ABSTRACT

A PROGRAMMABLE ROUTING CONTROLLER SUPPORTING MULTI-MODE ROUTING AND SWITCHING IN DISTRIBUTED REAL-TIME SYSTEMS.

by
James William Dolter

Chair: Kang G. Shin

Distributed systems with point–to–point interconnection networks are natural candidate architectures for supporting embedded real-time applications due mainly to their potential for high-performance and high-dependability with the multiplicity of processors and internode routes. In addition, distributed systems based on point–to–point interconnection networks also hold great promise for delivering systems with some level of intermediate dependability: something not as expensive and complete as the ultra-dependable systems used in mission-critical applications but more dependable than systems constructed using standard networking techniques and commercially-available components.

This dissertation investigates an area of the design space in which a homogeneous distributed computing system is constructed that has the potential for an intermediate level of dependability. Most of the characteristics of this system can be traced to either the parallel computing domain, the distributed systems domain, or the ultra-dependable systems domain. The unifying feature not explicitly present in the three "parent" domains is the flexibility of the underlying communication support hardware which allows these other features to co-exist while supporting the end goals.

One of the central themes of this dissertation is that if flexibility is provided in the front-end routing hardware, the management of issues related to operating in this hybrid domain are then possible. To that end, most of the results presented here focus on the development and implementation of a programmable routing controller (PRC) that acts both as a single sample in the design space and supports further exploration of the design space. Specifically, this work proposes an architecture and an implementation for a front-end programmable routing controller that supports multiple routing and switching schemes. We then proceed to analyze the performance of one of the supported switching schemes for a hexagonal mesh interconnection network based on the capabilities provided by the PRC. We then conclude by showing how the capability of supporting multiple routing and switching schemes simultaneously offers great potential for supporting real-time communication subsystems.

A PROGRAMMABLE ROUTING CONTROLLER SUPPORTING MULTI-MODE ROUTING AND SWITCHING IN DISTRIBUTED REAL-TIME SYSTEMS.

by

James William Dolter

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
1994

Doctoral Committee:
- Professor Kang G. Shin, Chair
- Assistant Professor William P. Birmingham
- Associate Professor Richard B. Brown
- Professor Trevor N. Mudge
- Associate Research Scientist Chinya V. Ravishankar

To Karen and my parents

ACKNOWLEDGEMENTS

It is an impossible task to thank all those who have contributed in one form or another to this dissertation. I would, however, like to take this opportunity to thank a few who have provided me great support in this undertaking.

First, I would like to thank my advisor, Professor Kang G. Shin. Without Professor Shin's continuous and unconditional support I would not have been able to complete the work presented here. Through his guidance I have gained invaluable expertise in approaching, analyzing, and solving the problems that every researcher hopes to find. I would also like to thank the other members of my doctoral committee, Professors William Birmingham, Richard Brown, Trevor Mudge, and Chinya Ravishankar for their constructive input on this dissertation. Thanks also needs to go to the Office of Naval Research, Martin Marietta, and the National Science Foundation for providing the funds necessary for this endeavour.

I would also like to gratefully acknowledge:

- My wife Karen, for her love, support, and putting up with what appeared at times to be a snail's pace at completing this work.
- Parmesh Ramanathan and Dilip Kandlur, for many of the daily discussion/arguments (ask the neighbors a couple of offices down) that formed many of the earlier ideas presented here.
- Professor John F. Meyer for instilling in me a great respect for the use of performability modeling and SANS.
- Stuart Daniel for having the patience to work in detail on the existing PRC design and tolerate what at times may seem like complete lunacy for the rigor that I impose on the project.
- Dave Musliner for being a fellow hacker in many ways and a great sounding board.
- Jennifer Rexford and Ashish Mehra, for reading draft material and providing an enthusiastic audience for some of my preaching.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF APPENDICES

APPENDIX

CHAPTER 1

INTRODUCTION

1.1 Motivation

As we approach the 21st century the once separate disciplines of parallel computing and distributed systems are now starting to exhibit significant overlap. Parallel computing, initially driven by the need for high-performance scientific computing, traditionally has resulted in highly regular interconnection networks and tightly-coupled processing elements. In contrast, distributed systems grew out of the need to share expensive resources such as disk space, printers, and data between possibly heterogeneous network-based machines. The once clean separation between these two fields exists no more.

At the same time it is becoming commonplace to use digital computers for such real–time applications as fly–by–wire, industrial process control, computer–integrated manufacturing, electric power distribution/monitoring, and medical life-support systems. The use of computers can increase productivity but the applications impose stringent timing *and* dependability requirements on the computer system. These requirements are a direct consequence of the fact that a disruption of service caused by a physical failure or inadequate response time can result in a catastrophe.

The need for ultra-dependable computers that can function in real-time has been recognized for some time and resulted in the initial developments of the Software Implemented Fault-Tolerance (SIFT) computer[22] and the Fault-Tolerant Multiprocessor (FTMP)[44]. This commitment continued with the research and implementation of both the Fault-Tolerant Processor(FTP)[43] and its incorporation into the Advanced Information Processing Systems (AIPS). Most of the effort in these systems was focused on the placement and the management of redundancy in different forms to achieve the desired dependability. Each of the above systems uses explicit redundancy in the physical interconnect between the processing elements. These architectures could be summarized as ultra-dependable multiprocessors or multi-computers and form roughly the first or second generation of dependable computers.

Due mainly to their potential for high-performance and high-dependability with the multiplicity of processors and internode routes, distributed systems with point–to–point interconnection networks are natural candidate architectures for supporting embedded real-time applications. The key to the success in using distributed systems for these applications is the timely execution of computational tasks which usually reside on different nodes and communicate with one another to accomplish a common goal. Deadline guarantees for these real–time tasks are not possible without a carefully designed communication subsystem which supports the timely delivery of messages. In some cases it may be necessary to provide specific hardware support in the communication subsystem in order to satisfy these guarantees. In addition to the potential for supporting ultra-dependable computing, distributed systems based on a point–to–point interconnection networks also hold great promise for delivering systems with some level of intermediate dependability: something not as expensive and complete as the ultra-dependable systems used in mission-critical applications but more dependable than systems constructed using standard networking techniques and commercially-available components.

1.2 Research Objectives

This dissertation investigates an area of the design space in which a homogeneous distributed computing system is constructed that has the potential for an intermediate level of dependability. The distinguishing features captured in this system are depicted by the enclosing ellipse in Figure 1.1. Most of these characteristics are "borrowed" from either the parallel computing domain, the distributed systems domain, or the ultra-dependable systems domain. The unifying feature not explicitly present in the three "parent" domains is the flexibility of the underlying communication support hardware which allows these other features to co-exist while supporting the end goals.

One of the central themes of this dissertation is that if flexibility is provided in the front-end routing hardware, the management of issues related to operating in this hybrid domain are then possible. To that end, most of the results presented here focus on the development and implementation of a programmable routing controller (PRC) that acts both as a single sample in the design space and supports further exploration of the design space. Specifically, this work

- Proposes an architecture and an implementation for a front-end programmable routing controller that supports multiple routing and switching schemes.

- Analyzes the performance of one of the supported switching schemes for a hexagonal mesh interconnection network based on the capabilities provided by the programmable routing controller.

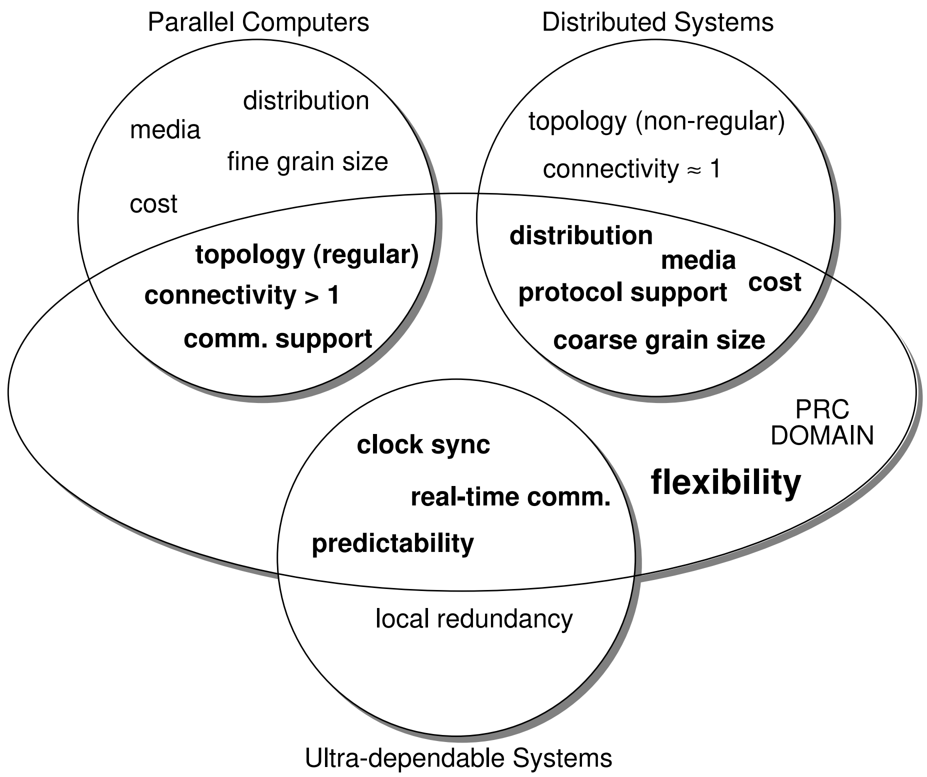


Figure 1.1: Domain specific characteristics

- Shows how the capability of supporting multiple routing and switching schemes simultaneously offers potential for supporting a real-time communication subsystem.

1.3 Domain Clarification and Assumptions

This section highlights some of the important characteristics of the region of the design space we are exploring. These domain characteristics either pertain to topological and architectural components of the system or relate to requirements of the applications to be supported by the system. We discuss each of the domain characteristics that was selected and attempt to illuminate the insight that led to their selection. Our domain can be effectively characterized in terms of *topology and connectivity*, *transmission media*, *message/packet size*, *real-time support*, *fault-tolerance*, and *flexibility*.

Connectivity refers to the number of adjacent neighbors of a node, which directly translates into the number of network ports that need to be provided into the interconnect fabric. Distributed systems usually feature a low connectivity, providing only one or two network ports. Parallel systems typically exhibit a higher degree of connectivity as a function of the network topology. The desire to provide a system with enhanced dependability characteristics leads directly to the adoption of the topology, connectivity, and routing and switching features of parallel computers.

Transmission media refers to the technology used to implement the physical connections between nodes and is characterized by several performability metrics, including propagation delay, capacity, and reliability. Distribution relates to the length of the individual physical links between nodes and directly affects the propagation delays and error characteristics. Many of the real-time applications of which our system is intended to support have physical distance constraints and external environmental disturbances that tend to align themselves with those of the distributed systems.

The desire to be able to interoperate with existing distributed systems technology but also have multiple internode routes and a communication system capable of timely delivery of messages solidifies the remaining characteristics highlighted by Figure 1.1. Existing distributed systems use communication protocols that force message/packet sizes to be considerably larger than those traditionally used in parallel computers. Furthermore, most existing parallel computers provide no mechanism for making the protocols supported extensible if providing any protocol at all. We are faced with the problem of having to deal with the large packets of the distributed system domain along with the issues of routing these packets between the multiple ports inherited from the parallel domain. Our answer is a flexible routing controller that can intelligently handle these issues efficiently close to the physical network.

1.4 A Map of the Dissertation

This dissertation unfolds into seven chapters and several appendixes.

Chapter 2 presents background material that acts as a foundation for the remaining chapters. The network topology and experimental system for which the programmable routing was conceived are described. In addition to the system level descriptions provided, a brief tutorial on different routing and switching schemes is provided.

Chapter 3 presents the architecture and implementation of the programmable routing controller and the network interface. In this chapter we will justify the design decisions in light of our overall goals. It is the synergistic combination of many of the features present in the programmable routing controller that make it stand out from previous work. After the implementation is described, the design is compared to several related systems.

The problem of deriving an analytical model for virtual cut-through message passing in a hexagonal mesh is addressed in Chapter 4. To accomplish this, the hexagonal mesh is first characterized using a combinatorial analysis to determine the probability that a packet will establish a cut-through at an intermediate node. Given this parameter, the probability distribution function for packet delivery times is derived. The delivery times obtained from the model are then compared with simulated results of an earlier version of the proposed routing hardware.

Chapter 5 presents the second generation simulator developed to explore design issues in point-to-point distributed systems. The task specification language and some of the design issues handled to allow for accurate modeling of the programmable routing controller are described.

Chapter 6 presents some of the most interesting results derived in this dissertation: significant advantages can be gained by simultaneously supporting multiple routing and switching algorithms. First, an analytic argument is presented for the mean value delivery times in a hexagonal mesh using wormhole switching and compared against delivery times of virtual cut-through. The differences in the delivery times lead to the conclusion that there may be situations in which one scheme is preferred over another. The work is then extended to show that it is possible to carry multiple classes of traffic and capitalize on the best features of both classes.

The dissertation concludes with Chapter 7, which summarizes the contributions and presents a discussion of the many directions that can be pursued with the results uncovered thus far.

CHAPTER 2

PRELIMINARIES

Many of the design decisions incorporated into the research described in this dissertation were influenced by the operating environment. This chapter presents a brief background on the target system (i.e., HARTS), as well as examining and classifying a number of routing and switching schemes.

2.1 C-wrapped Hexagonal Mesh

One of the interconnection topologies that has received considerable attention throughout this dissertation is the *C-wrapped hexagonal mesh* [6, 49].

Definition 1 *A* C-wrapped *H–mesh of edge size e, denoted by H_e, is comprised of $N = 3e^2 - 3e + 1$ nodes, labeled from 0 to $N-1$, such that each node s has six neighbors $[s+1]_N$, $[s+3e-1]_N$, $[s+3e-2]_N$, $[s+3e^2-3e]_N$, $[s+3e^2-6e+2]_N$, and $[s+3e^2-6e+3]_N$, where $[a]_b$ denotes $a \bmod b$.*

This topology can be visualized as an ordinary hexagonal mesh with links added to the nodes on the periphery that wrap around to other nodes on the periphery to form a homogeneous surface. For each mesh of edge size e the mesh of edge size $e+1$ is formed by adding a new hexagon of $6(e+1)$ nodes around the existing nodes, relabeling the nodes, and connecting the wraps according to the definition. Figure 2.1 shows an H_4 with the wrap links shown in gray and labeled with the nodes to which they are connected. See

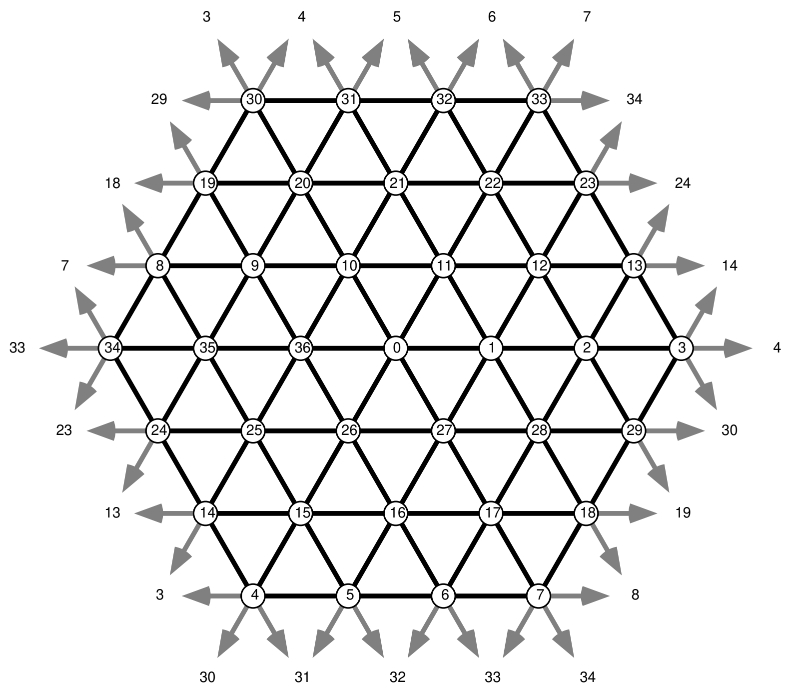


Figure 2.1: A C-wrapped hexagonal mesh of edge size 4 (H_4).

[6] for a more detailed discussion of some of the properties of the C-wrapped H–Mesh and comparisons with other existing topologies.

Several of the properties discussed in [6] are relevant to the development of this work. First, the C-type wrapping results in a homogeneous network. Consequently, any node can view itself as the center (labeled as node 0) of the mesh. This allows the physical interface to the network to be independent of the network size. Second, in terms of routing messages throughout the surface, the diameter of an H_e is $e-1$. Third, there is a simple, transparent addressing scheme such that the shortest paths between any two nodes can be determined by a $\Theta(1)$ algorithm given the address of the two nodes. (At each node on a shortest path there are at most two different neighbors of the node to which the shortest path runs.) Fourth, based on this addressing scheme it is possible to devise a simple routing algorithm that can be efficiently supported in hardware. Last, since the total number of nodes in an

H_n grows as a function of $\Theta(n^2)$, the incremental cost of expanding the network is low when compared to many other topologies.

The six neighbors of a node in a C-wrapped H–mesh can be thought of as being in the directions d_0, d_1, ..., d_5. To send a message using the base routing algorithm presented in [6], the source node calculates the shortest paths to the destination and encodes this routing information into three integers denoted by m_0, m_1 and m_2. These three integers represent the number of hops from the source node to the destination node along the d_0, d_1 and d_2 directions, respectively. Before sending the packet to an appropriate neighbor, intermediate nodes update these values to indicate the remaining hops in each direction to the destination. Hence, $m_0 = m_1 = m_2 = 0$ indicates that the packet has reached its destination.

For example, the (m_0, m_1, m_2) triple for routing a message from node 2 to node 14 is $(1, 1, 0)$ (see Figure 2.1). This triple encodes all the shortest paths from node 2 to node 14, i.e., node 14 can be reached by either going from node 2 to node 3 in the d_0–direction and then from node 3 to node 14 in the d_1–direction or by first going from node 2 to node 13 in the d_1–direction and then from node 13 to node 14 in the d_0–direction. Note that in routing this message from the source to the destination the "wrap" link was used *transparently* by all nodes in the message's path.

The C-wrapped mesh is the basis for the interconnection network of the HARTS discussed in the next section.

2.2 Hexagonal Architecture for Real-Time Systems (HARTS)

HARTS is an experimental testbed being designed and built to investigate various issues in distributed real–time computing. The primary goal of HARTS is to allow low–level architectural and OS issues such as message buffering, scheduling, and routing to be studied in a setting that allows the researchers internal access to many of the system parameters. To meet these goals, a hybrid system based on commercially–available processors and a custom–designed communication interface is being developed. Up to three processor cards are grouped together to form a cluster of Application Processors (APs). Each cluster then serves as a multiprocessor node and is interconnected to other clusters or nodes using the Network Processor (NP) to form a distributed system. The structure of a single node is shown in Figure 2.2.

In addition to the APs and the NP, each node has an additional processor that functions both as the VME bus system controller and as a network monitor (NMON). All network monitors are interconnected via a separate LAN. The NMON serves several purposes in the node architecture. First, it provides a necessary non-experimental means of distributing code and data amongst the nodes. This function is especially important during the early stages of development of the NP. Second, the NMON allows portions of the operating system to be developed in parallel with the development of the NP. Third, the NMON will be used to collect experimental data by monitoring the APs and the NP with minimal interference. Due to both multiprocessor and distributed aspects in the system, a wide variety of architectural and OS effects on real–time issues can be easily investigated.

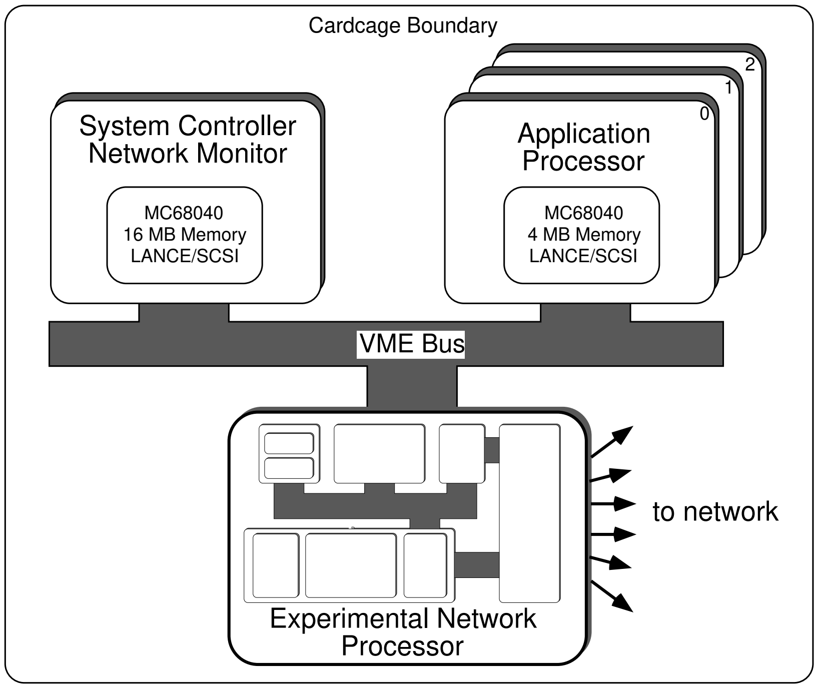


Figure 2.2: Block diagram of a HARTS node.

2.3 Network Processor (NP)

2.3.1 Functions supported by the NP

The support functions that the NP provides HARTS fall under three main categories: communication protocol processing, low latency message transmission, and direct support for real–time communication. Each of these areas is covered briefly, followed by a presentation of the proposed architecture.

The NP's main function will be to offload communication processing from the host processor. When an AP needs to transmit a message, it will provide the corresponding NP with information about the intended message recipient and the location of the message data. The NP should then execute the operations necessary to pass the message data through the various layers of protocol down to the physical layer where it can be transmitted.

In terms of the OSI reference model, the NP will be responsible for the functions from the transport layer down to the physical layer. At the transport level, the NP establishes transport connections dependent only on the source and destination nodes, without concern for the route to be used. It will also handle end–to–end error detection and handling. At the network level, the NP will select primary and alternate routes for establishing virtual circuits, form data blocks and segments, and reassemble packets at the destination node. Depending on traffic conditions in the network and the message type, the NP will choose an appropriate switching method for the message as discussed in Section 2.4. Detection and correction of errors will also be performed at this level. At the data link level, the NP will provide access to the network for the messages. It will perform framing and synchronization, and packet sequencing.

Low communication latency is another key goal of the NP. This goal impacts upon task migration and distribution, load sharing, and real–time communication. Latency considerations extend from the application tasks on the system down to the hardware components. It is important to recognize that a significant portion of latency occurs in communication processing; therefore achieving low latency communication is intimately related to implementation of communication protocols.

Timely delivery of messages requires a global time–base across the different nodes in HARTS. The NP will provide hardware support for clock synchronization and time–stamping of messages that are the basis for the implementation of various real–time communication algorithms. At the same time, operations such as checkpointing, timeouts, and deadline checking are made possible with the global time–base.

The NP will be required to implement buffer management policies that maximize utilization of buffer space, while guaranteeing the availability of buffers to the highest priority messages. Similarly, if non-critical messages hold other resources that are needed by more critical ones, it will be necessary to provide a means for preemption of such resources for use by the critical messages.

Another important function of the NP will be to monitor the state of the network in terms of traffic load and component failures. The traffic load will affect the ability of the NP to send real–time messages to other processors, while link failures will affect the system reliability. It will also be possible for the NP to keep track of the processing load of its host (or hosts), and use the information for load balancing/sharing and task migration operations.

2.3.2 Proposed NP architecture and implementation

There are five major components of the NP architecture: the main communications processor, which will be referred to as the interface manager unit (IMU), the programmable routing controller (PRC), the network interface (NI), the buffer memory, and the application processor interface (API), interconnected as shown in Figure 2.3.

The API transfers data between the NP and the APs, while the PRC moves data between the NP and the network with the assistance of the NI. Within the NP, the IMU controls the movement and processing of message data. The buffer memory acts as a staging area for data that is to be transmitted to, or received from, the network. Messages that need to be temporarily stored at the node due to unavailability of outgoing links are also stored in the buffer memory. The PRC, in conjunction with the NI, implements the physical layer and data link protocols for accessing the network and routing data to neighboring nodes. The remainder of this section briefly discusses each of the major components but defers the

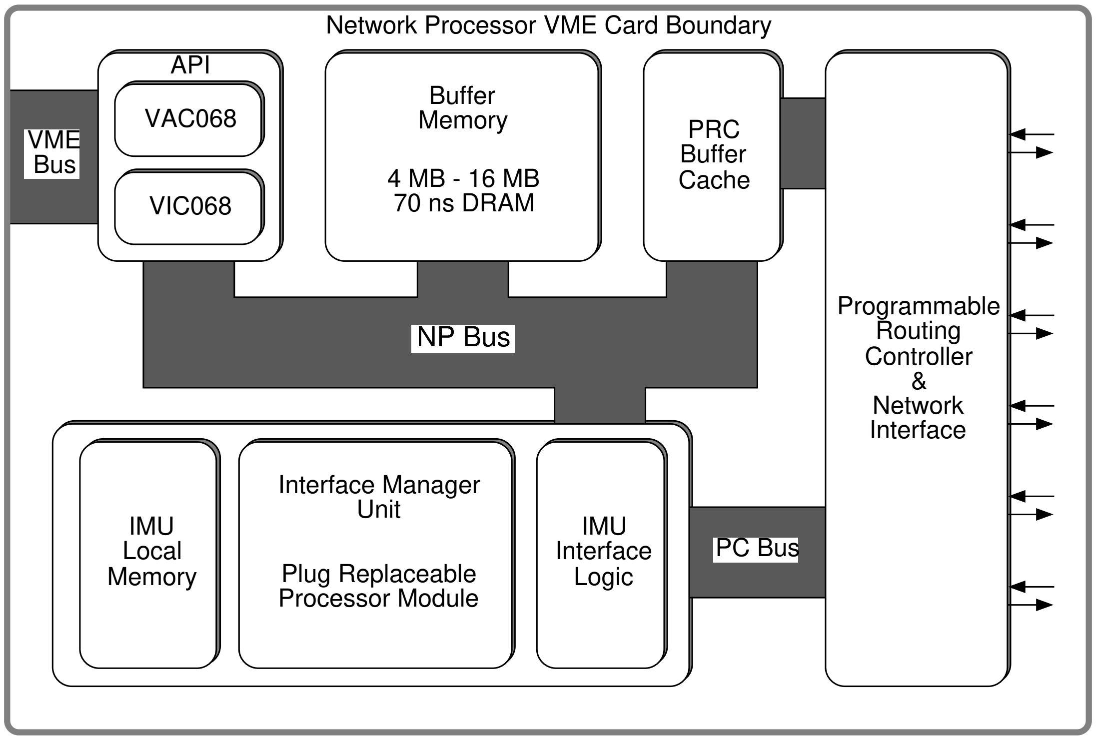


Figure 2.3: A block diagram of the NP.

detailed description of the PRC and NI to Chapter 3.

Interface Manager Unit: The IMU will control the operation of the rest of the NP. In particular, it will construct packets from the original messages and reassemble them at the final destination, schedule messages with different levels of priority, decide on routing strategies based on message priority levels and network load state, monitor the network state, perform error correction and message acknowledgment, and implement various real–time communication algorithms. One of the unique features of the IMU is the processor module. Rather than restrict this study to one particular processor architecture, the IMU is designed with a generic interface to a plug-replaceable processor module. This approach has many advantages when designing experiments that can be pursued with the NP. The processor module will also have external connectors that allow address and data traces to be collected in real-time.

In addition to its own local memory, the IMU will also have access to the buffer memory. To minimize the need of copying data, the buffer memory will in most instances serve as the data memory of the IMU. Hence, the buffer memory will be part of the address space of the IMU. Also under implementation is the separation of the headers from the body of the messages, because such a separation allows the expansion and contraction of the headers independent of the location of the message body. This separation allows the NP to reduce the broadcast and/or multi-cast latency when it is necessary to transmit copies of the message *simultaneously* over the different links. These copies would contain the same data and differ only in the header information.

Buffer Memory: The buffer memory subsystem consists of the RAM required for storing messages and a management unit. It will store messages that are waiting to be transmitted to or from the current node, and may act as a temporary storage area for messages routed through the current node. The amount of memory needed is being determined by the usage patterns of the application tasks and is not expected to be greater than a few megabytes.

The word size used for the buffer memory is 32 bits. With the current access speeds of DRAMs at 70 ns, this gives a memory bandwidth as high as 457 Mb/s. This bandwidth appears to be sufficient for access by the PRC, the API, and the IMU, and for refresh cycles.

The buffer manager arbitrates access to the buffer from the IMU, API, and the PRC. It also handles the refresh of the buffer memory by periodically accessing rows in the DRAM. The access priorities given to these different sources can be static, dynamic or random, depending on the buffer management policy being adopted.

Application Processor Interface: The interface between the NP and the APs is through a VME bus. Data copying between the AP and the NP will be done with the help of the API, which will have a DMA interface to the VME bus. There are two ways of designing this interface for data transfer: mapping the NP data memory into the APs' address space, or copying data from the APs' memory to the NP data memory. It may appear that mapping the NP into the address space of the APs is efficient as it avoids the overhead of a system call. However, this mapping requires dedicated memory management hardware and kernel support for mapped address spaces, and also incurs the overhead of data access over the VME bus. Depending on the size of typical messages, it may be more efficient to use burst mode DMA transfer from the APs memory to the NP memory.

The APs will initiate data transfer to the NP by writing to a control register in the API. The API will then contend for the host VME bus and the NP buffer memory. When both resources are acquired, it will proceed to copy the message data in burst mode directly from the host to the NP buffers. Upon completion of the transfer, the IMU will be notified, and the communication processing can begin. A similar sequence of operations will be performed in the reverse order for the reception of messages.

This interface is being implemented using the Cypress VIC068/VAC068 VME bus controller chip set. These devices provide a complete VME bus interface controller and arbiter, support burst–mode data transfers, and map local bus addresses to VME bus addresses.

2.4 Communication Terminology

Up to this point HARTS and the NP have been presented without precisely defining the meaning of *messages*, *packets*, *routing*, *switching*, and other terms commonly used when describing communication subsystems. In this section we discuss the communication paradigms and definitions that are identified to be relevant to the area of the design space that is explored as well as how these paradigms differ as some of the parameters of the design are varied. We will now present a set of definitions that closely follows the terminology and framework outlined by Reed and Fujimoto [40]. It is also important to recognize the location of HARTS, including the PRC, within the entire spectrum of communication architectures. This work investigates systems with a homogeneous point-to-point interconnection supporting coarse-grain parallel computations, rather than trying to operate in the domain of tightly-coupled multiprocessor networks. In terms of real systems this work is located somewhere between the high performance hypercubes/k-ary n-cube machines and the computation environment that can be provided by the Open Software Foundation's Distibuted Computing Environment (OSF/DCE).

2.4.1 Messages and Packets

In this framework a *message* is a unit of information that an entity, most commonly a user task on an application processor, submits to the communication subsystem for dissemination to a set of entities located elsewhere in the system. The set of entities will be referred to as the *destinations* of the message. The most common case is that the destination set identifies a single entity, which corresponds to a task sending a simple message to another task. At the other extreme, the destination set can identify all entities present in the system, in which case a message is being *broadcast*. The node in which the sending entity resides will be called the *source node*, and the nodes that correspond to thoses in which the destination entities reside will be called the *destination nodes*.

The communication subsystem, for efficiency reasons, may break the messages into a number of smaller *packets* with their sizes usually chosen from a set of fixed sizes. If this occurs, it becomes necessary to perform reassembly of the packets into the messages at each of the destination nodes. The PRC and the NI assume this to be the case and deal with packets. The API, on the other hand, usually deals with messages. This does not imply that parts of messages or packets are stored in contiguous memory in either the NI or the APs.

The NP derives its ability to act as an ideal tool for an experimental testbed by being able to move in several dimensions of the design space. The key dimensions that define its position are the switching and the routing paradigms, buffer management policies, and flow control issues. The switching paradigm defines how the message/packet is physically forwarded in the system. The routing paradigm determines paths that the messages/packets will traverse in order to reach the destination nodes. Buffer management determines how and where to store part or all of the messages/packets if communication links are busy. Flow control determines how the incoming traffic is controlled.

2.4.2 Switching Paradigms

When observing the different switching paradigms there are several characteristics that allow identification of the differences between various techniques. First, the unit for which resources are allocated will be called the *allocation unit*, and can vary from a single bit up to multiple messages. Second, the overhead associated with making the routing decisions that can be attributed to the switching technique will be identified as the *routing overhead*. Third, the policy by which the communication link bandwidth is allocated will be referred to as the *bandwidth allocation* policy. The final parameter is the *buffering complexity* imposed by the switching technique.

Point–to–point interconnection networks can be loosely classified as either *circuit-switched* or *store-and-forward.* The most common example of a circuit-switched network is the Plain Old Telephone Service (POTS) in which the switching mechanism is implemented via the Public Switched Telephone Network (PSTN). When a call is placed, a circuit is created between the source of the call and the desired destination. This circuit remains established until either party decides to terminate the call. In terms of the distinguishing characteristics the allocation unit is of arbitrary length, since it is equal to the duration of the call which is user-determined. The routing overhead is only incurred as the circuit is being established. The telephone company provides a guarantee that the voice-grade line will be able to carry signals from approximately 300Hz to 3300Hz; hence the bandwidth allocation is static. Finally, there is no need to provide any support for buffering. Another example of this technique involving computer data is the circuit-switched data networks based on the X.21 protocols.

One of the common objections to circuit-switched networks is that since the bandwidth is statically allocated using the peak signal rate, much of the network bandwidth is wasted if the communication is sporadic in nature. Store-and-forward networks avoid this problem by allocating the communication link to either a packet or a message. Store-and-forward networks can be further subdivided to into those supporting *datagram* service, *packet-switched* service, or *virtual circuit* service.

A comparison of these key characteristics is shown in Table 2.1. One of the main points that can be noted from the table is that both packet-switched and datagram-based services incur routing overhead on every hop that the packet must take to reach its destination. Virtual circuits are an attempt to combine the best points of both circuit-switched and

	Store-and-forward type of service		
	Datagram	Packet-Switched	Virtual Circuit
Allocation Unit	Message	Packet	Packet
Routing Overhead	per Message	per Packet	once per Message

Table 2.1: Basic Types of Store-and-Forward Networks

packet-switched techniques. The routing overheads are mainly incurred during the circuit establishment, during which the route through the network for a particular message is established. Subsequent packets are routed according to the circuit path that was previously established. Virtual circuits are not without their disadvantages. Since virtual circuits obtain their reduced routing overhead by storing state information in the network, the presence of node or link failures can be problematic.

Two more recently proposed variants on the above switching methods are *virtual cut-through switching* and *wormhole routing*. Virtual cut-through switching, a variant on packet-switching, was first presented and its average performance analyzed in [26]. Virtual cut-through switching reduces packet latency by immediately forwarding the packet out of an intermediate node the moment the node's outgoing link on the packet's path is available, even if the packet has not been received in its entirety. The determination of the outgoing link can only be made when the destination of the packet is known, and this information is normally found in the header of the packet. Hence, virtual cut-through can begin only after the header of the packet has been received, and when the outgoing link is available. Should the link be busy, the packet is *buffered* at the node itself.

Wormhole routing, a variant on circuit switching, is similar to virtual cut-through switching in that a packet is forwarded to the next node once the header is received and the outgoing link is available. The difference lies in the way the packet is handled when the packet is not able to cut through a node. Whereas virtual cut-through buffers the packet into the storage area of the node, wormhole routing buffers only a small portion of the packet in an on-line buffer while signaling to the preceding node to stop transmission of the packet. All the communication links from the source to the blocking node — the node whose outgoing link is busy — continue to be reserved for use of the packet. Viewed from a different perspective, wormhole routing is circuit-switching with the data portion of the packet being sent immediately after the circuit establishment information. One benefit of this, as compared to virtual cut-through, is that links need not be reacquired once they have been initially acquired when encountering a busy link. Thus, one can save the time needed to buffer the packet into a blocking node, and retrieve it from the node's buffer memory when the associated outgoing link of the blocking node becomes free.

One problem that wormhole routing has, as a consequence of not relinquishing acquired links when a packet is blocked, is the possibility of deadlock occurring in the network. Since packets can hold a link while simultaneously requesting the use of other links, and circular waits are possible in the network, conditions are satisfied for the possibility of deadlock. Deadlock-free routing algorithms have been discussed previously in [10] to get around this problem and usually involve multiplexing multiple channels on the physical links and some ordering constraint on how the links are assigned.

2.4.3 Routing Methods

As mentioned earlier, routing is the process that selects the path(s) that a packet will traverse when traveling from the source node to the destination node(s). When examining the possible routing methods that are appropriate for the domain under study, techniques can be used from both the loosely-coupled networks above and the tightly-coupled multicomputer networks below. Since both the IMU and the PRC are programmable devices, it isn't necessary to focus on any particular algorithm at the moment. However, it is important to ensure that the architecture can support a reasonable number of the possible alternatives.

There are many properties that can be used to classify routing algorithms [39, 47, 46, 23, 41]. Figure 2.4 shows a simplified collection of properties that will be considered for the discussion of routing algorithms.

Routing decisions can be either *distributed*, *centralized*, or *directed (source-list)* by the source node. For distributed routing algorithms, in which each intermediate node along the path participates, the amount and source of information required to make the routing decisions is important. Centralized and distributed algorithms that depend on non-local information suffer from the need to either collect or disseminate the state information throughout the network. This adds additional complexity and uses a portion of the available network bandwidth. The information type and quantity can vary from no information to complete global information with algorithms using something in between these extremes, providing the most interesting results. For example, algorithms using information only available at the local node or information available up to k-hops distant from the node have provided good performance without unduly loading the network.

Routing algorithms can also be classified as either *static* or *adaptive*. A routing scheme is static if the path traversed by the packet only depends on the source node and the destination node. This path is not influenced by the presence of other packets or resource

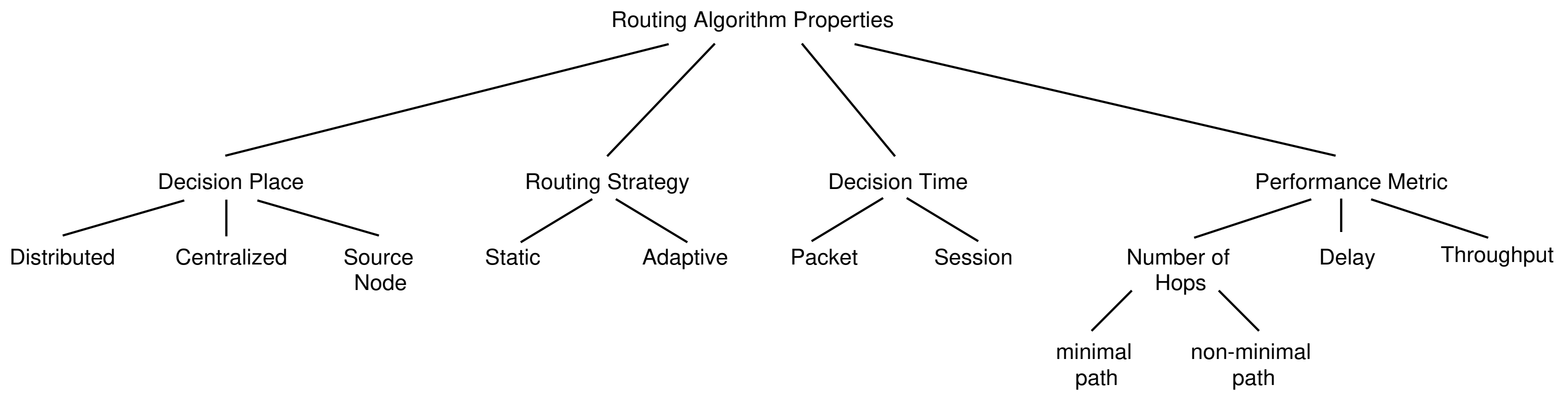


Figure 2.4: Classification of routing algorithms

conflicts. For an adaptive scheme, the path may be and usually is influenced by other entities in the network. Static schemes have the desirable properties that they are usually simple to implement and preserve the order of packets sent between nodes. Unfortunately, these advantages are gained at the cost of higher packet latencies and the inability to deal with any type of traffic congestion. Adaptive schemes overcome some of these disadvantages but at the cost of algorithm complexity and can introduce problems of deadlock.

Another characterization of routing algorithms is the performance metric of the number of hops the packet will traverse when being routed between a source–destination pair. These algorithms are usually characterized as either providing *minimal* or *non-minimal* routing. For minimal routing, only routes which traverse a minimum number of hops for the particular source-destination pair and topology are considered. In non-minimal routing, packets may be routed around busy or broken links and this approach can improve performance and tolerance to link and node failures. The greater flexibility offered by non-minimal routing can be quite costly, especially if the introduction of packets that have been "misrouted" introduces the possibility of livelock.

Finally, when and how often the algorithm must be executed is another way of characterizing routing algorithms. This is clearly demonstrated when comparing virtual circuits versus packet switching. For virtual circuits, the routing algorithm need only be executed at each node during the establishment of the circuit through the network. All subsequent packets will use the circuit identification registered by the first packet and will be routed with practically no overhead. This is in contrast to packet switching, in which the routing algorithm must be executed at every node for each packet.

