Flexible Router Architectures for Point-to-Point Networks

by

Stuart Willard Daniel

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
1996

To Tami.

ACKNOWLEDGEMENTS

I would to acknowledge the contributions of many people:

- I am grateful to my graduate advisor, Dr. Kang Shin, for his support and guidance throughout the long, and sometimes tortuous, process that has finally culminated in this dissertation.
- I would also like to thank James Dolter, who provided most of the inspiration and was, to a large extent, the driving force behind much of the PRC design. The strength of his work can be seen by how much of it has lived on in the PRC through the many design revisions.
- The other members of my dissertation committee, for their suggestions and feedback. Dr. Brown deserves particular thanks for all of his advice on VLSI issues.
- Jennifer Rexford has assumed a critical role in the PRC project by providing me with a sounding board for ideas both grandiose and mediocre.
- The PRC simulations in this work would not have been possible without the simulator developed by Jim and later extended by Jennifer and Wu-chang Feng.
- Finally, the SPIDER board itself has been a group project, with notable contributions from Ashish Mehra, Jaehyun Park, and Atri Indiresan.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF APPENDICES

CHAPTER 1

Introduction

"Our device conforms to all international standards for communications."

"In other words, it doesn't do anything useful ... "

— Scott Adams' *Dilbert*

In parallel and distributed systems, efficient communication is a major factor in enabling fine-grained cooperation between processing nodes. Maximizing system performance requires matching application characteristics and performance requirements with a suitable network design. However, parallel applications employ a wide variety of communication paradigms that affect the quantity and frequency of communication between nodes. Since most networks are fairly inflexible in their design, maximizing performance often requires redesigning the application to utilize the network efficiently.

Rather than forcing the system to adjust its communication patterns to the network, we wish to adapt the network to the system's requirements and traffic workloads. Since changing the actual network interface hardware is impractical, we introduce and evaluate flexible communication hardware that allows network policies to be tuned to these diverse characteristics. In particular, we will focus on the low-level policies implemented at the *router*-level in the network. In this thesis, our discussion will focus on two complementary themes: the performance benefit from a flexible network architecture, and the cost (in both implementation and performance) of this flexibility.

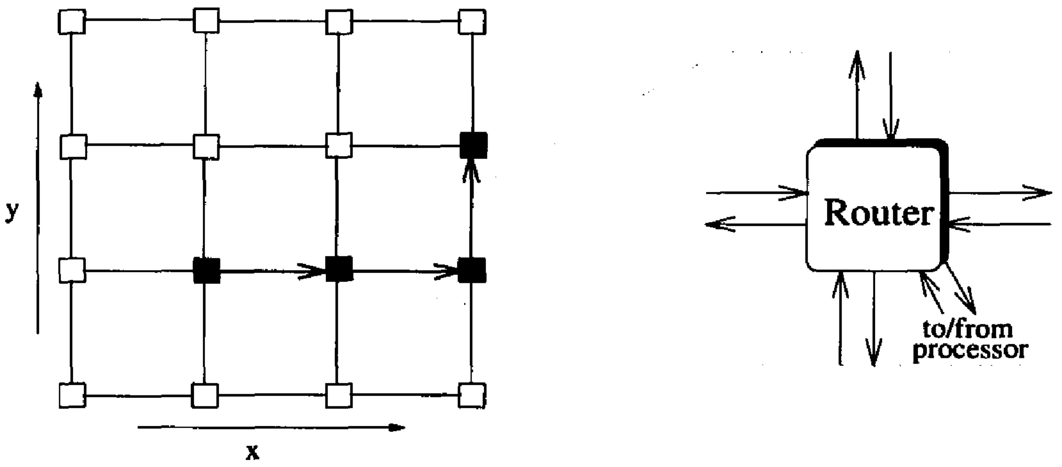


Figure 1.1: A 4×4 unwrapped square mesh network.

