1	*2319321*	0	*8241581*	0	*8241581*	*8241581*	0	*8241581*
4	*2319321*	23040	1863019	66163	**1928809**	1863122	80708	**1942007**
8	*2319321*	30464	781103	87820	**840279**	781141	110729	**858536**
16	*2319321*	34560	380554	100566	**467081**	380546	132080	**497386**
32	*2319321*	36736	204997	132377	**322489**	205025	177653	**367745**
64	*2319321*	37824	117567	152952	**255266**	117484	205974	**307624**

Table B.8: Calculated instructions per cycle, speedup and efficiency of Gauss.

p	Zero communication cost			DRAM latency = 0		DRAM latency = 10	
	IPC_{max}	SU_{max}	Eff_{max}	SU_{act}	Eff_{act}	SU_{act}	Eff_{act}
1	0.281	1.00	1.00	1.00	1.00	1.00	1.00
4	1.24	4.42	1.11	4.27	1.07	4.24	1.06
8	2.96	10.6	1.32	9.81	1.23	9.60	1.20
16	6.08	21.6	1.35	17.6	1.10	16.6	1.04
32	11.3	40.2	1.26	25.6	0.799	22.4	0.700
64	19.7	70.1	1.10	32.3	0.504	26.8	0.419

Table B.9: Significance of communication, actual throughput, percentage of ideal throughput and effective throughput of Gauss.

p	DRAM latency = 0				DRAM latency = 10			
	S_{comm}	TP	TP_{pct}	TP_{eff}	S_{comm}	TP	TP_{pct}	TP_{eff}
4	0.032	0.348	23.9	0.350	0.039	0.285	32.1	0.292
8	0.085	0.347	35.8	0.515	0.107	0.275	46.4	0.394
16	0.195	0.344	47.3	0.399	0.256	0.262	58.9	0.296
32	0.514	0.278	47.7	0.313	0.690	0.207	58.2	0.226
64	1.19	0.247	51.0	0.275	1.60	0.184	62.0	0.199

Table B.10: Measurements of Jacobi successive over-relaxation (Jacobi).

			DRAM latency = 0			DRAM latency = 10		
p	$I_{comp}(1)$	B_{comm}	C_{comp}	C_{comm}	C_{ol}	C_{comp}	C_{comm}	C_{ol}
1	*3874266*	0	*10794936*	0	*10794936*	*10794936*	0	*10794936*
4	*3874266*	17120	2387345	50504	**2426428**	2387381	55214	**2431534**
9	*3874266*	17636	1144649	66172	**1160492**	1144628	74260	**1163988**
16	*3874266*	14232	624173	80404	**674009**	624097	87508	**679351**
36	*3874266*	15556	309582	94242	**366027**	309694	103490	**372502**
64	*3874266*	9926	174477	103069	**232521**	174456	111538	**237956**

Table B.11: Calculated instructions per cycle, speedup and efficiency of Jacobi.

	Zero communication cost			DRAM latency = 0		DRAM latency = 10	
p	IPC_{max}	SU_{max}	Eff_{max}	SU_{act}	Eff_{act}	SU_{act}	Eff_{act}
1	0.359	1.00	1.00	1.00	1.00	1.00	1.00
4	1.62	4.52	1.13	4.45	1.11	4.44	1.11
8	3.39	9.43	1.05	9.30	1.03	9.27	1.03
16	6.21	17.3	1.08	16.0	1.00	15.9	0.993
32	12.5	34.9	0.969	29.4	0.819	29.0	0.805
64	22.2	61.9	0.967	46.4	0.725	45.4	0.709

Table B.12: Significance of communication, actual throughput, percentage of ideal throughput and effective throughput of Jacobi.

	DRAM latency = 0				DRAM latency = 10			
p	S_{comm}	TP	TP_{pct}	TP_{eff}	S_{comm}	TP	TP_{pct}	TP_{eff}
4	0.019	0.339	11.7	0.438	0.020	0.310	17.4	0.388
8	0.055	0.267	9.16	1.11	0.062	0.237	13.4	0.911
16	0.119	0.177	6.09	0.286	0.130	0.163	9.15	0.258
32	0.314	0.165	5.67	0.276	0.345	0.150	8.46	0.248
64	0.611	0.096	3.31	0.171	0.661	0.089	5.01	0.156

Table B.13: Measurements of Jacobi successive over-relaxation without global combine (JacNoGC).