CHAPTER 3

PROGRAMMABLE ROUTING CONTROLLER

This chapter presents the architecture and the implementation of the Programmable Routing Controller (PRC). This device is a custom-designed application-specific IC (ASIC) that has been developed using Cascade Design Automation's EPOCH system and Cadence Design Systems' Verilog-XL. The PRC combines the functionality of the previously-designed Routing Controller [20] and Packet Controller [15] as well as exploiting the availability of the AM79168 TAXIchip Transmitters and the AM79169 TAXIchip Receivers[2, 1]. This design greatly improves performance over its predecessors while preserving the experimental flexibility and support for distributed real-time communication required for HARTS. Since the design of the PRC and the Network Interface (NI) are closely-coupled and interdependent, the architecture and interface requirements of the NI are also discussed in this chapter.

We open with a high-level description of the architecture and then illustrate the different components through several packet-routing scenarios. After having demonstrated how the PRC and NI interoperate we show how the PRC deals with some of the fundamental issues that must be answered during the design of any communication subsystem. We conclude by comparing our design with several other systems.

3.1 Architectural Overview of the PRC and NI

Interaction with the PRC is carried out through three distinct interfaces: the Page Control Bus (PCBUS), the Cut-Through Bus (CTBUS), and the NP Bus (NPBUS). This structure can be seen in Figure 3.1. The PCBUS is a bidirectional 32-bit bus that provides a communication path used by the IMU to control the PRC. This bus, with some additional decoding logic, is then used to map the PRC into the IMU's physical address space as a memory-mapped I/O device. The CTBUS is an 8-bit bidirectional, time-division multiplexed bus that facilitates communication between the I/O devices of the Network Interface (i.e., AMD TAXI Transmitters and Receivers) and the internal components of the PRC. Arbitration for a slot on the CTBUS is implemented on the PRC and supports a centralized demand assignment allocation policy that is based on a binary priority tree. The NPBUS interface stores data in and retrieves data from the buffer pool (or PRC cache) of the NP. This interface is a master-only interface; all traffic through this interface is initiated by the PRC.

Most of the internal architecture of the PRC is organized around the concept of a physical channel. The PRC has twelve such channels, each of which is dedicated to communication over a particular unidirectional link. There are six "inbound" channels (one for each receiver interfaced to the CTBUS), and six "outbound" channels (one for each transmitter interfaced to the CTBUS). Most of the components of the PRC can be classified as providing support for either an inbound channel, an outbound channel, or an interface unit. For example, the six transmitter fetch units in the lower center part of the PRC in Figure 3.1 support data transmission on the six "outbound" channels. The *Cyclic Redundancy Code* (CRC) check and CRC generator units are used for manipulating the CRC values for the "inbound" and "outbound" channels, respectively. The microprogrammed receivers are used to support both "inbound" and "outbound" channels. This support takes the form of low-level route determinations and reservations of transmitters in the NI for packets that are attempting to cut through the node. The receivers also perform the more mundane tasks of byte-to-longword construction and control of the CRC check on packets destined for the communications subsystem residing at the PRC's node.

Like the PRC, the NI is organized around twelve physical channels. For each receiver in the PRC the NI has a corresponding receiver that recovers data from the physical interconnect and presents the data in a parallel form to the CTBUS. This functional coupling is depicted in Figure 3.2. "Outbound" transmission is supported by the NI with six transmitters that take parallel data from the CTBUS and perform the physical transmission on the interconnect. Under PRC control, the receivers of the NI can be configured to automatically forward incoming data to either a set of NI transmitters, a PRC receiver, or both. The NI transmitters in conjunction with the CTBUS protocol support a command channel and data channel in the forward direction and a command channel in the reverse direction. The command channels are used to implement the low-level flow control between adjacent nodes.

3.2 Packet Routing Example

Rather than simply listing each of the components and their respective functions we will present the internal structure and operation of the PRC by describing the activities required to transmit a packet, receive a packet, and assist in establishing a cut-through. The block diagram presented in Figure 3.1 will be the focus of our discussion.

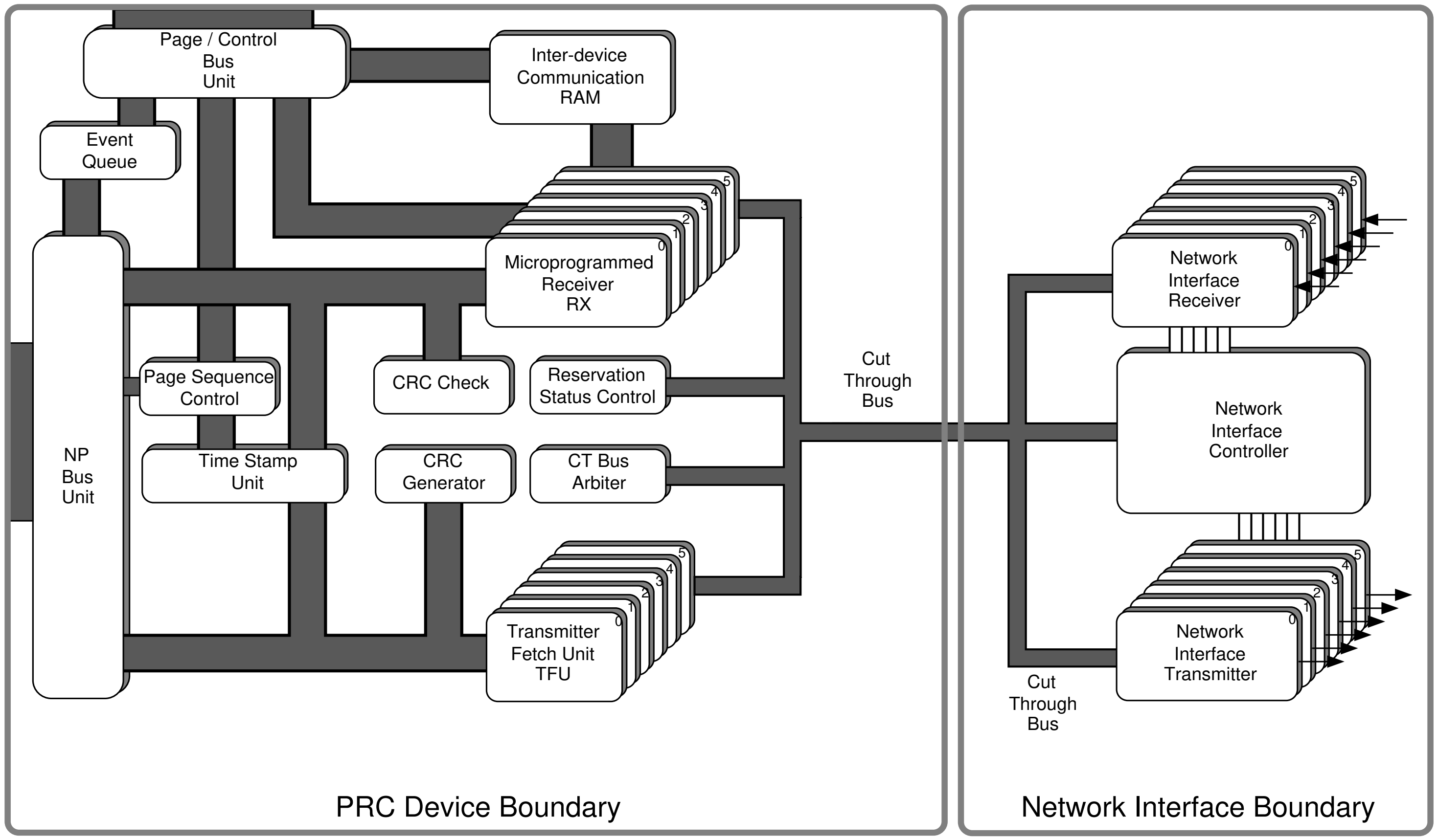


Figure 3.1: Programmable Routing Controller architecture

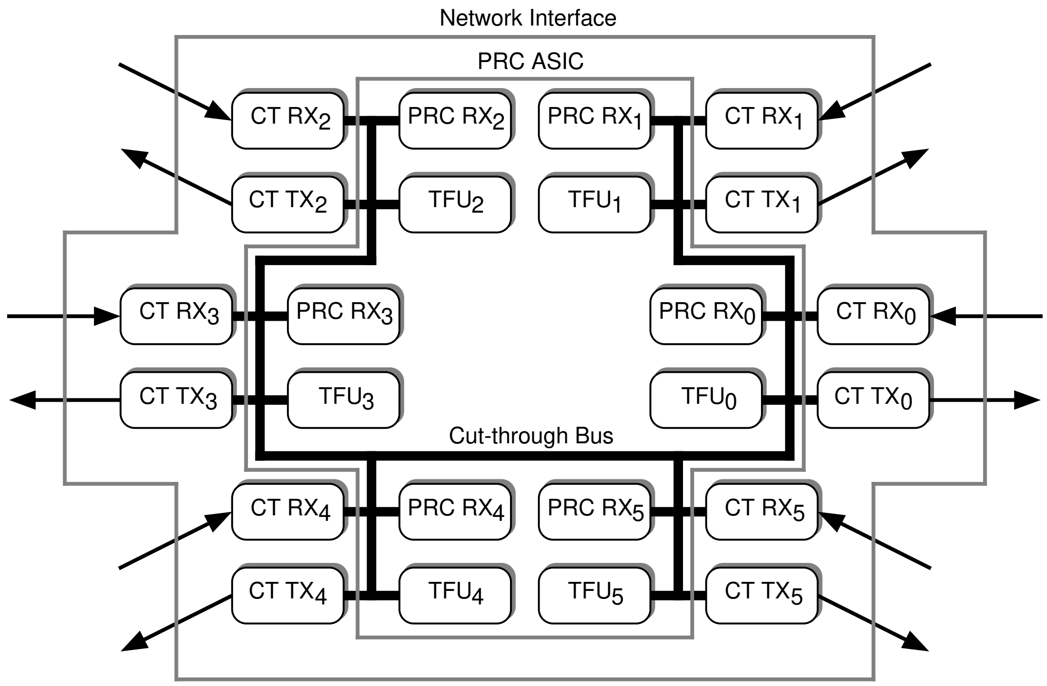
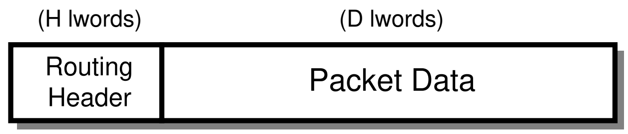


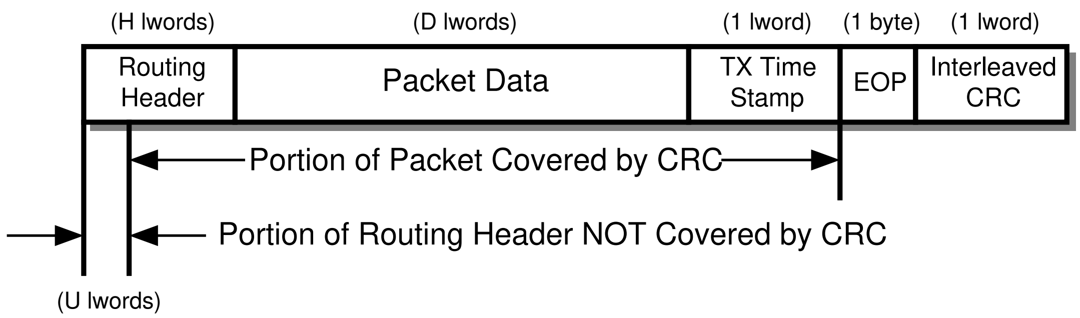
Figure 3.2: PRC to NI functional coupling

Before proceeding to the examples, the packet format and other restrictions imposed by the PRC need to be clarified. Figure 3.3 summarizes the three packet formats that the PRC uses. There are several important points to note in the different formats. First, time-stamps are added by the transmitting and receiving PRCs. These time-stamps can be accumulated to obtain the total transmission time of a message and are useful for distributed information services such as clock synchronization and reliable broadcast as described in [38]. The accumulation of time-stamps and modification of the packet size in order to accommodate the time-stamps is the responsibility of the IMU. Second, a portion of the routing header is not covered by the CRC. This allows the PRCs along the path that the packet traverses to modify the routing headers without invalidating the CRC covering the rest of the packet.

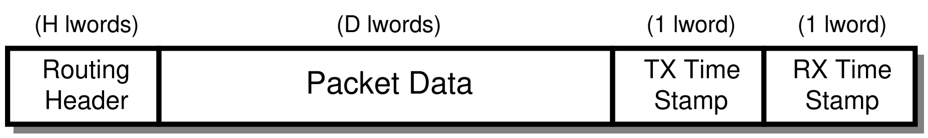
Packets that are transmitted and received by the PRC are stored in NP buffer memory in a linked page structure similar to the indirect MBUF cluster structures used by many UNIX



(a) Prior to Transmission



(b) Transmitted on Physical Links



(c) After Reception

$$H < 64$$

$$U < min(H, 16)$$

Figure 3.3: Packet formats

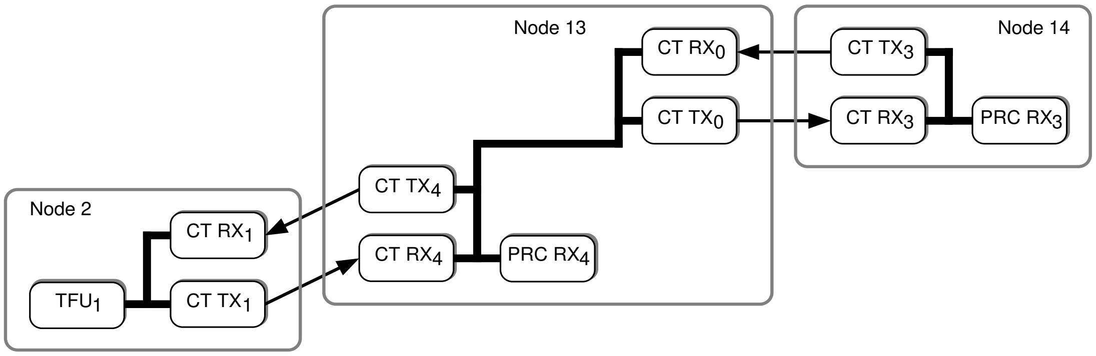


Figure 3.4: Components used in common routing example.

communication subsystems [32]. Furthermore, the pages referenced by these structures have the following characteristics: the page size is 256 bytes, pages are longword aligned, and pages must contain an integral number of longwords starting from an offset of 0 from the base of the page. This structure, along with the offset indicating where to start the CRC calculation is conveyed to the PRC by the IMU through a stream of *page control tags* (PCT). Each PCT is a longword containing four data fields: the unchecked length, a last page flag, the page base address, and the length of data within the page. The IMU manages a separate stream of these PCTs for each of the outbound channels and a single stream for the inbound channels.

Throughout the remainder of this section we will expand the routing example presented in Section 2.1 and describe the transmission of a packet from the buffer memory on node 2 to node 14 (through the intermediate node 13). This common example as depicted in Figure 3.4 will allow us to explain packet transmission at node 2, the PRC's role in supporting a cut-through at node 13, and packet reception at node 14. For our example, the packet will consist of 75 longwords of data occupying three partially-filled pages in the buffer memory, as shown in Figure 3.5. Five of these longwords will make up the header of the packet and are stored at address `0x0C220000`[1]. Only the first longword of the header will not be covered by the packet CRC. The remainder of the packet is split into two blocks of 25 and 45 longwords, stored at addresses `0x0C225000` and `0x0C222000`, respectively.

3.2.1 Packet Transmission

Once the IMU has selected a packet for transmission and verified that there is space in the *page control tag queue* (PCTQ) for the appropriate channel, the IMU loads the PCT for the packet into the channel's PCTQ. If space allows, the IMU may load up to three more PCTs for the successive pages of the packet. At this point responsibility for the transmission of the packet is taken over by the PRC, and the IMU must only provide a stream of PCTs. Requests for more PCTs are conveyed through the PRC's interrupt mechanism.

The PRC retrieves packet data from the buffer memory via the NPBUS, stores it temporarily in the *transmitter fetch units* (TFUs), and then places it into the CT devices, one byte at a time. The TFUs, each of which is associated with a particular outbound channel (and thus a single CTBUS transmitter), control the process of packet transmission.

When idle, each TFU constantly monitors its PCTQ for the presence of a PCT. When the TFU finds a PCT (as the result of the IMU loading the PCT through the PCBUS), packet transmission commences, following the algorithm described in Figure 3.6. This code, while not complete, describes the general procedure for packet transmission.

At the start of a packet, the TFU retrieves the unchecked length from the PCT at the head of the PCTQ and stores it internally. Until this value has been decremented to 0, the retrieved data is excluded from the calculations of the CRC for the packet. This allows packet headers to be altered by the PRC receivers without affecting the error detection for

[1]The PRC can only address 4M longwords and is mapped into the buffer memory at base address `0x0C000000`

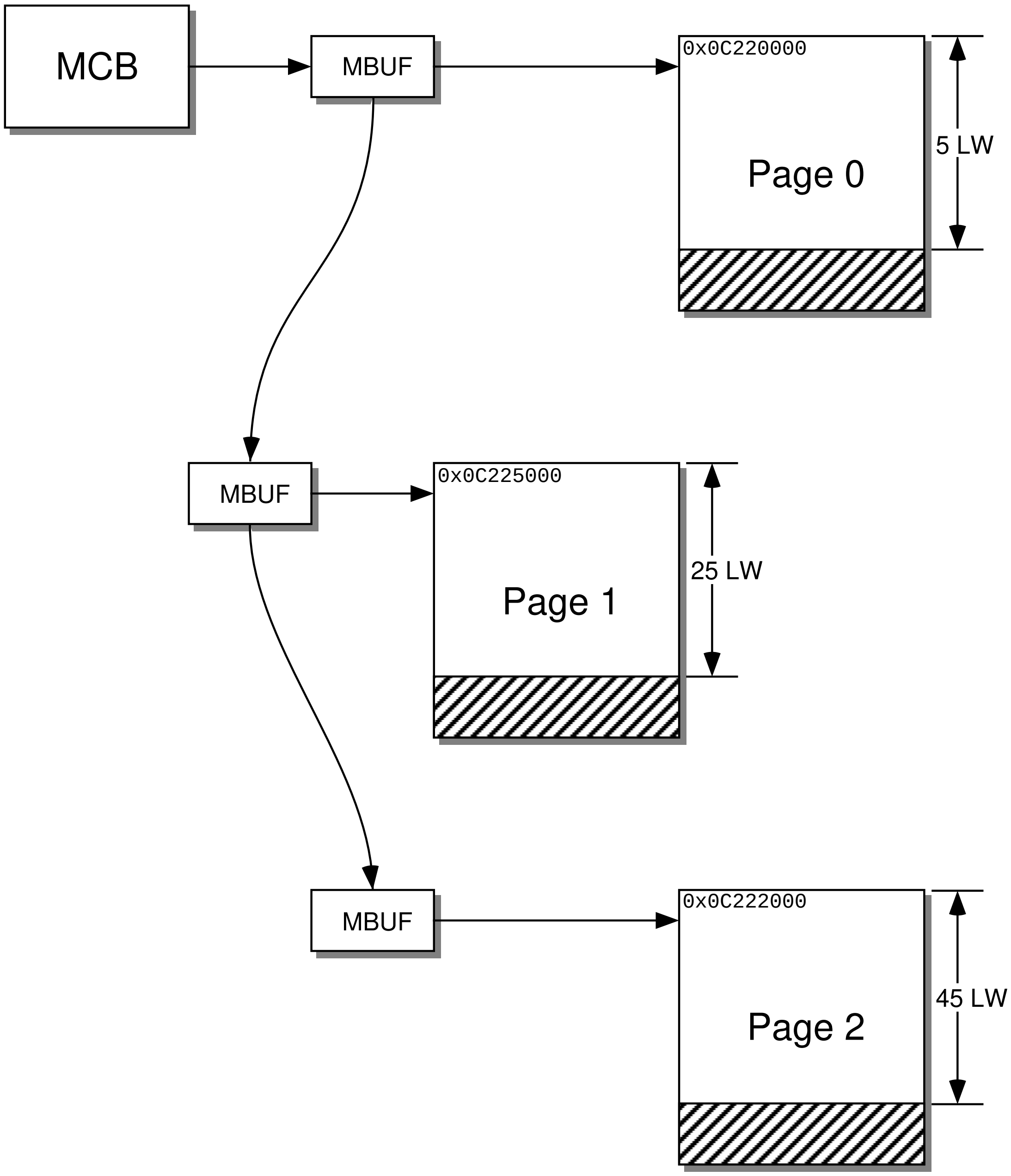


Figure 3.5: Example packet structure

```
transmit_word:
        if(eopsig)
                request_ctbus;
                transmit_byte(eop);
        if(marksig)
                request_ctbus;
                transmit_byte(mark);
        for (i = 3; i ≥ 0; i - -)
                request_ctbus;
                transmit_byte(i);

transmit_packet:
        while(transmitter_reserved);
        reserve_transmitter
        eopsig = false;
        while ((unchk > 0) & ¬eopsig)
                eopsig = fetch_word(data, ¬CRCENBL);
                transmit_word;
                unchk = unchk - 1;
        while (¬eopsig)
                eopsig = fetch_word(data, CRCENBL);
                transmit_word;
        fetch_word(time-stamp, CRCENBL);
        transmit_word;
        fetch_word(CRC,¬CRCENBL);
        transmit_word;
```

Figure 3.6: Pseudocode description of TFU operation

the remainder of the packet. Simultaneously, the TFU attempts to reserve its associated transmitter and issues a request to the NPBUS for the first longword of the packet. The TFU checks the reservation status of its associated transmitter, and if the transmitter is currently unreserved, requests access to the CTBUS to reserve the transmitter.

Once the first longword of the packet has been retrieved from the buffer memory and a reservation achieved for the appropriate transmitter, packet transmission begins. The TFU requests access to the CTBUS, and when this is granted, drives the CTBUS data lines with byte 3 (the high byte) of the longword in its local buffer. This word and its control information are latched by the previously reserved CTBUS transmitter and transmitted to the adjacent node.

The TFU now spins until the **IRDATA** line[2] of its CTBUS transmitter returns to TRUE. When this is observed, the TFU again requests control of the CTBUS, placing the next byte of data on the CTBUS when this access is granted. This process continues until the TFU has transmitted all four bytes of data in its local register. In parallel with the transmission of the bytes associated with the current longword, the TFU prefetches up to 4 longwords. Hence, once the fourth byte of the current longword been transmitted, the TFU should be able to immediately switch to transmitting the next longword.

The PRC design allows the page structure of a packet to be preserved during transmission (to allow received packets to only partially fill their pages). This is done by tagging the longword requested by the TFU with a flag indicating that the associated data is the last longword on a page. If this is the case, the TFU will transmit the CTBUS "mark" command. This mark can then be used by downstream nodes to reconstruct the page structure. This same tag when used in conjunction with the last page bit of the PCT is used to detect when the TFU has reached the end of the current packet. In addition to the EOP that must be sent when dealing with the end of packet condition, two additional longwords are transmitted: the first is a time-stamp representing the time at which the transmission of the packet was completed; the second (transmitted after the EOP) is a CRC for error detection.

Returning to our example, packet transmission begins when the PCT (`0x02220005 (Addr=0x0C220000/Len=05/Unchk=1/Lastpage=false)`) written by the IMU at node 2 reaches the head of the PCTQ of TFU_1. This causes TFU_1 to attempt to reserve its CTBUS transmitter (CTBUS TX_1 connected to node 13) and also to retrieve the longword stored

[2]A line used to indicate the CTBUS transmitter's willingness to accept forward channel data. The detailed semantics are explained in the description of the CTBUS implementation.

at `0x0C220000`. This first longword will not be included in the future CRC calculations since the unchecked length counter is greater than zero. Once the reservation has been achieved and the data retreived, the TFU begins sending the data over the CTBUS in one-byte chunks. The first NPBUS access also decrements the unchecked length counter to 0; hence, all future longwords retrieved from the buffer memory will be included in the CRC calculation.

A page fault is generated when the data word at `0x0C220004` is retrieved. This has several effects: the PCT at the head of the PCTQ is shifted out, the page offset is restored to 0, and the "mark" code is sent before transmitting the next byte of data. This code notifies the destination PRC that a page fault has occurred.

The value `0x00225019` (`Addr=0x0C225000/Len=25/Unchk=0/Lastpage=False`) now appears at the head of the PCTQ, and the next memory access is to location `0x0C225000`. Data is retrieved and transmitted as before, and a page fault eventually occurs when `0x0C225019` is accessed, bringing `0x0122202D` (`Addr=0x0C222000/Len=45/Unchk=0/Lastpage=True`) to the head of the PCTQ. When the page fault is generated at the end of this page, however, the NPBUS interface logic will also log a TX EOP event, since the **lastpage** bit is currently set. The retrieved longword is transmitted normally, but the TFU exits the main data transmission loop. On the next data request, the TFU sets its control lines to indicate that it needs to read the time-stamp register. The time-stamp is latched in and transmitted; when this is finished, the TFU retrieves the CRC. The CRC is transmitted across the links; the only difference is that this final longword is preceded by a EOP signal rather than a MARK signal. Once the final byte of the CRC is acknowledged, the TFU accesses the CTBUS one final time and unreserves the transmitter. The PRC is designed so that the time-stamp and CRC accesses do not reach the NPBUS interface; thus, the inbound channels can use the NPBUS while these accesses are made.

3.2.2 Packet Cut-Through and Reception

Packet reception by the PRC begins when data is transferred from a CTBUS receiver to the PRC receiver (PRC RX) associated with it. Each receiver is an 8-bit micro-controller intended to perform serveral functions: routing mode identification, route determination, channel establishment, and packet reception.

Upon detecting the arrival of data, the receiver first allows each data byte to be acknowledged until it has accumulated enough of the routing header to determine how the packet is being routed and the packet's final destination. It then stops signalling its willingness to accept data (stalling the inbound channel) until a routing determination has been made and the routing header is flushed by buffering at the local node and/or by retransmitting it out through a CTBUS transmitter(s). Once the routing strategy for the packet has been determined, the receiver must determine which resources are available for a packet of this priority. If the RX decides to forward the packet, it attempts to reserve the appropriate CTBUS transmitter(s). If this attempt succeeds, the RX establishes the channel cut-through and reconfigures the inbound channel to forward to the outgoing link.

If the packet is destined for the local node or the RX is unable to immediately forward the packet, it is usually buffered in the local node. In order to support broadcast algorithms the design of the CTBUS and the RX allow the RX to simultaneously buffer and forward a packet. In cases where local resources are scarce or a deadlock may be present, the RX may even drop the packet entirely by acknowledging it and then simply losing the data internally (i.e., neither buffering nor forwarding it). The sender at a higher level in the communication protocol stack will retransmit the packet after the corresponding timeout period elapses.

When a packet is stored at the local node, the data is transferred from the PRC RXs to the buffer memory via the NP interface of the PRC. Once an RX has accumulated a full longword, it places the data in the RX FIFO (access is arbitrated by the same scheme as the CTBUS). The bus connecting the PRC RXs to this FIFO is known as the RXBUS. The CRC check is computed during this storage. At the end of the packet, as the receive time-stamp is read out from the local clock and placed into the RX FIFO, a flag is set to indicate the outcome of the CRC check. Data is then retrieved from the FIFO by the NP interface and stored in the buffer memory.

We return once again to our packet reception example. The first event at node 13, indicating the begining of reception from the perspective of PRC RX_4, is the availability of data in its inbound CTBUS interface. The PRC RX_4 removes this data and stores it in an internal FIFO (after possible modification) and signals to the CTBUS receiver its willingness to accept more data. This process then continues until the PRC RX_4 has accumulated enough of the routing header to make a routing decision. In the case of the algorithm presented by Chen *et al.* [6], the first 4 bytes of data are sufficient to determine the packet's type, its priority, and its destination. Based on this, the PRC RX_4 can then choose how to handle the packet.

Since the current node (13) is not the final destination of the packet, the PRC RX_4 determines that it should check the reservation status of the CTBUS TX_0. If the transmitter is free, PRC RX_4 simply reserves the transmitter and transmits the data in its internal FIFO to node 14 via this transmitter. After all the data buffered in the internal FIFO of PRC RX_4

has been sent, the CTBUS receiver (CTBUS RX_4) is configured to forward all received data to CTBUS TX_0 without interacting further with the PRC RX_4.

On node 14, a similar process takes place. Since the packet is destined for this node, PRC RX_3 simply leaves the CTBUS receiver (CTBUS RX_3) configured in this default mode which is to transfer all received data to PRC RX_3. As data is received from the CTBUS, the PRC RX transfers it to the RXBUS interface for storage in the buffer memory. When *mark* commands are received from the CTBUS, PRC RX_3 can choose to convey them to the NP interface as page faults and thus preserve the page structure of the packet or ignore them, thereby compressing the incoming packet into a minimal number of pages. After receiving the EOP command, PRC RX_3 instructs the RXBUS interface to store the receive time-stamp in the RXFIFO, thereby reading out the CRC error flag and signaling an RX EOP event.

3.3 PRC Implementation

There are many specific issues that need to be addressed when approaching the implementation of any communication subsystem [4, 52, 51, 16, 34, 24, 29, 7] These include accessing shared resources, flow-control, error detection, and routing. This section discusses how some of these issues were resolved in the implementation of the PRC.

3.3.1 Media Access and Flow Control

The primary shared media in the PRC design is the CTBUS, an 8-bit wide Time-Division Multiplexed (TDM) bus that provides a gateway into the interconnection network. The CTBUS and its associated protocol play a fundamental role in supporting many of the features provided by the PRC. The CTBUS is necessary for the implementation of multiple switching techniques such as virtual cut-through and wormhole routing. It also provides the primitives necessary for low-level flow control and broadcast algorithms [25]. Finally, since the CTBUS is a critical resource, the arbitration method emphasizes fairness and scales well under varying loads.

The CTBUS interconnects 25 separate components: 6 CTBUS transmitters, 6 CTBUS receivers, 6 PRC receivers, 6 PRC transmitter fetch units, and the Network Processor's IMU. Of these 25 devices 19 can act as "master" and can participate in arbitrating for control of the CTBUS (only CTBUS transmitters do not function as masters). All of the devices except the PRC transmitter fetch units and the IMU can also function as "slave" devices on the CTBUS. The external CTBUS signals are functionally divided into five groups: master, control, address, arbitration, and data. The master and address lines identify both the device that has control of the bus (the master) and the selected slave device(s) for the current bus cycle. The control lines are driven by the master device and identify the current command.

Bus Arbitration Scheme

Arbitration for the CTBUS is based on a binary priority tree in which the different levels of the tree are permuted according to the state of a counter. This scheme is derived from the work of Kovaleski *et al.* [28] and provides the basis for a demand slot access method. This scheme was chosen for its “fairness” and scalable performance over varying loads. Figure 3.7 shows an example in which we wish to construct an arbiter for eight masters. In this example a 3-bit counter is used to control the permutation of the tree. This counter is hooked up such that the least significant bit controls the nodes just above the leaves, the middle bit controls the nodes below the root, and the most significant bit controls the root of the tree.[3] The order in which access to the bus will be granted is the left-to-right order of the leaves. For cycle 0, the priority order is $0, 1, 2, 3, 4, 5, 6, 7$ with device 0 having the highest priority. To obtain the order for cycle 1 we exchange the left and right children of the tree used for cycle 0 and obtain a device ordering of $1, 0, 3, 2, 5, 4, 7, 6$. The device orders for remaining cycles can be read off from the figures. One of the main advantages of this type of arbitration scheme is that no device has higher priority than any other device over the major period[4] of the priority cycle. For our simple 3-bit example device 0 has higher priority than device 4 four times (Cycles 0,1,2,3) and it has lower priority four times (Cycles 4,5,6,7). Another advantage is that access to the bus scales well over a wide range

[3] In the full CTBUS arbiter the bits are not a simple linear assignment to prevent master devices from falling into a lockstep access pattern in which a particular master has an unfair advantage when accessing the bus in a periodic fashion.

[4] Where the *major period* would be 8 bus cycles for the example and 32 cycles for the CTBUS managed by the PRC.

of bus activity. If a device does not wish to use the current-cycle access, then access will be granted to devices of lower priority that need bus access. This means that in a PRC with only a single channel active, the entire bandwidth of the CTBUS is at its disposal.

The actual arbitration for the CTBUS is implemented by the PRC with arbitration for the next bus cycle overlapped with the current cycle. A device which has been granted master status may suspend the arbitration until it is finished with the bus. This allows for short bursts of multiple bytes and can be used to speed up the flow-control handshaking described below.

More specific details concerning the actual arbitration order or device addressing may be found in Appendix A.

Commands, Flow Control, and Broadcast Support

In order to present the measures undertaken to provide hardware support for broadcast algorithms and flow control, it is necessary to provide a brief overview of the commands used in implementing the CTBUS protocol. The commands shown in Table 3.1 can be categorized as supporting either device configuration or forward channel data transmission. The configuration commands support communication with the Network Interface Controller, resource management, and CTBUS receivers. The forward channel commands provide for transmission of data and data delimiters.

Each device on the CTBUS that can function as a slave and respond to a forward channel command has an *Input Ready Data* (**IRDATA**) line that signals the device's willingness to accept data on the forward channel. A master device uses the status of the **IRDATA** line in deciding if it should attempt to acquire the CTBUS and perform a transaction. This line forms the basis of the CTBUS flow control mechanism.

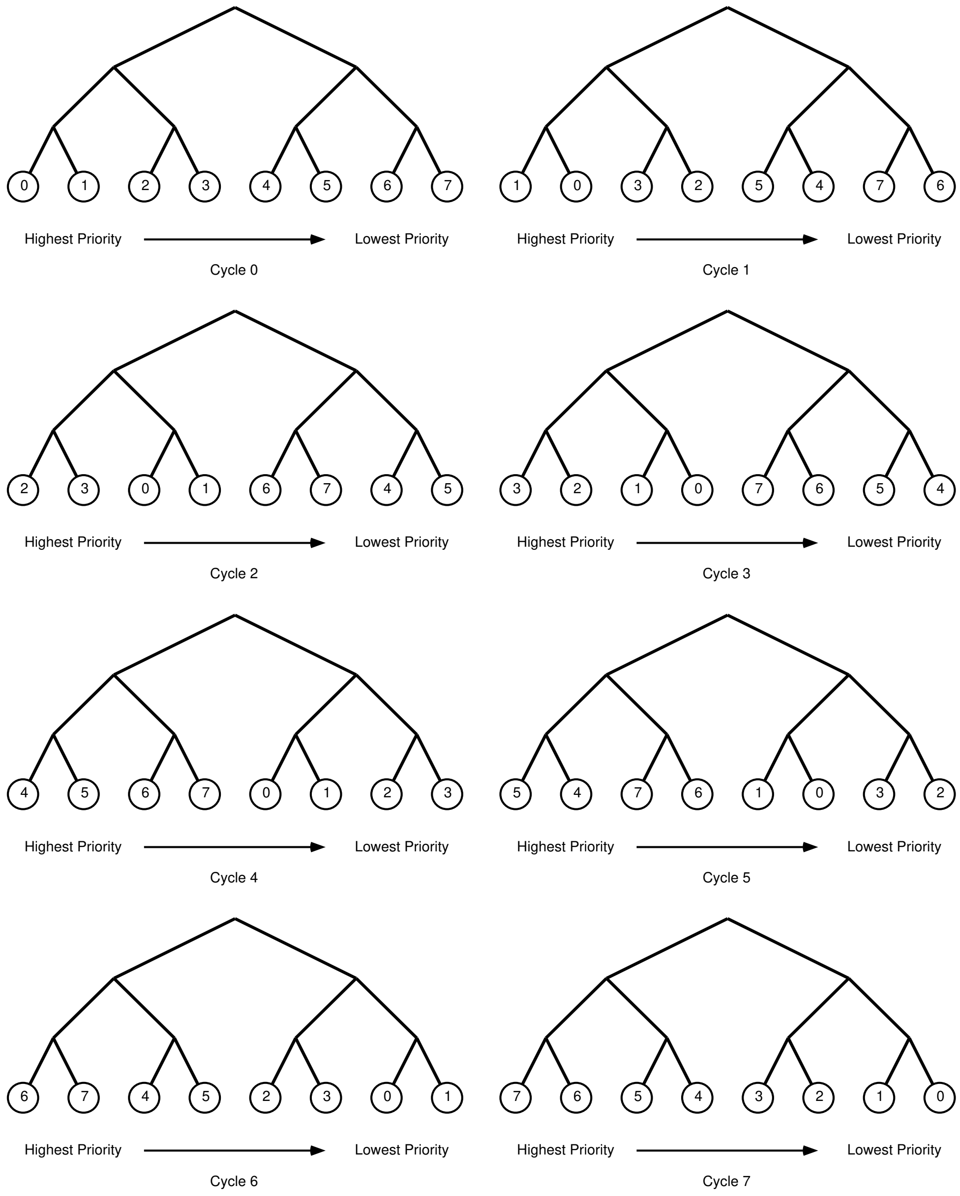


Figure 3.7: Binary tree priority arbiter example

Cut-Through Commands	
Command	Channel Direction
NOOP (No Operation)	N/A
DTX (Data Transfer)	Forward
RESV (Transmitter Reservation Request)	Configuration
FREE (Transmitter Free Request)	Configuration & Forward
HOLD (Transmitter Hold Request)	Configuration
CHK (Transmitter Hold Check)	Configuration
CFG0 (Load CTBUS Receiver Address Reg 0)	Configuration
CFG1 (Load CTBUS Receiver Address Reg 1)	Configuration
MARK (Byte Marker)	Forward
EOP (End of Packet)	Forward

Table 3.1: Cut-Through Bus Commands

The set of policies outlined in Figure 3.8 provides support for several of the primitives necessary to satisfy the design goals of the PRC. First, the use of the **IRDATA** line implements a tightly-coupled byte-level flow control mechanism that will prevent data overruns in both the Network Interface and the PRC. Second, making the multiple **IRDATA** lines available to the appropriate bus masters and using an address mapping that allows the selection of multiple slave devices provides efficient hardware support for broadcast algorithms [25]. This feature is significant in that it allows a one-to-many transaction in a single bus cycle. Last, at the cost of providing the **IRDATA** lines to the possible CTBUS masters, unnecessary traffic is greatly reduced when compared to schemes in which the current CTBUS cycle would have to be canceled if all the devices addressed during the transfer were not ready to accept the data.

Support for Multiple Routing Strategies

The primitives necessary to support multiple routing strategies are provided by making the CTBUS transmitters "reservable" resources. Prior to using a forward data channel, bus masters must have obtained a reservation by arbitrating for the CTBUS and issuing

- No CTBUS master will attempt a forward channel command unless all of the **IRDATA** lines of the selected CTBUS transmitters are true.
- When a CTBUS transmitter is addressed as a slave and a forward channel command has been issued the **IRDATA** line for that transmitter will become false until the command is properly acknowledged (as explained below).
- Upon receiving forward channel data, a CTBUS receiver will arbitrate for the CTBUS and forward the data once all of the **IRDATA** lines of the devices selected by its internal address register are true. After the data has been sucessfully forwarded, the CTBUS receiver will indicated to its paired transmitter[a] to acknowledge the last forward channel command.
- upon receiving a reverse channel command indicating acknowledgement of the last forward channel command, a CTBUS receiver will indicate to its paired transmitter that the transmitter may raise its **IRDATA** line.

[a]The term *paired transmitter* refers to the CTBUS transmitter that is connected to the same neighboring node as the receiver.

Figure 3.8: CTBUS protocol policies

a *RESV* command. If the command is positively acknowledged during the bus cycle, then a reservation has been granted and the bus master has sole control of the forward data channel. Once the bus master is finished with the forward channel and wishes to surrender its reservation, it arbitrates for the CTBUS and issues a *FREE* command. In addition to releasing the CTBUS transmitter on the current node, the *FREE* command is transmitted down the forward channel providing a mechanism to dismantle any circuit that may have been established.

The reservation status of all six CTBUS transmitters is maintained by the PRC's Reservation Status Unit. This unit performs the bookkeeping duties on behalf of the CTBUS transmitters for the *RESV*, *FREE*, *CHECK*, and *HOLD* CTBUS commands and issues acknowledgments accordingly. The *HOLD* command guarantees the issuer the next possible reservation on the target device[5]. After issuing this command the master monitors

[5]It is assumed, although not strictly enforced, that only the transmitter fetch units on behalf of their IMU will use the *HOLD* and *CHECK* commands

the CTBUS transmitter's reservation status until the current reservation is surrendered. At this point a *CHECK* command is used to release the *HOLD* command and capture the reservation.

With the *RESV* and *FREE* primitives alone we can easily support switching strategies such as virtual cut-through, circuit switching, packet switching, and wormhole routing. An example of how these commands are used to support virtual cut-through is presented in Section 3.3.1. When operating in a circuit switching mode, *FREE* commands are only issued when a circuit is to be dismantled, instead of at the end of every packet as in systems using other routing strategies. For virtual cut-through, packet switching and wormhole routing, a reservation is obtained and then released for every packet.

The *HOLD* and *CHECK* commands were added to allow the IMU to influence the CTBUS reservation policy. These commands can then be used sparingly to guarantee a single high-priority channel by bounding the delay required to reserve a CTBUS transmitter.

Packet Transmission Example on the CTBUS

In returning to our example, we describe the sequence of activities occurring on the CTBUS. We begin at node 2, where the packet transmission begins, and then proceed to node 13, which the packet cuts through. Finally, we look at node 14's CTBUS as the packet is received.