1.1 Application Domain

This thesis concentrates on the network policies used in message-passing multicomputers with a *point-to-point* (or direct) network topology. In a point-to-point, every node has a direct connection with several other nodes; these nodes are then connected to still others to form a network topology. One such topology — a 4×4 square mesh with 16 nodes — is shown in Figure 1.1; in this network, every node has a direct connection through its *router* to up to four other nodes. Point-to-point networks, with their multiplicity of processors and internode routes and scalable communication bandwidth, provide a natural platform for applications that require both high performance and dependability [16, 29, 31, 43].

The particular policies we are examining have traditionally belonged within the parallel computer domain, but we are also examining them from the context of a distributed system. Parallel computing has been motivated primarily by the need for high-performance scientific computing, resulting in regular interconnection networks and tightly-coupled processing elements. Distributed systems, on the other hand, arose from the need for connectivity, communication, and resource sharing between network-based machines. While parallel computers and distributed systems have traditionally been employed in different application domains, technological advances in VLSI, networking, and operating systems have expanded the domain of distributed computing, facilitating the merger of these seemingly disparate disciplines.

Many of the protocols used in point-to-point networks are also applied in other network architectures. Many multistage networks, for example, use protocols originally developed for direct networks inside the communication switches [41, 65, 68]. Even though nodes usually have only one port into the switching fabric, there are several routes to each of the other nodes within the network; this gives rise to many of the same issues of route selection and competition for transmission links.

In this research, we will concentrate our attention on message-passing multicomputers. In this context, a *message* is a unit of information that the application submits to the operating system for transmission to some process on another *destination* node (or nodes). The node where the message originates will be referred to as the *source*. For efficiency, the source node often breaks larger messages into one or more *packets*. The largest unit handled by the network, therefore, is the packet. At the destination, packets are reassembled into messages before delivery to the application. Networks that deliver packets in the order in which they were transmitted greatly simplify this task. In this research, we will focus on supporting a wide range of packet sizes, which allows the operating system to choose the packet size (or sizes) that are most appropriate to its needs.

Figure 1.2 shows a generic router. Several input and output ports connect the router to the host processor, while the input and output channels constitute the internode communication links for that node. Packets are transferred over the internode channels one *phit* (physical unit) at a time. In addition, packets are often subdivided into *flits* (flow control units); a flit is the smallest unit used for flow control. Typical flit sizes range from as small as a single phit up to as large as an entire packet. In this model, the router is tasked with delivering packets that arrive at its input channels and ports to the appropriate output.

Much of this work has been done as part of the communication support for Hexagonal Architecture for Real-Time Systems (HARTS) [31, 62], a point-to-point, distributed system targeted for real-time applications. One goal in designing the HARTS communication subsystem was to provide support for predictable, dependable communication while retaining the flexibility to experiment with a variety of existing and future communication protocols over several network topologies and under a variety of traffic characteristics. As part of this work, we have designed and fabricated a router to show how this goal can be met through a low-overhead, integrated solution which supports, but does not dictate, higher-level host policies.

Figure 1.3 shows a typical node in HARTS. Each node consists of one or more *application processors* (APs) and a *network processor* (NP) that executes the communication protocols.