p	$I_{comp}(1)$	B_{comm}	DRAM latency = 0			DRAM latency = 10		
			C_{comp}	C_{comm}	C_{ol}	C_{comp}	C_{comm}	C_{ol}
1	*3874266*	0	*10794936*	0	*10794936*	*10794936*	0	*10794936*
4	*3874266*	14336	2365565	14525	**2379334**	2365673	17111	**2382784**
9	*3874266*	14336	1138511	16586	**1142703**	1138547	22387	**1145862**
16	*3874266*	10752	620330	14888	**628860**	620406	19068	**633859**
36	*3874266*	11947	307300	15181	**314371**	307224	19682	**319487**
64	*3874266*	6272	172706	13001	**178136**	172744	15369	**180308**

Table B.14: Calculated instructions per cycle, speedup and efficiency of JacNoGC.

p	Zero communication cost			DRAM latency = 0		DRAM latency = 10	
	IPC_{max}	SU_{max}	Eff_{max}	SU_{act}	Eff_{act}	SU_{act}	Eff_{act}
1	0.359	1.00	1.00	1.00	1.00	1.00	1.00
4	1.64	4.56	1.14	4.54	1.13	4.53	1.13
8	3.40	9.48	1.05	9.45	1.05	9.42	1.05
16	6.25	17.4	1.09	17.2	1.07	17.0	1.06
32	12.6	35.1	0.976	34.3	0.954	33.8	0.939
64	22.4	62.5	0.977	60.6	0.947	59.9	0.935

Table B.15: Significance of communication, actual throughput, percentage of ideal throughput and effective throughput of JacNoGC.

p	DRAM latency = 0				DRAM latency = 10			
	S_{comm}	TP	TP_{pct}	TP_{eff}	S_{comm}	TP	TP_{pct}	TP_{eff}
4	0.005	0.987	33.9	1.04	0.006	0.838	47.1	0.838
8	0.014	0.864	29.7	3.42	0.019	0.640	36.0	1.96
16	0.022	0.722	24.8	1.26	0.028	0.564	31.7	0.799
32	0.051	0.787	27.1	1.69	0.066	0.607	34.1	0.974
64	0.077	0.482	16.6	1.16	0.091	0.408	23.0	0.829

Table B.16: Measurements of bucket sort (Sort).

p	$I_{comp}(1)$	B_{comm}	DRAM latency = 0			DRAM latency = 10		
			C_{comp}	C_{comm}	C_{ol}	C_{comp}	C_{comm}	C_{ol}
1	*1372014*	0	*2056942*	0	*2056942*	*2056942*	0	*2056942*
4	*1372014*	12619	475122	6876	**481998**	478200	9307	**487508**
8	*1372014*	7966	251492	6221	**257713**	251453	8718	**260171**
16	*1372014*	5604	136494	9727	**146221**	136533	11376	**147909**
32	*1372014*	5658	76541	17478	**94019**	76538	19207	**95745**
64	*1372014*	8516	46624	34318	**80942**	46626	36966	**83592**

Table B.17: Calculated instructions per cycle, speedup and efficiency of Sort.

p	Zero communication cost			DRAM latency = 0		DRAM latency = 10	
	IPC_{max}	SU_{max}	Eff_{max}	SU_{act}	Eff_{act}	SU_{act}	Eff_{act}
1	0.667	1.00	1.00	1.00	1.00	1.00	1.00
4	2.89	4.33	1.08	4.27	1.07	4.22	1.06
8	5.46	8.18	1.02	7.98	0.998	7.91	0.988
16	10.1	15.1	0.942	14.1	0.879	13.9	0.869
32	17.9	26.9	0.840	21.9	0.684	21.5	0.671
64	29.4	44.1	0.689	25.4	0.397	24.6	0.384

Table B.18: Significance of communication, actual throughput, percentage of ideal throughput and effective throughput of Sort.

p	DRAM latency = 0				DRAM latency = 10			
	S_{comm}	TP	TP_{pct}	TP_{eff}	S_{comm}	TP	TP_{pct}	TP_{eff}
4	0.013	1.84	63.1	1.84	0.018	1.35	76.2	1.35
8	0.024	1.28	44.0	1.28	0.034	0.914	51.4	0.914
16	0.076	0.576	19.8	0.576	0.088	0.493	27.7	0.493
32	0.272	0.324	11.1	0.324	0.299	0.295	16.6	0.295
64	1.068	0.248	8.53	0.248	1.150	0.230	13.0	0.230

Appendix C

MEASUREMENTS OF COMMUNICATION COST IN GAUSS UNDER MODIFICATIONS TO Cranium

Section 6.3 provides the background for the measurements in Table C.1. The following list summarizes the modifications to Cranium used in the measurements of the Gauss benchmark. In all cases, the DRAM access latency is zero cycles for the network interface.