On node 2, TFU_1 (assigned to outbound channel #1) arbitrates for the CTBUS and issues a *RESV* command for the CTBUS transmitter connected to node 13 (CTBUS TX_1). Once TFU_1 succeeds in obtaining a reservation on the transmitter, it will issue a *DTX* bus cycle and wait for the **IRDATA** line to return to true. This process will repeat until TFU_1 reaches the last word of the first page, at which point it will inject a *MARK* command to delimit the page boundary[6]. Page two is transmitted in a similar manner. At the end of page three, the end of the packet is signaled with an *EOP* command[7]. Finally, TFU_1 surrenders its reservation of the CTBUS TX_1 with a *FREE* command.

On node 13, the CTBUS receiver (CTBUS RX_4) is initially configured to transfer any received data to the PRC RX (PRC RX_4). Thus, when the first byte of data is received, CTBUS RX_4 simply transfers it to PRC RX_4 by arbitrating for the CTBUS and using a *DTX* transfer. After transferring the first byte to PRC RX_4, CTBUS RX_4 acknowledges this byte using the reverse channel of CTBUS TX_1. Meanwhile, PRC RX_4 stores this first byte internally and signals its willingness to accept the next byte of data by raising its **IRDATA** line. During this time the CTBUS RX_4 has received the second byte of data and is waiting for the **IRDATA** line to go true. This process continues until the PRC RX_4 has enough of the header to reach a routing decision and reserve CTBUS TX_0, which is connected to node 14, with a *RESV* command. PRC RX_4 then forwards the portion of the header it used to reach its decision out CTBUS TX_0. At this point, PRC RX_4 reconfigures CTBUS RX_4 to automatically forward the received data to CTBUS TX_0. When the *FREE* command arrives from node 2, CTBUS RX_4 forwards the command to node 14 via CTBUS TX_0 and reconfigures itself to its initial mode.

The activities on node 14 closely resemble those on node 13. Since the packet is destined for node 14, PRC RX_3 does not reserve an outbound channel and instead leaves CTBUS RX_3 in its default configuration that forwards all data to PRC RX_3.

3.3.2 Routing Algorithm Support

[6] Prior to transmitting the last word.

[7] The *EOP* supersedes the *MARK* that would regularly be transmitted.

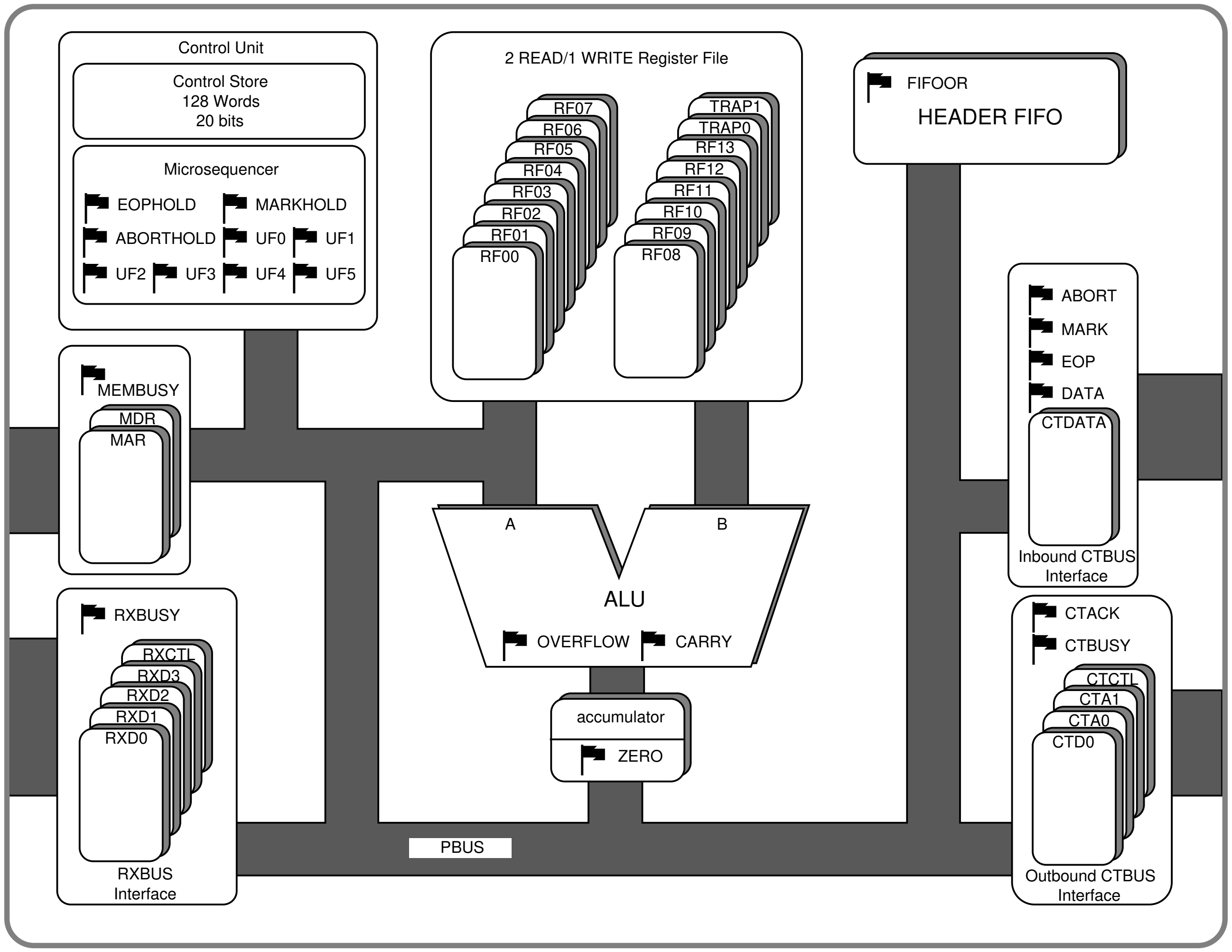


Figure 3.9: The architecture of a PRC Receiver

The flexibility required for efficient support of a wide variety of routing algorithms was achieved by having each of the PRC's six RXs designed around a microprogrammed controller containing a 128-word control store. During system initialization, the IMU downloads microcode to each PRC RX through the PCBUS. After the microcode downloads, each PRC RX executes independently of both the IMU and other PRC RXs. The internal architecture of an individual PRC RX is shown in Figure 3.9.

Local storage in the PRC RX is provided in the form of a small FIFO, six user-defined flags, and a 16-byte register file. The main datapath includes a multi-function ALU, allowing arithmetic and logical operations to be performed on incoming data. The FIFO allows the PRC RX to store an incoming packet header after possible modification until a routing determination has been made. Data may exit the PRC RX through either the CTBUS unit for retransmission to an adjacent node, the RXBUS unit for storage in the buffer memory, or both.

The operation of the PRC RX is controlled by the microsequencer, which decodes the current instruction (retrieved from the control store) and then configures the remainder of the datapath to execute it. The microsequencer operates as a standard two-stage pipeline and thus needs to stall to refill the instruction decoding unit in the case of jump or subroutine calls.

The instruction set contains instructions for general register/constant transfers, flag and ALU operations, and control-flow instructions. All of these instructions have been tailored to support communications, with the goal of reducing both execution time and microprogram size. For example, the instructions that specify a destination register in either the outbound CTBUS interface or the RX bus interface automatically stall the microsequencer if the interface is busy with the last operation. This type of instruction greatly reduces both the size and execution time of the microcode by eliminating the need to check the status of the interface prior to attempting loads; the drawback is that routing algorithms must be written in a manner that won't deadlock if the microsequencer stalls on a busy interface.

The WAIT instruction is one of the more extreme examples of an instruction that has been customized to support microcoded communication. In its simplest form the WAIT instruction stalls the microsequencer until the selected condition becomes true. In its most common use, the WAIT instruction stalls on one condition but also enables trap handlers that allow the instruction execution to jump to addresses specified by the TRAP0 and TRAP1 registers on alternate conditions. This provides a three-way branch that greatly simplifies the implementation of several supported routing strategies. For example, one usually waits for data to arrive but needs to branch to an alternate routine if either an error occurs or an *EOP* command is received.

In addition to the local registers in each PRC RX, the PRC RX has read/write access to a 256 byte inter-device communication RAM located within the PRC. This RAM is accessible to all PRC RXs and is mapped into the PCBUS memory map. This RAM provides a way for the PRC RXs or the IMU to convey information to each other and therefore influence the routing algorithms dependent on either local state information or packet header contents.

For more information concerning the instruction encoding, timing, or syntax required to use the accompanying micro-assembler see Appendix B.

Packet Routing Example using the PRC RX

We return one last time to our example and examine the microcode that could be executed by the PRC RX on node 13 to assist in establishing a cut-through from node 2 to node 14. In this code example we have assumed that the network is only routing virtual cut-through messages which allows us to ignore the byte that identifies the packet type. We have also assumed that the transmitter required to establish the cut-through is unreserved when describing the control flow.

Figure 3.10 shows the prolog code that most routing algorithms need. These are constant expressions usually used as the immediate operand of a load constant instruction when configuring one of the external interfaces in the PRC RX. These expressions do not use any space in the control store and are only for the benefit of the routing algorithm designer. The details necessary to derive these constants can be found in Appendices A and B.

The code depicted in Figure 3.11 performs two functions. Lines 41–50 initialize the default flag values and prepare for the collection of the routing header. Lines 58–80 actually collect the first four bytes of the packet and store them in the local registers RF00 through RF03. While this is taking place notice that each transfer from the CTDIN register triggers the IR flag (Lines 60, 64, and 71). This is coupled to the byte-level flow control and allows the PRC RX to control the CTBUS RX behavior. Also, notice that one of the offset values is being updated (lines 72-74) on behalf of the transmitter sending the packet. Although not being used in this code example, if the header FIFO is used to store the header is often difficult to modify the header to reflect the direction the packet will be transmitted. Since we are operating in a point-to-point network the receivers of the packet know what direction the packet was received from and can update the header appropriately. The important point concerning Figure 3.11 is that on the reception of the last byte of the header the PRC RX has not raised its IR line. This will allow the PRC RX to reconfigure the CTBUS RX.

```
microprogram vcexample;
begin

# Identify this program as belonging to receiver 1
receiver 1;

#  Constant Declarations for RXBUS Interface
const rxcmd_sop_w_crc    0x36;  # Signal SOP and accumulate CRC
const rxcmd_sop_wo_crc   0x26;  # Signal SOP and don't accumulate CRC
const rxcmd_data_w_crc   0x34;  # Signal DATA and accumulate CRC
const rxcmd_data_wo_crc  0x24;  # Signal DATA and don't accumulate CRC
const rxcmd_mark_wo_crc  0x25;  # Signal MARK and accumulate CRC
const rxcmd_mark_w_crc   0x35;  # Signal MARK and don't accumulate CRC
const rxcmd_final_crc    0x14;  # Accumulate CRC with no shift (prep for EOP)
const rxcmd_timestamp    0x3b;  # Signal EOP and timestamp packet


# Constants for controlling outbound CTBUS interface (loaded into CTCTL)
const ctcmd_noop  0xf; # CTBUS NOOP
const ctcmd_dtx   0x0; # CTBUS DTX   (Data transfer)
const ctcmd_abort 0x5; # CTBUS ABORT (Abort packet and circuit)
const ctcmd_resv  0x7; # CTBUS RESV  (Attempt CTTX reservation)
const ctcmd_free  0x3; # CTBUS FREE  (Free CTTX)
const ctcmd_cfg0  0x8; # CTBUS CFG0  (CTRX configuration command)
const ctcmd_mark  0x1; # CTBUS MARK  (Data delimiter)
const ctcmd_eop   0x2; # CTBUS EOP   (End of Packet)

# Address for common devices
const ctaddr_ctrx0 0x41;
const ctaddr_ctrx1 0x42;
const ctaddr_ctrx2 0x44;
const ctaddr_ctrx3 0x48;
const ctaddr_ctrx4 0x50;
const ctaddr_ctrx5 0x60;
const ctaddr_cttx0 0x81;
const ctaddr_cttx1 0x82;
const ctaddr_cttx2 0x84;
const ctaddr_cttx3 0x88;
const ctaddr_cttx4 0x90;
const ctaddr_cttx5 0xa0;

```

Figure 3.10: Code example constant declarations

```
init:
    # Clear out user flags, set IR high, clear ABORT, HOLD, EOP
    clear all;

    # Load trap handler addresses
    ldc eop_handler, trap0;
    ldc abort_handler, trap1;

    # Prepare rxbus interface for first part of data
    ldc  rxcmd_sop_wo_crc,rxctl;

    # For this example we will assume VC offset routing with the following
    # format
    #    {type,x-offset,y-offset,z-offset}
    #
    # Since we are receiver 1 we need to update (increment) the y offset.

    # Get message type and store in RF00
    wait data,trap1(abort);
    xfer ctdin,rf00,go ir;  # IR set true to allow more data

    # Get x-offset, save zero/non-zero status in UF0, and store in RF01
    wait data,trap1(abort);
    xfer ctdin,rf01,go ir;  # IR set true to allow more data
    alu rf01;
    flag zero,f0;

    # Get y-offset, update offset, save zero/non-zero status in UF1,
    # and store in RF02
    wait data,trap1(abort);
    xfer ctdin,rf02,go ir;  # IR set true to allow more data
    alu rf02+1;
    flag zero,f1;
    xfer acc,rf02;

    # Get z-offset, save zero/non-zero status in UF2, and store in RF03
    wait data,trap1(abort);
    xfer ctdin,rf03;        # NOTE/WARNING: IR still pending
    alu rf03;
    flag zero,f2;

```

Figure 3.11: Code example – Header collection

```
    # Enough of the header has been collected to make a route determination
check_y:
    jump f1,check_x;            #  Jump around the -y transmitter check
    ldc  ctaddr_cttx4,   rf04;  #  Save address for later reconfiguration
    jump rsvstat4,check_x;      #  Check the reservation stat of TX3
    jump true,grab_transmitter; #  Try to actually reserve TX3

check_x:
    jump f0,check_z;            #  Jump around the -x transmitter check
    ldc  ctaddr_cttx3,   rf04;  #  Save address for later reconfiguration
    jump rsvstat3,check_z;      #  Check the reservation stat of TX2
    jump true,grab_transmitter; #  Try to actually reserve TX2

check_z:
    jump f2,buffer_packet;      #  Jump around the -z transmitter check
    ldc  ctaddr_cttx5,   rf04;  #  Save address for later reconfiguration
    jump rsvstat5,buffer_packet;#  Check the reservation stat of TX3
    jump true,grab_transmitter; #  Try to actually reserve TX3


    # We either lost the reservation or the packet really goes here
buffer_packet:

    # Load up collected header into rx interface and trigger it
    # Here we have assumed that only the first long word is non-crc data
    #
    # Also turn our input ready back on, allowing CTRX to dump data into us
    xfer rf00,rxd3, go ir;
    xfer rf01,rxd2;
    xfer rf02,rxd1;
    xfer rf03,rxd0, go rxbus;

```

Figure 3.12: Code example – Route determination

3.3.3 Error Detection

Any communication subsystem must deal with errors in transmitted and received packets. As error correction usually incurs too much overhead in transmitted check bits, one of the accepted means of error control and recovery is to detect errors with a cyclic redundancy code (CRC) and request retransmission of erroneous packets. Calculating CRCs in software, however, is a time-consuming and resource-intensive task. The processor must access every part of a packet and compute the remainder, which takes several operations for every word covered. By implementing these calculations in hardware, however, they can

```
    #
    # Main Data loop: Collect 4 bytes (Maybe 5 bytes if tagged with EOP or MARK)
    # The key is that the EOP or MARK condition is held until we want to look at
    # it after we have received the data bytes.
data_loop:
    ldc  rxcmd_data_w_crc,rxctl;        # Assume default non-MARK data with CRC
    wait data, trap1(abort);            # Wait on byte 3
    xfer ctdin, rxd3, go ir;            # Xfer byte 3 and set IR high for next byte
    wait data, trap1(abort);            # Wait on byte 2
    xfer ctdin, rxd2, go ir;            # Xfer byte 2 and set IR high for next byte
    wait data, trap1(abort);            # Wait on byte 1
    xfer ctdin, rxd1, go ir;            # Xfer byte 1 and set IR high for next byte.
    wait data, trap0(eop), trap1(abort); # Wait on byte 0 or trap on EOP
    # Handle possible mark by reloading control reg.
    jump ~mark, d0;
    ldc  rxcmd_mark_w_crc,rxctl;

d0:
    xfer ctdin, rxd0, go rxbus, go ir;   # Fire off RXBUS interface and start over
    jump true, data_loop;

eop_handler:
    wait data, trap1(abort);            # Make sure that data has arrived.
    ldc  rxcmd_final_crc, rxctl;        # Prepare for final CRC
    xfer ctdin, rxd0, go rxbus;
    ldc  rxcmd_timestamp, rxctl, go rxbus; # Timestamp packet and signal EOP
    jump true, init;


    # It isn't totally clear what actions that abort handler should do to inform
    # the upper layers of the communication sub-system.
abort_handler:
    jump true, init;

```

Figure 3.13: Code example – Main data loop

be made relatively transparent to the system. Consequently, this is the approach we have taken in the PRC. The rest of this section describes the implementation and use of the CRC generation and CRC check modules in the PRC.

The conventional implementation of a CRC generator/checker is a linear feedback shift register. The obvious advantage of this design is its simplicity — for an n-degree generator polynomial, all that is needed to implement CRC error detection is n flip-flops and several exclusive-or gates. This implementation, however, depends upon being able to access the data serially. Thus, its use is generally limited to error detection over serial interconnects,

```
    # The address of CTTX to try to reserve is passed in RF04
grab_transmitter:
    xfer rf04,        ctaddr0;
    ldc  ctcmd_resv,  ctctl, go ctbus;
    # This jump will spin until cycle is complete
    jump ~ack,buffer_packet;

    # We got the transmitter we wanted and now forward the packet header.
    ldc  ctcmd_dtx,      ctctl;
    xfer rf00,           ctdout, go ctbus;
    xfer rf01,           ctdout, go ctbus;
    xfer rf02,           ctdout, go ctbus;
    xfer rf03,           ctdout, go ctbus;

    # Reconfigure the CTRX to use the reserved transmitter.
    xfer rf04,           ctdout;
    ldc  ctcmd_cfg0,     ctctl;
    ldc  ctaddr_ctrx1,   ctaddr0;
    ldc  0xff,           ctaddr1,go ctbus;
    # The clear all at the beginning will reset our IR to high
    jump true,init;

end
```

Figure 3.14: Code example – Transmitter reservation

although designers have been known to convert parallel data to serial form to run it through a serial CRC register, and then to reconvert the data to parallel [35].

Another drawback to serial CRC generators and checkers is the time required to process a single n-bit word of data. As noted in [3], it takes n clock pulses for a serial CRC implementation to process a single word of data. By contrast, an n-bit parallel CRC generator requires only a single cycle. There is thus a break-even point of $nt_s = t_p$ (where t_s is the clock period for the serial implementation and t_p is the period of the parallel). For this reason, and since data is not always available in serial form, parallel CRC generators have been studied by a number of researchers. Implementations have ranged from variations on the software approach using a lookup table stored in ROM [31] to fully parallel encoders [3, 37].

For the PRC, packet data is only available in parallel form and at data rates such

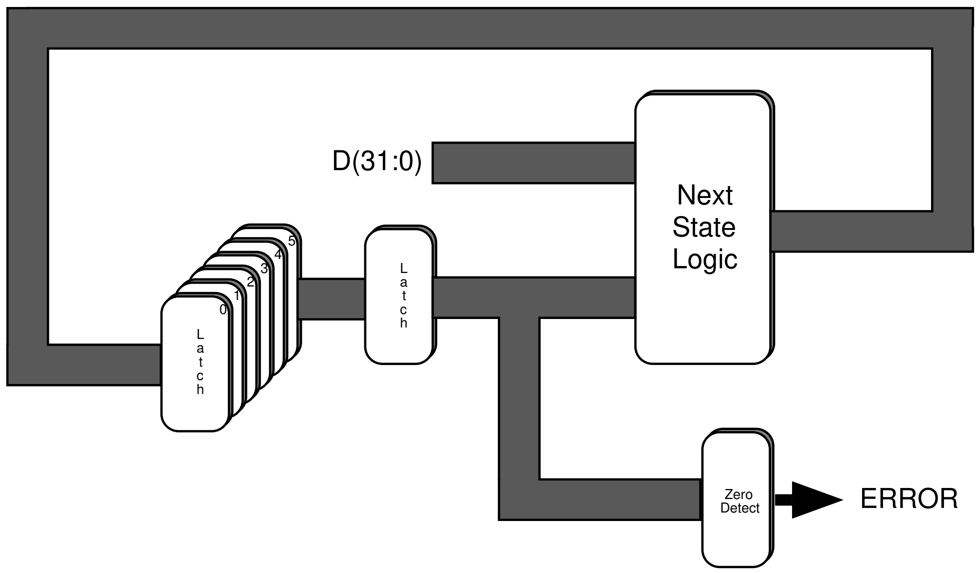


Figure 3.15: Parallel CRC Check Unit

that serial computation is not feasible. This leads to the selection of a parallel CRC with the obvious drawback of the number of gates required for an implementation. To remedy these problems we have proposed and implemented bit-interleaved CRC generator and CRC check units based on the CRC-CCITT polynomial. Theses units snoop the internal TXBUS/RXBUS for data that is transferred from/to the buffer memory and maintain separate 32 bit-interleaved CRCs for each of the outbound/inbound channels. (See Figures 3.1, 3.15 and 3.16.) The CRC check bits are generated in parallel, with even and odd bits being covered by independent polynomials. This approach results in a 2.4 times reduction in complexity and reduced the number of logic levels by 2 for both the CRC generator and CRC check units as compared to a parallel CRC-32.

The equations used in the generator and check units may be found in Appendix D. Information concerning the error coverage for the bit interleaved approach can be found in [13, 14].

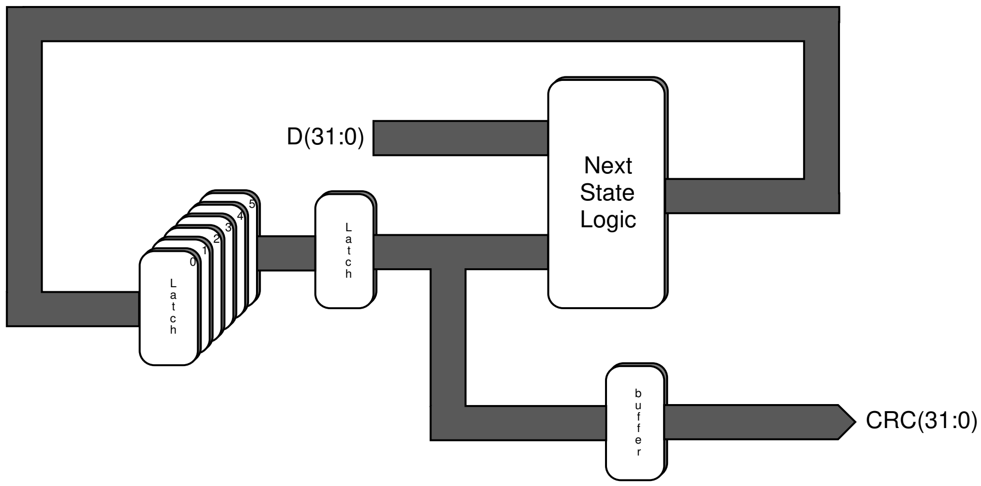


Figure 3.16: Parallel CRC Generator Unit

3.3.4 Operating System Support

The PRC interacts with, and supports, the operating system running on the IMU in several ways. These include inbound and outbound page sequencing, interrupt management, precise time-stamping of packets, and the error detection as described previously. All of this interaction, excepting data writes and reads to the buffer memory, takes place using the PCBUS, which is mapped into the IMU's physical address space.

One of the key functions supported by the PRC is automatic page sequencing on both the outbound and inbound channels. As alluded to earlier, the IMU controls the page sequencing by managing seven independent streams of Page Control Tags (PCTs). For the inbound packet streams, the IMU loads PCTs that describe the base location of each page. For each page associated with an outbound message, the IMU loads a PCT into the PRC. Each PCT contains information about the base location of the page and the amount of data on the page. In addition, PCTs associated with outbound streams contain a field specifying how many longwords at the beginning of each page should be excluded from the CRC calculation. This allows the downstream PRC RXs to modify the lowest-level routing headers without affecting the packet's CRC. Finally, outbound PCTs have a flag that can indicate that the page is the last of the packet.

The PCTs for each outbound stream are stored in FIFOs that are part of the specific TFU that manages transmission in the direction in which the streams are associated. The inbound streams have a single FIFO such that each inbound channel takes the next available PCT. In the current design the outbound FIFOs are 4 PCTs deep and the inbound FIFO is 16 PCTs deep. The intention is to allow the different components of the PRC to be serviced by the IMU in bursts. For example, in a single PCBUS read the IMU may determine the capacity of all of the outbound PCT FIFOs and subsequently service all of them in a single invocation of the PRC interrupt service routine. Since the PRC is designed to support a tightly-coupled flow-control scheme, the unavailability of a PCT for a particular channel doesn't cause the loss of a packet but rather idles transmission in the case of outbound channels or eventually suspends acknowledgements for inbound channels.

With the possibility of having twelve active channels, the potential volume of interrupts requiring service can be quite large. The design of the PRC specifically addresses this potential problem. This support takes two forms: the event queue and the interrupt masks that determine which events generate interrupts.

Information is conveyed from the PRC to the IMU through the generation of events on the PRC and eventual retrieval of the events by the IMU. An event may be any of the following: a page fault, an end-of-packet signal (EOP), or a start-of-packet signal (SOP). A page fault occurs when a channel reaches the end of a page. For an outbound channel this is a function of the length field of the active PCT. For an inbound channel the page faults can be attributed to two sources: the NPBUS interface because the current page has been exhausted or the PRC RX managing that particular channel (i.e. The PRC RX has conveyed the mark command to the IMU). In either case the page fault for the inbound channels indicates the channel and the length of data in the current page. The EOP and SOP events are associated with a particular channel and indicate the end or start of a packet, respectively.

Events are logged by the PRC in an internal FIFO called the *event queue*. Due to the first-in first-out nature of the queue, the events can be read out by the IMU via the PCBUS in the order in which they occurred. The only events which are not recorded in the event queue are page faults on outbound channels. The length information on inbound pages is conveyed back to the IMU through page fault events. These events contain enough information (i.e., the actual number of longwords used) to construct PCTs that can be used in subsequent outbound transmission. An interrupt of the IMU is triggered by any SOP or EOP event, or by a channel needing more page addresses. The PRC incorporates an interrupt masking scheme that allows the IMU to individually mask interrupts due to TX SOPs, TX EOPs, RX SOPs, and RX EOPs. Interrupts due to page requirements, however, cannot be masked inside the PRC.

Another feature specifically incorporated into the implementation of the PRC to support OS functions is send time-stamps and receive time-stamps. The ability to accurately time-stamp messages close to the network hardware and how this aids in synchronization of the clocks in a distributed system is discussed in [38] and motivated its inclusion in this work.

3.3.5 Design Decisions Revisited

At many points during the implementation of the PRC we were faced with design decisions that shaped the final capabilities of the PRC. This subsection sheds some light on a few of the key decisions and discusses how we arrived at our solutions.

The interconnect structure of the CTBUS was one of the first decisions that needed to be evaulated. The choice was limited to either a cross bar with the TFUs, PRC RXs, and CTBUS RXs requesting service, or the demand slot TDM scheme described earlier. The former was eventually rejected due to the excesive number of I/O pins that would be required for a parallel implementation. The demand slot arbitration method was chosen over other TDM arbitration techniques since it fairly distributes the bus bandwidth in times of high utilization but allows idle cycles to be used in the event of unbalanced traffic.

The desire to minimize unnecessary traffic on the CTBUS motivated several of the design decisions revolving around interaction with the CTBUS. The first question was how to maintain the reservation status. The inclusion of the reservation status unit in the PRC rather than the Network Interface Controller or the individual CTBUS transmitters maintaining their own status allowed both the TFUs and RXs to concurrently check the status of the CTBUS transmitters they are attempting to reserve. This allows the units to issue the *RESV* command only after they know the unit has been previously freed. Several other alternatives were considered, including requiring the units to continously issue *RESV* commands until the device positively acknowledged the reservation attempt or to snoop the CTBUS for *RESV/FREE* commands. The former was rejected due to the overhead placed on the CTBUS. The latter was rejected since its hardware complexity is equivalent to the current implemented unit.

Similar to the handling of reservation status, the availability of individual **IRDATA** lines was chosen over a scheme in which the device would positively or negatively acknowledge each data transaction on the CTBUS.

The implementation of the page sequencing is another area where several different methods were evaluated. The current implementation requires that the IMU provides streams of PCTs. One would think that an obvious improvement would have the PRC actually fetch the next PCT automatically. This was not done for two reasons. First, this assumes that the PRC has detailed knowledge of the data structures used by the IMU. Second and most important, automatic page following would prevent partial cut-throughs using the buffer memory. With the current implementation, a packet can be resubmitted for transmission as soon as a single page has been received. If the source nodes are not able to provide data fast enough, then the TFU will exhaust its page queue and simply wait for the next PCT. The alternative approaches that follow the page structures automatically would have to use some form of page locking to indicate that the next page is available. This was deemed to be too expensive to implement for the amount of traffic that would take advantage of this capability.

The decision to support the stalling I/O operations in the RX was made after observing the excessive number of WAIT instructions required to stay in synchronization with the CTBUS and RXBUS interfaces. The cost of adding these features was minimal since the microsequencer must already handle the WAIT, JUMP, and RETURN instructions.

3.4 Comparison to Related Commercial and Research Devices

In this section we compare the features of the PRC to those available in current research and commercial devices. It is important to note that none of the systems discussed were designed for the particular region of design space that the PRC is targeted, therefore making direct comparisons difficult. This does not mean that we cannot learn from previous mistakes or successes. Furthermore, it does not mean that future systems for these alternative domains can't integrate some of the features that we have chosen to incorporate.

The differences between the related systems and the PRC can be divided into several dimensions. These include the decision to use centralized hardware components, the grain size of the computation tasks, and the level of support services provided. In terms of the spectrum of possible designs, the communication subsystems that are intended to provide support for a network of workstations tend to provide high-level services through a single port to the interconnect fabric. On the other end of the spectrum are the communication subsystems targeted for today's tightly-coupled multiprocessors. Here the packets are short, the number of connections to the interconnect fabric larger, the interconnect assumed to be reliable, and the host processor required to interact quite closely with the hardware. The PRC is placed somewhere in between these two extremes; targeted for performing and supporting experiments on fault-tolerant distributed real-time systems, the PRC needs to support intermediate performance communication with multiple ports into an interconnect fabric.

In examining the systems described below we will focus on the following characteristics when applicable:

- The operating domain originally envisioned.
- The interconnection topology or fabric.
- The number of network ports.
- The routing algorithm(s) supported.
- The switching methodology.
- The packet/message sizes supported.
- The assumed media characteristics.
- The physical interconnection cost issues.
- The type of operating system support.

Post Office (Mayfly/FAIM-1) The Post Office (PO) of the Mayfly system [5, 17, 18] (developed by Hewlett-Packard Laboratories) is a system whose architecture and functionality most closely matches that of the PRC. The PO is a independent packet delivery subsystem that has evolved from its original inception as the communication support for Schlumberger's FAIM-1 symbolic multiprocessor [19, 49, 50] to its current role. The end target environment for the system has changed from supporting an ultra-concurrent symbolic multiprocessor for AI systems to the existing design goals of being a scalable general-purpose parallel processing environment for modern programming languages. Surprisingly enough, the PO architecture and supported functions have remained relatively constant even though most of its surrounding components have radically changed.

The interconnection topology provided by the PO is a two-level processing surface based on a regular hexagonal surface with a wrap structure using a twisted torus. This wrap structure is isomorphic to the C-wrapped structure being used for HARTS as described earlier. This two-level structure is achieved by placing special three way switches on the periphery of the lower level surface and connecting two adjacent surfaces. For example, consider the original H_4 surface presented in Figure 2.1. Instead of directly connecting the gray wrap links as originally described, insert a three-way switch so that a packet may either take the wrap link or exit the surface. The exit links are then connected to an adjacent $H-4$ surface. This hierarchical scheme doesn't provide real benefits until the total number of nodes and surfaces are reasonably large. A H_7 surface containing 127 nodes has a diameter of six compared against a two-level surface constructed from a series of H_3 surfaces (a S-2 E-3 surface using Davis's notation), which has a diameter of five. The benefits are clearly presented when comparing the difference in the diameters of a E-140 to a series of E-10 surfaces (139 vs 89). This reduction is gained by having the routing algorithms deal with two sets of offsets; one for the base surface and one for the surface constructed by tessellating the base surface.

The PO claims to support the dynamic selection of four types of routing algorithms: virtual cut-through, best path, no farther, and random. In reality, the PO doesn't really select one of these algorithms but sequences through these four sub-algorithms as a single routing/switching algorithm. Only the router modules associated with input ports perform the virtual cut-through algorithm, which implements the same policies as the best path algorithm but attempts to directly connect the input port to the output port prior to the entire packet arriving at the input port. If an inbound packet can not be handled successfully by the virtual cut-through phase, then best path algorithm is attempted. This policy attempts to transmit the packet out a port that will result in the most routing options for the router in the next node. If that isn't immediately possible, best path attempts to select a route that still preserves minimal path routing. Once a certain time threshold for a particular packet is exceeded, the router modules switch from attempting best path routing to attempting no farther routing by routing the message to a node which is the same number of hops distant. Finally, after another time threshold is exceeded, the routers use the random approach in which the packet goes out in any direction.

The PO uses a fixed packet size of 36 words and can carry a payload of 32 words. The 4 word header specifies the surface, the node within the surface, the original source, the message id, the packet id, and the total number of packets associated with the message. The PO deals solely with packets, but the message id is used by its managing processor (MP) and allows the received packets to be directly transferred to the appropriate locations in the MP's memory.

Each PO has six 12 bit bidirectional buses used for communicating with the POs in adjacent nodes and a 32 bit interface onto the MP's bus. Arbitration for the bus between two POs is handled by one of the POs and is granted for the period of a packet with the option of the receiving PO NACKing the packet at any time during the transfer. Retransmission for packets that are corrupted or refused due to lack of buffer space are handled directly by the port controllers associated with each link.

The PO interface with the operating system and its managing processor (the MP) is accomplished in several ways. First the PO can act both as a co-processor to the MP and/or an instruction assist device. This allows efficient use of the MP's instruction set (a HP Sterling 1.5 which is a 16 Mhz 32 bit implementation of Hewlett-Packard's RISC architecture) for loading the transmit and receive packet FIFO. The PO is designed to operate in either polled or interrupt driven mode, thus allowing the OS to use the mode that is appropriate for the current network load (or set of device drivers available).

There are many differences between some of the fundamental decisions that the designers of the PO made and those of the PRC. These include the fixed packet size, placement of buffer space, the type of service expected from the MP, and resource allocation policies.

The fixed packet size of the PO is acceptable for the MIMD multi-computer which the PO is designed to support. This fixed packet size is also necessary for the PO to make guarantees that it will not deadlock on its internal packet buffers. Since the PRC is intended to support a wide variety of communication protocols, many of which have a much larger packet size, the PRC can not place such a restriction on the packet size and therefore externally buffers transit packets. The PRC assumes that the external packet buffers are large enough that higher level protocols can prevent deadlock due to buffer starvation.

The type of interaction and control required differs considerably between the PRC and the PO. The PO operates as an autonomous packet delivery subsystem with only the initiating MP and receiving MP required to interact with the PO. This also means that the MP has no way of influencing the routes or resources being used. In contrast, the PRC requires the IMU to schedule the packets for transmission on each individual TFU (both for injected packets and transit packets that could not cut-through). This allows the IMU and PRC to adapt to changing network conditions and mission requirements. For example, the IMU has at its disposal the packet scheduling policies for each of the TFUs, the reservation release policies of the TFUs, and the inter-device communication RAM. This type of control provides the necessary support for both fault-tolerant and real-time communication.

There are several lessons that we can learn from the P.O design. First, the flexibility gained from having multiple routing algorithms (or phases of an algorithm) can be quite important. This can be seen from the evolution of the routing support originally proposed for the FAIM/1 system and what was finally implemented in the PO for the Mayfly system. Second, the OS interaction with the PO can be dealt with in many ways. The PO acts as an autonomous sub-system but requires the MP to calculate the checksum on the packets submitted. In contrast, the PRC requires the IMU to initiate packet transmission in the TFUs but does not require the IMU to scan the entire packet in order to accumulate a CRC. Last, it is easy to design a system that will give a large variance for the packet delivery times. There are several examples of this in the PO. First, the PO does not implement a FIFO policy in selecting which packet will be selected for transmission out of the packet memory. Second, the PO has seven routers attempting to gain access to an outbound port in order to transmit a packet. Six of these are assigned to servicing ports that are receiving packets and thus will attempt a cut-through. This appears to lead to the situation in which a node can have its packet delivery latency significantly influenced by cut-through traffic.

Torus Routing Chip The torus routing chip (TRC) is designed as a building block for high-throughput, low latency message-passing concurrent computers based on a byte-wide parallel k-ary n-cube interconnection network [9, 11]. The TRC implements a provable deadlock-free interconnection network by routing variable length messages using wormhole switching on two virtual channels per physical link. The TRC is implemented using a 5x5 10 bit crossbar with five 5-way arbiter built into the crossbar switch, 5 input controllers that make routing decisions, and 5 output controllers that handle the flow control and physical multiplexing onto the output links.

Although the TRC is now quite dated, some of its contributing features can be found in many of the second generation parallel machines. Most notable is the efficient use of wormhole routing that appeared in the Mesh Routing Chip (MRC) used in the Amtex 2010 [42] and the theory behind constructing deadlock-free routing by defining an order in which the packets use the available virtual channels[10].

When evaluating the TRC in the intended operating domain of the PRC the fixed routing algorithm, lack of end-to-end error detection, and lack of operating system support could be considered major drawbacks. The beauty of the TRC is that it is small and simple.

Message Driven Processor The Message-Driven Processor (MDP), like its predecessors (TRC/NDF/ADC), is designed to be part of a multicomputer capable of supporting fine grain, message passing, parallel computation[12, 11, 36, 45]. What is novel about their approach is that the processor that comprises the core of each node has been made network aware. This has been accomplished by integrating a 36 bit (32 bits data/4 bit tag) integer processor, a memory controller, a 3-D mesh router, 4K SRAM, and a 1 MB DRAM controller all into a single VLSI device. This provide both the end user and the operating system with low overhead mechanisms for communication, synchronization, and translation.

As with many of the parallel machines, the physical proximity of the nodes allows for synchronous parallel interconnect, and the media characteristics are assumed to be error free. These assumptions immediately allow the MDP to ignore the need for any end-to-end error checking. Furthermore, the designers of the MDP have chosen to deemphasize any notion of the interconnection topology and use a simple 3-D mesh with dimension order routing with wormhole switching.

Messages are submitted to the mesh router using the SEND and SENDE instructions which enqueue either one or two words per instruction. There is no DMA support and the SENDE instruction signals the end of the message under construction and initiates its transmission. For any time lost in constructing messages the MDP certainly compensates with its hardware support for the automatic interrupt and execution of a thread upon arrival of a new message. This may occur on two separate message priority levels which further enhances the MDP flexibility.

Certain key characteristics of the MDP can be observed in portions of the PRC architecture and implementation. The simple control interface in which the IMU submits a single page tag for a single page message serves the same purpose as the SEND/SENDE instructions. The desire to minimize interrupt overhead by allowing the packet to be collected completely before dispatching the receive thread is a dual to the PRC's ability to filter out interrupting events until the end of packet is detected.

The major differences between the MDP and the PRC are mainly an artifact of the intended operating domains. The router of the MDP is designed solely to support low latency communication for the J-machine, and the intention is to hide any notion of the network from the user. The PRC is designed to allow the operating system designer the maximum amount of information and control of the network so that a communication subsystem capable of supporting distributed real-time applications can be constructed.

Nectar CAB We now move from comparisions of the PRC to routers supporting closely coupled parallel computation to more general network adaptors or communication controllers. CMU's Nectar project's design and implementation of the Communication Accelerator Board (CAB) clearly identifies some of the important features that a general purpose communication system should efficiently support [8, 30, 33, 48]. Heterogeneity, scalability, low-latency, and high-bandwidth communication were the stated goals of the Nectar project. To achieve these goals they designed an efficient parallel crossbar interconnection network that can be cascaded in multiple levels to support more nodes; hence satisfying the scalability, low-latency, and high-bandwidth requirements. The designers' answer to the heterogeneity requirement was the CAB, a flexible and highly programmable network processor. Similar to the PRC, the CAB had to balance the desire for flexibility against the need for fast basic operations.

As with the HARTS Network Processor, the CAB architecture was divided into three main function blocks: the host interface, the processing unit, and a network interface. For purposes of comparison to the PRC we will only consider the functions supported by their network interface.

The network interface is composed of two independent sections: a transmitter and a receiver. These two sections operate completely independently of one another and are controlled as separate devices by the managing processor. Both the transmitter and the receiver have a private DMA channel that allows for unattended data transmission/reception once set up by the processor. Physical connection to the network is performed by AMD TAXI transmitters and receivers.