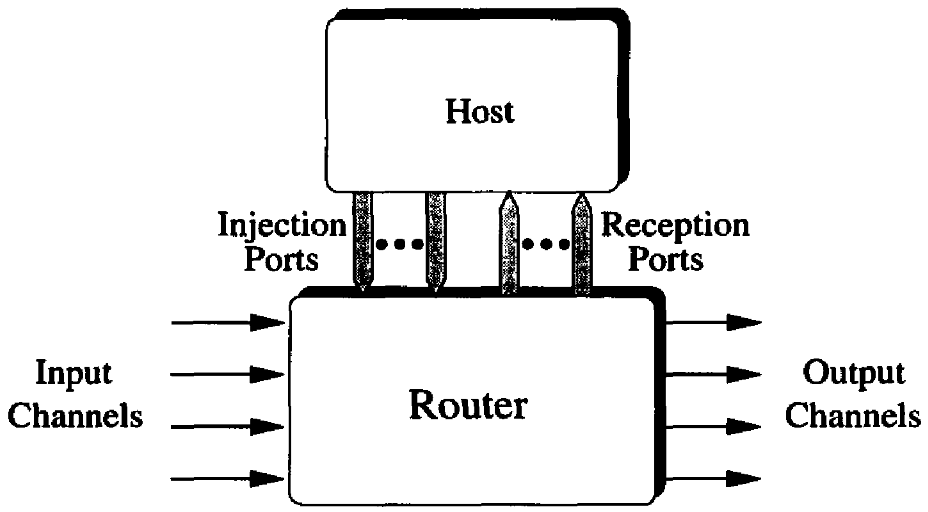


Figure 1.2: A generic router architecture.

Internode communications are provided in the prototype system either through the Ethernet connections on each processor board and a commercial fiber-channel interface, the Ancor CIM. The final system will have internode communications provided by the SPIDER board, which is built around the router presented in Chapter 4. pSOS+ [67], a commercial real-time executive, provides system support to application threads within a node while x-kernel [28] coordinates communication between nodes.

To support HARTS, the router described in this work needs several specific features. Since it interacts directly with the x-kernel, support for variable packet sizes and header/data separation are useful. In addition, support (via timestamping of packets on transmission and reception) for a distributed clock synchronization algorithm [31] is required.

1.2 Application Requirements for Communication

Parallel applications vary in the *quality-of-service* that they require from the network. Most parallel applications require low *communication latency*. Communication latency is defined by Ni and McKinley as being the sum of three values: start-up latency, network latency, and blocking time [46]. The start-up latency includes the delay from the operating system handling the packet at both the source and destination nodes; the operating system and the host-side interface provided by the network are crucial in determining this factor. Network latency is defined as the time from the packet header entering the network at the source to the tail exiting at the destination. The blocking latency covers all delays caused

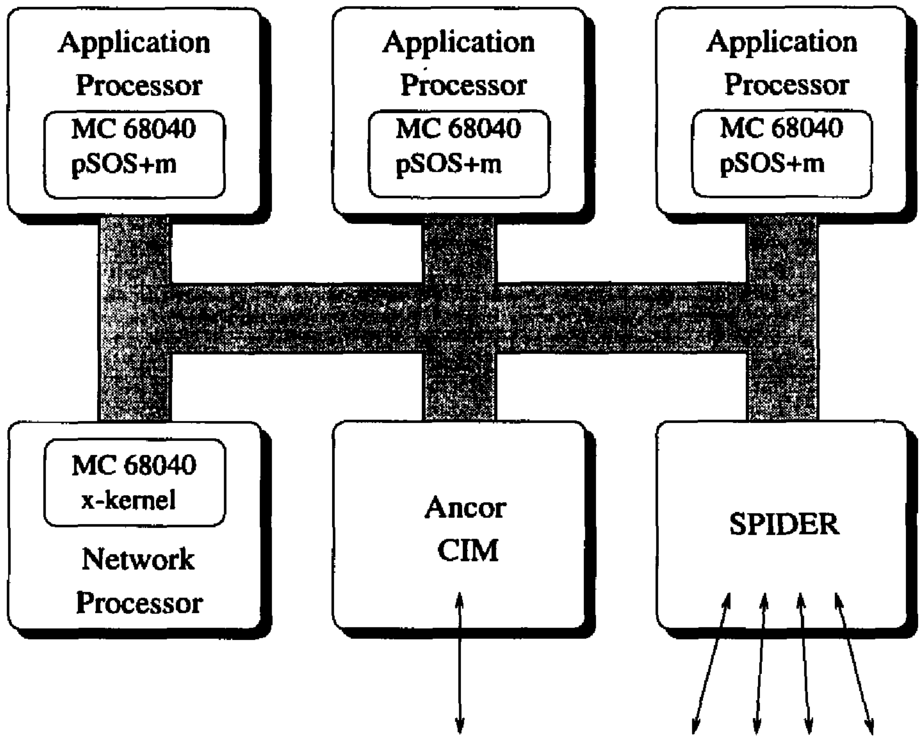


Figure 1.3: Architecture of a HARTS node.

by competition for network resources such as links and buffers.