- **M1**: Each send command results in the injection of one packet into the network.
- **M2**: Each incoming packet causes the processor to be notified. Packet data are volatile and must be read into processor registers or copied to memory before accessing data from subsequent packets.
- **M3**: There is a one-to-one mapping between a physical message buffer on the sending node and a physical message buffer on the receiving node.
- **M1+M2**: Modifications M1 and M2 are applied together to the benchmark and the simulator.

Table C.1: Cost of communication in Gauss under Cranium modifications M1, M2, M3 and M1+M2.

	Ccomm				
p	Cranium	M1	M2	M3	M1+M2
4	66163	108515	247626	206219	269183
8	87820	166400	304223	308314	341894
16	100566	199440	314230	338057	383108
32	132377	280056	342193	502234	460063
64	152952	310569	466154	607040	586091

Appendix D

DESCRIPTION OF LOW-LEVEL DETAILS OF TESCHIO

Teschio is a paper design of the Cranium network interface architecture. Its context and overall structure are described in Chapter 7. This appendix contains low-level descriptions of the following aspects of Teschio:

- P link handshake
- Node mapping using the Chaos network
- Sending and receiving a packet

D.1 Handshaking signals and protocol of the P link

Figure D.1a describes the signals used in the processor-network link (*P link*). The P link uses five signals for handshaking: RTS (ready to send), CTS (clear to send), RTR (ready to receive), CTR (clear to receive) and TB (tie breaker). (All the names of the handshaking signals assume the point of view of Teschio.) There are also state registers associated with the P link, represented by the virtual signal bus PLState. PLState does not need to be exposed as an external signal because both Teschio and the router chip maintain separate copies of the state registers. Hence, the bus is drawn using a dotted line. There are 23 states for the P link: an idle state (0), eleven states for sending a packet (1-11) and eleven states for receiving a packet (12-22) (Figure D.1b). State changes are synchronous and occur every clock cycle. The states of the P link are unconstrained by the states of the ADU bus; each state machine sequences independently. When both the handshake signals RTS and CTS are true, then Teschio has a packet ready for injection while the router is ready to accept a packet. The signal pair RTR and CTR work similarly for packet delivery. When all four signals are true, a packet may either be injected or delivered. Teschio and the router chip must negotiate the direction of packet flow. Sending and receiving

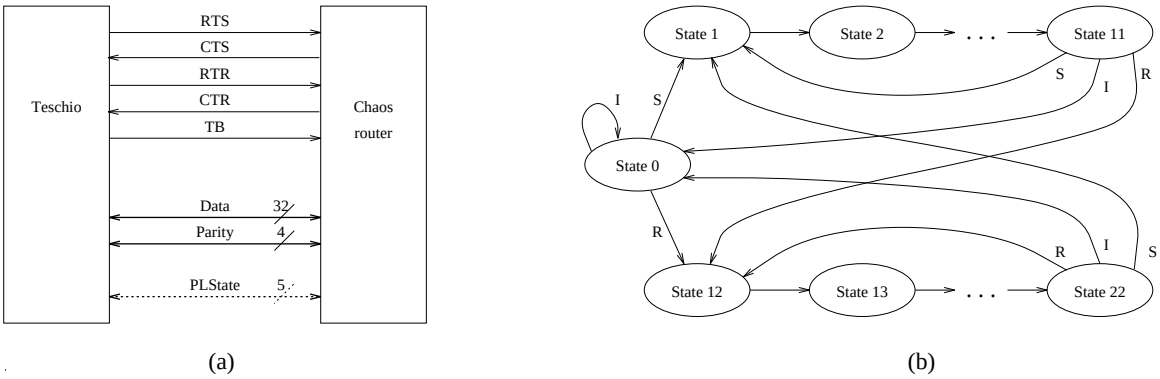


Figure D.1: Organization of the P link. Subfigure *a* describes the data path and handshaking signals. Subfigure *b* is a state transition diagram for PLState. Condition S is shorthand for (RTS and CTS and (TB or !RTR or !CTR)). Condition R is (RTR and CTR and (!TB or !RTS or !CTS)). Condition I is (!S and !R). The initial condition is that PLState is in state 0 and TB is false.