There are significant differences in some of the design decisions undertaken by the CAB designers when compared against the PRC/NI combination. First, the CAB uses only packet level flow control and claims that byte-level flow control would too greatly impact the peak performance. While this is true for the SRAM based CAB design, the fact that the PRC is managing six TAXI device pairs and writing into DRAM invalidates this claim. Also, routing in a crossbar can occur in a single cycle, which is not possible in the context of more complicated routing/switching algorithms for point-to-point networks that the PRC is designed to explore. These requirements force the PRC to support byte-level flow control, but its costs are partially amortized by having up to 12 channels independently active.

The method in which DMA is implemented also differs between the two designs. The PRC preloads the page tags that identify where incoming data will be received and then has the IMU and API interfaces of the NP construct the contiguous message images as they are copied into the address space of the application processors. The CAB requires the processor to respond to the incoming packet and set up the DMA prior to overrunning a FIFO inside the receiver of the network interface. This allows for the possibility of directly transferring the received data into the host memory but places a significant response time limit on the processor that can only be done for a limited number of incoming channels. With the possibility of 6 packets arriving almost simultaneously, the PRC must take a more automated approach to the reception. The same type of differences exist on the transmitter side. Here the PRC loads a stream of page tags into the appropriate TFU's, where as the CAB must set up the DMA for each contiguous block of data.

3.5 Low-Level Simulation and Geometry Results

The PRC design presented in this chapter has been completed and gate level simulations performed. Designed in a 1.0μm CMOS process, the PRC measures 14.005mm by 13.4mm. The PRC requires a total of 180 pins, divided into four main groups. Power, ground, and clock lines consume 15 pins, while the remainder are divided amongst the PCBUS, the CTBUS, and the NPBUS.

During the design of the PRC, we targeted for a CTBUS cycle period of no more than 50ns (20MHz). Timing analysis, however, has shown that this interface will be able to function with a 30ns cycle (33MHz). This exceeds the 22.5 MHz cycle time if the AMD TAXIchips are run in synchrounous mode which currently is our mode of choice for the NI. In the current design of the Network Processor, this speed is sufficient to saturate the memory subsystem.

With an unloaded system, the PRC can begin transmission of a packet within 500ns of the first page control tag being latched into the page queue. During this time, the PRC retrieves the first word of the packet from the buffer memory, obtains a reservation of the CTBUS transmitter, and latches the first byte of the first word into that transmitter.

In addition to the low-level hardware simulations we have developed an intermediate level event-driven simulator presented in Chapter 5 that allows us to model the entire point-to-point network based on the operating characteristics of the PRC. The simulator is able to capture activities with a granularity as small as a bus cycle, where necessary, in order to provide accurate estimates of the network performance.

3.6 Summary

Throughout this chapter, the flexibility of the PRC has been emphasized as a key desirable attribute. Why is this flexibility important? The flexibility of the PRC will be crucial to experiments that vary routing strategies in order to determine which is the "best" for a particular application or interface. The intelligence of the PRC receivers allow complex routing decisions to be made at a level below that of the main processor of the NP. The PRC even provides support for priority based decisions: if a particular channel (for whatever reason) needs to be reserved to high-priority traffic, low-priority traffic can be prevented from cutting through.

One feature of the PRC design which has not been previously emphasized is how the CTBUS protocol insulates the PRC from the physical implementation details of the Network Interface. For example, HARTS uses the AMD TAXI chip set to create the physical links between nodes. These could easily be replaced with CTBUS-compatible parallel interconnect without requiring modifications to either the PRC or the software. This separation also serves to highlight the distinction the PRC provides between the routing algorithm and switching scheme. The CTBUS protocol is used solely for controlling switching.

Several primitives were directly provided to support real-time communications. These include the transparent addition of send and receive time stamps, which are useful for clock synchronization and distributed agreement algorithms. The ability to implement channel priorities allows the IMU to influence decisions made by the low-level routing strategies as a function of high-level goals of the IMU.

The PRC provides the flexibility required for an experimental system combined with the performance of a hardware implementation. Although the PRC is not designed to compete with commercial devices on raw performance, the insights gained from its design and use in an experimental environment can be used in the design of future communication subsystems that operate in its domain.

CHAPTER 4

DELIVERY TIME DISTRIBUTIONS

4.1 Problem Description

This chapter derives an analytical model to evaluate the virtual cut-through message-passing scheme in a distributed computing system based on a hexagonal mesh architecture described in Chapter 2.

Since real-time applications normally require short response times, simple store-and-forward message passing schemes may not always be suitable. Consequently, we look at a message passing scheme *virtual cut-through* as described earlier.

Although virtual cut-through was proposed almost a decade ago, it has not been implemented in real systems until recently. Since custom ASICs have become economically viable, several distributed systems are being designed and implemented that use virtual cut-through (or some variant thereof) as their basic message passing scheme. It is easy to see that virtual cut-through will perform better than a conventional packet-switching scheme in terms of packet delivery times. However, the actual improvement it offers over a packet-switching scheme for packet deliveries has not yet been accurately evaluated.

Kermani and Kleinrock carried out a mean value analysis of the performance of virtual cut-through for a general interconnection network [26]. However, a mean value analysis is not adequate for real-time applications because worst-case communication delays often play an important role in the design of real-time systems. For example, the mean value analysis cannot answer questions like what is the probability of a successful delivery given a delay or what is the delay bound such that the probability of a successful delivery is greater than a specified threshold.

The authors of [26] wanted to avoid any dependence on the interconnection topology in their analysis. As a result, they assumed that the probability of a packet getting buffered at an intermediate node is a given parameter. Since one cannot get a reasonable estimate of the performance of virtual cut-through without an accurate estimate of the probability of buffering, the approach in [26] becomes useful only if we can accurately determine the probability of buffering for a given interconnection topology. However, determining the probability of buffering at an intermediate node for a given topology is not simple. This is because each node in a distributed system handles not only all packets generated at the node but also all packets passing through the node (called *transit packets*). Consequently, to evaluate the probability of buffering, we have to account for the fraction of packets generated at other nodes that pass through each given node.

In contrast to [26], we first derive the probability that a packet is destined for a particular node by characterizing the hexagonal mesh topology. This *probability of branching* is then used as a parameter in a queueing network to determine the *throughput rates* at each node in the mesh. After the throughput rates are found, the probability that a packet can establish a cut-through at an intermediate node is derived. From these parameters we approximated the probability distribution function of delivery times for a packet traversing a specified number of hops. The importance of this kind of analysis in a real-time system, as opposed to a mean value analysis, is then illustrated through some numerical examples and compared with simulation results that are based on some of the relevant parameters in **HARTS**.

The chapter is organized as follows. The terms and notation used are introduced in Section 4.3. Analytical expressions for the branching probability and buffering probability are derived in Section 4.4 and the probability distribution function of packet delivery times is derived in Section 4.5. Numerical results from both the analytic model and simulations are presented and compared in Section 4.6. We conclude with Section 4.7.

4.2 Message Model

This section presents the derivation of the probability distribution of packet delivery times in a C-wrapped H–mesh that implements virtual cut-through. A queueing network will be used to carry out this analysis.

To make the analysis tractable, we make the following assumptions:

A1: Poisson packet generation with rate λ_G at each node.

A2: Exponentially-distributed packet lengths with mean $\overline{\ell}$.

A3: The length of a packet is regenerated at each intermediate node of its route independently of its length at other intermediate nodes.

A4: Nodes have no preferential direction for communication.

Assumptions A1–A3 are consistent with Kermani and Kleinrock's assumptions in [26]. Although not completely accurate, it has been shown through empirical studies that these assumptions lead to a fairly accurate characterization of message arrivals. Assumption A4 implies that all minimal length paths between a source and destination are equally used.

A4 does not imply uniform communication over all nodes of the mesh, but implies uniform communication with nodes reachable in the same number of hops. So, let q_k denote the probability of a node communicating with a node which is k hops away. The definition of q_k will be used to derive some of the base parameters for the queueing network.

Due to the homogeneity of a C-wrapped H-mesh, any node can be considered as the origin of the mesh and labeled 0. Without loss of generality, we can concentrate on evaluating the distribution of the packet delivery times for the packets generated at node 0. In order to determine the distribution of the delivery times it will be necessary to evaluate the transit load handled by node 0. This transit load is a function of both the packet generation rate at each node and the interconnection topology. Another parameter necessary to determine the distribution of the delivery times is the probability that a transit packet (at node 0) will be buffered (at node 0) as a result of not being able to establish a circuit to the neighboring node. The derivation of the analytical expressions for the transit load and the probability of buffering is presented in Section 4.4. The distribution for the packet delivery times is then presented in Section 4.5.

4.3 Terms and Notation

In the following analysis let e be the dimension of the H-mesh and let $[j]_i$ denote j mod i. Also let $N = \{0, 1, \ldots, 3e(e-1)\}$ be the set of all nodes in the H-mesh.

Definition 2 *A* route *from a source node, $s \in N$, to a destination node, $d \in N$, is a sequence $n_0 n_1 \cdots n_i \cdots n_{k-1} n_k$ of nodes, $n_i \in N$, $\forall\, i \in \{0, 1, \ldots, k\}$, such that (i) $n_0 = s$, $n_k = d$, and (ii) there exists a direct link in the H-mesh between n_i and $n_{i+1}, \forall i \in \{0, \ldots, k-1\}$. The* length *of a route r is the number of components in the sequence and will be denoted by $len(r)$.*

Definition 3 *A* minimal route *from $s \in N$ to $d \in N$ is a route r_1 from s to d such that $len(r_1) \leq len(r_2)$ for all routes r_2 from s to d.*

Definition 4 *An* anchored route *is an ordered pair $(n_0 \cdots n_k,\ x)$ consisting of a route $n_0 \cdots n_k$ and $x \in N$ such that*

1. *$k \geq 2$,*

2. *$n_0 \cdots n_k$ is a minimal route from n_0 to n_k, and*

3. *$\exists\ i,\ 1 \leq i \leq k-1$, such that $n_i = x$.*

x is called the anchor *of $(n_0 \cdots n_k,\ x)$.*

Definition 5 *A* shape s *of length k, $2 \leq k \leq e-1$, is a sequence $a_1 a_2 \cdots a_i \cdots a_{k-1} a_k$, $a_i \in \{d_0, \ldots, d_5\}$, such that*

$$\begin{aligned}\bigcup_{i=1}^{k} \{a_i\} &\in \bigcup_{j=0}^{5} \{\{d_j\}\} \ \bigcup \ \bigcup_{j=0}^{5} \{\{d_j, d_{[j+1]_6}\}\} \\ &\equiv \{\ \{d_0\}, \{d_1\}, \{d_2\}, \{d_3\}, \{d_4\}, \{d_5\}, \\ &\qquad \{d_0, d_1\}, \{d_1, d_2\}, \{d_2, d_3\}, \{d_3, d_4\}, \{d_4, d_5\}, \{d_5, d_0\}\ \}.\end{aligned}$$

The length of shape s is denoted by $\ell(s)$.

A shape is a route that a packet can traverse. The above definition of a shape is motivated by the fact that all minimal routes between any pair of nodes are formed by links along one or two directions only [6]. For example, route A in Figure 4.1 corresponds to the shape $d_1 d_1 d_0 d_1$ such that $\cup_{i=1}^{4}\ \{a_i\} = \{d_0, d_1\}$ and route B corresponds to the shape $d_0 d_0 d_0$ such that $\cup_{i=1}^{3}\ \{a_i\} = \{d_0\}$. A shape can represent routes between several different pairs of communicating nodes. For example, routes A and C in Figure 4.1 correspond to the same shape but represent routes between two different pairs of communicating nodes.

Definition 6 *An* anchored shape *p is an ordered pair (s, k), where s is a shape, and $1 \leq k \leq \ell(s) - 1$ marks a position within the shape. The length of an anchored shape (s, k) is defined to be the length of the associated shape s.*

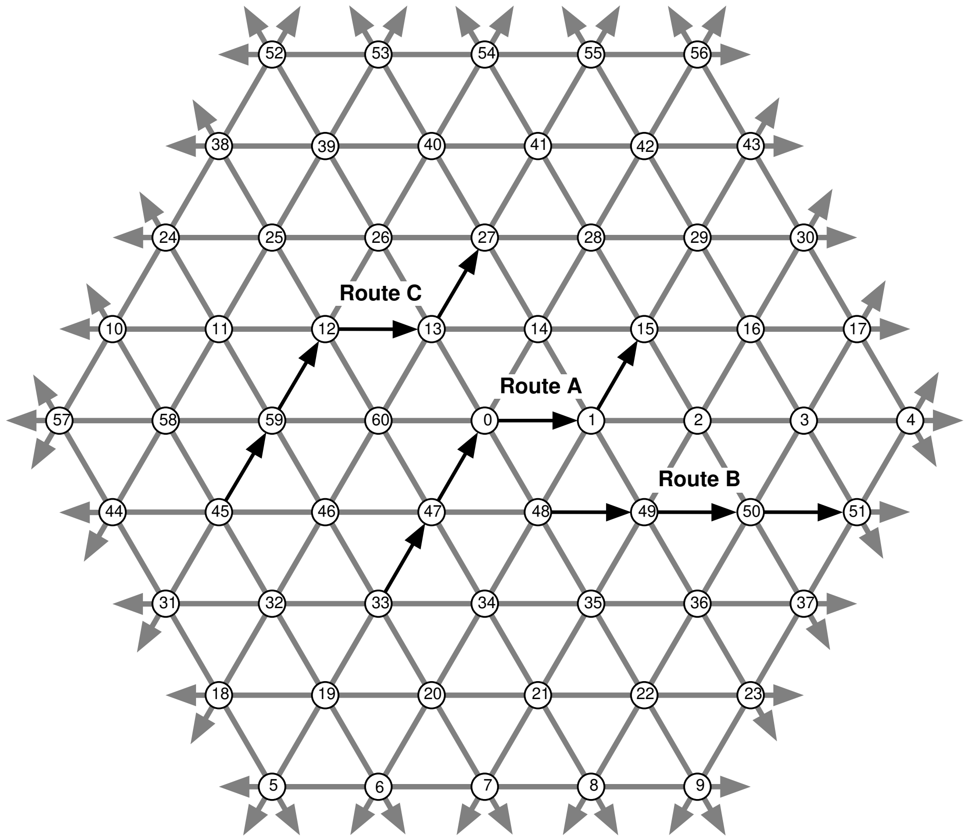


Figure 4.1: Example shapes in a H–mesh of dimension 5.

There exists a one-to-one correspondence between the set of all anchored shapes and the set of all anchored routes with their anchor at 0. (In order to not detract from the main goal of this chapter the proof that there is a one-to-one correspondence between these mappings has been given in Appendix E.) The mapping from an anchored shape $(a_1 \cdots a_\ell \cdots a_k, \ell)$ to an anchored route $(n_0 \cdots n_k, 0)$ is done as follows:

$$n_i = \begin{cases} cwhm(n_{i+1}, \overline{a_{i+1}}) & \text{if } 0 \leq i \leq \ell - 1 \\ 0 & \text{if } i = \ell \\ cwhm(n_{i-1}, a_i) & \text{if } \ell + 1 \leq i \leq k \end{cases} \tag{4.1}$$

where $\overline{a_i} = d_{[m+3]_6}$ if $a_i = d_m$, $0 \leq m \leq 5$. The mapping from an anchored route $(n_0 \cdots n_k, 0)$ to an anchored shape $(a_1 \cdots a_\ell \cdots a_k, \ell)$ is

$$\begin{aligned} a_i &= cwhm^{-1}(n_{i-1}, n_i) \quad 1 \leq i \leq k \\ \ell &= \arg_{1 \leq j \leq k-1}(n_j = 0). \end{aligned} \tag{4.2}$$

where $\arg_{1 \leq j \leq k-1}(n_j = 0)$ refers to the value of j such that $n_j = 0$. For example, consider the anchored route $(33 \rightarrow 47 \rightarrow 0 \rightarrow 1 \rightarrow 15,\ \ 0)$ obtained from route A in Figure 4.1. From Definition 1, we know that $cwhm^{-1}(33, 47) = d_1$, $cwhm^{-1}(47, 0) = d_1$, $cwhm^{-1}(0, 1) = d_0$ and $cwhm^{-1}(1, 15) = d_1$. Since 0 is the third node in the route $\arg_{1 \leq j \leq 4}(n_j = 0) = 2$. It then follows from Equation 4.1 that the anchored shape corresponding to the anchored route $(33 \rightarrow 47 \rightarrow 0 \rightarrow 1 \rightarrow 15,\ \ 0)$ is $(d_1 d_1 d_0 d_1,\ 2)$.

Similarly, the anchored route $(n_0 n_1 n_2 n_3 n_4,\ 0)$ corresponding to the anchored shape $(d_1 d_1 d_0 d_1,\ 2)$ can be obtained as follows. Since the second element in the anchored shape is 2, $n_2 = 0$. Since $\overline{d_1} = d_4$ and $cwhm(0, d_4) = 47$, $n_1 = 47$. Proceeding further, we get $n_0 = 33$ because $cwhm(47, d_4) = 33$, and $n_3 = 1$ because $cwhm(0, d_0) = 1$. Finally, $n_4 = 15$ since $cwhm(1, d_1) = 15$. As expected, combining these results we get the anchored route corresponding to the anchored shape $(d_1 d_1 d_0 d_1, 2)$ as $(33 \rightarrow 47 \rightarrow 0 \rightarrow 1 \rightarrow 15,\ \ 0)$.

The minimal route corresponding to the anchored shape p is one of the possibly many minimal routes between the source and the destination. The other minimal routes between the source and the destination can be obtained by permuting the components of the shape associated with p and applying a mapping function similar to the one above.

All of the routes associated with these permutations will not necessarily go through node 0. Only the fraction of the total number of routes from the source to the destination that pass through node 0 will influence the transit load at node 0.

4.4 Network Model Derivation and Parameter Calculation

The packet transmission in the H-mesh can be modeled as a Jackson queueing network, consisting of $3e(e-1)+1$ service centers of the M/M/1 type. For each service center a packet completing its service may go to either of its six immediate neighbors or exit from the system. Packets whose final destinations are immediate neighbors will not use the service centers of their immediate neighbors and will exit the system at the current service center. Packets whose final destination are not immediate neighbors travel to a neighboring service center.

Let $p_{b_{ij}}$ denote the probability that a packet completing its service at a node i will be routed to neighboring node j. Using assumption A4 and the fact that the C-Wrapped H-Mesh is a homogeneous surface it is easily seen that all the $p_{b_{ij}}$ have to be equivalent and thus will be denoted by p_b. Figure 4.2 shows a portion of the queuing network centered around node i.

The rest of this section concentrates on deriving an expression for p_b. Once given an expression for this, we can derive the probability that a packet will establish a cut-through when arriving at a node in the H-mesh.

Calculation of p_b

The following symbols are used to identify the different packet arrival rates:

- λ_G : the rate of generating packets at a node.
- λ_{G^2+} : the rate of generating packets at a node that are not destined for an immediate neighbor.
- λ_T : the rate of transit packets arriving at a node.

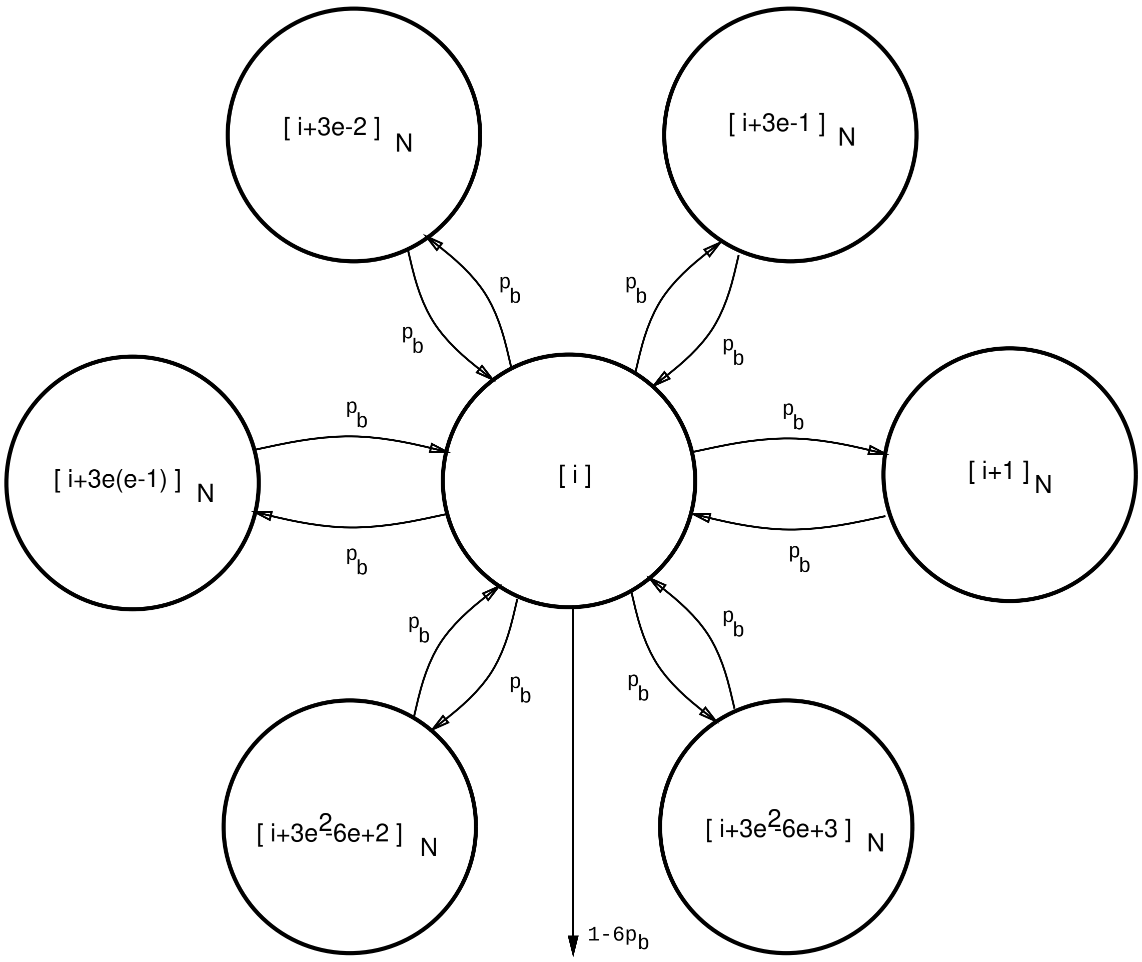


Figure 4.2: Network model around node i.

- λ_{T^2+} : the rate of transit packets arriving at a node that are not destined for an immediate neighbor.

It is convenient to define a function $\Phi(d_j, p)$ that counts the total number of d_j's in the shape associated with the anchored shape p, that is,

$$\Phi(d_j, p) = |\ \{a_k : a_k = d_j,\ a_k \text{ is in the shape associated with anchored shape } p\ \}\ |.$$

Considering the anchored shape A from the previous example, $\Phi(d_0\ ,\ (d_1d_1d_0d_1, 2)) = 1$ and $\Phi(d_1\ ,\ (d_1d_1d_0d_1, 2)) = 3$. This function is used to derive the transit load associated with an anchored shape on node 0.

Lemma 1 *The contribution of an anchored shape p to the transit load of node 0 is*

$$L(p) = \frac{\lambda_G \cdot q_k}{M(p)}$$

where $k = \sum_{j=0}^{5} \Phi(d_j, p)$ *and* $M(p) = \frac{\left[\sum_{j=0}^{5} \Phi(d_j, p)\right]!}{\prod_{j=0}^{5} \left[\Phi(d_j, p)!\right]}$.

Proof: It follows from the definition of anchored shape and a simple combinatorial analysis that the total number of shortest routes between a source–destination pair is $M(p)$.

By the definition of q_k, the rate at which a source sends packets to a destination is $\lambda_G \cdot q_k$. By 4, all routes between the source and the destination are equally used. Hence, $L(p) = \frac{q_k \cdot \lambda_G}{M(p)}$. ■

Lemma 1 allows us to calculate the transit load for a single route through node 0. In order to calculate the total transit loads λ_T and λ_{T^2+}, we will need to determine the total number of minimal routes passing through node 0 for all pairs of communicating nodes. To determine this number we will partition the set of all anchored shapes into sets that can be counted. Since there is a one-to-one correspondence between the anchored shapes and anchored routes with their anchor at node 0, counting all anchored shapes is equivalent to counting all pair of nodes that have a minimal route passing through node 0.

Partition the set of all anchored shapes P into the sets $P_{mn} \stackrel{def}{=} \{\ p : \ \Phi(d_m, p) \geq 1 \ \}$ for $0 \leq m \leq 5$ and $n = [m+1]_6$. Intuitively, each P_{mn} contains anchored shapes with one or more d_m and possibly some d_n components.

Lemma 2 *The sets* $P_{mn} \stackrel{def}{=} \{\ p : \ \Phi(d_m, p) \geq 1, p \in P \ \}, \quad n = [m+1]_6, \quad 0 \leq m \leq 5$ *partition* P.

Proof: We will first show that sets P_{mn} cover the entire set P. For an anchored shape $p \equiv (a_1 a_2 \cdots a_i \cdots a_k, \ell)$, there are two cases to consider.

In the first case,

$$\bigcup_{i=1}^{k} \{a_i\} \quad = \quad \{d_{i_1}\}, \quad \text{for some } i_1 \in \{0,1,2,3,4,5\}.$$

From this fact we can conclude that $p \in P_{i_1[i_1+1]_6}$.

In the second case,

$$\bigcup_{i=1}^{k} \{a_i\} \quad = \quad \{d_{i_1}, d_{[i_1+1]_6}\}, \quad \text{for some } i_1 \in \{0,1,2,3,4,5\}.$$

In this case, $p \in P_{i_1[i_1+1]_6}$.

We will now show that the sets P_{mn} are disjoint. Suppose not. Then, $\exists\ P_{i_1j_1}$ and $P_{i_2j_2}$, $i_1 \neq i_2$, such that $P_{i_1j_1} \cap P_{i_2j_2} \neq \emptyset$. Consider an anchored shape $p \in P_{i_1j_1} \cap P_{i_2j_2}$ with the shape $a_1a_2\cdots a_k$.

Case 1: $j_1 = i_2$.

$p \in P_{i_1j_1}$ implies $d_{i_1} \in \bigcup_{i=1}^{k} \{a_i\}$ and, $p \in P_{i_2j_2}$ implies $\bigcup_{i=1}^{k} \{a_i\} \in \{\{d_{i_2}, d_{j_2}\}, \{d_{i_2}\}\}$. Since by construction $j_1 = [i_1+1]_6$ and $j_2 = [i_2+1]_6$, and by the case under consideration $i_2 = j_1$ we can conclude that $i_1 \neq i_2$ and $i_1 \neq j_2$. But, $d_{i_1} \notin \{d_{i_2}, d_{j_2}\}$ and $d_{i_1} \notin \{d_{i_2}\}$, a contradiction.

Case 2: $j_1 \neq i_2$.

$p \in P_{i_1j_1}$ and $p \in P_{i_2j_2}$ imply $\{d_{i_1}, d_{i_2}\} \subseteq \bigcup_{j=1}^{k} \{a_j\}$. This would violate the definition of an anchored shape since $j_1 = [i_1+1]_6 \neq i_2$. ■

In order to calculate the total transit load λ_T we will need to further refine the partition P_{mn} into the sets $P_{mn}^{ab} \ \equiv\ \{\ p :\ p \ \in\ P_{mn},\ \Phi(d_m,\ p)\ =\ a,\ \Phi(d_n,\ p)\ =\ b\ \}$. The proof that P_{mn}^{ab} is a refinement of P_{mn} is straightforward and thus omitted.

We are now in a position to derive λ_T.

Lemma 3 *The total transit load at node 0 is given by*

$$\lambda_T = \lambda_G \sum_{k=2}^{e-1} 6k(k-1) \cdot q_k$$

where λ_G is the total rate of packet generation at a node, and q_k is the probability of a node communicating with a node k hops away.

Proof: Since there is a one-to-one correspondence between the anchored shapes and all minimal routes through node 0,

$$\begin{aligned}\lambda_T &= \sum_{p\in P} L(p), \quad \text{where } P \text{ is the set of all anchored shapes} \\ &= \sum_{i=0}^{5} \sum_{p\in P_{i[i+1]_6}} L(p).\end{aligned}$$

From the definitions of shapes and anchored shapes, the length of the shape associated with the above anchored shape p lies between 2 and $e-1$. From the definition of P_{mn} we know that $\Phi(d_m,\ p) \geq 1$. It follows from these observations that

$$\lambda_T = \sum_{i=0}^{5} \sum_{\substack{1\leq a\leq e-1 \\ 2\leq a+b\leq e-1}} \sum_{p\in P^{ab}_{i[i+1]_6}} L(p). \tag{4.3}$$

Note that all of the anchored shapes $p \in P^{ab}_{mn}$ have length $a+b$. Furthermore, since each shape associated with the anchored shapes of P^{ab}_{mn} has only components in the d_m and d_n directions, there are $\begin{pmatrix} a+b \\ a \end{pmatrix}$ shapes in P^{ab}_{mn}. Given each shape one can then derive $a+b-1$ anchored shapes. Therefore

$$\begin{aligned}\left|P^{ab}_{i[i+1]_6}\right| &= \begin{pmatrix} a+b \\ a \end{pmatrix} \cdot (a+b-1) \\ &= \frac{(a+b)!}{a!\cdot b!} \cdot (a+b-1) \\ &= \frac{[\sum_{l=0}^{5} \Phi(d_l, p)]!}{\prod_{l=0}^{5} [\Phi(d_l, p)!]} \cdot (a+b-1), \quad p \in P^{ab}_{i[i+1]_6}.\end{aligned} \tag{4.4}$$

Combining Equations (4.3) and (4.4) with Lemma 1, we get

$$\lambda_T = \sum_{i=0}^{5} \sum_{\substack{1\leq a\leq e-1 \\ 2\leq a+b\leq e-1}} \lambda_G \cdot q_{a+b} \cdot (a+b-1). \tag{4.5}$$

Since Equation (4.5) depends only on $(a+b)$, we can substitute k for $a+b$ to obtain

$$\begin{aligned}\lambda_T &= \sum_{i=0}^{5}\sum_{k=2}^{e-1}\lambda_G \cdot q_k \cdot (k-1)\cdot k \\ &= \lambda_G \sum_{k=2}^{e-1}\sum_{i=0}^{5} k(k-1)\cdot q_k \\ &= 6\lambda_G \sum_{k=2}^{e-1} k(k-1)\cdot q_k. \qquad \blacksquare\end{aligned}$$

Lemma 4 *The transit load at node 0 for packets not bound for an immediate neighbor is given by*

$$\lambda_{T^{2+}} = \lambda_G \sum_{k=3}^{e-1} 6k(k-2)\cdot q_k.$$

Proof: The proof of this lemma follows closely that of Lemma 3 with the additional restriction that node 0 cannot be in the last position for the anchored shapes being counted. Having node 0 in the last position of a anchored shape corresponds to having the anchored shape terminate in an immediate neighbor. This is exactly the traffic that we are trying to eliminate.

Since there is a one-to-one correspondence between the anchored shapes and all minimal routes through node 0,

$$\begin{aligned}\lambda_{T^{2+}} &= \sum_{\substack{(s,k)\in P \\ k\neq len(s)-1}} L((s,k)), \quad \text{where } P \text{ is the set of all anchored shapes} \\ &= \sum_{i=0}^{5} \sum_{\substack{(s,k)\in P_{i[i+1]_6} \\ k\neq len(s)-1}} L((s,k)).\end{aligned}$$

In contrast to Lemma 3, the lengths of shape s associated with the above restricted anchored shape (s,k) lies between 3 and $e-1$. From the definition of P_{mn} we know that $\Phi(d_m,\ (s,k)) \geq 1$. It follows from these observations that

$$\lambda_{T^{2+}} = \sum_{i=0}^{5} \sum_{\substack{1\leq a\leq e-1 \\ 3\leq a+b\leq e-1}} \sum_{\substack{(s,k)\in P^{ab}_{i[i+1]_6} \\ k\neq len(s)-1}} L((s,k)). \tag{4.6}$$

Note that all of the anchored shapes $(s,k) \in P^{ab}_{mn}$ have length $a+b$. Furthermore, since each shape s associated with the anchored shapes of P^{ab}_{mn} has only components in the d_m and d_n directions, there are $\left(\begin{array}{c} a+b \\ \\ a \end{array}\right)$ shapes in P^{ab}_{mn}. Given each shape one can then derive $a+b-2$ anchored shapes. The number of anchored shapes generated from each shape differs by 1 from Lemma 3 since we cannot use the last position in the shape. Therefore following that same steps as in Lemma 3 we arrive at

$$\begin{aligned}
\lambda_{T^2+} &= \sum_{i=0}^{5} \sum_{\substack{1 \le a \le e-1 \\ 3 \le a+b \le e-1}} \lambda_G \cdot q_{a+b} \cdot (a+b-2) \\
&= \sum_{i=0}^{5} \sum_{k=3}^{e-1} \lambda_G \cdot q_k \cdot (k-2) \cdot k \\
&= \lambda_G \sum_{k=2}^{e-1} \sum_{i=0}^{5} k(k-2) \cdot q_k \\
&= 6\lambda_G \sum_{k=3}^{e-1} k(k-2) \cdot q_k . \qquad \blacksquare
\end{aligned}$$

Theorem 1 *The branching probability, p_b, between adjacent service centers in the model is*

$$p_b = \frac{1}{6} \cdot \left(1 - \frac{1}{6 \cdot \sum_{k=1}^{e-1} k^2 \cdot q_k}\right)$$

Proof: p_b can be derived as the ratio of traffic bound for immediate neighbors to all traffic leaving a service center.

$$p_b = \frac{1}{6} \cdot \frac{\lambda_{G^2+} + \lambda_{T^2+}}{\lambda_G + \lambda_T}$$

Using the results of Lemmas 3 and 4 along with

$$\begin{aligned}
\lambda_{G^2+} &= \lambda_G (1 - 6q_1) \\
\sum_{k=1}^{e-1} 6k \cdot q_k &= 1
\end{aligned}$$

the theorem follows after some algebraic manipulation. $\blacksquare$

It should be noted that p_b only depends on q_k and the topology.

Lemma 5 *The throughput at each service center is* $6 \cdot \lambda_G \cdot \sum_{k=1}^{e-1} k^2 \cdot q_k$.

Proof: Jackson's theorem[27] states that the total throughput T_i at service center i is given by the solution to the set of traffic flow equations

$$T_i = \lambda_G + \sum_{k=0}^{3e(e-1)} p_b \cdot T_k \ , \ i = 0, \ldots, 3e(e-1).$$

By assumption A4 and the homogeneous nature of the C-wrapped H-mesh all T_i are equal. Therefore,

$$T_i = \frac{\lambda_G}{1 - 6p_b} \ , \ i = 0, \ldots, 3e(e-1).$$

Substituting p_b from Theorem 1 the lemma immediately follows. ■

Theorem 2 *The probability of a packet cutting-through an intermediate node is*

$$p_c = 1 - \left(\lambda_G \sum_{k=1}^{e-1} k^2 \cdot q_k \right) \cdot \overline{\ell}$$

where $\overline{\ell}$ *is the mean length or service time for packets.*

Proof: A packet can establish a cut-through at an intermediate node only if there are no packets being serviced or waiting for service at that node. Using Lemma 5 and Jackson's theorem that the probability of having zero packets at any node is $1 - \rho$ where ρ is the traffic intensity. ρ in terms of the throughput and service rate μ is given as follows:

$$\rho = \frac{T}{\mu} = \frac{T \cdot \overline{\ell}}{6} = \left(\lambda_G \sum_{k=1}^{e-1} k^2 \cdot q_k \right) \cdot \overline{\ell}.$$

and hence the theorem follows. ■

4.5 Distribution of Message Delivery Times

In a virtual cut-through message passing scheme, the delay that a packet incurs at a node depends on whether the packet is able to establish a cut-through at that node. If the packet establishes a cut-through, the delay incurred is negligible and assumed to be 0. Otherwise, the packet incurs both waiting and service time delays. Furthermore, since a packet cannot establish a cut-through unless there are no other packets waiting for service at that node, the FCFS queueing discipline is preserved at each node. From Jackson's theorem we know that the queueing network described in Section 4.4 has a product form solution. Therefore, each service center behaves as an M/M/1 queueing system.

The delivery time for a packet traveling n hops, denoted by D_n, can be expressed as

$$D_n = Y_0 + X_{n-1},$$

where Y_0 (X_{n-1}) is a random variable that represents the total time spent by a packet at the source node ($n-1$ intermediate nodes). Also let Y_k be a random variable that represents the total time spent by a packet buffered in an intermediate node.

Therefore,

$$\begin{aligned} P[D_n \leq t] &= P[Y_0 + X_{n-1} \leq t] \\ &= \sum_{m=0}^{n-1} P[Y_0 + X_{n-1} \leq t \mid \text{buffered at } m \text{ int. nodes}] \cdot P[\text{buffered at } m \text{ int. nodes}] \\ &= \sum_{m=0}^{n-1} P[\sum_{k=0}^{m} Y_k \leq t] \cdot P[\text{buffered at } m \text{ int. nodes}]. \end{aligned}$$

Note the $P[\sum_{k=0}^{m} Y_k \leq t]$ corresponds to an Erlang distribution with parameters $\mu(1-\rho)$ and $m+1$, i.e., $\mathrm{ERL}(\mu(1-\rho), m+1)$. This allows us to derive the probability density function of D_n as:

$$f_{D_n}(t) \quad = \quad \sum_{m=0}^{n-1} \left(\begin{array}{c} n-1 \\ \\ m \end{array} \right) (1-p_c)^m {p_c}^{n-1-m} \mu(1-\rho)^{m+1} \cdot \frac{t^m e^{-\mu(1-\rho)t}}{m!}.$$

Using the result

$$\int x^n e^{ax} dx = \frac{e^{ax}}{a} \sum_{k=0}^{n} \left(\begin{array}{c} n \\ \\ k \end{array} \right) (-1)^k k! \frac{x^{n-k}}{a^k}$$

and integrating $f_{D_n}(t)$ from 0 to t we get the delivery time distribution as

$$F_{D_n}(t) \quad = \quad \sum_{m=0}^{n-1} \left(\begin{array}{c} n-1 \\ \\ m \end{array} \right) (1-p_c)^m {p_c}^{n-1-m} \frac{[\mu(1-\rho)]^{m+1}}{m!} \cdot$$
$$\left\{ \frac{m!}{[\mu(1-\rho)]^{m+1}} - \frac{e^{-\mu(1-\rho)t}}{\mu(1-\rho)} \sum_{k=0}^{m} \left(\begin{array}{c} m \\ \\ k \end{array} \right) \frac{k! \cdot t^{m-k}}{[\mu(1-\rho)]^k} \right\}.$$

4.6 Numerical Examples and Simulation Comparison

In this section, parameters derived from the actual **HARTS** routing hardware are used to evaluate the probability distribution function for delivery times discussed in the previous section. Also presented is a comparison of the analytic results against a low-level functional simulation of the routing hardware of **HARTS**.

In contrast to the analytical model, the simulator makes very few simplifying assumptions in modeling the behavior of virtual cut-through in **HARTS**. The simulator accurately models the delivery of each message by emulating the timing of the routing hardware [21] along the route of a packet at the microcode level. Also captured are the internal bus access overheads that the packets experience if they are unable to cut-through an intermediate node. For example, when a transit packet arrives at an intermediate node, the following sequence of timed events are set into action. First, the receiver for that particular direction waits for the packet header to become available to attempt a routing decision. For the case of the H-mesh any incoming packet may have either arrived at its final destination or could be transmitted in one of possibly three directions. Second, the receiver schedules an access to an internal bus to reserve the first choice for a direction to transmit the packet. If the transmitter for this direction is free, the packet will cut-through this node with only the slight delay of waiting for the header and the single status query of the transmitter. If the first attempt to reserve the transmitter was unsuccessful, an attempt at an alternate transmitter is made, if applicable. If both of these attempts are unsuccessful, the packet is queued at this node for later transmission. Third, the receiver schedules events to signal the completion of the packet at this node. This may involve either unreserving a transmitter if the packet successfully cut-through or informing the module that simulates the handling of buffered messages. This detailed timing and tracking of messages allows different message scheduling, access protocols, and memory management strategies to be investigated. However, for the results presented in this section only a First-Come First-Serve single queue with unlimited memory was used.

In addition to the exponentially distributed packet lengths, the simulator can also use a discrete distribution of packet lengths where the user specifies the number of different types of messages, their lengths, and the probability of each type of message. Similar to the analytic model, packet arrivals are assumed to follow a Poisson arrival process.

For the examples presented in this section the following parameters were used. (Note that choice of these parameters is arbitrary and will not in general change our conclusions drawn in this section.) The dimension of the mesh was 7 resulting in 127 nodes in the system. The probability of a node communicating with a specific node k hops away was assumed to be inversely proportional to the number of hops, i.e., $q_k = \frac{1}{36k}$. The mean packet length for the analytic model was assumed to be 185.6 bytes. The distribution of packet lengths for the simulation were 64, 128, 512 bytes, each with probability 0.3, 0.5, 0.2, respectively. The results for three different packet generation rates are obtained. These correspond to 15%, 30%, 45% of the peak packet generation rate that can be supported by the routing hardware. Currently, the peak packet generation rate that can be supported by the routing hardware is 4 MBytes per second. All the distributions either generated or collected were for messages having their destination 5 hops from their source node.