Another key performance metric is *throughput*, which is the sustained rate of data delivery given some applied load. The network throughput is equal to the applied load until the network saturates (i.e., the throughput is less than the applied load). The actual load at which the network saturates, and the throughput of the network for loads past this point, determine both how often a network might saturate and whether it can recover from this state. Since no application can execute efficiently with a continually-saturated network, the network's saturation point and how it degrades under these heavy loads are important considerations for applications that need to transfer large amounts of data.

It is also becoming commonplace to use digital computers for real-time applications such as fly-by-wire, industrial process control, computer-integrated manufacturing, and medical life-support systems. These applications typically impose stringent timing requirements on the computer network. Since these real-time applications often coexist with other applications, multiple classes of traffic with different requirements for predictability and performance may require simultaneous service [60].

It is often difficult to improve one of these characteristics without impacting the others.

For example, network policies that improve latency may often lower the saturation point of the network while also decreasing predictability. For this reason, a router architecture that allows the system to select the policies that meet its needs is desirable.

1.3 Application Traffic Patterns

Parallel applications generate a wide range of communication workloads that vary according to the application (or applications) running on the system and the mapping of processes onto the systems nodes. Consequently, communication characteristics such as message interarrival times, lengths, and target destinations vary substantially on modern parallel machines [9, 25, 27].

Message/packet arrival: Message arrivals have typically been modeled as a Poisson process with exponentially-distributed interarrival times, due to the simplicity of the model and the lack of more realistic data. Recent studies, however, have shown that applications, due in part to packetization of longer messages, often communicate through bursty traffic [9, 27]. In addition, group communications (such as barrier synchronization and global reduction/combine operations) often send several copies of a single message to different destinations at the same time, which suggests a need for multicast communication [51]. These traffic models have significant impact on network evaluation, since Poissonian arrival processes typically yield overly optimistic performance results.

Message/packet length: Message and packet lengths depend on several factors including packet-size restrictions and the mixture of data and control messages. Although fixed-length packets or exponentially-distributed lengths simplify analytic models, recent work shows that real multicomputer applications typically generate *bimodal* packet-length distributions [9, 27], where inter-node communication consists of large data transfers with small request and acknowledgement packets.

Message destination: Message destination distributions vary significantly, based on the network topology and the mapping of processes onto nodes. Many studies use a simple uniform distribution of destination nodes, but this does not capture the characteristics of many applications. Hop-uniform traffic distributions can represent spheres of spatial locality, but these still do not capture the communication structure of specific parallel algorithms or applications. In particular, many scientific programs generate permutation patterns such as matrix-transpose (dimension-reversal), bit-complement, and bit-reversal [13, 56]. Other application constructs, such as synchronization or multicast operations, may induce "hot-spots" of heavily-utilized nodes and links [4, 56]. Finally, dynamic models [27] can produce variation in target destinations during the course of application execution.

1.4 Approach and Contributions

For routers in a point-to-point network (such as the one in Figure 1.2), the basic problem can be summed up: how do we move packets from the input ports and channels to the outputs, while meeting the current network requirements for quality of service? This movement is controlled by several low-level policies.

Addressing: Every router uses some addressing format to specify the destination (or set of destinations) for a packet. The destination might be given as a node ID, as an offset from the current location, and so on.

Routing: The routing policy for a network determines the path taken by a packet between its source and its destination.

Selection: The selection scheme determines the order in which channels will be considered. For example, a dimension-ordered scheme might prefer a link on the x-axis over one on the y-axis.