is interleaved at the packet level, but not at the phit (physical transfer unit) level. Therefore when the first phit of a packet is placed on the data bus of the P link, the other ten phits of the packet follow in succession. Fairness is ensured using the signal TB: when TB is false, packet delivery has higher priority, and when TB is true, packet injection has higher priority. The initial state for TB is false. TB changes state if and only if it is used to break a tie. In the implementation, TB is an output from Teschio and an input into the router chip. In principle, TB could be an internally generated signal like PLState, but by making it an explicit external signal, there is more leeway for experimenting with different algorithms to provide fairness.

Figure D.1b shows the transitions among the states of the P link according to the conditions S, R and I. Condition S means that RTS and CTS are true, and either RTR or CTR are false or TB is true. Condition R means that RTR and CTR are true, and at least one of RTS, CTS and TB is false. Condition I is true if neither condition S nor condition R holds. The data direction changes only after states 0, 11 and 22, whereupon a whole packet has been sent or received. The state transition diagram demonstrates that there are no wait states needed to reverse the direction of the data path, to ensure that the P link delivers its peak throughput under heavy utilization.

D.2 Node mapping

In the Chaos network, the physical name for a remote node is its relative position in the Y dimension (the north-south axis) and the X dimension (the east-west axis). To send a packet from node A to node B, the network interface at node A specifies the tuple [y,x] indicating the difference in the Y dimension and the X dimension. The north and west directions are negative and the south and east directions are positive. Note that the tuple for sending a packet in the opposite direction from B to A is calculated simply by taking the complement of both values, i.e. [$-y$,$-x$]. In the implementation of the network, each dimension is represented by a signed byte, allowing a maximum configuration of 256×256 nodes.

Because physical addressing is relative, each node requires a unique configuration of the node lookup table. Figure D.2 illustrates the relationship between the logical node names and the physical node names for logical node `0x1C` in an 8×8 node network. The diagram shows the 4×4 node local neighborhood of logical node `0x1C`. Because the network is a torus and there are no edges to the network, this scheme generalizes very easily. Each rectangle in Figure D.2 represents the processing node connected to a Chaos network router. Three lines of text describe the node's names: the logical node identifier (Logical), the physical node identifier to indicate the destinations of packets to be sent (Phy Dst) and the physical ID for the source node of packets that are received (Phy Src). Logical node `0x1C` is highlighted using a thick rectangle to indicate that all the physical names in this diagram are relative to this node. Since the displacement from node `0x1C` to itself is zero, both the destination and source physical IDs are zero.

When a packet is sent, the sending node places the destination physical ID into the first 16 bit field of the packet header and the source physical ID into the second 16 bit field. As the packet traverses the network, the network routers update the destination ID field in place, by incrementing or decrementing the X or Y displacement field at each network hop. When both X and Y are zero, then the packet has arrived at its destination node and it is ejected from the network. Since the destination ID field has been modified, the receiver's network interface must detect and verify the packet's sender by using the source ID field. In principle, the receiver's interface circuit could send a reply packet back to the sender directly, by copying the source ID field of the original packet into the first 16 bit field of the reply packet, taking the complement