Figure 4.3 shows a plot of the probability distribution function of the delivery times of a message traveling 5 hops in a H-mesh of dimension 7. The three curves in the figure show the variation in the probability distribution function with respect to the assumed message generation rate λ_G at each node. As would be expected, the delivery time distributions shift to the right as the load on the network is increased.

Figure 4.4 shows the inverse of the probability distribution functions in Figure 4.3. The inverse of the distribution function is useful to determine design parameters like delay bounds. For instance, one can select a delay bound such that the probability of a message being delivered within that bound is greater than a specified threshold. This would provide a probabilistic measure on the guarantees that can be provided in a real-time system during its operation.

Figures 4.5, 4.6, and 4.7 compare the analytic model against a low-level functional simulation of the routing hardware in **HARTS**. The results show that the analytic model predicts, with a reasonable accuracy, the delivery times for the loads shown. The jumps in the simulation results are due to the discrete distribution of the message length. It is

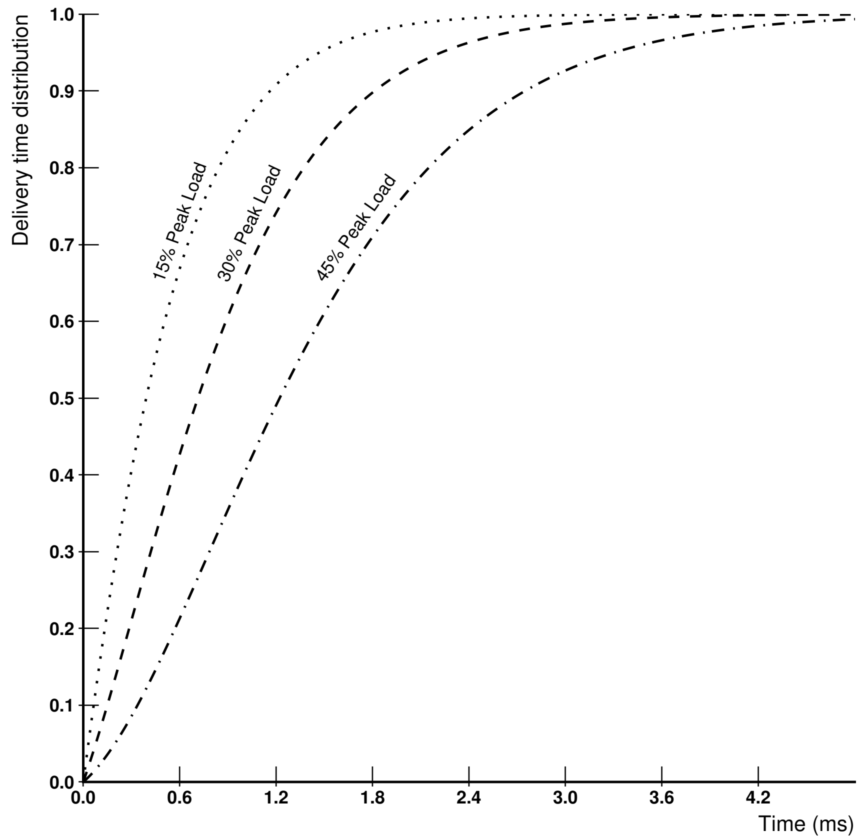


Figure 4.3: Probability distribution of D_5 in a H-mesh of dimension 7.

found that at higher loads (greater than 65% of the peak load) the differences between the simulation and the model can be significant. Reasons for these differences are currently being investigated. Also note, the analytic model overestimates the actual delivery times and therefore the model produces a pessimistic result. The slight discrepancy at small delivery times between the model and the simulation result from the model not taking into account the overheads of processing the message headers.

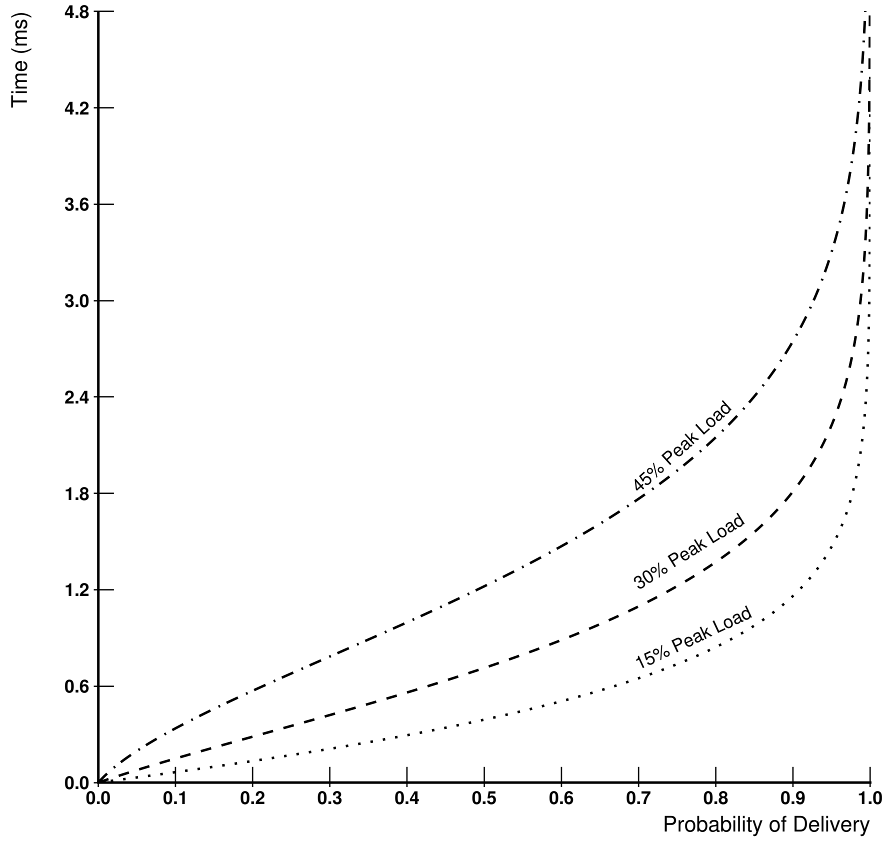


Figure 4.4: Delivery time vs. Probability of successful delivery.

4.7 Discussion of Results

The main contribution of this chapter is the derivation of the probability of cut-through for a C-wrapped H–mesh that has virtual cut-through switching capabilities. The techniques used in here can be extended to other interconnection topologies like hypercubes or rectangular meshes because the techniques depend only on ability to determine the fraction of minimal routes between a pair of nodes passing through a given node. In a hypercube or a rectangular mesh we can easily determine the required fraction of minimal routes between a pair of nodes that pass through a given node.

The distribution functions derived are useful in the design of real-time systems with hard

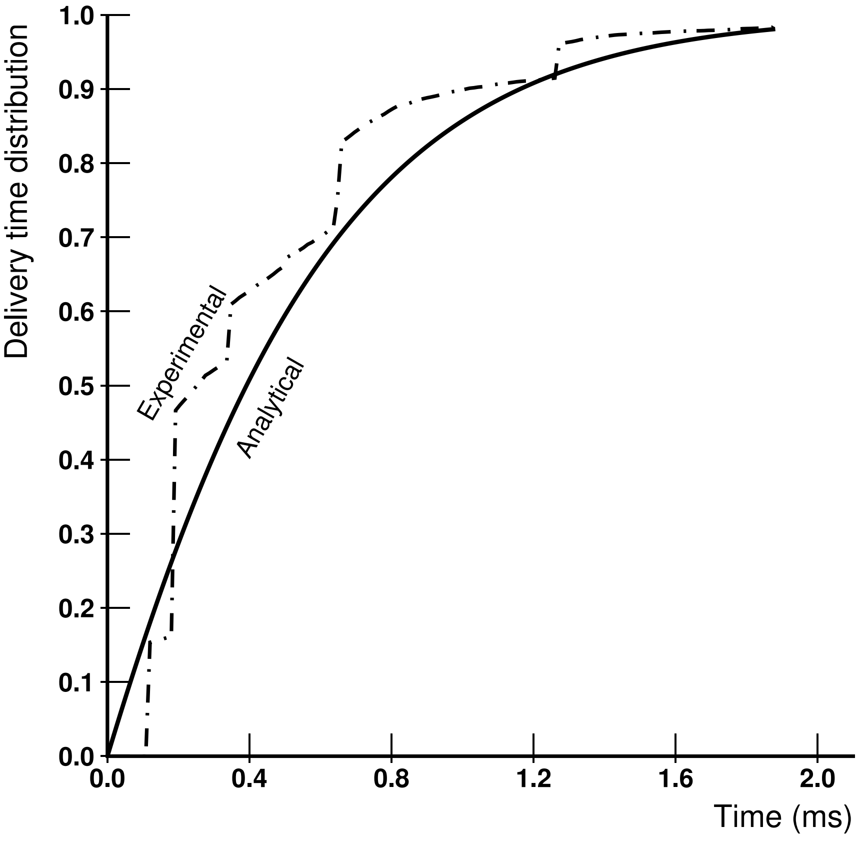


Figure 4.5: F_{D_5} at 15% Peak Load (H-mesh dimension 7.)

deadlines. They provide a probabilistic measure on the guarantees that can be provided for message exchanges during the operation of a real-time system.

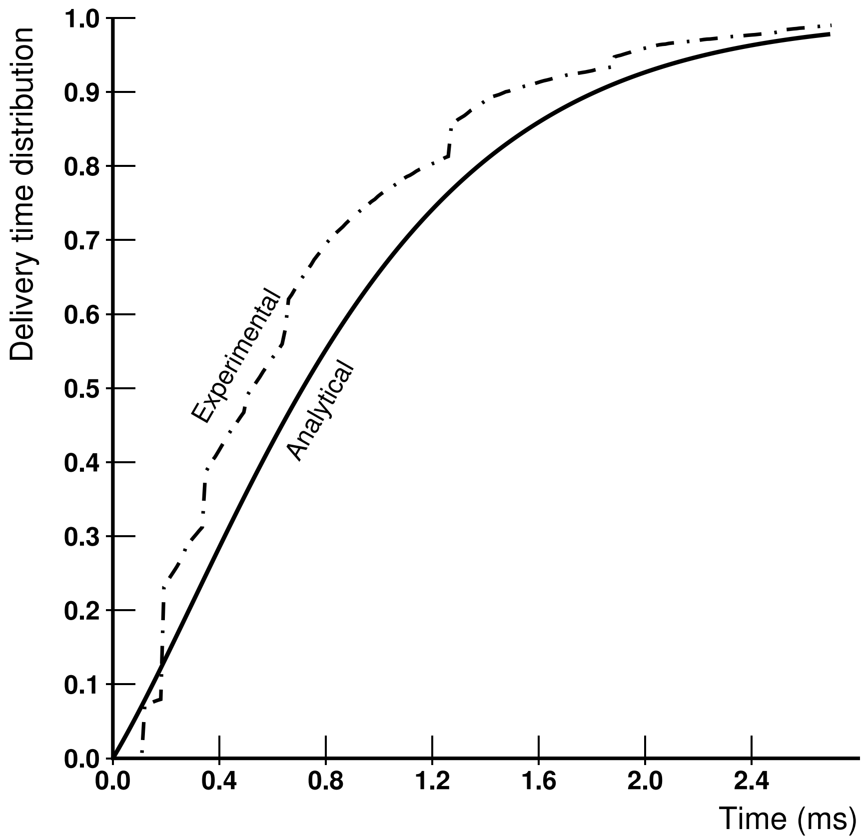


Figure 4.6: F_{D_5} at 30% Peak Load (H-mesh dimension 7.)

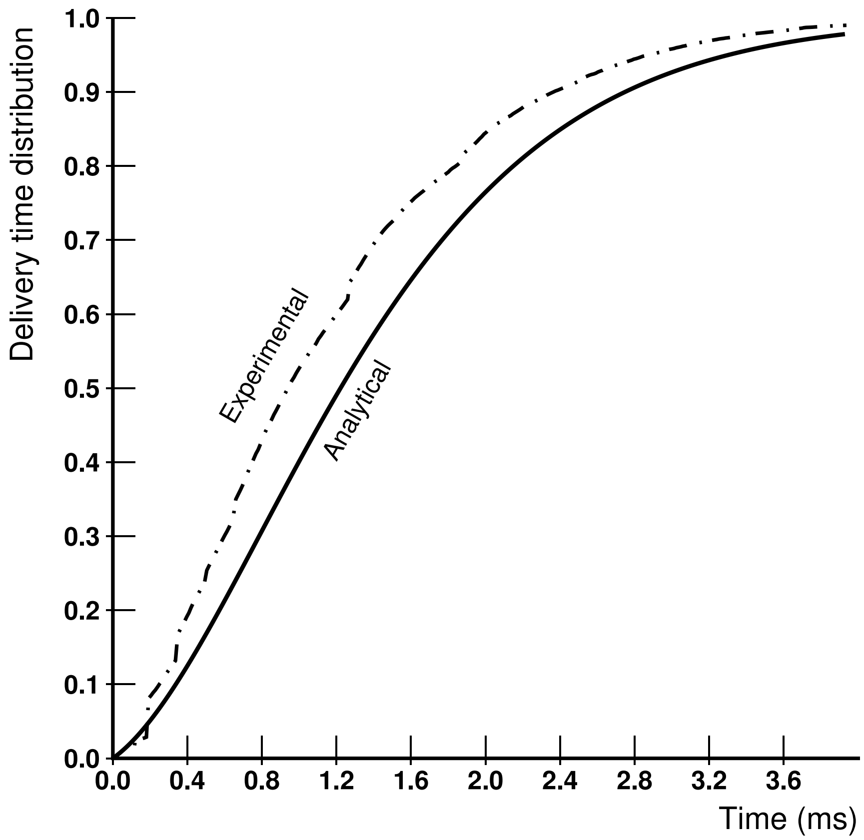


Figure 4.7: F_{D_5} at 45% Peak Load (H-mesh dimension 7.)

CHAPTER 5

THE POINT-TO-POINT MESSAGE SIMULATOR

This chapter discusses the design rationale and implementation insights gained during the development of the Point-to-Point Message Simulator. The long-term high-level goal of this effort was to develop a simulation environment for investigating issues related to supporting real-time communication systems interconnected by point-to-point interconnection networks. The short-term goal was to create a simulator that could be used to size some of the internal components of the PRC when operating in realistic conditions. Finally, the immediate goals were to provide the tools needed to perform the multi-mode routing and switching experiments described in Chapter 6.

The long-term goal of this work was motivated by the observation that many researchers working on different topics related to communication using point-to-point networks seemed to be constantly writing their own private simulators to show the one feature they were interested in exposing. These simulators usually only provided support for examining one free variable with the remaining parameters being implicitly fixed. This constant duplication of effort resulted in a great amount of wasted manpower and, in the case of inexperienced programmers, often produced results of questionable integrity.

The ideal solution would be to provide the researcher with a set of building blocks that can be composed using a high-level language or graphical editor to construct a custom simulator. These types of tools are commercially available for modeling LANs and signal processing systems but, unfortunately, at least for the features that we were interested in investigating, do not provide sufficient modeling detail.

On the other end of the spectrum, an obvious avenue that can be exploited is the existing gate level simulation tools. With sufficient encapsulation of the base components, one would think that this could be a viable alternative. For small systems this can be an alternative partially due to the built-in support some of the gate level simulators provide. For example, the negative exponential time delays that Cadence Verilog supports can prove to be quite useful. The problem in simulating point-to-point systems is that they require replication of the individual nodes, which quickly makes this approach impractical[1].

Our solution to this problem of balancing the lack of modeling detail against the computation overheads imposed by the replication necessary in simulating point-to-point networks is to provide a base set of primitives and a structure for combining these primitives. The idea is that the user can model in detail the components of interest to him and hopefully reuse previously designed components necessary to form the complete system under simulation.

The results of this work are the "Point-to-Point Message Simulator" (pp-mess-sim) that has the following features:

- Supports C-wrapped hexagonal mesh, wrapped square mesh, and binary hypercube.
- Accurately models the behavior of the low-level routing hardware (based on the PRC architecture).
- Supports multiple routing strategies.

[1] A Verilog simulation of the PRC in isolation currently requires at least 400MB of swap and 48 MB of main memory.

- Provides a textual specification language for mapping tasks onto the topology being simulated, thus allowing the end user to easily configure the message workload.
- Provides a structured framework for making extensions.

What should be of interest to those not interested in the PRC is the methods and structures that we use to achieve the specific capabilities mentioned above and how easily pp-mess-sim can be re-targeted for different topologies, routing hardware, and communication patterns.

5.1 General Organization and Structure

Figure 5.1 shows the directory structure of the pp-mess-sim source code tree. This figure also quite accurately depicts the functional separation that was identified, and the interfaces that needed to be developed between the different components of pp-mess-sim. These functional separations can be classified into a set of C++ classes supporting: the different network topologies (the `net` directory), the workload run on the simulated system (the `workload` directory), the parsing and storage of the run specification (the `spec` directory), or the implementation of the particular routing hardware of interest (currently the `imu-prc-ni` directory). The interaction between classes in these different functional areas is through a well-defined set of member functions such that the impact of adding new interconnection topologies and routing hardware is minimized to one area.

There are five fundamental classes that need to be briefly introduced before the remainder of pp-mess-sim can be explored. These are Net, Node, Task, Message, and Event.

The Net class serves as the base class for the classes CWHMesh, SQMesh, and HyperCube. Each of these derived classes implements the specific details of address translation and

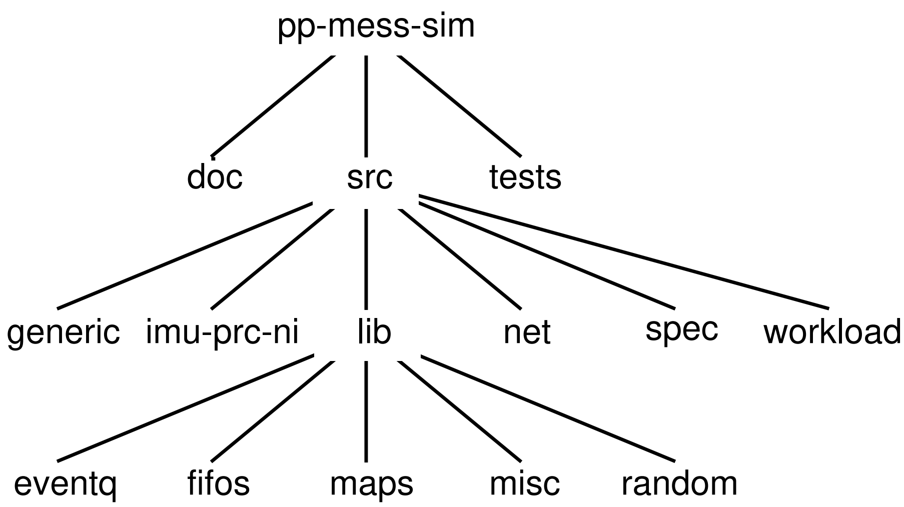


Figure 5.1: Organization of pp-mess-sim source tree

mapping for its particular topology. All of the other classes requiring information on the current topology must use the appropriate member functions of these derived classes.

The Node class implements the router/node-specific functions, and is dynamically allocated according to the size of the network under simulation and referenced through the derived net classes. This class and the subclasses stored within it can be easily changed to model different hardware or software.

The Task class implements the behavior of the workload imparted on the system for the duration of the simulation. Each instantiation of a Task has certain properties that are either explicitly or implicitly specified in the simulation specification. Once instantiated, the Task is assigned to a node much like its counterparts in a standard operating system.

The Message class is used just as an object to pass and possibly subdivide. This is the case for the PRC model in which the messages are actually subdivided into bytes that are individually simulated.

The Event class serves as a base class for all of the different types of events that implement the behavior of the system under simulation. This class contains pure virtual functions, thus requiring the support of a certain minimal set of operations in the derived classes.

5.2 Simulation Specification Language

Part of the flexibility and power of pp-mess-sim is derived from the language used to specify the parameters for each simulation run. This language, coupled with the built-in primitives, has enough expressive power to concisely construct a series of workloads that exhibit a wide range of communication patterns and resource usage. This is primarily accomplished by allowing the user to assign tasks with specific behavioral characteristics on a node-by-node basis if necessary to generate the desired workload. For the most part, these assignments can be performed implicitly by allowing the default rules to resolve the tasks assigned to each node.

The behavior of each communication task is characterized by eight parameters:

- the generation process
- the length process
- the maximum packet length
- the minimum packet length
- the target selection process
- the default routing algorithm
- the total number of packets

The generation process is used to generate the packet inter-arrival times in base unit of bus cycles and may be selected from a wide range of the standard random processes used in simulation. This process is guaranteed to have its own assigned random number stream that does not overlap with any others' during the lifetime of the simulation.

The length process is used to generate the length of a newly-generated packet in long-words. This process may be also selected from either negativeexpntl or lengthdiscrete random processes. The negativeexpntl provides the standard negative exponential distribution on packet lengths used commonly in analytical models. The maximum and minimum length parameters can be used to clip the lengths produced by this process negativeexpntl to prevent unrealistically small or large packets. The lengthdiscrete process allows the user to individually specify the probability of a discrete number of packet lengths. These processes, like the generation processes are guaranteed to have independent random number streams.

The target selection process selects the node that is the recipient for each generated packet. The user can currently choose from either NodeUniform or HopUniform. When NodeUniform is selected, each node, excluding the originating node, is equally likely to be selected as the recipient. If HopUniform is selected, the user specifies the probability of the packet traveling x hops. Once it is determined that the packet is traveling x hops, then all nodes x hops distant are equally likely to be selected as the recipient of the packet.

The default routing algorithm determines the routing algorithm and switching methodology that will be used to deliver the packet to the recipient. Users can currently choose from virtual cut-through, wormhole, packet switching, or source-list. The first three choices use a minimal path approach in selecting the route as the packet traverses the network and use their respective switching method. As currently implemented, source-list utilizes a user-specified route and a virtual cut-through switching method to deliver the packet.

The selection of a routing algorithm that the underlying hardware doesn't support is considered a specification error and aborts the simulation run during the initialization of the instantiation of the task in error.

The last key parameter describing a task's behavior is the total number of packets that need to be generated. This is actually the minimum number of packets the task will generate. A task will continue generating packets until all tasks in the simulation have satisfied their respective specification of the total number of packets. This approach allows the data collections routines to detect when they should shut down in order to prevent falsely collecting data as the network is "draining out" at the end of the simulation run.

In addition to the parameters describing the behavioral characteristics, the user must select the type of data collection that needs to be performed with respect to the packet being generated by the task. Currently, there are several different levels of detail, with the simplest collecting the mean, variance, min, max, and confidence intervals of packet delay. More complex data collection schemes are currently being investigated to explore posible correlation effects such as those observed during higher loads in the previous chapter.

The aggregate behavior of a node is specified by the number and different types of tasks that are assigned to it. This can be accomplished either by a default specification, an override for a particular node number, or by a named node specification.

It is easier to explain the process of fully resolving an input specification to a complete simulation specification through an example. Figure 5.2 shows a simple yet functionally complete specification file that we will use to illustrate both the simplicity and the power of the specification language.

There are three major blocks that the user must provide: the topology selection, the

```
topology begin
   select cwhm;
   size 4;
end

node default begin
   tasks 4;
   select task task_rt_real 2;
end

node 3 begin
   tasks 2;
   select task task_rt_real 1;
end

task default begin
   arrival NegativeExpntl(3368.421053);
   length  LengthDiscrete(0.3,8,0.5,24,0.2,88);
   target  NodeUniform();
   routing vc();
   history simple;
   packets 1000;
   drop 250;
   renew 1000;
end

task task_rt_real begin
   arrival NegativeExpntl(16000.00000);
   length  NegativeExpntl(8.0);
   target  NodeUniform();
   routing wormhole();
   history complex;
   packets 500;
   drop 100;
   renew 500;
   max 8;
   min 8;
end

general begin
   random seed 1353625084;
   timeout 640;
   option resources true;
   output  simple_example.out;
   errors  simple_example.err;
   results simple_example.results;
   debug   simple_example.debug;
end
```

Figure 5.2: Example of a simulation specification

default node, and the default task.

The first is the *topology selection* block. For our example lines 0–4 select a C-wrapped hexagonal mesh with 31 nodes (edge size 4).

The next block that must be provided is a *default node* specification. This is shown on lines 4–8 and directs pp-mess-sim to instantiate each node such that there are a total of 4 tasks, two of which will have the characteristics of the named task "task_rt_real" unless a specification of higher precedence for that node exists. The remaining tasks (two in this case) required to bring the total number of tasks up to the total specified number (line 6) will have the characteristics of the default task.

Lines 15–24 and lines 26–37 are two examples of task specification blocks. The first describes the behavior of the *default task* that is created as a last resort in order to satisfy the total number of tasks needed on a particular node. In this example, the default task generates packets with an inter-arrival time using a negative exponential with a mean of 3368 bus cycles. The packets will have a 3 in 10 chance of being 8 longwords, a 1 in 2 chance of being 24 longwords, and a 1 in 5 chance of being 88 longwords. Each node of the other 36 nodes will be equally likely to receive the packets generated. The packets will be routed using minimal-path routing and virtual cut-through. The user will be guaranteed that the task will generate at least 1000 packets of which the first 250 will not be considered for data collection. The renew line specifies that the random number streams allocated to this task should grab blocks of 1000 samples at a time from the master random number stream. The second task specification is an example of a named task which can be selected as shown in lines 7 and 12.

Lines 10–13 demonstrate the use of a node override in which the task mix assigned to node 3 is individually specified.

The *general block* (lines 39–47) is used for general parameters such as output file names and random number seeds. Although not explicitly required, it usually has to be provided since it is the only way the seed for the master random number stream can be specified.

The grammar describing the complete language is given in Appendix F.

5.3 Topology Generation and Support Functions

The Net class and its derived topology-specific classes play a key role in enabling pp-mess-sim to function as a reusable software tool. Much of this power is a result of being able to identify a small set of functions and parameters that allow the external interfaces to the topology support routines to be identical for all of the topologies that are implemented. The user is then forced to utilize only these functions when accessing topology specific information in their implementation of the routing hardware and routing/switching algorithms.

There are several assumptions made in creating a representation scheme for the generic interconnection topology that pp-mess-sim provides. First, nodes must be able to be assigned an unique integer label that is handled by pp-mess-sim as the *typedef* NodeId. Second, all references to adjacent nodes can be made in terms of an integer (*typedef* Direction in pp-mess-sim) that ranges from 0 to some maximum value that can be dependent on the topology under simulation. Given these two typedefs the user is then provided three functions for maneuvering within the topology and a function that returns a reference to a Node based on a NodeId. The basic structure of this scheme is shown in Figure 5.3.

The expressiveness of this generic approach will be shown by briefly studying these

```
class Net
{
public:
   SimSpecPtr sim_spec;
   Dimension edge_dimension;
   Direction max_direction;
   unsigned int total_nodes;
   unsigned int diameter;
   MTACG *rootacg;
   TaskIdTaskPtrAVLMap tasks;
   unsigned int generating_tasks;
   unsigned int holding_tasks;
   unsigned int delivered;
   MessageIdMessagePtrAVLMap messages;
   EventQ eventq;
   Net( SimSpecPtr );
   virtual ~Net() ;
   virtual Node& node( NodeId ) = 0;
   virtual Node& operator[]( NodeId ) = 0;
   virtual NodeId neighbor( NodeId , Direction ) = 0;
   virtual NodeId translate( NodeId , OffsetVec ) = 0;
   virtual OffsetVec map( NodeId , NodeId ) = 0;
   virtual void error (const char* );
};
```

Figure 5.3: Net class definition

basic functions in terms of the implementation of the support for the C-Wrapped hexagonal mesh and the wrapped square mesh. These are shown in Figures 5.4, 5.6, and 5.8 for the C-Wrapped hexagonal mesh and Figures 5.5, 5.7, and 5.9 for the wrapped square mesh.

The Net class doesn't force any particular addressing/storage method for the nodes but only requires that the derived class resolve the node() and operator[] methods since they are defined as pure virtual functions in the base class. Moving the implementation of the storage policy of nodes into the derived classes is done to allow for more efficient policies dependent on the topology. For all of the topologies implemented thus far this has been accomplished as an array of pointers to dynamic instantiations of the nodes in the system. The node() and operator[] methods then just index off of this pointer as shown on lines 19 and 21 of Figure 5.4 for the hexagonal mesh and lines 17 and 19 ofFigure 5.5 for the square mesh.

```
class CWHMesh : public Net
{
public:
  NodePtr *nodes; // Array Used to index instantiated nodes

  CWHMesh( SimSpecPtr );
  virtual ~CWHMesh();
  virtual Node& node( NodeId );
  virtual Node& operator[]( NodeId );
  virtual NodeId neighbor( NodeId , Direction );
  virtual NodeId translate( NodeId , OffsetVec );
  virtual OffsetVec map( NodeId , NodeId );
  virtual void error ( const char* );
private:
  unsigned short dir_v[6];
};

inline CWHMesh::~CWHMesh () {}

inline Node& CWHMesh::node( NodeId n ) { return *(nodes[n]); }

inline Node& CWHMesh::operator[]( NodeId n ) { return *(nodes[n]);}

inline NodeId CWHMesh::neighbor( NodeId n , Direction d )
{
  if ( d < max_direction )
    return (n + dir_v[d]) % total_nodes;
  else {
    error("Illegal neighbor mapping requested");
    return 0;
  }
}

```

Figure 5.4: C-wrapped hexagonal mesh class definition

The derived topology classes are allowed to define private data structures and methods in order to support the neighbor(), translate(), or map() methods if necessary. For the CWHMesh the dir_v[] is defined to assist the neighbor() and translate() methods. In the case of the H_4 mesh shown in Figure 2.1 the dir_v[] of Figure 5.4 would be initialized to {1,11,10,36,26,27} when the network is instantiated.

The neighbor() method is used to obtain the NodeId of an adjacent node for a given Direction and the current NodeId. The user can then utilize this function to implement the transport of information from one node to another. In both of the cases of the CWHMesh

```
class SQMesh : public Net
{
public:
    NodePtr *nodes; // Array Used to index instantiated nodes

    SQMesh( SimSpecPtr );
    virtual ~SQMesh();
    virtual Node& node( NodeId );
    virtual Node& operator[]( NodeId );
    virtual NodeId neighbor( NodeId , Direction );
    virtual NodeId translate( NodeId , OffsetVec );
    virtual OffsetVec map( NodeId , NodeId );
    virtual void error( const char* );
};

inline SQMesh::~SQMesh () {}

inline Node& SQMesh::node( NodeId n ) { return *(nodes[n]); }

inline Node& SQMesh::operator[]( NodeId n ) { return *(nodes[n]);}

inline NodeId SQMesh::neighbor( NodeId n , Direction d )
{
    switch ( d ) {
    case 0:
        return
            (n+1)%edge_dimension + (n/edge_dimension)*edge_dimension;
        break;
    case 1:
        return
            (n+edge_dimension) % total_nodes;
        break;
    case 2:
        return
            (n+edge_dimension-1)%edge_dimension + (n/edge_dimension)*edge_dimension;
        break;
    case 3:
        return
            (n+edge_dimension * (edge_dimension-1)) % total_nodes;
        break;
    default: // Big error time.... What should we do now!

        error("Illegal neighbor mapping requested");
        return 0;
        break;
    }
}
```

Figure 5.5: Wrapped square mesh class definition

```
NodeId CWHMesh::translate( NodeId src , OffsetVec offset )
{

   NodeId dest = src;

   for ( unsigned int i = 0 ; i < max_direction ; i++ ) {
      dest += offset[i] * dir_v[i];
   }

   dest %= total_nodes;
   return dest;

}
```

Figure 5.6: C-wrapped hexagonal mesh translate function

```
NodeId SQMesh::translate( NodeId src , OffsetVec offset )
{
   int src_row = src / edge_dimension;
   int src_col = src % edge_dimension;

   int delta_row = mod(src_row + offset[0] - offset[2], edge_dimension);
   int delta_col = mod(src_col + offset[1] - offset[3], edge_dimension);

   return edge_dimension * delta_row + delta_col;
}
```

Figure 5.7: Wrapped square mesh translate function

class and SQMesh class it is relatively straightforward to implement as shown on lines 23–31 in Figure 5.4 and lines 21–46 in Figure 5.5.

Both the map() and translate() functions manipulate data of the types NodeId and OffsetVec. The data type OffsetVec is just a max_direction element vector representing the number of hops in each direction some entity should take. The map() functions then provides a method in which the OffsetVec can be produced given source and destination nodes. The translate() functions provide the inverse operation of the map() functions. These topology support functions have proven to be quite adequate and have successfully insulated the implementation of the PRC-specific portions of pp-mess-sim from the knowledge of the interconnection topology or its implementation.

```
OffsetVec CWHMesh::map( NodeId src, NodeId dest )
{
  int N = int(total_nodes);
  int E = int(edge_dimension);
  OffsetVec retval(max_direction,0);
  int k = mod(dest - src, N);

  if (k < E) {
    retval[0] = k;
  }
  else {
    if (k > N - E )
      retval[3] = N - k;
    else {
      int t = mod((k - E), (3 * E - 2));
      int r = (k - E) / (3 * E - 2);

      if (t ≤ E + r - 1) {
        /* destination is in the lower part of H-mesh centered at source */
        if (t ≤ r) {
          retval[3] = r - t ;
          retval[4] = E - r - 1 ;
        }
        else {
          if (t ≥ E - 1) {
            retval[0] = t - E + 1;
            retval[5] = E - r - 1;
          }
          else {
            retval[5] = t - r;
            retval[4] = E - t - 1;
          }
        }
      }
      else {
        /* destination is in the upper part of H-mesh centered at source */
        if (t ≤ 2 * E - 2) {
          retval[3] = 2 * E - t - 2;
          retval[2] = r + 1;
        }
        else {
          if (t ≥ 2 * E + r - 1) {
            retval[0] = t - 2 * E - r + 1;
            retval[1] = r + 1;
          }
          else {
            retval[2] = 2 * E + r - t - 1;
            retval[1] = t + 2 - 2 * E;
          }
        }
      }
    }
  }
  return retval;
}
```

Figure 5.8: C-wrapped hexagonal mesh map function

```
OffsetVec SQMesh::map( NodeId src, NodeId dest )
{
   OffsetVec retval(max_direction,0);

   src %= total_nodes;
   dest %= total_nodes;

   int src_row = src / edge_dimension;
   int src_col = src % edge_dimension;

   int dest_row = dest / edge_dimension;
   int dest_col = dest % edge_dimension;

   int delta_row = dest_row - src_row ;
   int delta_col = dest_col - src_col ;

   if ( delta_row )
      retval[2] = - delta_row;
   else
      retval[0] = delta_row;

   if ( delta_col )
      retval[3] = - delta_col;
   else
      retval[1] = delta_col;

   return retval;
}
```

Figure 5.9: Wrapped square mesh map function

5.4 Event Management and Flow

5.4.1 Event queue

Two of the major problems that needed to be overcome during the development of pp-mess-sim were the representation of time and the efficient management of the queue of pending events. The importance of both of these problems is amplified by the fact that we have both high-level and detailed low-level activity being modeled. In the case of the PRC model, the transmission of each byte of a message can generate up to 6–8 events per node.

In order to accurately size some of the components of the PRC it was decided early in the development of pp-mess-sim that at a minimum it had to be able to accurately resolve an individual bus cycle on the CTBUS. Given our target speed of a 40ns CTBUS cycle, if an unsigned long int (32 bits) were to be used to represent time this would only allow a simulation of approximately 3–4 minutes of elapsed time. This was deemed to be too short and a two-level scheme was adopted in which global time is represented as an `unsigned long long` (64 bits) and the delta time measurements represented as a unsigned long. This two-level scheme was chosen over a scheme in which all time intervals are represented as an unsigned long long since the operations on long long integer data types are open coded on most of the 32 bit processor architectures on which pp-mess-sim was expected to execute.

It was also noticed during the course of debugging the implementation of the PRC-specific modules that a few of characteristics of the events being processed should influence the implementation of the event queue manangement routines. First, there was a significant number of events being processed that were being submitted with a delta time of zero time units (a mature event). Second, as an artifact of how some of the low-level bus arbiters were being implemented, there were a fair number of events being canceled and removed from the event queue. These observations led us to partition the event queue into two parts and to support a lazy delete mechanism for canceling events that are located in the event queue.

Taking all of the above factors into consideration resulted in the implementation of the event queue as a container class of pointers to the base event class that has the following characteristics:

- Global time is represented as a unsigned long long (glb_time).
- Events are submitted to the event queue management class with a delta time field represented as an unsigned long and are only allowed to use half of the available resolution of that field.
- A priority heap keyed on an unsigned long is used for the non-mature event list.
- A FIFO is used for the mature event list.
- Events submitted with a non-zero delta time are stored in the priority heap with a key that is equal to the submitted delta time plus the key of the last item removed from the priority heap.
- When an event is removed global time is advanced by the difference between the key of previous item removed from the priority heap and the key of the item being currently removed.
- If at any time the key of the last item removed exceeds half of the resolution of the delta time data type, all keys within the priority heap are adjusted by subtracting the value of the last key removed from the priority heap. This is usually accomplished

```
class Event
{
public:
   delta_time delta; // Time from current time to event occurance
   glb_time e_time; // Global time event occurs (Set on dequeue.)
   EventLoc loc; // Event Location (Indicates ownership)
   bool cancel; // Event has been canceled.
   unsigned long id; // Unique Identifier
   NetPtr net; // Network that event is taking place in.

   Event ( NetPtr , delta_time );
   virtual ~Event () ;
   virtual void cancel_event();
   virtual void error ( const char *);
   virtual bool mtrace() = 0;
   virtual void handler() = 0;
private:
   static unsigned long last_id; // Id Allocator

};

typedef Event* EventPtr;
```

Figure 5.10: Base Event class definition

by removing all items and placing them in the mature event FIFO while adjusting the delta time field. The FIFO is then emptied back into the priority heap. After repacking the priority heap the key of the last item removed is set to zero.

5.4.2 Event hierarchy and structure

All events in pp-mess-sim are implemented as classes derived from the Event class shown in Figure 5.10. What is important about this structure is that all derived events must implement the handler() and the mtrace() functions. This structure also allows both main event handler and hardware units to cause the execution of the code associated with the events with no knowledge of what the event actually does, i.e., the dispatch of the event is handled by the C++ virtual function mechanism.

The events derived from the base class store supplemental information so that they can be executed in isolation. For example, Figure 5.11 shows the event that is responsible for

```
class MessCreateEvent : public Event
{
public:
  TaskPtr task;
  NodePtr node;
  MessCreateEvent ( NetPtr , NodePtr, TaskPtr , delta_time );
  virtual ~MessCreateEvent ();
  virtual bool mtrace();
  virtual void handler();
  static void* operator new(size_t);
  static void operator delete( void* );
};

typedef MessCreateEvent* MessCreateEventPtr;
```

Figure 5.11: Message create event class definition

message generation. When the handler for this event is executed, it needs to determine what node and task was responsible for its creation. Hence, a pointer to task and node are stored locally.

Like all discrete event simulators, most of the effort in modeling the system is spent in writing and debugging the event flow. pp-mess-sim has a crude built-in message trace capability to assist in debugging. This feature is implemented by requiring all events to implement a mtrace() member function that returns a boolean. There is a global associative map stored in the network under simulation that contains the identifiers of messages that should be traced. It is the responsiblility of the mtrace() function to return true if it is an event that is related to a message identifier that exists in the global map. For events that have no relationship to particular messages, the mtrace() function is just an inline for a "return false." What is surprising is that the consistent availability of such a function can reduce the debugging time by orders of magnitude.

5.5 Insights into Modeling Hardware

During the construction of the classes that model the low-level behavior of the PRC, several techniques for modeling different types of components were developed that were not immediately intuitive. These problems usually centered around modeling the different types of asynchronous behavior present in the PRC. We describe below the techniques that we used for modeling the bus arbiters, the bus interfaces, and the implementation of the flow control behavior.

5.5.1 Bus behavior

The problem that we were immediately faced with when modeling the bus units was when to consider the transaction complete. Tightly-coupled to this problem is how to model the actual bus arbitration algorithm. Our technique solves both of these problems in an efficient manner.

Each bus that operates in an asynchronous mode has three major types of event classes defined: an arbitration event, a cycle event, and a complete event. Consider the following scenario, where an event handler for the interface has been triggered due to the availability of data. The proper behavior is to arbitrate for a shared bus, and once becoming master of the bus to write data to a slave unit across the bus. The event handler dealing with this data accomplishes this by first creating a complete event that will be used to indicate to the interface unit that the future bus cycle is complete. The initial handler then creates a cycle event that describes the nature of the future bus cycle such as the destinations of the transfer and of the data. One of the fields of the cycle event is a pointer to a complete event. This pointer is set to the complete event that the handler previously created. The handler then calls a "submit" member function for the class that actually models the bus arbiter. This submit function stores the cycle event internally and creates an arbitrate event if one is not already pending for the next possible arbitration time. The handler for the arbitrate event will then retrieve the appropriate cycle event, as determined by the arbitration algorithm, that was previously stored, execute its handler, and then if possible, execute the handler for the complete event. Finally, if there are remaining cycle events still stored, a new arbitrate event is created and placed in the global event queue.

There are two advantages to this approach. First, buses that are inactive will produce no events. Second, the arbitration algorithms and timing behavior are located in a single event handler and can be easily modified.

5.5.2 Flow control

Flow control for the PRC is physically coupled to the state of certain signals. When the **IRDATA** line from a CTBUS transmitter is true, then the device that has the transmitter reserved can arbitrate for the bus and write forward channel data. The simple approach would be to schedule an event every bus cycle that would check the status of the **IRDATA** and take the appropriate action. The problem with this approach is that the number of events scheduled becomes excessively large very quickly. Efficiently modeling this type of interaction can be quite challenging.