Switching: The switching policy determines how data is moved between the input and output ports of a node.

Queueing: The queueing scheme determines where packets are buffered — in a central queue, at an output, or at the inputs.

Chapter 2 discusses these policies in more detail. We will examine how a single router may support a wide range of addressing, routing, selection, and switching schemes. We will neglect the queueing policy, as this reflects the actual location of the buffers in the router. We will consider, however, methods for varying the decision on *when* to use these buffers.

A major theme of this research is that since no policy is best-suited for all situations, the proper answer might be to support a number of policies and dynamically choose the one best suited to the current workload. In meeting this goal, the thesis makes several contributions:

- We analyze how routers typically handle packets, and use this to determine the functions necessary for the desired degree of flexibility.
- To provide this flexibility, we develop two routing engines: a microcontroller-based approach that maximizes flexibility, and a more limited architecture that stresses performance.
- We also develop and evaluate a new switching scheme, *hybrid switching*, that allows the system to balance resource consumption and improve overall performance.
- We have fabricated a prototype flexible router, the Programmable Routing Controller, that meets the goals we have outline above.

1.5 Organization of the Dissertation

The remainder of this dissertation consists of seven chapters and four appendices. The next chapter provides a background on low-level multicomputer network policies. In particular, an emphasis is placed on discussing the routing, selection, and switching schemes. The chapter concludes with an examination of the major actions required to route a packet from the inputs of a router to its outputs.

Drawing on this discussion, Chapter 3 introduces a microcontroller-based router architecture to support flexible routing and switching. Since the routing engine is an expensive resource, we develop an architecture for sharing each routing engine amongst several input channels without degrading performance. We then discuss in detail the microcontroller architecture and functions necessary to support a wide range of routing and switching schemes. To illustrate these functions, and to demonstrate the power of this routing engine, we also present several routing and switching microprograms.

Chapter 4 introduces the *Programmable Routing Controller* (PRC), which combines the routing engines of the previous chapter with a powerful host interface. Besides the microcontroller-based routing engines, this ASIC has several features that distinguish it from other multicomputer routers. The main data switch is based on a time-division multiplexed bus, rather than the more common crossbar; this simplifies channel allocation and enables multicast operations without impacting performance.

We then introduce a new switching scheme, *hybrid switching*, which combines aspects of both wormhole and virtual cut-through switching by buffering a small fraction of blocked packets and limiting the number of links that blocked packets can hold. Through cycle-level simulations of a network of PRCs, we show that this switching scheme significantly reduces the buffer requirements for in-transit packets when compared to virtual cut-through, while providing higher maximum throughput than wormhole switching. In this manner, hybrid switching bridges the performance gap between other cut-through switching schemes.

Through simulation, we also demonstrate how the PRC can be used to tailor network policies to the application requirements. Four different case studies are performed that compare routing, switching, and selection schemes under differing application workloads. Through these simulations, we can observe the strengths and weaknesses of the PRC and of our flexible router concept.

Drawing on these observations, Chapter 7 addresses the tradeoff between performance and flexibility by developing the z-channel routing engine. Unlike the routing engine described in Chapter 3, the z-channel limits its flexibility in order to match or exceed the performance level of multicomputer routers using fixed network policies.

Chapter 8 concludes this dissertation with a brief summary of our contributions and future directions for this research. The appendices provide additional implementation details on the PRC and the routing engines.

CHAPTER 2

Background

This next transparency is an incomprehensible jumble
of complexity and undefined acronyms
— Dilbert, in Scott Adams' *Dilbert*

For point-to-point networks, application characteristics affect the performance of particular network policies [23, 26, 35, 36, 58]. Parallel applications often communicate via message-passing, where a message is a unit of information submitted to the communication subsystem for transmission to a process on another node (or nodes). Depending upon the application, messages may have several different properties. The network designer should account for messages with a range of sizes, priorities, and deadlines.