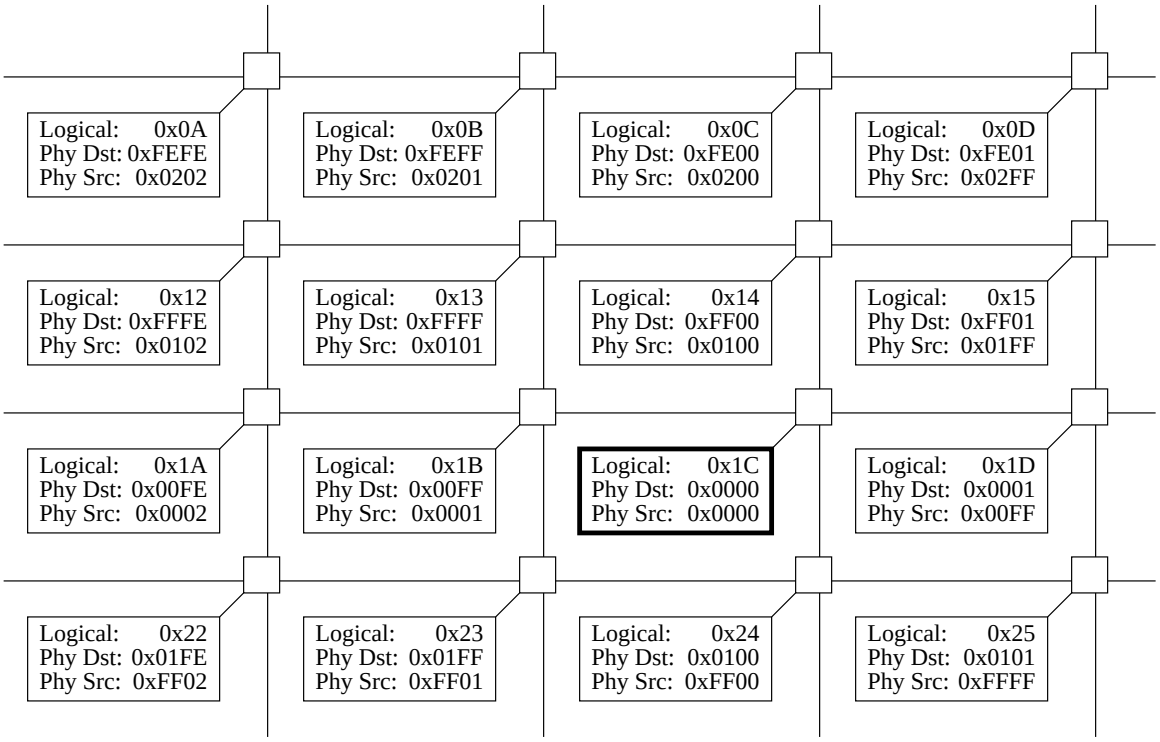


Figure D.2: Node naming scheme used by Teschio. The node names known to the application program are called the *logical* or *linear* node names (see Section A.1.1 in Appendix A). The node names known to the router hardware are called the *physical* node names. Teschio performs the logical-to-physical mapping using a lookup table. The diagram shows a 4×4 submesh of an 8×8 torus network. In this example, all physical names are relative to logical node ID `0x1C`, highlighted by a thick rectangle. Physical names are 8-bit signed displacements in both the Y and X dimensions. The physical identifier for the source node (Phy Src) can be calculated from the physical identifier for the destination node (Phy Dst) by negating both 8-bit fields in the identifier.

of both the X and Y displacements and placing this value into the second 16 bit field of the reply packet. However, this version of Teschio does not automatically generate reply packets.

D.3 Sending and receiving a single-packet message

This section describes the behavior of Teschio at its external interfaces – the ADU bus and the P link. We explain behavior in two parts: the values that are propagated and the timing of these values on the chip's pins.

Figure D.3 shows the organization of the bit fields that define a send or receive command issued by the processor. In assembly language the instruction is of the form

```
store An, Dm
```

meaning that the quantity in the data register `Dm` is stored into memory at the address held in the address register `An`. The address in `An` is a user virtual address that is mapped by the processor's MMU into a physical address corresponding to Teschio. The page table for this MMU page is marked non-cacheable, to mandate an ADU bus transaction whenever this instruction is executed. Two different mappings are maintained in the MMU: one for user processes and another for the operating system. The MMU prevents user programs from accessing the mapping that is privileged to the OS.

The amount of data to be transferred in a send or receive command is 71 bits. Since the number of bits is larger than size of a data register (64 bits), the remaining bits must come from somewhere else; in our implementation they come from the address register. The lowest five bits of the address register are ignored. The next 8 bits select the channel number. The next 35 bits select the MMU page. The top 16 bits are ignored.

Figures D.4, D.5 and D.6 describe the timing of Teschio's external environment: the ADU bus and the P link. In these diagrams the wide busses are divided into multiple 16-bit fields for readability. In this example we show what happens when a packet is sent from logical node 0x1C to logical node 0x24. Figure D.4 displays the timing of a host-initiated send or receive command on the ADU bus. All the wide busses are three-state (one, zero or high-Z). The line in the center of its high-low range represents the high-impedance state across all 16 wires in each field. In this