Our solution is based on a technique used in the implementation of many windowing systems. Each unit that has a signal that other devices may be asynchronously basing their event flow on has a FIFO in which the other devices may enqueue a "wakeup" event. The class that controls the signal that others are watching will remove all the events in the FIFO and enqueue them with a delta time of zero in the event queue any time the state of the line changes. The other devices may then use the wakeup event to perform the actions based on the new state of the line.

In the PRC this technique was used for both the byte-level flow control and the TFU's decision on whether to attempt a reservation of the CTBUS transmitter.

5.6 Data Collection Support

Originally, data collection was handled in pp-mess-sim as part of the message receive event handler. This just stored the collected data into the Task object that generated the message that had triggered the event. This approach was sufficient for the results presented in the next chapter.

More recently, we have restructured the data collection facilities to simplify the addition of custom collection methods. These changes center around the creation and maintenance of a MessHistory list. As a message is routed through the network and passes through different hardware components, these components have the option of attaching an information record to the message history list. Currently the following type of records are maintained:

- Enter
- Cut
- Inject
- Buffered
- Tail

Each of these information history records also contains the node id and a global timestamp indicating where and when the record was added. This new structure allows the message receive event handler at the destination node to call a user provided routine that processes the list and extracts the metrics of interest to the end user. For example, the correlation between routing/switching decisions in adjacent nodes can be investigated by looking the the Cut and Buffered records in the MessHistory list.

5.7 Summary

In this chapter, we have presented the design rationale and provided a glimpse at the implementation of the point-to-point message simulator. There are several unique features about this tool that can be easily exploited. First, its ability to capture both the high-level and, when necessary, low-level details of the system provide an opportunity to understand some of the low-level interaction effects not previously investigated. Second, the interfaces are designed such that models for different routing hardware can be developed and compared within a common environment. Third, the generic topology approach provides unique opportunities for investigating how sensitive different routing hardware is to the topology specific features.

Although a relatively complex system (2 MB of sources), incremental modification is quite tractable as proven by some of the more recent enhancements being attempted by other researchers within our lab.

CHAPTER 6

MULTI-MODE ROUTING AND SWITCHING

This chapter consolidates the results of the three previous chapters to formulate one of the first experiments that fully exploits the flexibility of the PRC. Although the results from the experiments presented here are relatively clear, the entire series of experiments suggested by our findings is much more than we could hope to accomplish in this dissertation.

This chapter investigates how multiple classes of traffic using different switching methods and routing algorithms *interact* with one another. We call the ability to support multiple classes of traffic, each possibly with their own switching method, multi-mode routing and switching.

After motivating with a simple analytical argument why there might be great potential in pursuing a scheme supporting multi-mode routing and switching, we formulate an experiment to answer the following questions:

- Does the presence of a low percentage of traffic using minimal-path routing with wormhole switching adversely affect the behavior of the traffic being routed using minimal-path routing with virtual cut-through switching in a C-wrapped hexagonal mesh?
- Is the wormhole traffic in the above scenario unstable in the presence of the virtual

cut-through traffic?

But why are these questions important in the first place? The ability to support multiple classes of traffic simultaneously with minimal interference can play a fundamental role in forming the basis upon which a communication subsystem capable of supporting real-time communication can be constructed. For example, associate one class of packets with background traffic (non real-time) and the other class of traffic with message requiring real-time delivery. This example is not that unrealistic because in most distributed real-time systems the actual percentage of traffic actually requiring hard deadlines is small. The control surface of the airplane requires consistent and timely updates, in contrast to internal cabin temperature display where "best effort" will suffice.

The usefulness of supporting multiple classes of traffic does not have to be cast only in terms of real-time communication in order to still hold merit. We can look at the communication support provided in the common workstation. One class can support standard socket-based communication while the other to carries "out-of-band" data. The point is that the different classes will have different characteristics as a function of routing and switching schemes used to support the classes and each of these can be individually exploited.

These two questions are investigated within the assumed operating environment provided by pp-mess-sim. The choice of using virtual cut-through switching and wormhole switching for the two traffic classes in this experiment was modivated by both practical issues and issues of fairness. The minimal-path routing algorithms for both wormhole switching and virtual cut-through switching were the most tested of all the routing and switching combinations implemented in pp-mess-sim. This choice also allowed the same routing algorithm to be used for both classes of traffic and only have the switching scheme be different. The availability of the analytical models for each of the traffic classes in isolation also influenced the design of experiments.

6.1 Motivation: Wormhole vs. Virtual Cut-through

In this section we derive an approximation for the expected packet latency using minimal-path routing and wormhole swithing in a C-wrapped hexagonal mesh. This is compared against the expected packet latency using minimal-path routing and virtual cut-through switching to conclude that the network load and number of hops the packet needs to travel determines the switching scheme that minimizes the packet latency. These comparisons are made in isolation; that is, they don't account for the presence of the other class of traffic.

We will use the following notation which parallels Chapter 4:

λ_G	packet rate generated by each node into the network
$(\overline{k})k$	(average) node distance measured in hops
q_k	probability of a packet destination being located k hops away
α	ratio of header length to the entire packet length
t_m	service time for a packet in a channel
$\overline{T}_{wh}$	packet latency using wormhole switching
$\overline{T}_c$	packet latency using cut-through switching
p_w	probability of a packet being required to wait for a channel
c_s	a constant representing the link arbitration/scheduling mechanism used when a wormhole–switched message encounters a busy link.

We can directly use the result from Theorem 2 from Chapter 4 which gives us:

$$p_w = \lambda_G t_m \sum_{k=1}^{e-1} k^2 \cdot q_k$$

The average latency of a packet using minimal-path routing and wormhole switching in the C-wrapped H–mesh, $\overline{T}_{wh}$, may then be approximated by:

Theorem 3

$$\overline{T}_{wh} = \frac{(1+\alpha(\overline{k}-1)) \cdot t_m}{1 - \frac{c_s}{6}\overline{k}^2 \ \lambda_G \ t_m} \tag{6.1}$$

Proof: There are three components in the packet latency. The first is the transmission time of the packet header at each node. Cut-through at each node does not save time in header transmission, because the header must be received in full before the node can select the outgoing transmission link. The second component is the waiting time at each node due to blocking. The final component is the packet transmission time.

The average packet delay per node, T_d, due to waiting for the current link occupant to complete its in-progress transmission and possibly other waiting packets is $c_s p_w \overline{T}_{wh}$, where c_s is a constant representing the link arbitration/scheduling mechanism used. The minimum of c_s, which equals 0.5, occurs when the blocked packet is scheduled to be transmitted next. The header transmission time is given by $\overline{k}\alpha t_m$ and the packet body transmission time by $(1-\alpha)t_m$. Hence, we get

$$\begin{aligned}
\overline{T}_{wh} &= \overline{k}T_d + (1-\alpha)t_m + \overline{k}\alpha t_m \\
&= c_s\overline{k}p_w\overline{T}_{wh} + t_m(1+\alpha(\overline{k}-1)) \\
&= \frac{(1+\alpha(\overline{k}-1))t_m}{1-c_s\overline{k}p_w} \\
&= \frac{(1+\alpha(\overline{k}-1))t_m}{1-c_s\overline{k} \ \lambda_G \ t_m \sum_{k=1}^{e-1} k^2 \cdot q_k}.
\end{aligned}$$

Using the fact that $\sum_{k=1}^{e-1} 6k^2 \cdot q_k = \overline{k}$, we get

$$\overline{T}_{wh} = \frac{(1 + \alpha(\overline{k} - 1)) \cdot t_m}{1 - \frac{c_s}{6}\overline{k}^2 \lambda_G t_m} \quad \blacksquare$$

We can make a few observations concerning the latency of packets that use wormhole switching from the above equation. From the denominator we see that as the packet generation rate increases from zero, the routing time increases gradually at first, and then rapidly. When the packet rate approaches the value which makes the denominator zero, the latency becomes infinite. This value $\lambda_{G,max}$ is the maximum packet rate the network can support for wormhole switching, and is given by:

$$\lambda_{G,max} = \frac{6}{c_s t_m \overline{k}^2}.$$

We also see that the latency increases rapidly with the average hop distance $\overline{k}$. This agrees intuitively with the observation that the longer the hop distance of a packet is, the more likely it is to block other packets. This is because packet data is not removed from the network when the packet is not able to cut through a node in wormhole switching. This means that for a wormhole switching scheme, the placement of tasks and the minimization of task separation is of prime importance. The latency is also sensitive to c_s (a function of the resource scheduling policy), although its effects are less than those of $\overline{k}$.

Fig. 6.1 shows the packet latencies for H–meshes of dimensions 3, 5, 7 and 9, assuming an average packet length of 128 bytes, a four-byte routing header, a link bandwidth of 16.6 MBytes/sec, and $c_s = 1.0$. Each destination is equally probable. When the packet generation is low (near zero), the latencies for the different H–meshes are similar, but as the generation rate increases, the larger mesh ($e = 7$) quickly reaches link saturation, while the latency for the smallest mesh ($e = 3$) increases rather slowly. The maximum packet

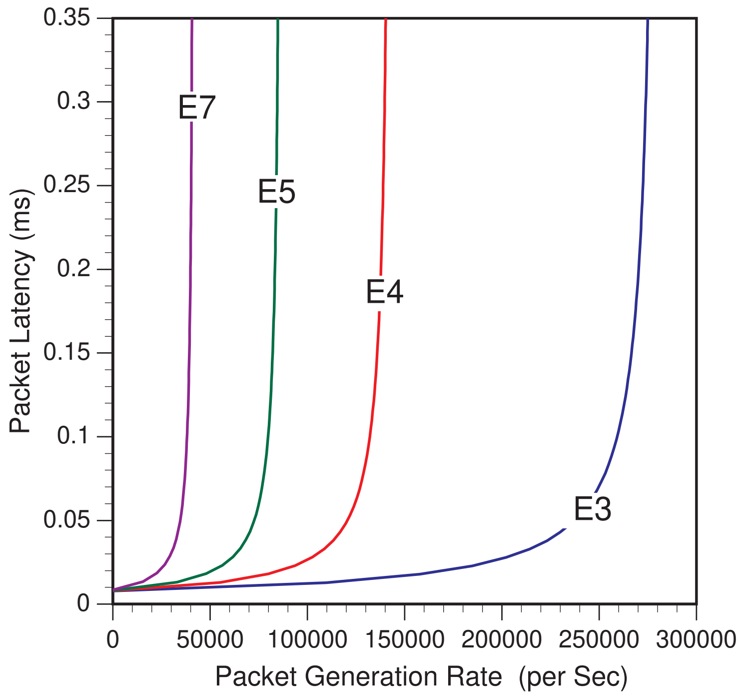


Figure 6.1: Wormhole switching latencies as a function of packet rate. ($c_s = 1.0$)

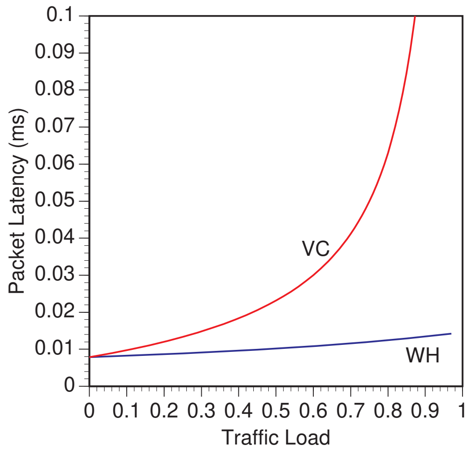
generation rate for the smaller H–meshes is higher due to the fact that the average packet distance is larger.

The packet latencies for virtual cut-through switching have been analyzed in [26], and given below:

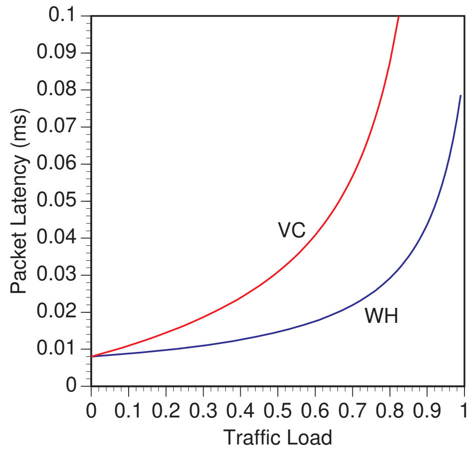
$$\overline{T}_c = \frac{\overline{k}t_m}{1-\rho} - (\overline{k}-1)(1-\rho)(1-\alpha)t_m.$$

It is instructive to compare the packet latencies for the two forms of switching as a function of channel load. The graphs of the latencies for different dimensions of H–mesh are shown in Fig. 6.2. Again, we have assumed a packet size of 128 bytes with a four byte routing header, a link bandwidth of 16.6 MBytes/sec, and $c_s = 1.0$.

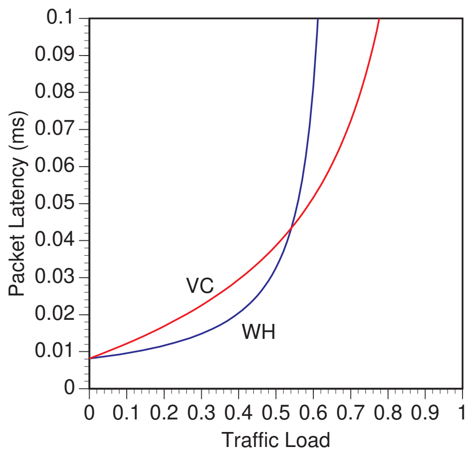
One can see that for low traffic, the latencies of both methods are almost identical. This is because at low traffic, the probability of blocking is very small, and therefore packets are not likely to be blocked at intermediate nodes. The latency is then determined mainly by



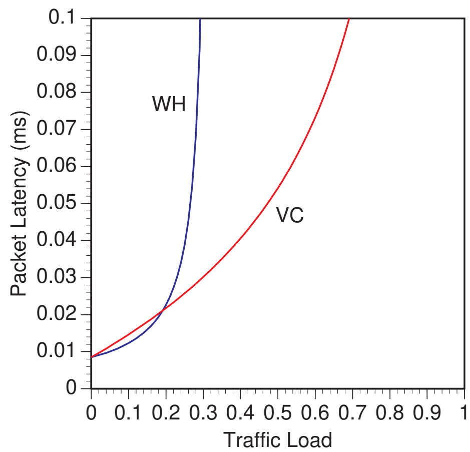
(a) $e = 3$



(b) $e = 4$



(c) $e = 5$



(d) $e = 7$

Figure 6.2: Packet delivery latencies for a C-wrapped H–mesh with edge dimension e. ($c_s = 1.0$)

link transmission time. For higher dimensions of the H–mesh, wormhole switching is not feasible for traffic load beyond a certain threshold. This threshold depends on the average number of hops for a packet. For a 3-dimensional H–mesh the packet distance is short enough[1] that the maximum traffic is limited by the bandwidth of the communication links themselves.

For low traffic loads, wormhole switching actually takes less time to deliver packets on average, while the opposite is true for high loads. The break-even point of traffic load also decreases as the size of the mesh increases. This is due to the average packet distance increasing with the size of the mesh. As pointed out earlier, the longer the packet, the more likely it is for wormhole switching to cause network congestion, resulting in higher average packet latencies. This also causes the maximum traffic load that can be supported by wormhole switching to decrease with the size of the mesh. For H–meshes of edge dimensions less than 5, wormhole switching always gives a lower latency than virtual cut-through. For an H–mesh of edge dimension 5, however, the break-even traffic load is 0.55. By the time the edge dimension reaches 7, the break-even point has dropped to 0.22. Consequently, depending on the traffic load and average packet distance, one switching method might be chosen in preference to the other.

6.2 Experimental Results

In the previous section we were able to conclude from the simplified models that one switching/routing scheme might be preferable to another as a function of the total network load and individual characteristics of the packet to be transmitted. Unfortunately these models do not provide much insight into how the real life issues such as the effects of flow

[1]The communication diameter of a 3-dimensional H–mesh is only 2.

control, routing algorithm processing time, and how multiple routing/switching methods interact with one another to change of overall behavior of the system.

To answer the two questions originally posed at the beginning of this chapter we performed a series of simulations of the IMU-PRC-NI triple on H_3, H_4, and H_5 meshes using pp-mess-sim. This simulations allows us to capture the real life factors and gain some confidence in the utility provided by having a flexible router. In the interest of brevity we will only present the results from the simulations on H_5 and note that the same observations and conclusions can be made concerning the data not shown.

The following experimental setup was used:

- Class I traffic
  - Minimal-path routing with virtual cut-through switching.
  - Inter-arrival process NegativeExpntl().
  - Packet length LengthDiscrete with $P[l = 8\,\text{longwords}] = 0.3$, $P[l = 24\,\text{longwords}] = 0.5$, and $P[l = 88\ \text{longwords}] = 0.2$.
- Class II traffic
  - Minimal-path routing with wormhole switching.
  - Inter-arrival process NegativeExpntl().
  - Packet length Fixed at 8 longwords.
- Parameters for inter-arrival processes are set as follows:
  1. Select percent peak link load as the primary parameter.
  2. Select percentage of traffic that should be Class II.

3. Solve for the two generation rates as a function of the two parameters above.

- Wormhole timeout set at 640 bus cyles (discussed below)

The characteristics of the traffic generated by the task specification that corresponds to the class I traffic is intended to act as the higher percentage "background" traffic which is then mixed with the minimal-path wormhole traffic. The task specification that corresponds to the class II is used to generate the low percentage of wormhole traffic injected into the nework. Both classes use a negative exponential interarrival process with their rates adjusted to achieve the desired mix of traffic.

One of the weaknesses in the current design of the PRC was uncovered during the simulations performed for this experiment. As currently implemented, the PRC only provides a single virtual channel across each physical link. For virtual cut-through switching this doesn't pose a problem since the packet is always allowed to make forward progress by buffering at an intermediate node. Wormhole switching does present problems once the load exceeds a certain threshold. We have two solutions to this problem. First, the PRC RX can implement a timeout on the period of time that a packet is willing to wait for a transmitter, after which the packet is buffered at the intermediate node and resubmitted. Second and more long term (future work), the PRC implementation can be modified to support more than a single virtual channel per physical link. We delay the discussion of this alternative to Chapter 7.

The results of this experiment are presented in three series of graphs at the end of this chapter. The first, examines the delivery latency of the class I traffic as the mix of class I and class II is varied. This first series is presented to answer the first question posed. The second series of graphs looks at the observered latency of the class II traffic as traffic is varied. This series is intended to address the second question. The last series superimposes some of the results of the first two series and allows to draw some conclusions that are discussed later.

Figures 6.3 through 6.5 show the delivery latencies of class I traffic (virtual cut-through) traveling 2, 3 or 4 hops in the mesh. Each separate curve is for a different percentage of class I versus class II traffic mix. Note that the curves remain tighly coupled as the class II traffic is varied from 5% to 30% of the total traffic being injected into the network. The conclusions that we can draw from these figures is that the class I traffic is not adversely affected by the class II traffic even as a function of the total number of hops the packets are required to travel. If this were not the case we would expect the delivery time to vary significantly as the percentage of the class II was increased.

Figures 6.6 through 6.9 show the delivery latencies of class II traffic (wormhole) traveling 1, 2, 3 or 4 hops in the mesh. Each separate curve is for a different percentage of class I versus class II traffic mix. Here again the curves do not significantly diverge from one another allowing use to conclude that the the class II traffic is not adversely affected by the increase in overall percentage class II traffic.

Figures 6.10 through 6.13 compares the delivery latencies of class I and class II traffic for the average number of hops and individually for 2, 3, or 4 hops. The top three curves in each figure represent the class I traffic for three different percentages of class I/class II traffic mixes. The lower three curves refer to the class II traffic for those same three traffic mixes. The key point of this series is that the relative benefits that one can gain by operating in one class or the other is not lost as the class II traffic percentage is varied in this range. If this were not true the two sets of curves should not maintain their relative differences from one another or worse yet actually cross.

6.3 Discussion

We opened this chapter by posing the two questions:

- Does the presence of a low percentage of traffic using minimal path routing with wormhole switching adversely affect the behavior of traffic being routed using minimal path routing with virtual cut-through switching in a C-wrapped hexagonal mesh?
- Is the wormhole traffic in the above scenario unstable in the presence of the virtual cut-through traffic?

Before pursuing the answers to these questions we developed a simple model of the latency of messages delivery via minimal-path routing in a C-wrapped hexagonal that uses wormhole switching. When compared against the delivery latency of messages using virtual cut-through switching we saw that there was no clear winner all of the time. This fact motivated further studies into how a system capable of supporting multiple classes of message traffic might behave. For the experimental system described earlier in this dissertation this type of routing and switching interaction would not be possible without the flexibility provided by the PRC.

With the results of the simulations we were able to conclude that the answer to both of the questions is "no."

Were the answer "yes" to either question the cost of providing the support necessary for multi-mode routing and switching would require careful justification. The "no" answers only partially vindicate multi-mode routing and switching since this experiment only examined a single pair of routing-swithing combination. The good news is that for a scheme that didn't specifically try to isolate the different classes of traffic from one another the interactions effects were not high. This is a very encouraging result since the PRC was easily able to support the functions required for this experiment.

It is also interesting to postulate on the reasons why the two types of traffic in the experiment didn't interact in a significant way. One avenue of thought is that it might be due to the fact that wormhole and virtual cut-through switching effectively uses different types of resources. Virtual cut-through use memory buffers on intermediate nodes in contrast to wormhole which stores its data in the actual network. A logical question that immediately follows is how specific are these experimental results to the wormhole/virtual cut-through switching pair. In the experiments performed thus far all the network resources have been available to all the classes of traffic. A series of experiments that is currently being pursued investigates different schemes in which the PRC can be used to isolated the different classes of traffic from one another. For example, a certain amount of bandwidth could be allocated to each class or in the case of the newer versions of the PRC certain classes of traffic can have a preferences for certain virtual channels. Here again, the flexibility of the PRC allows these types of issues to be explored.

The answers to our questions so far open up a whole new frontier of experiments concerning routing algorithms and switching methods that can be investigated. For example in the current experiments, once created the packets always use the same routing/switching scheme to be delivered to the destination. What if we allow routing packets using virtual cut-through switching initially and then change to wormhole as the packet approaches the destination? This might provide improved performance since wormhole packets perform quite well over a small number of hops when compared to virtual cut-through packets. But when should we change switching methods? These type of functions the PRC can easily provide.

What we are claiming is that having the flexibility to support multiple routing and switching schemes provides the system designer with the tools necessary to construct systems capable of supporting a wide variety of characteristics. The experiments presented in this chapter take a first step at showing that this type of flexibility requires further experiments.

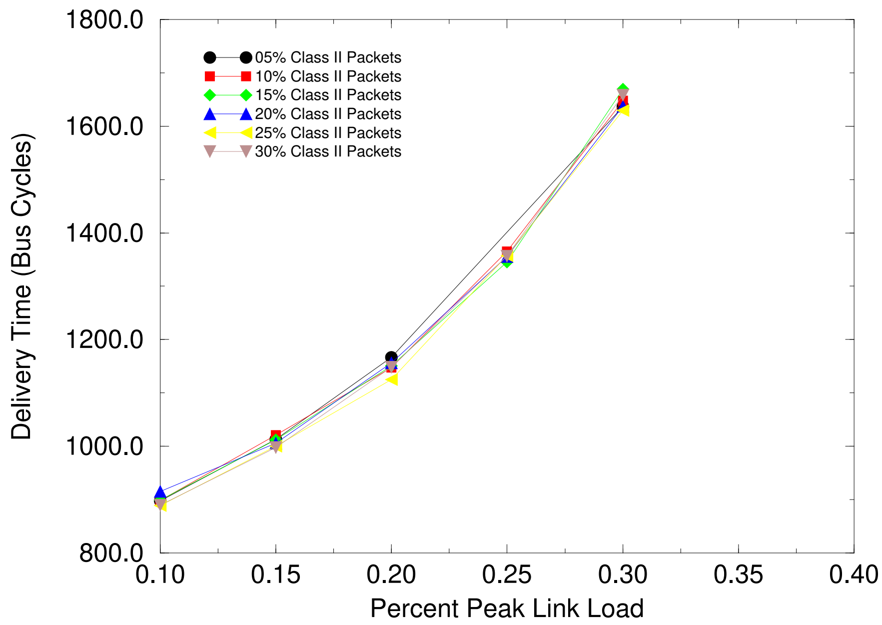


Figure 6.3: Delivery Latency Class I Traffic – 2 hops – H_5

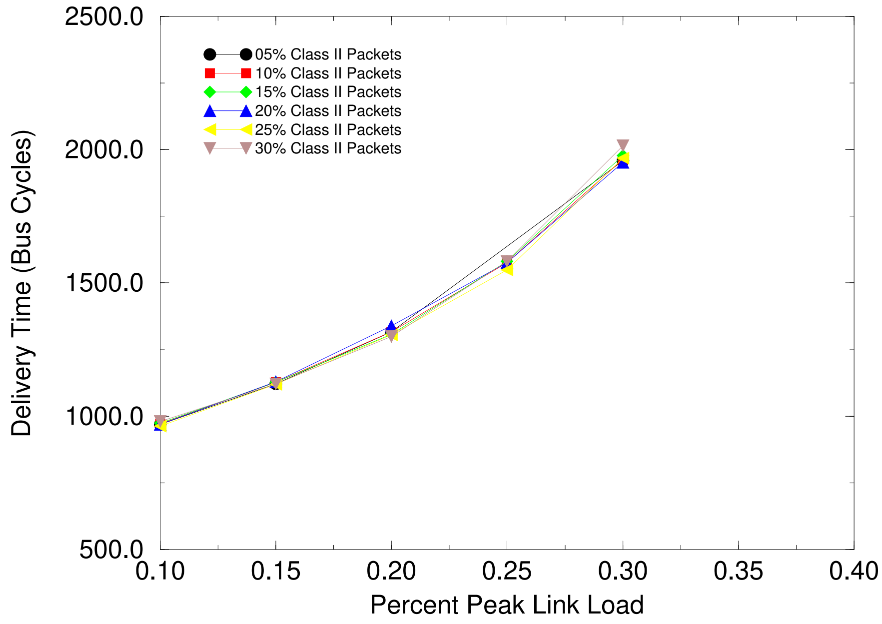


Figure 6.4: Delivery Latency Class I Traffic – 3 hops – H_5

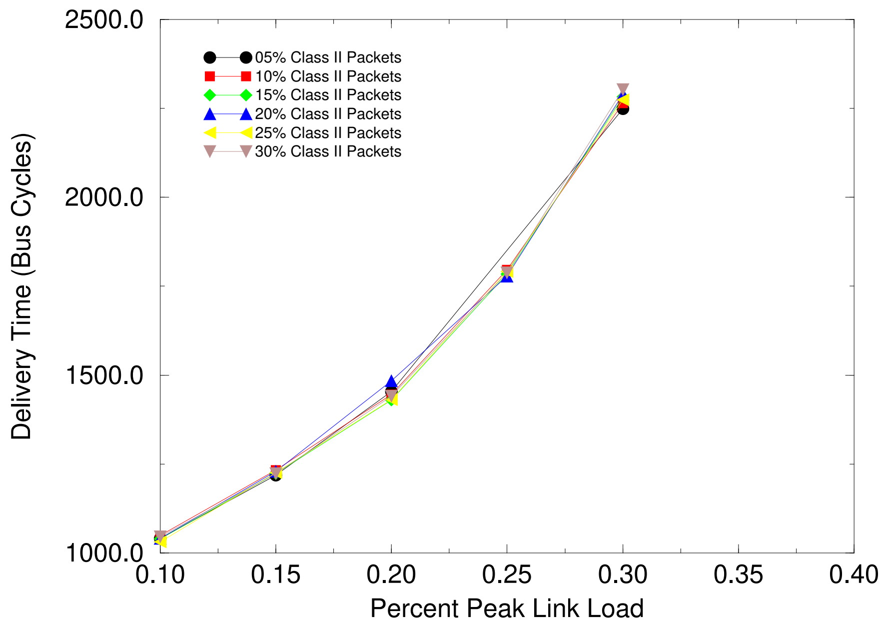


Figure 6.5: Delivery Latency Class I Traffic – 4 hops – H_5

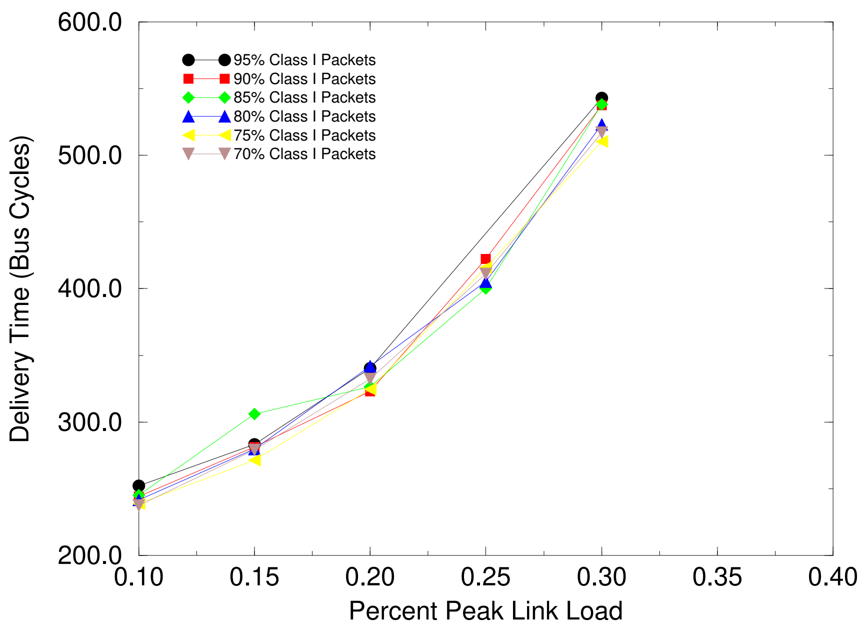


Figure 6.6: Delivery Latency Class II Traffic – 1 hop – H_5

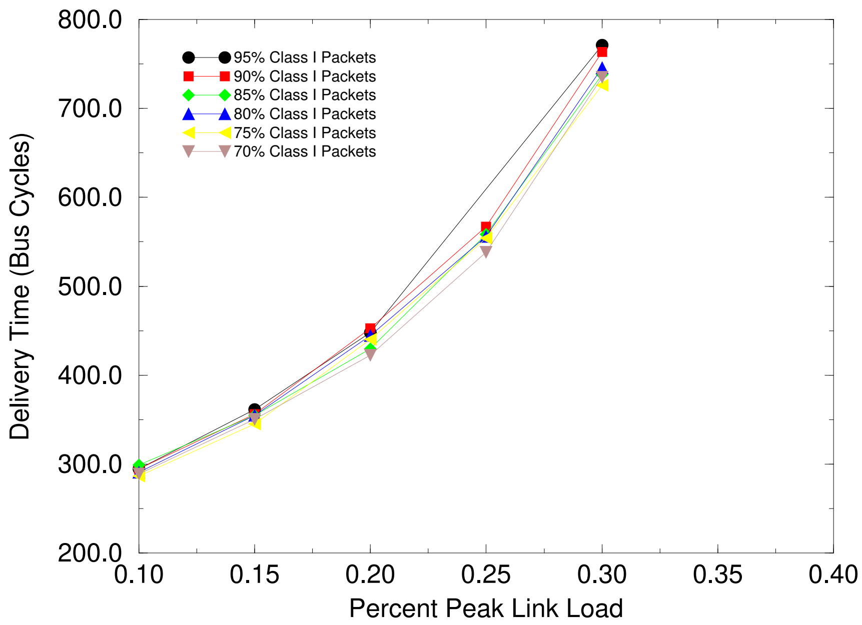


Figure 6.7: Delivery Latency Class II Traffic – 2 hops – H_5

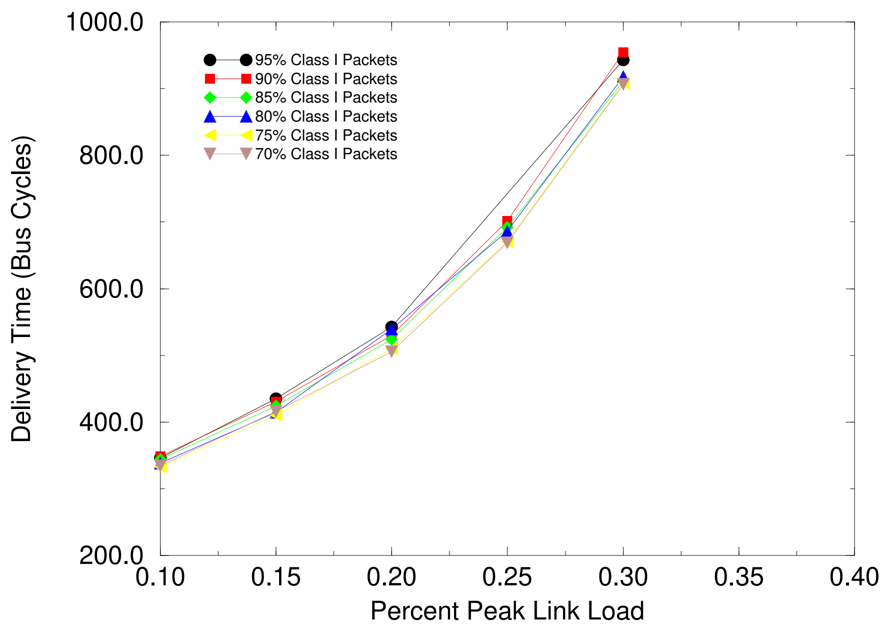


Figure 6.8: Delivery Latency Class II Traffic – 3 hops – H_5

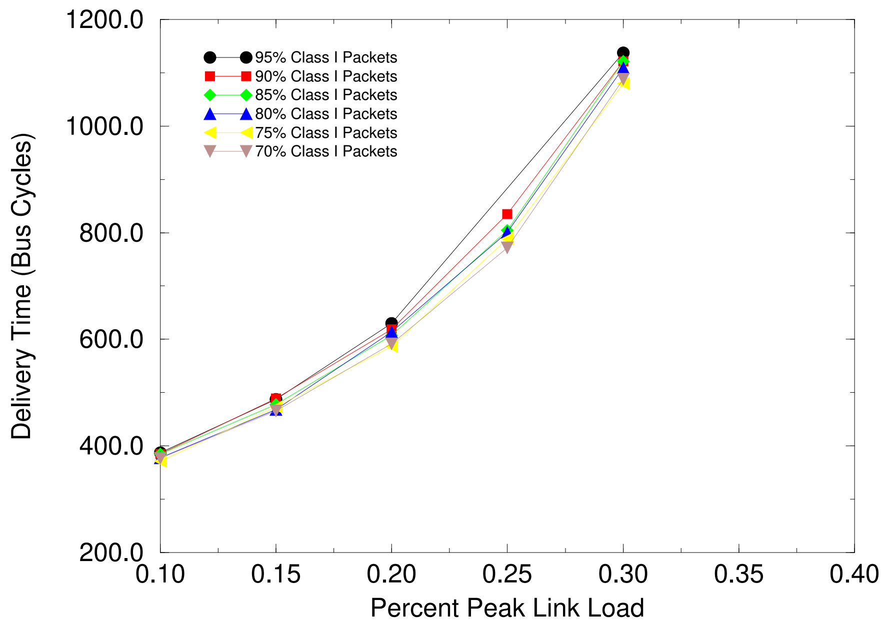


Figure 6.9: Delivery Latency Class II Traffic – 4 hops – H_5

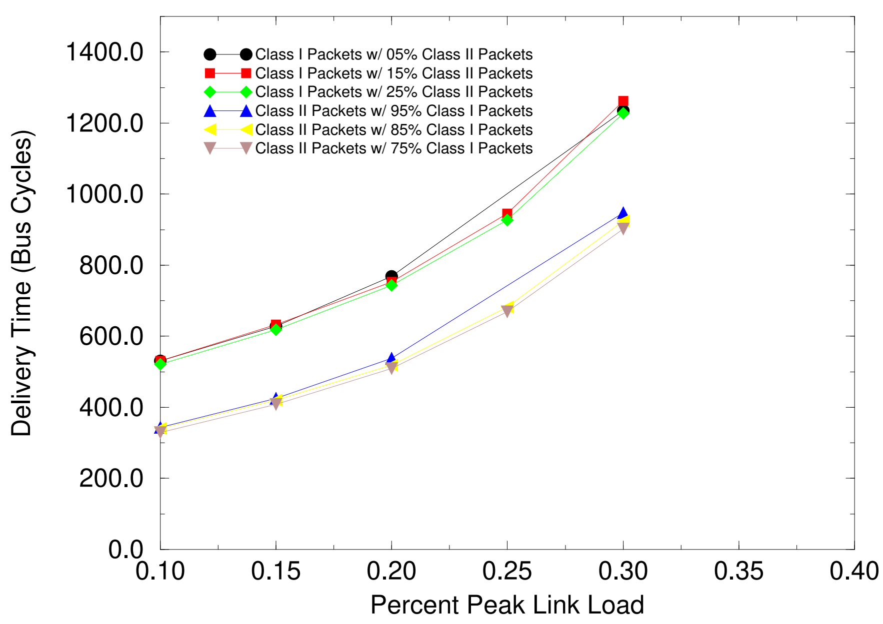


Figure 6.10: Comparison of Class I to Class II Traffic – Average hops – H_5

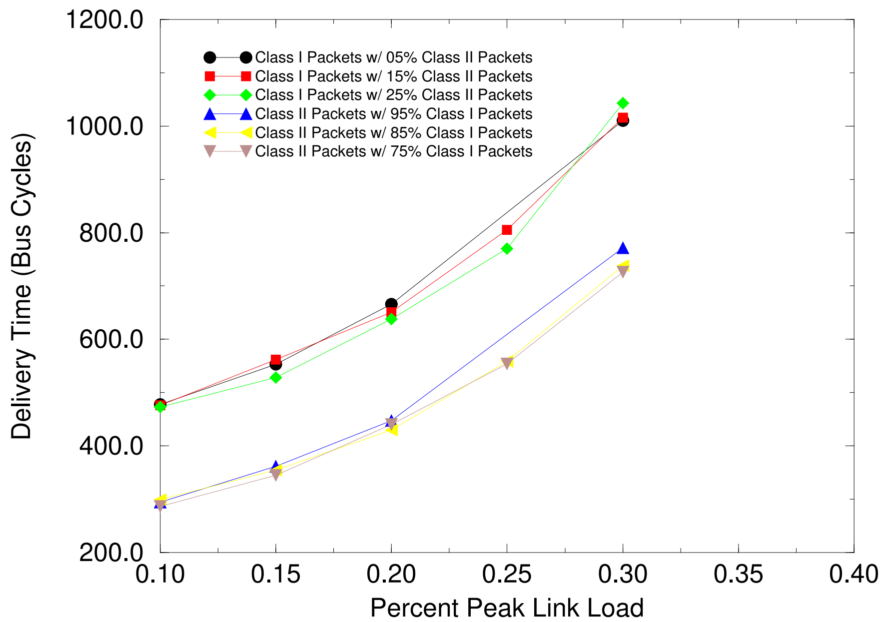


Figure 6.11: Comparison of Class I to Class II Traffic – 2 hops – H_5

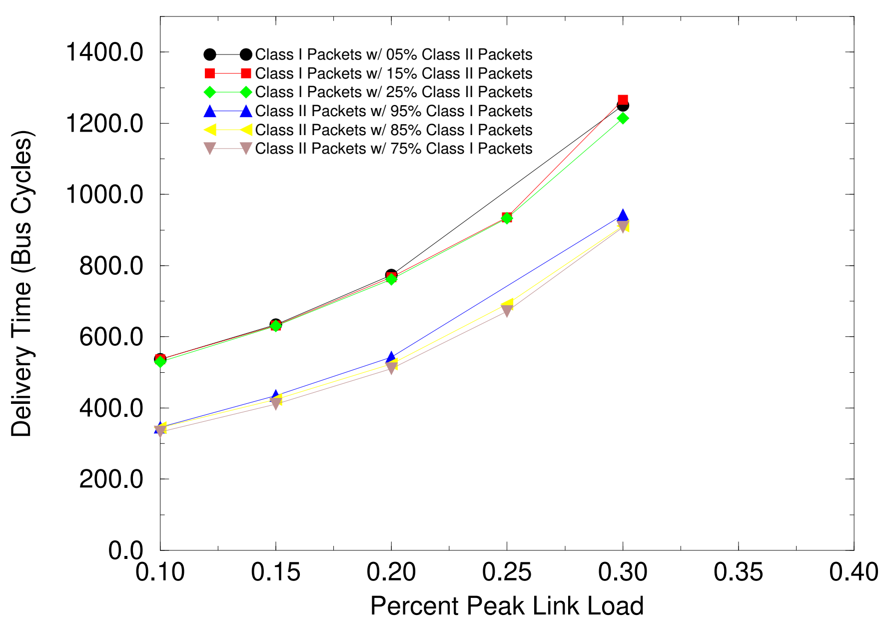


Figure 6.12: Comparison of Class I to Class II Traffic – 3 hops – H_5

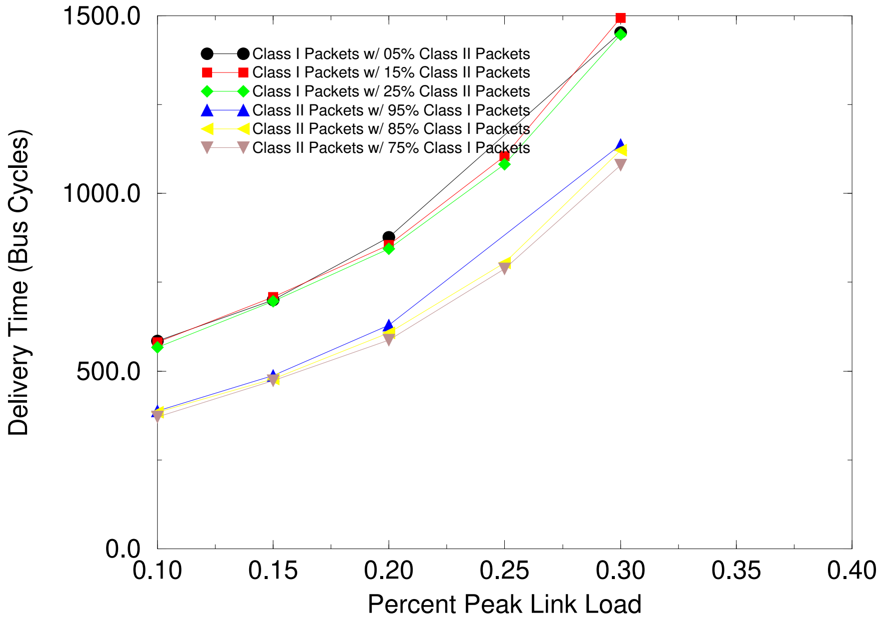


Figure 6.13: Comparison of Class I to Class II Traffic – 4 hops – H_5

CHAPTER 7

CONCLUSIONS AND FUTURE DIRECTIONS

In this chapter we review the contributions of this dissertation, and allude to the possible extensions and future research for the work presented.