2.1 Low-level Router Policies

The architecture of a router may generally be characterized by its routing, selection, switching, and queueing policies. Since this research is focusing on providing flexible routing and switching policies at the hardware level, we will consider the first three in more detail.

2.1.1 Routing

The routing policy for a network determines the path taken by a packet between its source and its destination. The route may be selected by the source node (*source* routing) prior to transmission, or computed at every intervening node (*distributed* routing). Source-routed schemes must carry information about the entire path in the header, which increases the size of the packet. In addition, the route generally cannot be varied once the packet has begun transmission.

Distributed routing schemes are divided into two general classes: *oblivious* and *adaptive*. Oblivious routing schemes always use the same path between two nodes, irregardless of network conditions, while adaptive schemes vary the path according to dynamic network conditions such as channel contention and faulty links. Routing schemes generally try to take a minimal-length path through the network from their source to the destination; every link chosen for the packet will bring it closer to its destination. Algorithms that may choose longer paths to circumvent congested links are termed *non-minimal*.

Finally, many applications and systems can benefit from hardware support for *multicast* packets, where a single packet is sent to multiple destinations. While few routers currently support multicast routing, it has benefits for many applications such as barrier synchronization, broadcast, and distributed shared memory [31, 47]. A multicast path is shown in Figure 2.1(d), where the source node S is transmitting a packet to two destinations, nodes 11 and 13. Rather than separately transmitting two identical packets, a network that supported multicast routing could transmit a single packet to node 9 and then "tee" it by transmitting it in two directions simultaneously [31].

Figure 2.1 illustrates the differences amongst the various routing schemes and selection functions. Figure 2.1(a) compares a dimension-ordered selection function, where links along the x-axis are preferred, and a diagonal selection function that reduces the larger offset. If the routing scheme is oblivious, it will always follow the route determined by this selection function. Adaptive schemes, however, will try alternate links if the first choice is not available. An adaptive minimal scheme is shown in Figure 2.1(b): once the packet reaches node 2, it finds that the $+x$ link to node 3 is busy. An oblivious routing scheme will then wait for the link to become available, while adaptive schemes would use the $+y$ link to node 6 instead. After forwarding the packet header to node 6, the link to node 7 is also found to be busy; the adaptive scheme then continues in the $+y$ direction to reach node 11. At this point, only one link remains that will take the packet to its destination; if it is busy any minimal scheme will be forced to wait. If an adaptive non-minimal scheme, however, may find an unblocked path to the destination by routing the packet through nodes 14 and 15.