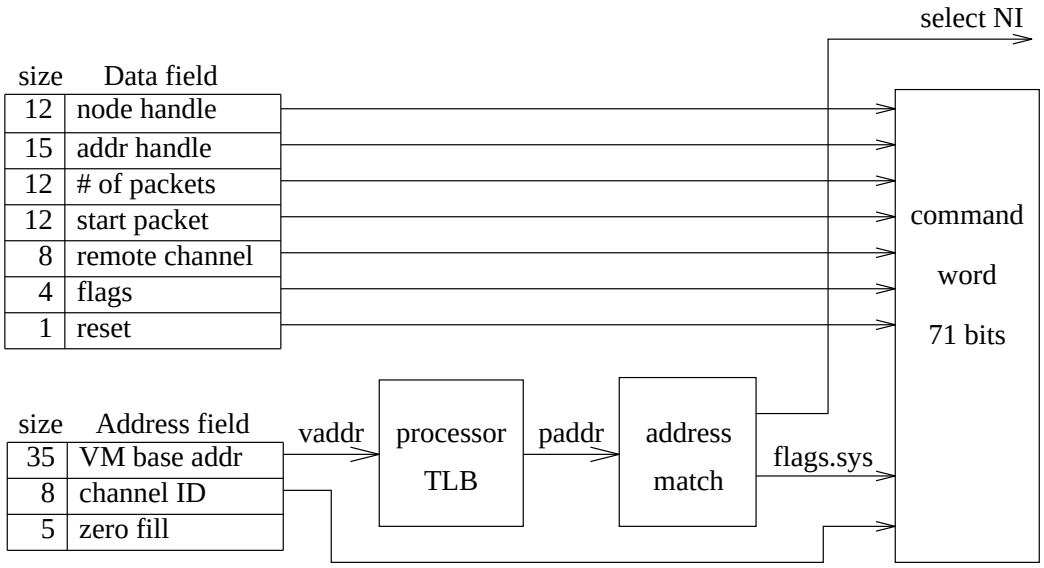


Figure D.3: Organization of bit fields placed on ADU bus by the processor to issue a send or receive command