7.1 Research Contributions

The recurring theme throughout this dissertation has been that flexibility in low-level routing hardware is a feature that the designer should promote. This is especially true for systems where the domain in which the routing hardware is operating may not have been fully characterized. This is certainly the case for the PRC, which is intended to operate somewhere in between parallel machines and networked computers, a region not previously investigated.

The PRC architecture offers several key contributions to the field. First, in no other architecture/design has the separation of the routing algorithms and switching schemes been made so explicit. This separation has many advantages over the more traditional schemes. Second, the identification of features important in reducing the overhead required to support real-time communication and their tradeoffs is key to understanding how future systems should be built. Last, the PRC itself can serve as a tool for investigating the hybrid domain of parallel computing and distributed systems. The PRC's ability to further explore this new domain will make a lasting impact in the field of distributed systems design.

The main contribution of the models developed in Chapter 4 is that the approach in obtaining the probability of cut-through can be extended to other partially connected point-to-point networks. It provides a more rigorous basis for justifying what previously had been derived intuitively.

The contributions of pp-mess-sim fall into three areas: the identification of the classes that enable topology independent modeling of routing hardware, the use of task and node specifications to easily generate custom communication workloads, and a reusable software tool in which plug-replaceable modules for different router hardware can be developed. The impact of the latter could be long lasting and greatly reduce the work required by others.

Chapter 6 provides evidence of the utility of both pp-mess-sim and the flexibility provided by the PRC. This initial set of experiments clearly demonstrates that the ability to dynamically vary the switching scheme independent of the routing algorithm deserves more investigation. The fact that the delivery performance was stable in the presence of other traffic is a significant finding.

7.2 Future Directions

We are currently working on three semi-independent research projects that are direct consequences of the work presented here: the enhancement of the PRC architecture to support 2 virtual channels per physical link, the addition of a virtual hardware router to pp-mess-sim, and the development of different variants of multi-mode routing and switching.

The need for having two virtual channels per link was quite clearly demonstrated while debugging the wormhole switching scheme during the development of pp-mess-sim. The addition of two virtual channels per physical link does not significantly alter the current architecture of the PRC. Both the PRC RX and TFU modules were modified during the last major design revisions to support the additional channels. For example, if you carefully examine the different transmitters for which the PRX RX can check the reservation status of, you will notice that there are twelve such possibilities. Until now the total size of the PRC was such that we have been able to avoid having to perform any manual intervention during the placement and route phases of compiling the PRC. This may no longer be true as we include six more TFUs and RXs.

The desire to develop a generic hardware router module for pp-mess-sim is motivated on several fronts. First, there are some effects of cut-through routing that can partially invalidate at higher loads some of assumptions necessary to make the models tractable. A router in which we don't have the overhead of byte-level flow-control would allow the cut-through effects to be measured. Second, it would allow the tradeoffs between on-chip memory and off-chip buffering to be evaluated in terms of the different routing and switching combinations.

The investigation of the different variants of multi-mode routing and switching is a direct result of just recently being enabled by the availability of pp-mess-sim and high performance workstations on which we can run pp-mess-sim.

In addition to the research efforts already being pursued there are several issues concerning the representativeness of the existing analytic models and extensions of the models beg for investigation. These questions center around what interactions are being lost in the models of the switching methods and how to capture the changes in switching and routing behavior.

APPENDIX A

PRC CTBUS IMPLEMENTATION

Cut-Through Bus Device Arbitration Order																			
Priority Cycle	← Highest							Device ID Priority Order											Lowest →
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	0	2	1	4	3	6	5	8	7	10	9	12	11	14	13	16	15	18	17
2	1	2	0	5	6	3	4	9	10	7	8	13	14	11	12	17	18	15	16
3	2	1	0	6	5	4	3	10	9	8	7	14	13	12	11	18	17	16	15
4	3	4	5	6	0	1	2	11	12	13	14	7	8	9	10	15	16	17	18
5	4	3	6	5	0	2	1	12	11	14	13	8	7	10	9	16	15	18	17
6	5	6	3	4	1	2	0	13	14	11	12	9	10	7	8	17	18	15	16
7	6	5	4	3	2	1	0	14	13	12	11	10	9	8	7	18	17	16	15
8	7	8	9	10	11	12	13	14	0	1	2	3	4	5	6	15	16	17	18
9	8	7	10	9	12	11	14	13	0	2	1	4	3	6	5	16	15	18	17
10	9	10	7	8	13	14	11	12	1	2	0	5	6	3	4	17	18	15	16
11	10	9	8	7	14	13	12	11	2	1	0	6	5	4	3	18	17	16	15
12	11	12	13	14	7	8	9	10	3	4	5	6	0	1	2	15	16	17	18
13	12	11	14	13	8	7	10	9	4	3	6	5	0	2	1	16	15	18	17
14	13	14	11	12	9	10	7	8	5	6	3	4	1	2	0	17	18	15	16
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	18	17	16	15
16	15	16	17	18	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
17	16	15	18	17	0	2	1	4	3	6	5	8	7	10	9	12	11	14	13
18	17	18	15	16	1	2	0	5	6	3	4	9	10	7	8	13	14	11	12
19	18	17	16	15	2	1	0	6	5	4	3	10	9	8	7	14	13	12	11
20	15	16	17	18	3	4	5	6	0	1	2	11	12	13	14	7	8	9	10
21	16	15	18	17	4	3	6	5	0	2	1	12	11	14	13	8	7	10	9
22	17	18	15	16	5	6	3	4	1	2	0	13	14	11	12	9	10	7	8
23	18	17	16	15	6	5	4	3	2	1	0	14	13	12	11	10	9	8	7
24	15	16	17	18	7	8	9	10	11	12	13	14	0	1	2	3	4	5	6
25	16	15	18	17	8	7	10	9	12	11	14	13	0	2	1	4	3	6	5
26	17	18	15	16	9	10	7	8	13	14	11	12	1	2	0	5	6	3	4
27	18	17	16	15	10	9	8	7	14	13	12	11	2	1	0	6	5	4	3
28	15	16	17	18	11	12	13	14	7	8	9	10	3	4	5	6	0	1	2
29	16	15	18	17	12	11	14	13	8	7	10	9	4	3	6	5	0	2	1
30	17	18	15	16	13	14	11	12	9	10	7	8	5	6	3	4	1	2	0
31	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Table A.1: Cut-Through Device Arbitration Order

Cut-Through Bus Command Encoding		
External	Internal	
CTCTL[3:0]	CTCTL[3:0]	Command
0 0 0 0	1 1 1 1	NOOP (No Operation)
0 0 0 1	1 1 1 0	DTX (Data Transfer)
0 0 1 0	1 1 0 1	DACK (Data Positive Acknowledgement)
0 0 1 1	1 1 0 0	DNACK (Data Negative Acknowledgement)
0 1 0 0	1 0 1 1	DRTRY (Data Transfer Retry)
0 1 0 1	1 0 1 0	RESV (Transmitter Reservation Request)
0 1 1 0	1 0 0 1	FREE (Transmitter Free Request)
0 1 1 1	1 0 0 0	HOLD (Transmitter Hold Request)
1 0 0 0	0 1 1 1	CHK (Transmiiter Hold Check)
1 0 0 1	0 1 1 0	CFG0 (Load TAXI Receiver Reg 0)
1 0 1 0	0 1 0 1	CFG1 (Load TAXI Receiver Reg 1)
1 0 1 1	0 1 0 0	CFG2 (Load TAXI Receiver Reg 2)
1 1 0 0	0 0 1 1	CFG3 (Load TAXI Receiver Reg 3)
1 1 0 1	0 0 1 0	CFG4 (Load TAXI Receiver Reg 4)
1 1 1 0	0 0 0 1	MARK (Byte Marker)
1 1 1 1	0 0 0 0	EOP (End of Packet)

Table A.2: Cut-Through Bus Command Encoding

Cut-Through Master Device Encoding					
External	Internal	Internal	Internal		
MST[2:0]	TAXIGRNT[5:0]	RXGRNT[5:0]	TFUGRNT[5:0]	Device	Name
0 0 0	0 0 0 0 0 1	0 0 0 0 0 0	0 0 0 0 0 0	0	TAXI RX_0
0 0 1	0 0 0 0 1 0	0 0 0 0 0 0	0 0 0 0 0 0	1	TAXI RX_1
0 1 0	0 0 0 1 0 0	0 0 0 0 0 0	0 0 0 0 0 0	2	TAXI RX_2
0 1 1	0 0 1 0 0 0	0 0 0 0 0 0	0 0 0 0 0 0	3	TAXI RX_3
1 0 0	0 1 0 0 0 0	0 0 0 0 0 0	0 0 0 0 0 0	4	TAXI RX_4
1 0 1	1 0 0 0 0 0	0 0 0 0 0 0	0 0 0 0 0 0	5	TAXI RX_5
1 1 1	0 0 0 0 0 0	0 0 0 0 0 1	0 0 0 0 0 0	6	PRC RX_0
1 1 1	0 0 0 0 0 0	0 0 0 0 1 0	0 0 0 0 0 0	7	PRC RX_1
1 1 1	0 0 0 0 0 0	0 0 0 1 0 0	0 0 0 0 0 0	8	PRC RX_2
1 1 1	0 0 0 0 0 0	0 0 1 0 0 0	0 0 0 0 0 0	9	PRC RX_3
1 1 1	0 0 0 0 0 0	0 1 0 0 0 0	0 0 0 0 0 0	10	PRC RX_4
1 1 1	0 0 0 0 0 0	1 0 0 0 0 0	0 0 0 0 0 0	11	PRC RX_5
1 1 1	0 0 0 0 0 0	0 0 0 0 0 0	0 0 0 0 0 1	12	PRC TFU_0
1 1 1	0 0 0 0 0 0	0 0 0 0 0 0	0 0 0 0 1 0	13	PRC TFU_1
1 1 1	0 0 0 0 0 0	0 0 0 0 0 0	0 0 0 1 0 0	14	PRC TFU_2
1 1 1	0 0 0 0 0 0	0 0 0 0 0 0	0 0 1 0 0 0	15	PRC TFU_3
1 1 1	0 0 0 0 0 0	0 0 0 0 0 0	0 1 0 0 0 0	16	PRC TFU_4
1 1 1	0 0 0 0 0 0	0 0 0 0 0 0	1 0 0 0 0 0	17	PRC TFU_5
1 1 0	0 0 0 0 0 0	0 0 0 0 0 0	0 0 0 0 0 0	18	IMU

Table A.3: Cut-Through Master Device Encoding

Cut-Through Addressed Slave Device Encoding				
External	External	Internal	Internal	
CTMST[2:0]	CTADDR[7:0]	CTMST[4:0]	CTADDR[7:0]	Device
x x x	1 x x x x x x 1	x x x	0 x x x x x x 0	TAXI TX_0
x x x	1 x x x x x 1 x	x x x	0 x x x x x 0 x	TAXI TX_1
x x x	1 x x x x 1 x x	x x x	0 x x x x 0 x x	TAXI TX_2
x x x	1 x x x 1 x x x	x x x	0 x x x 0 x x x	TAXI TX_3
x x x	1 x x 1 x x x x	x x x	0 x x 0 x x x x	TAXI TX_4
x x x	1 x 1 x x x x x	x x x	0 x 0 x x x x x	TAXI TX_5
0 0 0	x 1 x x x x x x	1 1 1	x 0 x x x x x x	PRC RX_0
0 0 1	x 1 x x x x x x	1 1 0	x 0 x x x x x x	PRC RX_1
0 1 0	x 1 x x x x x x	1 0 1	x 0 x x x x x x	PRC RX_2
0 1 1	x 1 x x x x x x	1 0 0	x 0 x x x x x x	PRC RX_3
1 0 0	x 1 x x x x x x	0 1 1	x 0 x x x x x x	PRC RX_4
1 0 1	x 1 x x x x x x	0 1 0	x 0 x x x x x x	PRC RX_5
1 1 1	x 1 x x x x x 1	0 0 0	x 0 x x x x x 0	TAXI RX_0
1 1 1	x 1 x x x x 1 x	0 0 0	x 0 x x x x 0 x	TAXI RX_1
1 1 1	x 1 x x x 1 x x	0 0 0	x 0 x x x 0 x x	TAXI RX_2
1 1 1	x 1 x x 1 x x x	0 0 0	x 0 x x 0 x x x	TAXI RX_3
1 1 1	x 1 x 1 x x x x	0 0 0	x 0 x 0 x x x x	TAXI RX_4
1 1 1	x 1 1 x x x x x	0 0 0	x 0 0 x x x x x	TAXI RX_5

Table A.4: Cut-Through Addressed Slave Device Encoding

Cut-Through Bus Signal Summary			
Pins	Type	Signal Names	Description
6	Input	CTREQ[5:0]	TAXI Receiver Bus Request Lines
0		CTREQ[11:0]	PRC Receiver Bus Request Lines
0		CTREQ[17:12]	PRC TFU Request Lines
1	Input	CTREQ[18]	IMU Request Line
3	Output	CTMST[2:0]	Bus Master Grant Status
1	Input	CTEXT	Bus Cycle Extend
1	BiDir	CTDSTRB	Data Strobe
1	BiDir	CTACK	Command Ack
4	BiDir	CTCTL[3:0]	Command Sub-Bus
8	BiDir	CTADDR[7:0]	Slave Address Sub-Bus
8	BiDir	CTDATA[7:0]	Data Sub-Bus

Table A.5: Cut-Through Bus Signal Summary

APPENDIX B

PRC RX INTERNALS

This appendix provides in depth information concerning the implementation of the PRC RX module. This currently consists of the instruction encodings, sample instruction timing diagrams, and some general figures used in the design of the RX.

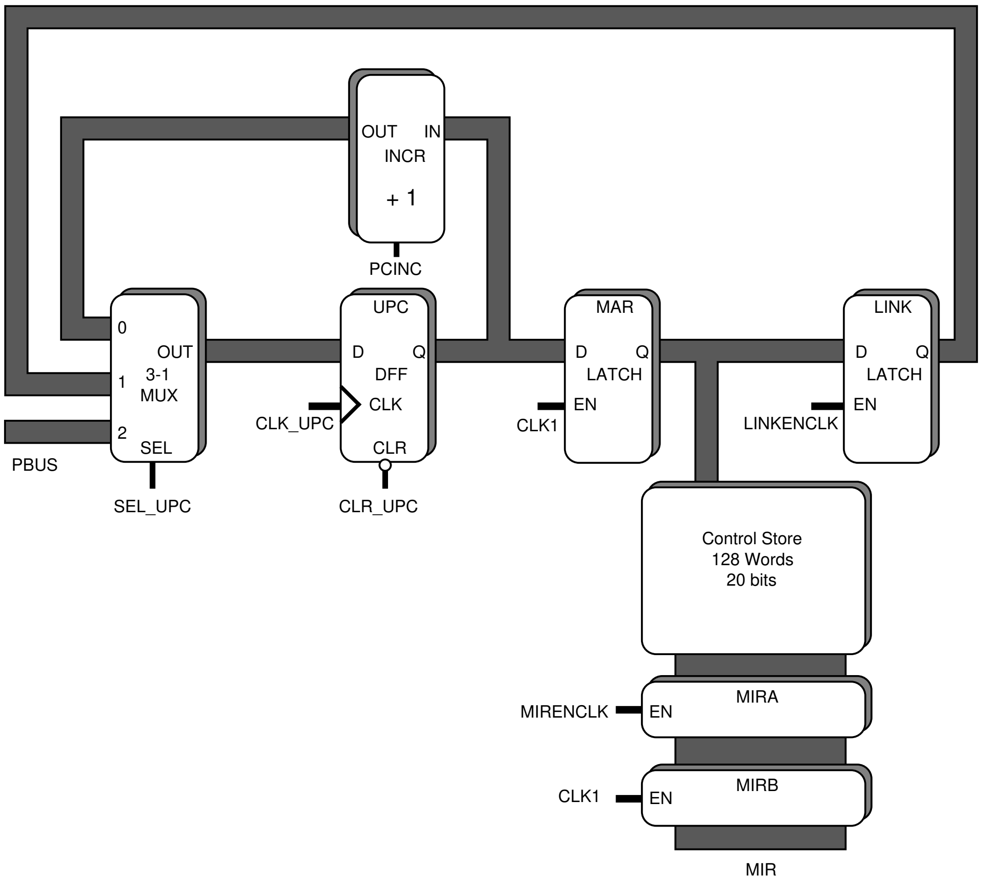


Figure B.1: Partial structure of the address sequencing hardware of the RX

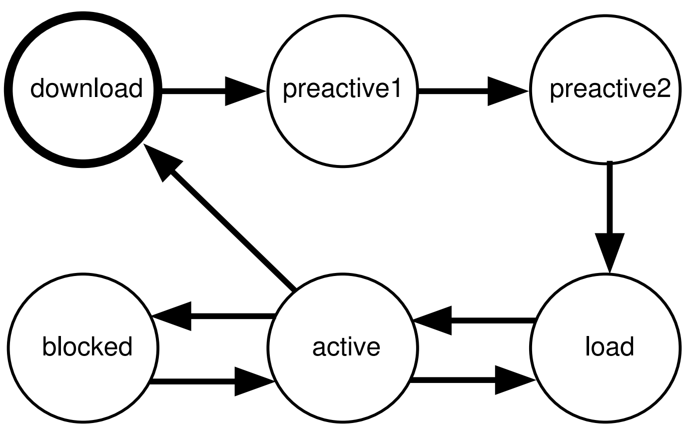


Figure B.2: Simplified state diagram for microsequencer controller

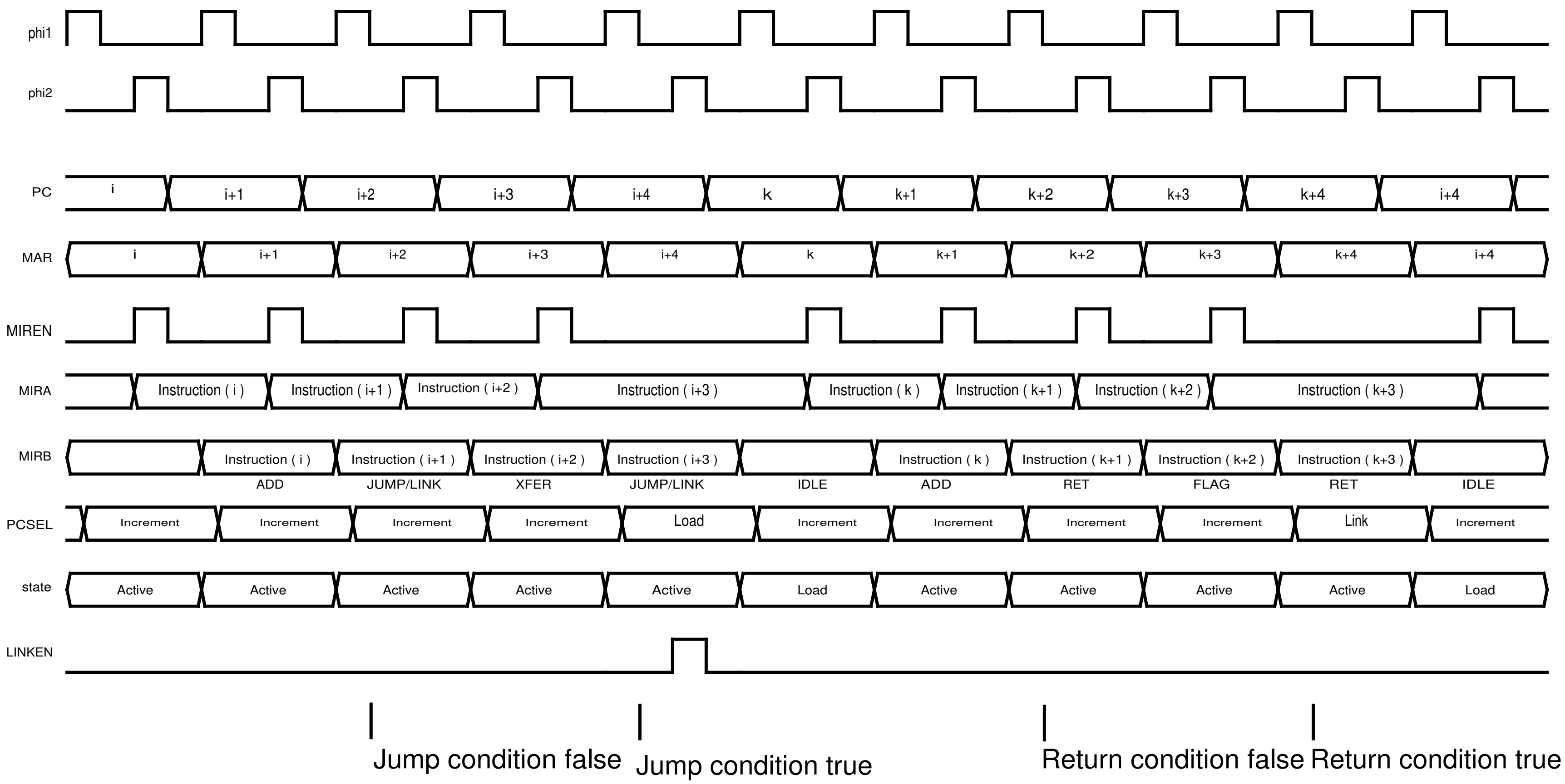


Figure B.3: Instruction sequence example #1

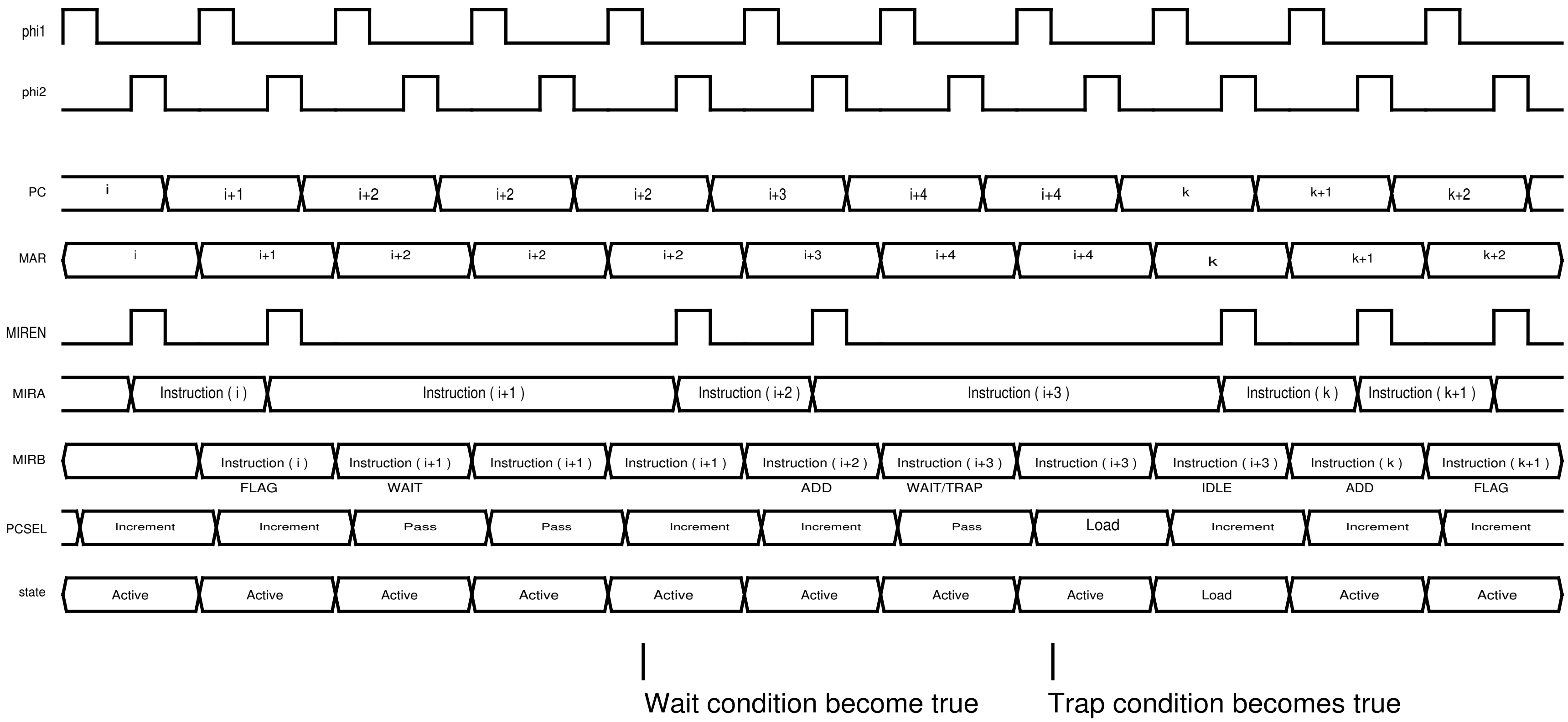


Figure B.4: Instruction sequence example #2

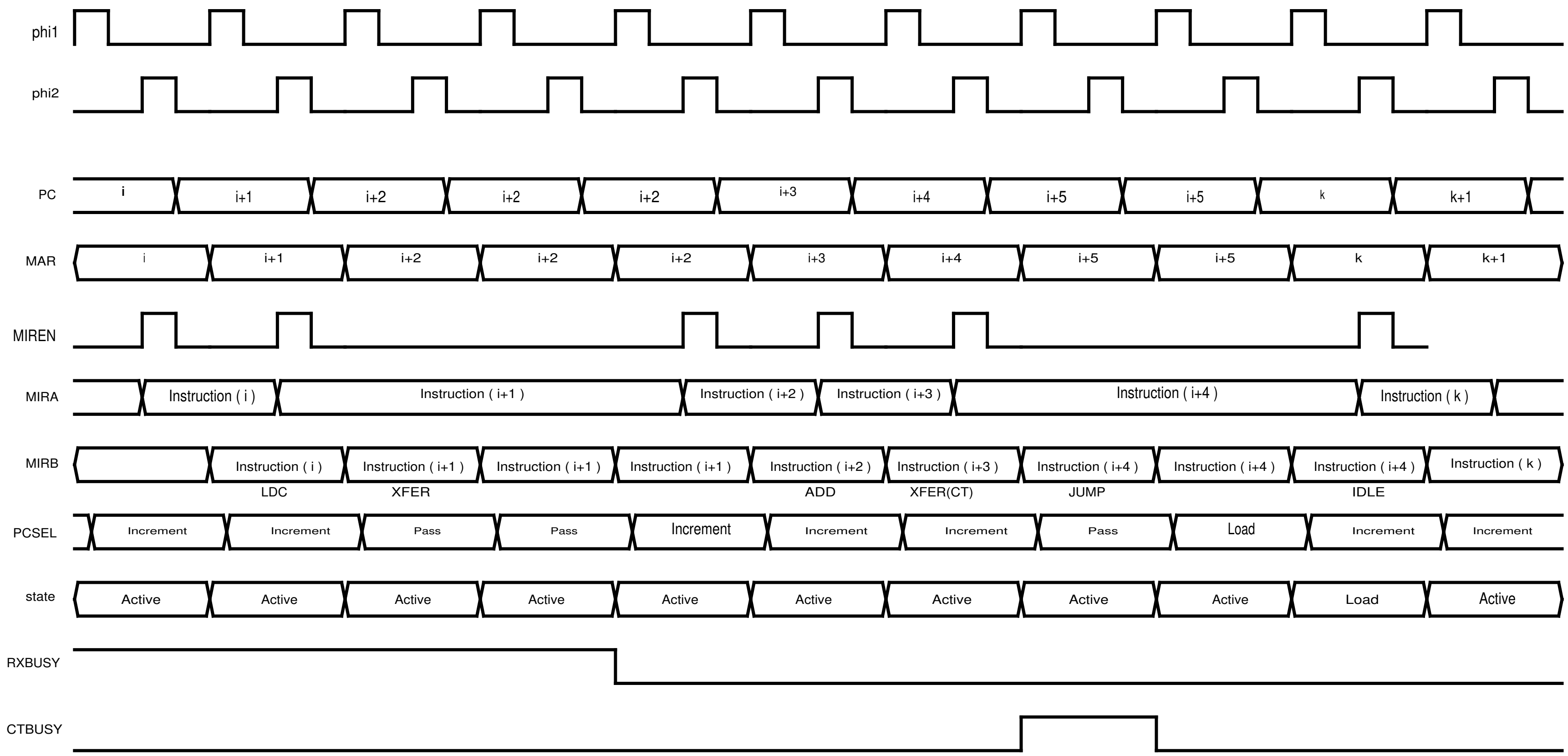


Figure B.5: Instruction sequence example #3

ALU Operations																			
M(19)	M(18)	M(17)	M(16)	M(15)	M(14)	M(13)	M(12)	M(11)	M(10)	M(9)	M(8)	M(7)	M(6)	M(5)	M(4)	M(3)	M(2)	M(1)	M(0)
1	1	0	Carry Control		A Port Select					B Port Select					ALU Function Control				

Carry Control		
M(16)	M(15)	ALU Carry Input
0	0	False (0)
0	1	Carry Flag
1	0	Zero Flag
1	1	True (1)

ALU Function Encodings						
				M(4) = 0	M(4) = 1; Arithmetic Operations	
M(3)	M(2)	M(1)	M(0)	Logic Functions	carryin = 0 (no carry)	carryin = 1 (with carry)
0	0	0	0	$\overline{A}$	A minus 1	A
0	0	0	1	$\overline{AB}$	AB minus 1	AB
0	0	1	0	$\overline{A} + B$	AB minus 1	A$\overline{B}$
0	0	1	1	1	minus 1 (2's comp)	zero
0	1	0	0	$\overline{A + B}$	A plus (A + $\overline{B}$)	A plus (A + $\overline{B}$) plus 1
0	1	0	1	$\overline{B}$	AB plus (A + $\overline{B}$)	AB plus (A + $\overline{B}$) plus 1
0	1	1	0	$\overline{A \oplus B}$	A minus B minus 1	A minus B
0	1	1	1	$A + \overline{B}$	A + $\overline{B}$	(A + $\overline{B}$) plus 1
1	0	0	0	$\overline{A}B$	A plus (A + B)	A plus (A + B) plus 1
1	0	0	1	$A \oplus B$	A plus B	A plus B plus 1
1	0	1	0	B	A$\overline{B}$ plus (A + B)	A$\overline{B}$ plus (A + B) plus 1
1	0	1	1	A + B	(A + B)	(A + B) plus 1
1	1	0	0	0	A plus A	A plus A plus 1
1	1	0	1	$A\overline{B}$	AB plus A	AB plus A plus 1
1	1	1	0	AB	A$\overline{B}$ plus A	A$\overline{B}$ plus A plus 1
1	1	1	1	A	A	A plus 1

Table B.1: ALU Instruction Encoding

ALU Port A Select					
M(9)	M(8)	M(7)	M(6)	M(5)	Device Selected
0	0	0	0	0	Reg 0 (RF00)
0	0	0	0	1	Reg 1 (RF01)
0	0	0	1	0	Reg 2 (RF02)
0	0	0	1	1	Reg 3 (RF03)
0	0	1	0	0	Reg 4 (RF04)
0	0	1	0	1	Reg 5 (RF05)
0	0	1	1	0	Reg 6 (RF06)
0	0	1	1	1	Reg 7 (RF07)
0	1	0	0	0	Reg 8 (RF08)
0	1	0	0	1	Reg 9 (RF09)
0	1	0	1	0	Reg 10 (RF10)
0	1	0	1	1	Reg 11 (RF11)
0	1	1	0	0	Reg 12 (RF12)
0	1	1	0	1	Reg 13 (RF13)
0	1	1	1	0	Trap Reg 0 (TRAP0)
0	1	1	1	1	Trap Reg 1 (TRAP1)
1	0	1	0	1	CTBus Data (CTDATA)
1	1	0	0	1	Memory Data Reg (MDR)
1	1	0	1	1	Header Fifo (FIFO)
1	1	1	1	0	Accumlator (ACC)

ALU Port B Select				
M(13)	M(12)	M(11)	M(10)	Device Selected
0	0	0	0	Reg 0 (RF00)
0	0	0	1	Reg 1 (RF01)
0	0	1	0	Reg 2 (RF02)
0	0	1	1	Reg 3 (RF03)
0	1	0	0	Reg 4 (RF04)
0	1	0	1	Reg 5 (RF05)
0	1	1	0	Reg 6 (RF06)
0	1	1	1	Reg 7 (RF07)
1	0	0	0	Reg 8 (RF08)
1	0	0	1	Reg 9 (RF09)
1	0	1	0	Reg 10 (RF10)
1	0	1	1	Reg 11 (RF11)
1	1	0	0	Reg 12 (RF12)
1	1	0	1	Reg 13 (RF13)
1	1	1	0	Trap Reg 0 (TRAP0)
1	1	1	1	Trap Reg 1 (TRAP1)

Table B.2: ALU Operand Selection Coding

Load Constant																			
M(19)	M(18)	M(17)	M(16)	M(15)	M(14)	M(13)	M(12)	M(11)	M(10)	M(9)	M(8)	M(7)	M(6)	M(5)	M(4)	M(3)	M(2)	M(1)	M(0)
1	0	1	RX	CT	Destination Select						Immediate Data								IR
			0	Do not trigger RXBUS interface															
			1	Trigger RXBUS interface															
				0	Do not trigger CTBUS interface														
				1	Trigger CTBUS interface														
																		Do not Set CTBUS IR	0
																		Set CTBUS IR	1

Transfer Operation																			
M(19)	M(18)	M(17)	M(16)	M(15)	M(14)	M(13)	M(12)	M(11)	M(10)	M(9)	M(8)	M(7)	M(6)	M(5)	M(4)	M(3)	M(2)	M(1)	M(0)
1	0	0	RX	CT	Destination Select					Source Select					x	x	x	x	IR
			0	Do not trigger RXBUS interface															
			1	Trigger RXBUS interface															
				0	Do not trigger CTBUS interface														
				1	Trigger CTBUS interface														
																		Do not Set CTBUS IR	0
																		Set CTBUS IR	1

Table B.3: Transfer Instructions

Source Select Field					
M(9)	M(8)	M(7)	M(6)	M(5)	Device Selected
0	0	0	0	0	Reg 0 (RF00)
0	0	0	0	1	Reg 1 (RF01)
0	0	0	1	0	Reg 2 (RF02)
0	0	0	1	1	Reg 3 (RF03)
0	0	1	0	0	Reg 4 (RF04)
0	0	1	0	1	Reg 5 (RF05)
0	0	1	1	0	Reg 6 (RF06)
0	0	1	1	1	Reg 7 (RF07)
0	1	0	0	0	Reg 8 (RF08)
0	1	0	0	1	Reg 9 (RF09)
0	1	0	1	0	Reg 10 (RF10)
0	1	0	1	1	Reg 11 (RF11)
0	1	1	0	0	Reg 12 (RF12)
0	1	1	0	1	Reg 13 (RF13)
0	1	1	1	0	Trap Reg 0 (TRAP0)
0	1	1	1	1	Trap Reg 1 (TRAP1)
1	0	1	0	1	CTBus Data (CTDATA)
1	1	0	0	1	Memory Data Reg (MDR)
1	1	0	1	1	Header Fifo (FIFO)
1	1	1	1	0	Accumlator (ACC)

Destination Select Field					
M(14)	M(13)	M(12)	M(11)	M(10)	Device Selected
0	0	0	0	0	Reg 0 (RF00)
0	0	0	0	1	Reg 1 (RF01)
0	0	0	1	0	Reg 2 (RF02)
0	0	0	1	1	Reg 3 (RF03)
0	0	1	0	0	Reg 4 (RF04)
0	0	1	0	1	Reg 5 (RF05)
0	0	1	1	0	Reg 6 (RF06)
0	0	1	1	1	Reg 7 (RF07)
0	1	0	0	0	Reg 8 (RF08)
0	1	0	0	1	Reg 9 (RF09)
0	1	0	1	0	Reg 10 (RF10)
0	1	0	1	1	Reg 11 (RF11)
0	1	1	0	0	Reg 12 (RF12)
0	1	1	0	1	Reg 13 (RF13)
0	1	1	1	0	Trap Reg 0 (TRAP0)
0	1	1	1	1	Trap Reg 1 (TRAP1)
1	0	0	0	0	RX Data Reg 0 (RXD0)
1	0	0	0	1	RX Data Reg 1 (RXD1)
1	0	0	1	0	RX Data Reg 2 (RXD2)
1	0	0	1	1	RX Data Reg 3 (RXD3)
1	0	1	0	0	RX Control Reg (RXCTL)
1	0	1	0	1	CTBUS Data Reg 0 (CTD0)
1	0	1	1	0	CTBUS Address Reg (CTADDR0)
1	0	1	1	1	CTBUS Address Reg (CTADDR1)
1	1	0	0	0	CTBUS Control Reg (CTCTL)
1	1	0	0	1	Memory Data Reg (MDR)
1	1	0	1	0	Memory Address Reg (MAR)
1	1	0	1	1	Header Fifo (FIFO)

Table B.4: Source and Destination Operand Coding

Set Flag Operation																			
M(19)	M(18)	M(17)	M(16)	M(15)	M(14)	M(13)	M(12)	M(11)	M(10)	M(9)	M(8)	M(7)	M(6)	M(5)	M(4)	M(3)	M(2)	M(1)	M(0)
1	1	1	Data Select		x	x	Flag Mask												

Flag Select Mask													
M(12)	M(11)	M(10)	M(9)	M(8)	M(7)	M(6)	M(5)	M(4)	M(3)	M(2)	M(1)	M(0)	Flag Modified
x	x	x	x	x	x	x	x	x	x	x	x	1	User Flag 0
x	x	x	x	x	x	x	x	x	x	x	1	x	User Flag 1
x	x	x	x	x	x	x	x	x	x	1	x	x	User Flag 2
x	x	x	x	x	x	x	x	x	1	x	x	x	User Flag 3
x	x	x	x	x	x	x	x	1	x	x	x	x	User Flag 4
x	x	x	x	x	x	x	1	x	x	x	x	x	User Flag 5
x	x	x	x	x	x	1	x	x	x	x	x	x	Eophold Flag
x	x	x	x	x	1	x	x	x	x	x	x	x	Markhold Flag
x	x	x	x	1	x	x	x	x	x	x	x	x	Aborthold Flag
x	x	x	1	x	x	x	x	x	x	x	x	x	Set IR (Set always)
x	x	1	x	x	x	x	x	x	x	x	x	x	Clear Mark (Clear always)
x	1	x	x	x	x	x	x	x	x	x	x	x	Clear Eop (Clear always)
1	x	x	x	x	x	x	x	x	x	x	x	x	Clear Abort (Clear always)

Data Select		
M(16)	M(15)	Flag Input
0	0	False (0)
0	1	Carry Flag
1	0	Zero Flag
1	1	True (1)

Table B.5: Flag Manipulation Instruction Encoding

Wait Operation																			
M(19)	M(18)	M(17)	M(16)	M(15)	M(14)	M(13)	M(12)	M(11)	M(10)	M(9)	M(8)	M(7)	M(6)	M(5)	M(4)	M(3)	M(2)	M(1)	M(0)
0	0	Mark	Condition Select								Trap 0 Mask					Trap 1 Mask			Link Enb
		0	Do not fall through on MARK																
		1	Fall through on MARK																
																		Do not store return Link	0
																		Store return Link	1

Trap 0 Select Mask					
M(8)	M(7)	M(6)	M(5)	M(4)	Trap Condition
x	x	x	x	1	$\overline{\text{CT Interface Busy}}$
x	x	x	1	x	$\overline{\text{RX Interface Busy}}$
x	x	1	x	x	Mark Flag
x	1	x	x	x	Eop Flag
1	x	x	x	x	Abort Flag

Trap 1 Select Mask			
M(3)	M(2)	M(1)	Trap Condition
x	x	1	Mark Flag
x	1	x	Eop Flag
1	x	x	Abort Flag