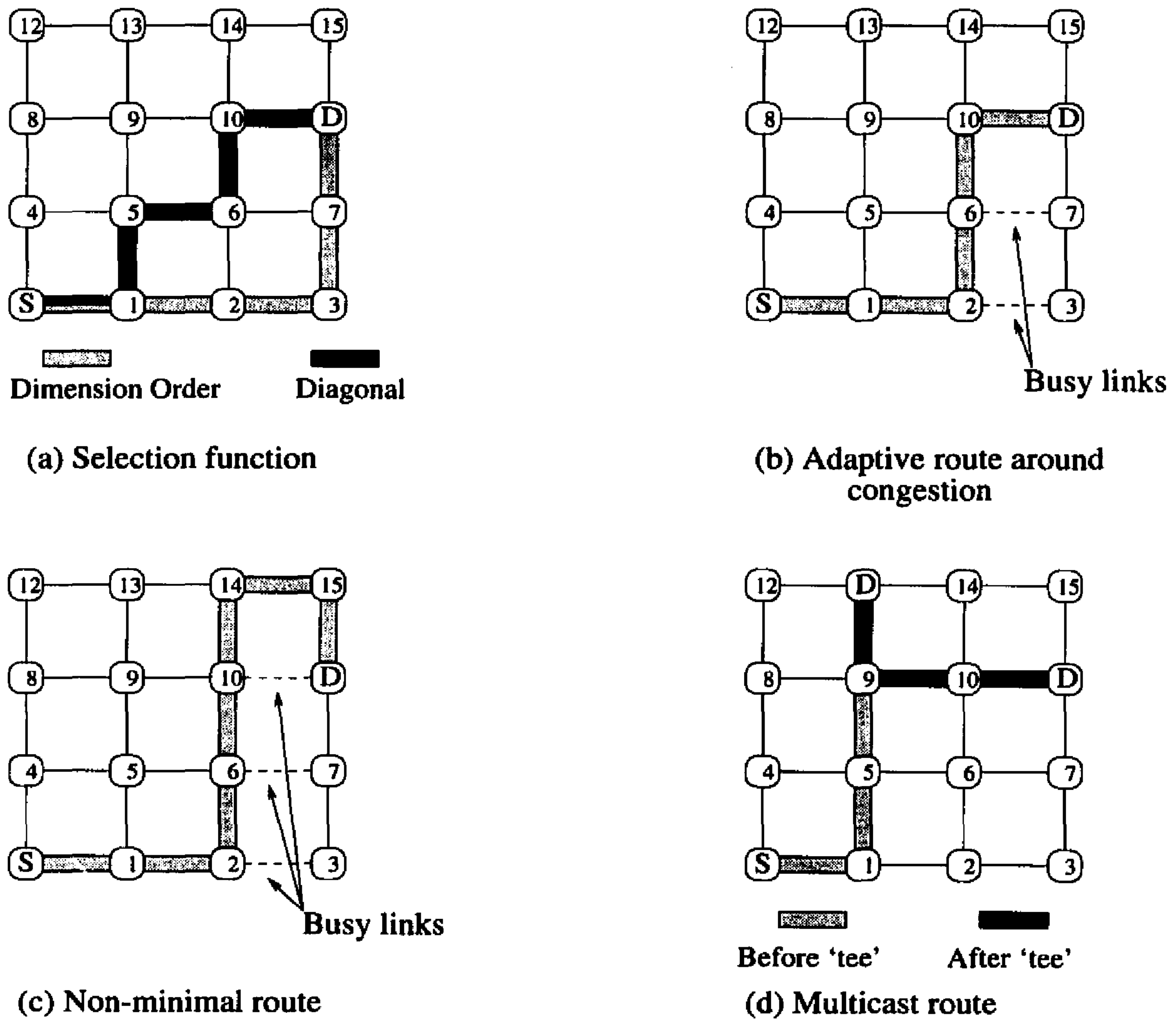


Figure 2.1: Example paths of routing algorithms.

2.1.2 Switching

The switching scheme impacts performance by determining the link and buffer resources a packet consumes at each node in its route. Packet switching requires an incoming packet to buffer completely before transmission to a subsequent node can begin. In contrast, cut-through switching schemes, such as *virtual cut-through* [33] and *wormhole* [10], try to forward an incoming packet directly to an idle output link; if the link is busy, virtual cut-through switching buffers the packet, whereas a wormhole packet stalls pending access to the link. While first-generation multicomputers employed packet switching, most contemporary routers utilize wormhole [10, 15, 53, 14, 61, 63, 71] or virtual cut-through [19, 20, 38, 66] switching for lower latency and reduced buffer space requirements.

Virtual cut-through and wormhole switching differ in how they handle packets that cannot immediately proceed to the next node because the appropriate output links are busy with other traffic. Virtual cut-through switching buffers blocked packets at the local node and releases the links currently held by the packet, but wormhole switching stalls the packet in the network, while holding all links the packet has acquired. Since packets never buffer at intermediate nodes with wormhole switching, nodes only receive those packets destined for them. Stalling the packet in the network, however, consumes network resources to "store" the packet, effectively dilating the packet's length. Virtual cut-through, on the other hand, minimizes the network bandwidth consumed by packets, but uses memory and control resources at intermediate nodes to store blocked packets.