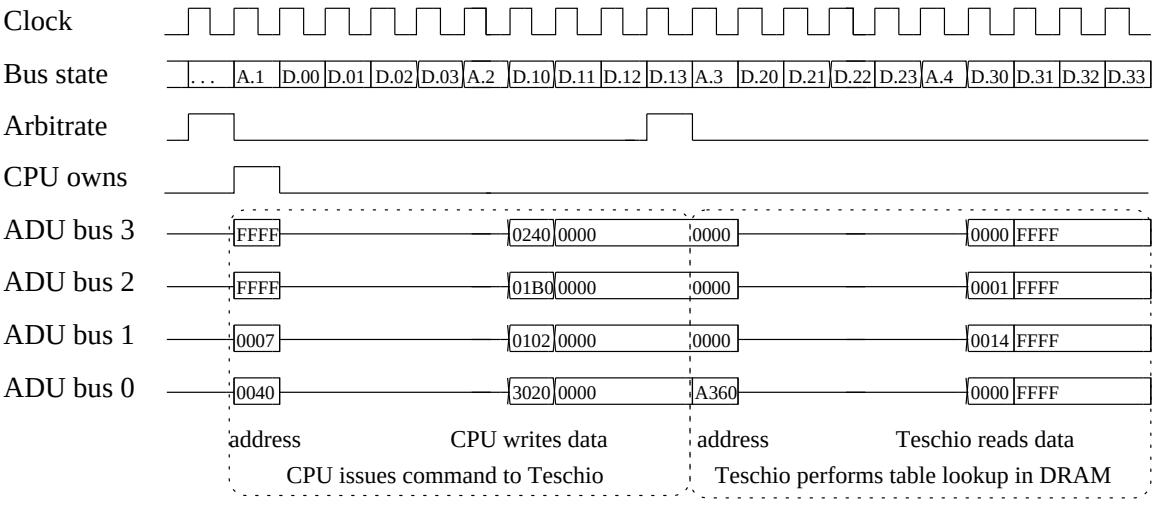


Figure D.4: Timing of a send or receive command on the ADU bus

example, user virtual addresses in the range 0xFFFF00070000 to 0xFFFF0007FFE0 are mapped to the Teschio send channels. The address 0xFFFF00070040 maps to Teschio send channel 2; in general 0xFFFF000700S0 maps to Teschio send channel S/2. In this example the data field of the send command arrives on the ADU bus during bus state D.10. The values that the processor places on the ADU bus during states D.11, D.12 and D.13 are null and are discarded by Teschio. Two other bus signals to note are Arbitrate and CPU Owns. Since both the processor and Teschio are bus masters, they must arbitrate for the bus one cycle prior to issuing the address. For the purpose of this diagram, Arbitrate is a wired-OR signal that is enabled whenever one or more bus masters are requesting the bus during the clock cycle just prior to an address field. Bus-master priority is maintained using a simple round-robin scheme that allocates the bus bandwidth fairly [64]. The signal CPU Owns shows that the processor has won the arbitration and is placing the address on the bus during bus state A.1. The value placed on the ADU bus in state D.10 is 0x024 001B 001 023 02 0 to represent logical node ID 0x24, address handle 0x001B, a message length of 1 packet, a starting address offset of 0x023 cache lines, auto-receive channel number 0x02 and flag values of 0. In Teschio the size of an MMU page is programmable at boot time. In this example the page size is 8K bytes; each page contains 256 (0x100) 32-byte cache lines.

The second bus access shown in Figure D.4 is a DRAM table lookup. Teschio becomes a bus master, arbitrates for the bus and wins the arbitration. Teschio places its DMA address on the ADU bus at bus state A.3. Teschio performs two table lookups: one to translate the address handle into a physical buffer address and the other to translate the logical node ID into a physical node ID. The table lookup shown is the translation of the address handle into the physical buffer address. The address handle specified in the previous bus transaction was 0x1B or binary 0011011. By shifting this value five bits to the left, the result is binary 001101100000 or 0x360. The base address of the table in DRAM is 0x00000000A000. Therefore the DMA engine issues a read command at ADU bus address 0x00000000A360. The value read at this location in DRAM is 0x000100140000, the physical address of the buffer page that is pinned and mapped into the user program's virtual address space. The complete DMA address of the packet payload is 0x000100140460, the base address plus the offset field (0x023 shifted left five bits, i.e. 0x460).

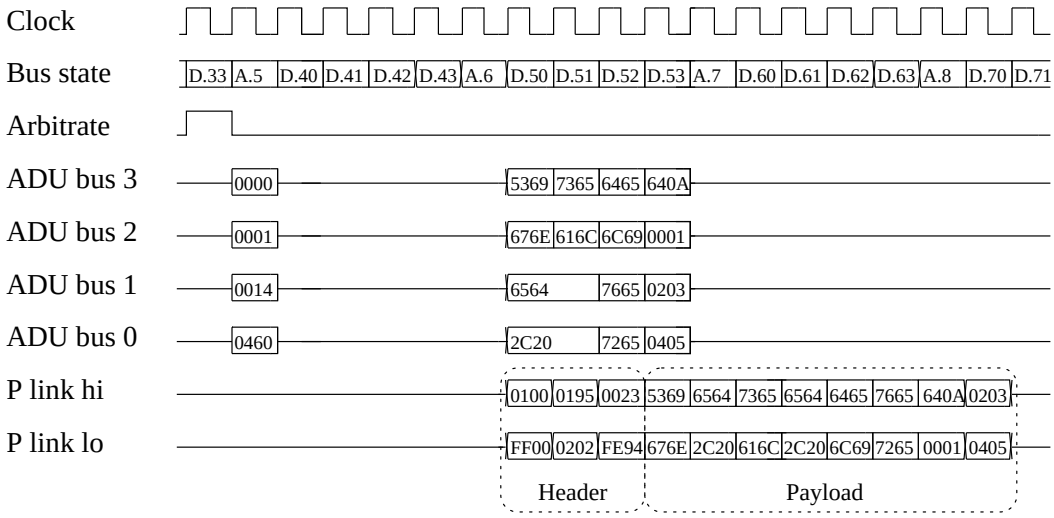


Figure D.5: Timing diagram for sending a single packet message

Upon the completion of the table lookup operations, Teschio is ready to start sending packets. Figure D.5 shows an example where the message to be sent is the ASCII string "Signed, sealed, delivered". In hexadecimal notation the string is `53 69 67 6E 65 64 2C 20 73 65 61 6C 65 64 2C 20 64 65 6C 69 76 65 72 65 64 0A`. The hex values 0 through 5 fill the final 6 bytes of the 32-byte payload. At bus state D.33, Teschio arbitrates for the bus and wins the arbitration. At bus state A.5, Teschio puts DMA address `0x000100140460` on the ADU bus. States D.40 through A.6 are wait states. Durings state D.50 through D.53, Teschio access both the ADU and the P link simultaneously. Teschio reads the payload data from DRAM while writing the first 16 bits of the header onto the P link. The pattern "6564 2C20" or "`ed, `" on the ADU bus repeats in states D.50 and D.51. At subsequent bus states D.53 through D.70, Teschio writes the payload onto the P link.