Table B.6: Wait Instruction Encoding

Jump Operation																			
M(19)	M(18)	M(17)	M(16)	M(15)	M(14)	M(13)	M(12)	M(11)	M(10)	M(9)	M(8)	M(7)	M(6)	M(5)	M(4)	M(3)	M(2)	M(1)	M(0)
0	1	0	Condition Select								Target Address								Link Enb
Do not store return Link																			0
Store return Link																			1

Return Operation																			
M(19)	M(18)	M(17)	M(16)	M(15)	M(14)	M(13)	M(12)	M(11)	M(10)	M(9)	M(8)	M(7)	M(6)	M(5)	M(4)	M(3)	M(2)	M(1)	M(0)
1	0	1	Condition Select								x	x	x	x	x	x	x	x	0

Table B.7: Jump and Return Instruction Encoding

Condition Select Part I								
M(16)	M(15)	M(14)	M(13)	M(12)	M(11)	M(10)	M(9)	Condition Selected
x	x	0	0	0	0	0	0	Resv Stat 0
x	x	0	0	0	0	0	1	Resv Stat 1
x	x	0	0	0	0	1	0	Resv Stat 2
x	x	0	0	0	0	1	1	Resv Stat 3
x	x	0	0	0	1	0	0	Resv Stat 4
x	x	0	0	0	1	0	1	Resv Stat 5
x	x	0	0	0	1	1	0	Resv Stat 6
x	x	0	0	0	1	1	1	Resv Stat 7
x	x	0	0	1	0	0	0	Resv Stat 8
x	x	0	0	1	0	0	1	Resv Stat 9
x	x	0	0	1	0	1	0	Resv Stat 10
x	x	0	0	1	0	1	1	Resv Stat 11
x	x	0	0	1	1	0	0	Timer Bit
x	x	0	0	1	1	0	1	Abort
x	x	0	0	1	1	1	0	Aborthold
x	x	0	0	1	1	1	1	Random Bit
x	x	0	1	0	0	0	0	User Flag 0
x	x	0	1	0	0	0	1	User Flag 1
x	x	0	1	0	0	1	0	User Flag 2
x	x	0	1	0	0	1	1	User Flag 3
x	x	0	1	0	1	0	0	User Flag 4
x	x	0	1	0	1	0	1	User Flag 5
x	x	0	1	0	1	1	0	Header FIFO OR
x	x	0	1	0	1	1	1	Mark Flag
x	x	0	1	1	0	0	0	Eop Flag
x	x	0	1	1	0	0	1	Data Flag
x	x	0	1	1	0	1	0	Ack Flag
x	x	0	1	1	0	1	1	RX Interface Busy
x	x	0	1	1	1	0	0	CT Interface Busy
x	x	0	1	1	1	0	1	Markhold Flag
x	x	0	1	1	1	1	0	Eophold Flag
0	0	0	1	1	1	1	1	False
0	1	0	1	1	1	1	1	Carry Flag
1	0	0	1	1	1	1	1	Zero Flag
1	1	0	1	1	1	1	1	True

Condition Select Part II								
M(16)	M(15)	M(14)	M(13)	M(12)	M(11)	M(10)	M(9)	Condition Selected
x	x	1	0	0	0	0	0	$\overline{\text{Resv Stat 0}}$
x	x	1	0	0	0	0	1	$\overline{\text{Resv Stat 1}}$
x	x	1	0	0	0	1	0	$\overline{\text{Resv Stat 2}}$
x	x	1	0	0	0	1	1	$\overline{\text{Resv Stat 3}}$
x	x	1	0	0	1	0	0	$\overline{\text{Resv Stat 4}}$
x	x	1	0	0	1	0	1	$\overline{\text{Resv Stat 5}}$
x	x	1	0	0	1	1	0	$\overline{\text{Resv Stat 6}}$
x	x	1	0	0	1	1	1	$\overline{\text{Resv Stat 7}}$
x	x	1	0	1	0	0	0	$\overline{\text{Resv Stat 8}}$
x	x	1	0	1	0	0	1	$\overline{\text{Resv Stat 9}}$
x	x	1	0	1	0	1	0	$\overline{\text{Resv Stat 10}}$
x	x	1	0	1	0	1	1	$\overline{\text{Resv Stat 11}}$
x	x	1	0	1	1	0	0	$\overline{\text{Timer Bit}}$
x	x	1	0	1	1	0	1	$\overline{\text{Abort}}$
x	x	1	0	1	1	1	0	$\overline{\text{Aborthold}}$
x	x	1	0	1	1	1	1	$\overline{\text{Random Bit}}$
x	x	1	1	0	0	0	0	$\overline{\text{User Flag 0}}$
x	x	1	1	0	0	0	1	$\overline{\text{User Flag 1}}$
x	x	1	0	1	1	1	0	$\overline{\text{User Flag 2}}$
x	x	1	1	0	0	1	1	$\overline{\text{User Flag 3}}$
x	x	1	1	0	1	0	0	$\overline{\text{User Flag 4}}$
x	x	1	1	0	1	0	1	$\overline{\text{User Flag 5}}$
x	x	1	1	0	1	1	0	$\overline{\text{Header FIFO OR}}$
x	x	1	1	0	1	1	1	$\overline{\text{Mark Flag}}$
x	x	1	1	1	0	0	0	$\overline{\text{Eop Flag}}$
x	x	1	1	1	0	0	1	$\overline{\text{Data Flag}}$
x	x	1	1	1	0	1	0	$\overline{\text{Ack Flag}}$
x	x	1	1	1	0	1	1	$\overline{\text{RX Interface Busy}}$
x	x	1	1	1	1	0	0	$\overline{\text{CT Interface Busy}}$
x	x	1	1	1	1	0	1	$\overline{\text{Markhold Flag}}$
x	x	1	1	1	1	1	0	$\overline{\text{Eophold Flag}}$
0	0	1	1	1	1	1	1	$\overline{\text{False}}$
0	1	1	1	1	1	1	1	$\overline{\text{Carry Flag}}$
1	0	1	1	1	1	1	1	$\overline{\text{Zero Flag}}$
1	1	1	1	1	1	1	1	$\overline{\text{True}}$

Table B.8: Condition Code Selection Coding

APPENDIX C

PRC MICROASSEMBLER INPUT GRAMMAR

This appendix lists the grammer accepted by the PRC microassembler used for created the microcode images downloaded to each individual PRX RX during system initialization. This BNF grammar was constructed from the parser grammar and lexical analyzer specification files. All non-terminals are bracketed by ⟨/⟩ and alternatives are separated by |. The lexical analyzer is configured to be case insensative. This notation my cause a little confusion since | is also used in the bitwise OR operations in the alu instruction.

⟨program⟩ → **microprogram** ⟨identifier⟩ ; ⟨body⟩

⟨body⟩ → **begin** ⟨statement_list⟩ **end**

⟨const_1⟩ → **const** ⟨identifier⟩ ⟨int⟩

⟨statement_list⟩ → ⟨statement_list⟩ ⟨statement⟩ | ⟨statement⟩

⟨statement⟩ →
    ⟨recv_decl⟩ ; |
    ⟨const_1⟩ ; |
    ⟨orgin⟩ ; |
    ⟨instruct_line⟩ ; |
    ⟨label⟩ ⟨instruct_line⟩ ;

⟨int⟩ → ⟨decint⟩ | ⟨octint⟩ | ⟨hexint⟩

⟨decint⟩ → ⟨nzdigit⟩ | ⟨decint⟩ ⟨digit⟩

⟨octint⟩ → **0** ⟨octdigit⟩ → ⟨octint⟩ ⟨octdigit⟩

⟨hexint⟩ → **0x** ⟨hexdigit⟩ → ⟨hexint⟩ ⟨hexdigit⟩

⟨octdigit⟩ → **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7**

⟨digit⟩ → ⟨octdigit⟩ | **8** | **9**

⟨hexdigit⟩ → ⟨digit⟩ | **a** | **b** | **c** | **d** | **e** | **f**

⟨alpha⟩ →
  a|**b**|**c**|**d**|**e**|**f**|**g**|
  h|**i**|**j**|**k**|**l**|**m**|**n**|
  o|**p**|**q**|**r**|**s**|**t**|**u**|
  v|**w**|**x**|**y**|**z**

⟨identifier⟩ →
  ⟨alpha⟩ |
  ⟨identifier⟩ ⟨alpha⟩ |
  ⟨identifier⟩ ⟨digit⟩

⟨label⟩ → ⟨identifier⟩ **:**

⟨orgin⟩ → **address** ⟨int⟩

⟨recv_decl⟩ → **receiver** ⟨int⟩

⟨instruct_line⟩ →
  ⟨noop_instruct⟩ |
  ⟨alu_instruct⟩ |
  ⟨ldc_instruct⟩ |
  ⟨xfer_instruct⟩ |
  ⟨wait_instruct⟩ |
  ⟨jump_instruct⟩ |
  ⟨ret_instruct⟩ |
  ⟨set_instruct⟩

⟨noop_instruct⟩ → **noop**

⟨alu_instruct⟩ →
  alu ⟨mux_select⟩ **,** ⟨alu_a⟩ **,** ⟨alu_b⟩ **,** **operation** ⟨int⟩ |
  alu ⟨alu_op⟩

⟨mux_select⟩ → **false** | **carry** | **zero** | **true**

⟨alu_op⟩ →
  ⟨alu_a⟩ **+** ⟨alu_b⟩ |
  ⟨alu_a⟩ **+** ⟨alu_b⟩ **+** **1** |
  ⟨alu_a⟩ **-** ⟨alu_b⟩ **-** **1** |
  ⟨alu_a⟩ **-** ⟨alu_b⟩ |
  ⟨alu_a⟩ |
  ⟨alu_a⟩ **+** **1** |
  ⟨alu_a⟩ **-** **1** |
  ∧ ⟨alu_a⟩ |

$\langle \text{alu\_a} \rangle$ | $\langle \text{alu\_b} \rangle$ |
$\langle \text{alu\_a} \rangle$ | $\wedge$ $\langle \text{alu\_b} \rangle$ |
$\wedge$ $\langle \text{alu\_a} \rangle$ | $\langle \text{alu\_b} \rangle$ |
$\langle \text{alu\_a} \rangle$ **&** $\langle \text{alu\_b} \rangle$ |
$\langle \text{alu\_a} \rangle$ **&** $\wedge$ $\langle \text{alu\_b} \rangle$ |
$\wedge$ $\langle \text{alu\_a} \rangle$ **&** $\langle \text{alu\_b} \rangle$ |
$\langle \text{alu\_a} \rangle$ **exor** $\langle \text{alu\_b} \rangle$ |
$\langle \text{alu\_a} \rangle$ **exnor** $\langle \text{alu\_b} \rangle$ |
-1 |
0 |
1

⟨alu_a⟩→
ctdata |
mdr |
fifo |
trap0 |
trap1 |
rf00 |
rf01 |
rf02 |
rf03 |
rf04 |
rf05 |
rf06 |
rf07 |
rf08 |
rf09 |
rf10 |
rf11 |
rf12 |
rf13

⟨alu_b⟩→
trap0 |
trap1 |
rf00 |
rf01 |
rf02 |
rf03 |
rf04 |
rf05 |
rf06 |
rf07 |
rf08 |
rf09 |
rf10 |
rf11 |
rf12 |
rf13

⟨ldc_instruct⟩ →
  ldc ⟨int⟩ **,** ⟨dest⟩ ⟨goop⟩ |
  ldc ⟨identifier⟩ **,** ⟨dest⟩ ⟨goop⟩

⟨goop⟩ →
  , go rxbus |
  , go rxbus , go ctbus |
  , go ctbus |
  , go ctbus , go rxbus |
  ϵ

⟨dest⟩ →
  trap0 |
  trap1 |
  ctd0 |
  ctaddr0 |
  ctaddr1 |
  ctctl |
  fifo |
  rxd0 |
  rxd1 |
  rxd2 |
  rxd3 |
  rxctl |
  rf00 |
  rf01 |
  rf02 |
  rf03 |
  rf04 |
  rf05 |
  rf06 |
  rf07 |
  rf08 |
  rf09 |
  rf10 |
  rf11 |
  rf12 |
  rf13 |
  mdr |
  mar

$\langle$source$\rangle \rightarrow$
ctdata |
acc |
mdr |
fifo |
trap0 |
trap1 |
rf00 |
rf01 |
rf02 |
rf03 |
rf04 |
rf05 |
rf06 |
rf07 |
rf08 |
rf09 |
rf10 |
rf11 |
rf12 |
rf13

$\langle$xfer_instruct$\rangle \rightarrow$ **xfer** $\langle$source$\rangle$ **,** $\langle$dest$\rangle$ $\langle$goop$\rangle$

$\langle$wait_instruct$\rangle \rightarrow$ **wait** $\langle$polarity$\rangle$ $\langle$condition$\rangle$ $\langle$mark_handler$\rangle$ $\langle$trap0_handler$\rangle$ $\langle$trap1_handler$\rangle$ $\langle$link_control$\rangle$

polarity $\rightarrow \sim \mid \epsilon$

$\langle$mark_handler$\rangle \rightarrow$ **,** **mark** $\mid \epsilon$

$\langle$trap0_handler$\rangle \rightarrow$ **,** **trap0** **(** $\langle$trap0_conds$\rangle$ **)** $\mid \epsilon$

$\langle$trap1_handler$\rangle \rightarrow$ **,** **trap1** **(** $\langle$trap1_conds$\rangle$ **)** $\mid \epsilon$

$\langle$trap0_conds$\rangle \rightarrow \langle$trap0_conds$\rangle$ **,** $\langle$trap0_cond$\rangle \mid \langle$trap0_cond$\rangle$

$\langle$trap0_cond$\rangle \rightarrow \wedge$ **ctbusy** $\mid \wedge$ **rxbusy** | **mark** | **data** | **eop**

$\langle$trap1_conds$\rangle \rightarrow \langle$trap1_conds$\rangle$ **,** $\langle$trap1_cond$\rangle \mid \langle$trap1_cond$\rangle$

$\langle$trap1_cond$\rangle \rightarrow$ **abort** | **mark** | **eop**

⟨condition⟩ →
mark |
eop |
data |
abort |
ack |
rxbusy |
ctbusy |
markhold |
eophold |
aborthold |
f0 |
f1 |
f2 |
f3 |
f4 |
f5 |
f6 |
rsvstat00 |
rsvstat01 |
rsvstat02 |
rsvstat03 |
rsvstat04 |
rsvstat05 |
rsvstat06 |
rsvstat07 |
rsvstat08 |
rsvstat09 |
rsvstat10 |
rsvstat11 |
random |
timer |
false |
carry |
zero |
true

⟨jump_instruct⟩ → **jmp** ⟨polarity⟩ ⟨condition⟩ **,** ⟨identifier⟩ ⟨link_control⟩

⟨ret_instruct⟩ → **ret** ⟨polarity⟩ ⟨condition⟩

⟨link_control⟩ → **,** **link** — ϵ

⟨set_instruct⟩ →
set ⟨valid_flags⟩ |
clear ⟨valid_flags⟩ |
flag ⟨mux_select⟩ **,** ⟨valid_flags⟩

$\langle$valid_flags$\rangle \rightarrow \langle$valid_flags$\rangle$ **,** $\langle$valid_flag$\rangle$ | $\langle$valid_flag$\rangle$

$\langle$valid_flag$\rangle \rightarrow$
f0 |
f1 |
f2 |
f3 |
f4 |
f5 |
f6 |
eophold |
markhold |
aborthold |
all

APPENDIX D

PRC BIT INTERLEAVED CRC IMPLEMENTATION

The standard CRC-CCITT polynomial

$$G(X) = X^{16} + X^{12} + X^{5} + 1 \tag{D.1}$$

was used to generate the next state equations assuming sixteen bits are clocked at one time. This results in the following equations that were used in implementing the CRC generator and check units. These equations where then feed into the multi-input multi-level logic minimization program to produce the final units.

$$
\begin{aligned}
Q_0(t+1) &= \sum_{Q(t)}(0,8,16,22,24) \oplus \sum_{D(t)}(6,8,14,22,30) \\
Q_1(t+1) &= \sum_{Q(t)}(1,9,17,23,25) \oplus \sum_{D(t)}(7,9,15,23,31) \\
Q_2(t+1) &= \sum_{Q(t)}(2,10,18,24,26) \oplus \sum_{D(t)}(4,6,12,20,28) \\
Q_3(t+1) &= \sum_{Q(t)}(3,11,19,25,27) \oplus \sum_{D(t)}(5,7,13,21,29) \\
Q_4(t+1) &= \sum_{Q(t)}(4,12,20,26,28) \oplus \sum_{D(t)}(2,4,10,18,26) \\
Q_5(t+1) &= \sum_{Q(t)}(5,13,21,27,29) \oplus \sum_{D(t)}(3,5,11,19,27) \\
Q_6(t+1) &= \sum_{Q(t)}(6,14,22,28,30) \oplus \sum_{D(t)}(0,2,8,16,24) \\
Q_7(t+1) &= \sum_{Q(t)}(7,15,23,29,31) \oplus \sum_{D(t)}(1,3,9,17,25) \\
Q_8(t+1) &= \sum_{Q(t)}(8,16,24,30) \oplus \sum_{D(t)}(0,6,14,22) \\
Q_9(t+1) &= \sum_{Q(t)}(9,17,25,31) \oplus \sum_{D(t)}(1,7,15,23)
\end{aligned}
$$

$$
\begin{aligned}
Q_{10}(t+1) &= \sum_{Q(t)}(0,8,10,16,18,22,24,26) \oplus \sum_{D(t)}(4,6,8,12,14,20,22,30) \\
Q_{11}(t+1) &= \sum_{Q(t)}(1,9,11,17,19,23,25,27) \oplus \sum_{D(t)}(5,7,9,13,15,21,23,31) \\
Q_{12}(t+1) &= \sum_{Q(t)}(2,10,12,18,20,24,26,28) \oplus \sum_{D(t)}(2,4,6,10,12,18,20,28) \\
Q_{13}(t+1) &= \sum_{Q(t)}(3,11,13,19,21,25,27,29) \oplus \sum_{D(t)}(3,5,7,11,13,19,21,29) \\
Q_{14}(t+1) &= \sum_{Q(t)}(4,12,14,20,22,26,28,30) \oplus \sum_{D(t)}(0,2,4,8,10,16,18,26) \\
Q_{15}(t+1) &= \sum_{Q(t)}(5,13,15,21,23,27,29,31) \oplus \sum_{D(t)}(1,3,5,9,11,17,19,27) \\
Q_{16}(t+1) &= \sum_{Q(t)}(6,14,16,22,24,28,30) \oplus \sum_{D(t)}(0,2,6,8,14,16,24) \\
Q_{17}(t+1) &= \sum_{Q(t)}(7,15,17,23,25,29,31) \oplus \sum_{D(t)}(1,3,7,9,15,17,25) \\
Q_{18}(t+1) &= \sum_{Q(t)}(8,16,18,24,26,30) \oplus \sum_{D(t)}(0,4,6,12,14,22) \\
Q_{19}(t+1) &= \sum_{Q(t)}(9,17,19,25,27,31) \oplus \sum_{D(t)}(1,5,7,13,15,23) \\
Q_{20}(t+1) &= \sum_{Q(t)}(10,18,20,26,28) \oplus \sum_{D(t)}(2,4,10,12,20) \\
Q_{21}(t+1) &= \sum_{Q(t)}(11,19,21,27,29) \oplus \sum_{D(t)}(3,5,11,13,21) \\
Q_{22}(t+1) &= \sum_{Q(t)}(12,20,22,28,30) \oplus \sum_{D(t)}(0,2,8,10,18) \\
Q_{23}(t+1) &= \sum_{Q(t)}(13,21,23,29,31) \oplus \sum_{D(t)}(1,3,9,11,19) \\
Q_{24}(t+1) &= \sum_{Q(t)}(0,8,14,16,30) \oplus \sum_{D(t)}(0,14,16,22,30) \\
Q_{25}(t+1) &= \sum_{Q(t)}(1,9,15,17,31) \oplus \sum_{D(t)}(1,15,17,23,31) \\
Q_{26}(t+1) &= \sum_{Q(t)}(2,10,16,18) \oplus \sum_{D(t)}(12,14,20,28) \\
Q_{27}(t+1) &= \sum_{Q(t)}(3,11,17,19) \oplus \sum_{D(t)}(13,15,21,29) \\
Q_{28}(t+1) &= \sum_{Q(t)}(4,12,18,20) \oplus \sum_{D(t)}(10,12,18,26) \\
Q_{29}(t+1) &= \sum_{Q(t)}(5,13,19,21) \oplus \sum_{D(t)}(11,13,19,27) \\
Q_{30}(t+1) &= \sum_{Q(t)}(6,14,20,22) \oplus \sum_{D(t)}(8,10,16,24) \\
Q_{31}(t+1) &= \sum_{Q(t)}(7,15,21,23) \oplus \sum_{D(t)}(9,11,17,25)
\end{aligned}
$$

APPENDIX E

ANCHORED SHAPES to ANCHORED ROUTES PROOF

Definition 7 *A* pseudo-shape *corresponding to the route* $n_0 \cdots n_k$ *is the sequence* $a_1 \cdots a_i \cdots a_k$ *of directions such that* $a_i = cwhm(n_{i-1}, n_i)$ *for* $1 \leq i \leq k$*.*

Note that a shape is a pseudo-shape with constraints on the permissible directions (to form minimal routes). Define an operator $\oplus$ between two directions in $\{d_0, d_1, \ldots d_5\}$ as follows.

$$d_i \oplus d_j = \begin{cases} d_{[i+1]_6} & \text{if } j = [i+2]_6 \text{ and } i \in \{0, \ldots 5\} \\ \emptyset & \text{if } j = [i+3]_6 \text{ and } i \in \{0, \ldots 5\} \\ d_{[i+5]_6} & \text{if } j = [i+4]_6 \text{ and } i \in \{0, \ldots 5\} \end{cases} .$$

The $\oplus$ operator is undefined between two directions d_i and d_j such that $j = i$ or $j = [i+1]_6$ or $j = [i+5]_6$. Intuitively, traveling first along direction d_i and then along d_j equivalent to traveling a single step along direction $d_i \oplus d_j$ if $d_i \oplus d_j$ is well-defined. For instance, traveling along d_0 and then along d_2 is equivalent to a single step along d_1.

Observation 1 *A route* $n_0 \cdots n_k$ *can be transformed to any other route* $m_0 \cdots m_{k'}$ *with* $k' < k$ *using the following procedure:*

1. *Transform* $n_0 \cdots n_k$ *to the pseudo-shape* $a_1 \cdots a_k$*.*
2. *Transform the* $a_1 \cdots a_k$ *to* $b_1 \cdots b_{k'}$ *by a finite number of applications of the following operations:*
   (a) *Replace a component* a_i *with a component* $a_{i'} \neq a_i$*.*

(b) Replace any two components a_i and a_j, $i \neq j$, by $a_i \oplus a_j$, where $a_i \oplus a_j$ is well-defined.

(c) Permute the components of $a_1 \cdot \cdots \cdot a_k$.

3. Transform the pseudo-shape $b_1 \cdots b_{k'}$ to the route $m_0 \cdots m_{k'}$.

Note that operation (2a) does not "preserve the source–destination node pair". This observation can be formally stated as follows. Let $a_1 \cdots a_k$ be the pseudo-shape associated with the route $n_0 \cdots n_k$. Let $b_1 \cdots b_k$ be a pseudo-shape obtained from $a_1 \cdots a_k$ by a single application of (2a). Also let $m_0 \cdots m_k$ be the route associated with the pseudo-shape $b_0 \cdots b_k$. Then by "preserving the source-destination pair" we mean $b_1 \cdots b_k$ is such that $m_0 = n_0$ implies $m_k = n_k$. With this definition of preserving the source-destination pair we can conclude that the operations (2b) and (2c) preserve the source–destination node pairs. Operation (2b) is the only operation that reduces the length of a pseudo-shape and the corresponding route.

Lemma 6 *A route $n_0 \cdots n_k$ is a minimal route iff the associated pseudo-shape is a shape.*

Proof: We will first prove that the pseudo-shape of a minimal route is a shape.

Suppose not. Then there exist components a_i and a_j in the pseudo-shape such that we can apply operation (2b) to reduce the length of the pseudo-shape. The route associated with this reduced pseudo-shape will be shorter than the assumed minimal route. A contradiction.

Now consider the reverse direction of the lemma, i.e., the route associated with a shape is minimal.

Suppose not. Then there exists a minimal route between the same source–destination pair whose pseudo-shape is shorter than the given shape. Therefore we should be able to reduce our given shape to the pseudo-shape of the minimal route using operations that preserve the source–destination pair. But this cannot happen since no operation of type (2b) can be applied to this shape. Thus our initial assumption the route associated with our shape is not minimal is false. ■

Theorem 4 *There is a one-to-one correspondence between anchored shapes and anchored routes anchored at node 0.*

Proof: We first show that Equation (4.2) transforms anchored shapes to anchored routes at node 0. Consider the anchored shape (s, ℓ). Construct the pair $(r, 0)$ using Equation (4.2). We show that $(r, 0)$ satisfies the three necessary properties of an anchored route.

1. Since s has a length of at least 2, the corresponding route r has a length at least 3.

2. Follows from Lemma 6 that r is a minimal route.

3. Follows directly from the construction that node 0 is contained in the route r.

We now show that Equation (4.1) transforms anchored routes at node 0 to anchored shapes. Consider the anchored route $(r, 0)$ anchored at 0. Construct the pair $(a_1 \cdots a_k, \ell)$ using Equation (4.1). By Lemma 6, $a_1 \cdots a_k$ will be a shape since the route r is minimal by definition. ℓ is bounded by construction between 1 and $k - 1$ as required by the definition of an anchored shape. ■

APPENDIX F

PP-MESS-SIM INPUT SPECIFICATION LANGUAGE

This appendix lists a pseudo BNF grammar for the language used to specific the simulation paramters for a pp-mess-sim run. This BNF grammar was constructed from the parser grammar and lexical analyzer specification files. All non-terminals are bracketed by ⟨/⟩ and alternatives are separated by |. The lexical analyzer is configured to be case insensative.

⟨specification⟩ → ⟨spec_lists⟩

⟨spec_lists⟩ → ⟨spec_lists⟩ ⟨spec⟩ | ⟨spec⟩

spec →
  topology begin ⟨topo_list⟩ **end** |
  node ⟨ident⟩ **begin** ⟨node_list⟩ **end** |
  node ⟨int⟩ **begin** ⟨node_list⟩ **end** |
  task ⟨ident⟩ **begin** ⟨task_list⟩ **end** |
  general begin ⟨gen_list⟩ **end**

⟨topo_list⟩ → ⟨topo_list⟩ ⟨topo_line⟩ | ⟨topo_line⟩

⟨node_list⟩ → ⟨node_list⟩ ⟨node_line⟩ | ⟨node_line⟩

⟨task_list⟩ → ⟨task_list⟩ ⟨task_line⟩ | ⟨task_line⟩

⟨gen_list⟩ → ⟨gen_list⟩ ⟨gen_line⟩ | ⟨gen_line⟩

⟨gen_line⟩ →
  output ⟨pathname⟩ **;** |
  random seed ⟨int⟩ **;** |
  random size ⟨int⟩ **;** |
  option ⟨switch⟩ **true ;** |
  option ⟨switch⟩ **false ;**

⟨switch⟩ →
flex | **bison** |
tasks | **nodes** | **prime** |
toplevel | **messgen** | **messrecv** |
npbus | **ctbus** | **rxbus** | **txbus** | **pcbus** |
rx | **tfu** | **ni** | **imu** |
npinterface | **pcinterface** | **taxitx** | **taxirx** |
cut | **header** | **drop** | **trace** | **arbitrate** |
spec | **topology** | **memstats** | **resources** | **postnode**

⟨param_list⟩ → ⟨param_list⟩ ⟨param⟩ | ⟨param⟩

⟨param⟩ → ⟨int⟩ | ⟨double⟩ | ϵ

⟨task_line⟩ →
arrival ⟨a_process⟩ **;** |
length ⟨l_process⟩ **;** |
target ⟨t_process⟩ **;** |
routing ⟨r_algorithm⟩ **;** |
history ⟨h_select⟩ **;** |
cutoff ⟨int⟩ **;** |
messages ⟨int⟩ **;** |
drop ⟨int⟩ **;** |
max ⟨int⟩ **;** |
min ⟨int⟩ **;** |
rngcycle ⟨int⟩ **;** |
renew ⟨int⟩ **;**

⟨a_process⟩ →
binomial ⟨i_d_parms⟩ |
erlang ⟨d_d_parms⟩ |
geometric ⟨d_parms⟩ |
hyperGeometric ⟨d_d_parms⟩ |
negativeexpntl ⟨d_parms⟩ |
normal ⟨d_d_parms⟩ |
lognormal ⟨d_d_parms⟩ |
poisson ⟨d_parms⟩ |
discreteuniform ⟨i_i_parms⟩ |
uniform ⟨d_d_parms⟩ |
weibull ⟨d_d_parms⟩ |
lengthdiscrete ⟨ld_parms⟩

⟨l_process⟩ →
negativeexpntl ⟨d_parms⟩ |
lengthdiscrete ⟨ld_parms⟩

⟨t_process⟩ →
nodeuniform **(** **)** |
hopuniform **(** ⟨d_list⟩ **)** ⟨d_parms⟩

⟨i_d_parms⟩ → **(** ⟨int⟩ **,** ⟨double⟩ **)**

⟨d_d_parms⟩ → **(** ⟨double⟩ **,** ⟨double⟩ **)**

⟨d_parms⟩ → **(** ⟨double⟩ **)**

⟨i_i_parms⟩ → **(** ⟨int⟩ **,** ⟨int⟩ **)**

⟨ld_parms⟩ → **(** ⟨di_list⟩ **)**

⟨di_list⟩ → ⟨di_list⟩ **,** ⟨di_single⟩ | ⟨di_single⟩

⟨di_single⟩ → ⟨double⟩ **,** ⟨int⟩

⟨d_list⟩ → ⟨d_list⟩ **,** ⟨double⟩ | ⟨double⟩

⟨h_select⟩ → **none** | **latency** | **correlation**

⟨node_line⟩ →
 tasks ⟨int⟩ **;** |
 select task ⟨ident⟩ ⟨int⟩ **;** |
 select node ⟨ident⟩ **;**

⟨topo_line⟩ →
 select cwhm ; |
 select sqmesh ; |
 select hypercube ; |
 select karycube ; |
 size ⟨int⟩ **;** |
 dimension ⟨int⟩ **;**

⟨int⟩ → ⟨decint⟩ | ⟨octint⟩ | ⟨hexint⟩

⟨decint⟩ → ⟨nzdigit⟩ | ⟨decint⟩ ⟨digit⟩

⟨octint⟩ → **0** ⟨octdigit⟩ → ⟨octint⟩ ⟨octdigit⟩

⟨hexint⟩ → **0x** ⟨hexdigit⟩ → ⟨hexint⟩ ⟨hexdigit⟩

⟨octdigit⟩ → **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7**

⟨digit⟩ → ⟨octdigit⟩ | **8** | **9**

⟨hexdigit⟩ → ⟨digit⟩ | **a** | **b** | **c** | **d** | **e** | **f**

⟨alpha⟩ →
 a|**b**|**c**|**d**|**e**|**f**|**g**|
 h|**i**|**j**|**k**|**l**|**m**|**n**|
 o|**p**|**q**|**r**|**s**|**t**|**u**|
 v|**w**|**x**|**y**|**z**

⟨identifier⟩ →
 ⟨alpha⟩ |
 ⟨identifier⟩ ⟨alpha⟩ |
 ⟨identifier⟩ ⟨digit⟩

BIBLIOGRAPHY

[1] *Am79168/Am79169-275 TAXI-275 Transmitter/Receiver Transparent Asynchronous Transmitter/Receiver Interface*, Advanced Micro Devices, 15765-b-0 edition.

[2] *Am79168/Am79169 TAXI^{tm}-275 Technical Manual*, Advanced Micro Devices, ban-0.1m-1/93/0 17490a edition.

[3] G. Albertengo and R. Sisto, "Parallel crc generation," *IEEE Micro*, pp. 63–71, October 1990.

[4] T. Anderson, H. Levy, B. Bershad, and E. Lazowska, "The interaction of architecture and operating system design," in *Proc. Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 108–120, April 1991.

[5] H. Carr, J. Evans, R. Kessler, L. Stoller, and M. Swanson, "Mayfly system software," Technical Report HPL-SAL-89-25, Hewlett Packard Company, April 1989. Available in RTCL library.

[6] M.-S. Chen, K. G. Shin, and D. D. Kandlur, "Addressing, routing and broadcasting in hexagonal mesh multiprocessors," *IEEE Trans. Computers*, vol. 39, no. 1, pp. 10–18, January 1990.

[7] D. D. Clark and D. L. Tennenhouse, "Architectural considerations for a new generation of communication protocols," in *Proc. of ACM SIGCOMM*, pp. 200–208, September 1990.

[8] E. C. Cooper, P. A. Steenkiste, R. D. Ransom, and B. D. Zill, "Protocol implementation on the Nectar communication processor," in *Proceedings of the SIGCOMM Symposium*, pp. 135–144. ACM, September 1990.

[9] W. J. Dally and C. L. Seitz, "The torus routing chip," *Journal of Distributed Computing*, vol. 1, no. 3, pp. 187–196, 1986.

[10] W. J. Dally and C. L. Seitz, "Deadlock-free message routing in multiprocessor interconnection networks," *IEEE Trans. Computers*, vol. C-36, no. 5, pp. 547–553, May 1987.

[11] W. Dally, *VLSI and PARALLEL COMPUTATION*, chapter Network and Processor Architecture for Message-Driven Computers, pp. 140–222, Morgan Kaufmann Publishers, Inc., 1990.

[12] W. Dally, A. Chien, S. Fiske, W. Horwat, J. Keen, P. Nuth, J. Larivee, and B. Totty, "Message-driven processor architecture version 11," MIT Concurrent VLSI Architecture Memo 14, MIT, August 1988.

[13] S. W. Daniel, "Improving distributed system communications through hardware support," Prelim, 1992.

[14] S. W. Daniel and J. W. Dolter, "Error-burst detection with bit-interleaved crcs," In preparation, 1993.

[15] S. W. Daniel, "A packet control for HARTS: An experiemental distributed real-time system," Real-Time Computing Laboratory Techinal Report RTCL-TR-1-91, Real-Time Computing Laboratory, 1301 Beal Ave, Ann Arbor, MI 48109-2122, 1991.

[16] B. Davie, "The architecture and implementation of a high-speed host interface," *IEEE Journal on Selected Areas in Communications*, vol. 11, no. 2, pp. 228–239, February 1993.

[17] A. Davis, R. Hodgson, B. Schediwy, and K. Stevens, "Mayfly system hardware," Technical Report HPL-SAL-89-23, Hewlett-Packard Company, April 1989.

[18] A. L. Davis, "Mayfly: A general-purpose, scalable, parallel processing architecture," *Lisp and Symbolic Computation*, vol. 5, no. 1/2, pp. 7–47, May 1992.

[19] A. Davis and S. Rovison, "The architecture of the faim-1: Symbolic multiprocessing system," in *Proc. Int'l Joint Conf. on Artificial Intelligence*, pp. 32–38, Los Angeles, August 1985.

[20] J. W. Dolter, P. Ramanathan, and K. G. Shin, "A microprogrammable VLSI routing controller for HARTS," Technical Report CSE-TR-12-89, Dept. of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, 1989.

[21] J. W. Dolter, P. Ramanathan, and K. G. Shin, "A microprogrammable VLSI routing controller for HARTS," in *International Conference on Computer Design: VLSI in Computers*, pp. 160–163, October 1989.

[22] J. Goldberg, M. W. Green, W. H. Kautz, K. N. Levitt, P. M. Melliar-Smith, R. L. Schwartz, and C. B. Weinstock, "Development and analysis of the software implemented fault-tolerance (sift) computer," Contractor Report 172146, NASA Langley Research Center, February 1984.

[23] J. M. Gordon, *Efficient Schemes for Massively Fault-Tolerant Parallel Communication*, PhD thesis, The Unversity of Michigan, 1990.

[24] Z. Haas, "A communication architecture for high-speed networking," in *IEEE INFOCOM*, pp. 433–441, June 1990.

[25] D. D. Kandlur and K. G. Shin, "Reliable broadcast algorithms for HARTS," *ACM Trans. Computer Systems*, vol. 9, no. 4, pp. 374–398, November 1991.

[26] P. Kermani and L. Kleinrock, "Virtual cut-through: A new computer communication switching technique," *Computer Networks*, vol. 3, no. 4, pp. 267–286, September 1979.

[27] L. Kleinrock, *Queueing systems*, volume I: Theory, John Wiley & Sons, 1975.

[28] A. Kovaleski, S. Ratheal, and F. Lombardi, "An architecture and interconnection scheme for time-sliced buses in real-time processing," *Proc. Real-Time Systems Symposium*, pp. 20–27, 1986.

[29] A. Krishnakumar and K. Sabnani, "VLSI implementations of communication protocols – a survey," *IEEE Journal on Selected Areas in Communications*, vol. 7, no. 7, pp. 1082–1090, September 1989.

[30] H. Kung, R. Sansom, S. Schlick, P. Steenkiste, M. Arnould, F. J. Bitz, F. Christianson, E. C. Cooper, O. Menzilcioglu, D. Ombres, and B. Zill, "Network-based multicomputers: An emerging parallel architecture," in *Supercomputing 91*, pp. 664–673. IEEE, ACM, New York, NY, USA, November 1991.

[31] R. Lee, "Cyclic code redundancy," *Digital Design*, vol. 11, no. 7, pp. 77–85, July 1981.

[32] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.3BSD Unix Operating System*, Computer Science, Addison Wesley, May 1989. ISBN 0-201-06196-1.

[33] O. Menzilcioglu and S. Schlick, "Nectar CAB: A high-speed network processor," in *Proc. Int'l Conf. on Distributed Computing Systems*, pp. 508–515, May 1991.

[34] R. Metcalfe, "Computer/network interface design: Lessons from Arpanet and Ethernet," *IEEE Journal on Selected Areas in Communications*, vol. 11, no. 2, pp. 173–180, February 1993.

[35] E. Miller, "High-speed cyclic redundancy check generation and verification," *IBM Technical Disclosure Bulletin*, vol. 21, no. 8, pp. 3065–3066, January 1979.

[36] M. Noakes, D. Wallach, and W. Dally, "The J-machine multicomputer: An architectural evaluation," in *Proc. Int'l Symposium on Computer Architecture*, pp. 224–235, 1993.

[37] A. Pandeya and T. Cass, "Parallel crc lets many lines use one circuit," *Computer Design*, vol. 14, no. 9, pp. 87–91, September 1975.

[38] P. Ramanathan, D. D. Kandlur, and K. G. Shin, "Hardware assisted software clock synchronization for homogeneous distributed systems," *IEEE Trans. Computers*, vol. C-39, no. 4, pp. 514–524, April 1990.

[39] D. A. Reed and R. M. Fujimoto, *Multicomputer Networks: Message-Based Parallel Processing*, M. I. T. Press, Cambridge, Massachusetts, 1987.

[40] D. A. Reed and D. C. Grunwald, "The performance of multicomputer interconnection networks," *IEEE Computer*, vol. 20, no. 6, pp. 63–73, June 1987.

[41] J. Rexford and K. G. Shin, "Shortest-path routing in homogeneous point-to-point netowrks with virtual cut-through switching," Computer Science and Engineering Techinal Report CSE-TR-146-92, University of Michigan, November 1992.

[42] C. L. Seitz, *VLSI and PARALLEL COMPUTATION*, chapter Concurrent Architectures, pp. 1–84, Morgan Kaufmann Publishers, Inc., 1990.

[43] T. B. Smith, "Fault tolerant processor concepts and operation," Contractor Report CSPL-P-1727, Charles Stark Draper Laboratory, May 1983.

[44] T. B. Smith and J. H. Lala, "Development and evaluation of a fault-tolerant multiprocessor (FTMP) computer volume I FTMP principles of operation," Contractor Report 166071, NASA Langley Research Center, May 1985.

[45] E. Spertus, S. Goldstein, K. Schauser, T. von Eicken, D. Culler, and W. Dally, "Evaluation of mechanisms for fine-grained parallel programs in the J-machine and the CM-5," in *Proc. Int'l Symposium on Computer Architecture*, pp. 302–313, 1993.

[46] J. D. Spragins, J. H. Hammond, and K. Pawlikowski, *Telecommunications Protocols and Design*, Addison-Wesley Publishing Company, 1991.

[47] W. Stallings, *Data and Computer Communications*, Macmillan Publishing Company, New York, second edition, 1988.

[48] P. Steenkiste, "Analyzing communication latency using the nectar communication processor," in *Proc. of ACM SIGCOMM*, pp. 199–209. ACM, ACM, New York, NY, USA, August 1992.

[49] K. S. Stevens, "The communication framework for a distributed ensemble architecture," AI Technical Report 47, Schlumberger Research Laboratory, February 1986.

[50] K. S. Stevens, S. V. Robison, and A. L. Davis, "The Post Office — Communication support for distributed ensemble architectures," in *Proc. Int'l Conf. on Distributed Computing Systems*, pp. 160–166, 1986.

[51] L. Svobodova, "Implementing osi systems," *IEEE Journal on Selected Areas in Communications*, vol. 7, no. 7, pp. 1115–1129, September 1989.

[52] C. A. Thekkath and H. M. Levy, "Limits to low-latency communication on high-speed networks," *ACM Trans. Computer Systems*, vol. 11, no. 2, pp. 179–203, May 1993.