The header of the packet is the hexadecimal string `0100 FF00 0195 0202 0023 FE94`. The first 16-bit field `0100` is the destination physical node identifier for logical node `0x24` from the point of view of logical node `0x1C`, and the second field `FF00` is the corresponding source physical node identifier (the inverse of the destination). The third field `0195` is the user process ID. The fourth field `0202` represents the send channel and the receive channel, both equal to 2. The fifth field `0023` four zero flag bits plus a 12-bit packet offset (sequence number) of `023`. The sixth field, `FE94`, is the redundancy code.

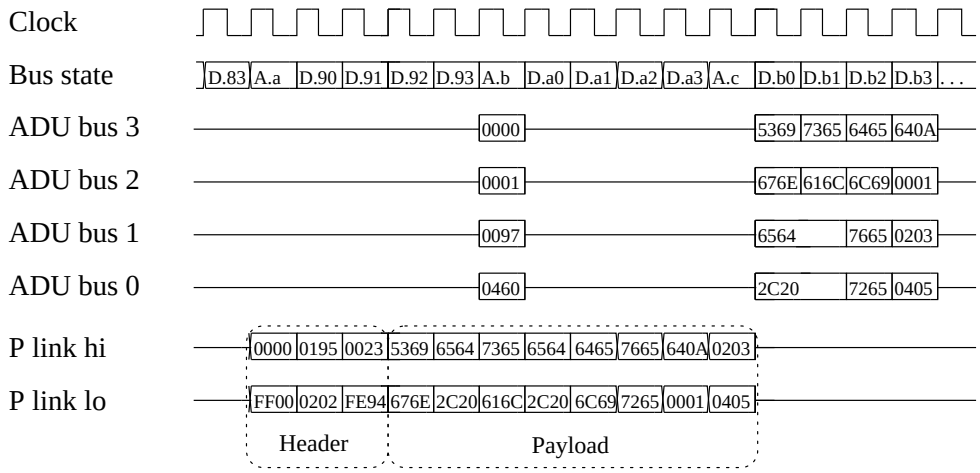


Figure D.6: Timing diagram for receiving a single packet message

To complete the example we follow the packet as it arrives at logical node `0x24`. Figure D.6 shows the timing for Teschio as it receives the packet from the P link and stores the data into DRAM. We assume that the receive command has been posted far enough in advance so that the translation steps have already completed, so that receive channel 2 is ready. When the packet arrives it is identical to the packet that was originally sent, with the exception that the first 16-bit field in the header becomes 0 because it was modified by the network routers. Teschio becomes a bus master, and arbitrates for and wins the ADU bus in preparation for a DMA write to memory. The DMA address in this case is `0x000100970460`. The top 35 bits of the address come from a combination of the way the application program is written and the operating system's virtual-to-physical mapping algorithm. The bottom 13 bits represent the memory offset within the 8K byte page, e.g. the sequence number (`0x023`) shifted five bits to the left.

Note that the timing constraints imposed by the design guidelines in Section 7.4.1 are reflected in Figures D.4, D.5 and D.6. In particular, it takes three cycles for information presented to Teschio at one side of the chip to have an influence on the data values at the other side of the chip. In Figure D.5, values that appear on the ADU bus due to Teschio's DMA read of DRAM during bus state D.50 affect values on the P link three cycles later during bus state D.53. Similarly in Figure D.6, values that arrive from the network on the P link during bus state D.91 are used to calculate the DMA address in bus state A.b.

Vita

Neil R. McKenzie was born in Watsonville, California on March 13, 1961. He received the Bachelor of Science degree in Electrical Engineering and Computer Science, With Honors, from the University of California at Berkeley in 1983. He entered the graduate school at the University of Washington in Seattle in 1987. He received the Master of Science degree in Computer Science and Engineering in 1989. He worked for LaserAccess Corporation in Bothell, Washington during 1990-91. He returned to the University of Washington in 1991. He completed the Ph.D. in Computer Science and Engineering in 1997. Dr. McKenzie is currently employed by MERL — a Mitsubishi Electric Research Laboratory, in Cambridge, Massachusetts.