Figure 2.2 illustrates the difference between three cut-through switching schemes: circuit, wormhole, and virtual cut-through. The packets are shown as they would appear when their forward progress is blocked by a busy link while trying to exit node C. Under circuit switching, the header flit has been forwarded to node C, while the body of the packet waits at the source until a circuit has been established between the source and the destination. This enables the header to easily backtrack if blocked. Wormhole switching, on the other hand, forwards each data flit as soon as there is an empty buffer at the next node; consequently, the blocked packet has a flit stored on each node. Since virtual cut-through provides sufficient flit buffers at each node to buffer blocked packets, all of the flits of the blocked packet are stored at node C. This frees the transmission links at the source, node

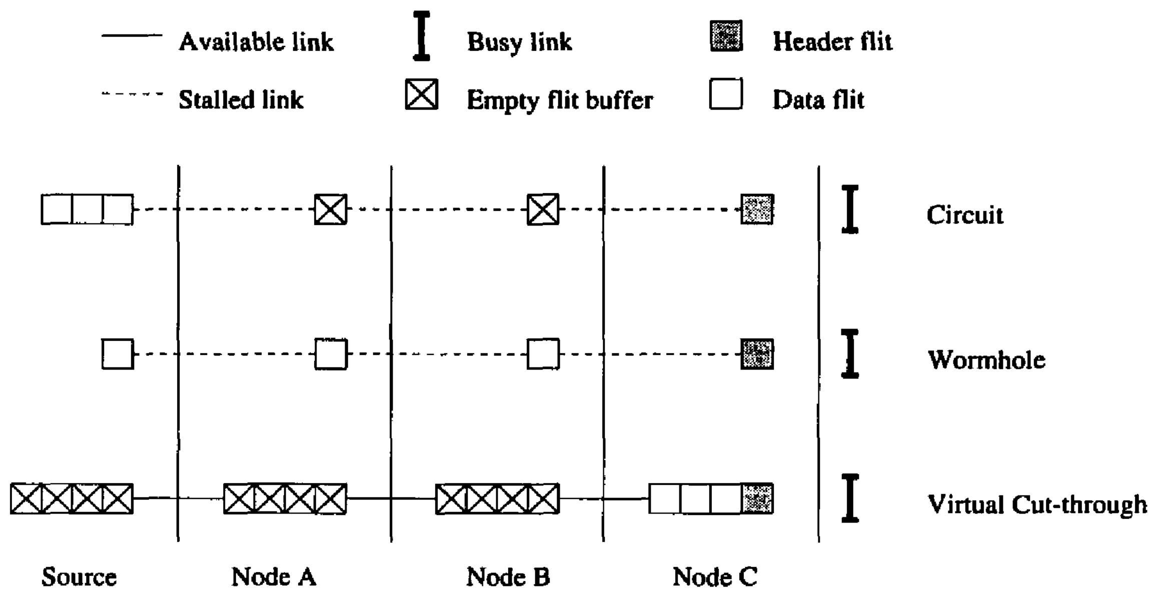


Figure 2.2: Comparison of cut-through switching schemes.

A, and node B for use by other packets.

One potential problem for wormhole-switched networks is *deadlock*, where packets may be blocked forever in the network. This occurs when two or more packets in the network form a cycle where each packet is holding resources (channels) and waiting for others. Typically, networks deal with deadlock by either *preventing* it or *recovering* from it. The most common solution is to prevent deadlock by restricting the routing algorithm to prevent cycles from forming [11].

Recently, researchers have noted that deadlocks are an uncommon event, and the common technique of restricting routing to prevent deadlock may often require more resources than recovering from the occasional deadlock [34, 70]. Recovery strategies typically use some form of a timeout to detect when a packet is deadlocked, and then either discard the packet or misroute it.

2.1.3 Virtual Channels

Wormhole-switched networks have also pioneered the use of *virtual channels*, where several packets may simultaneously share a single physical link [11]. Figure 2.3(a) shows a link with one channel, where link bandwidth is allocated on a per-packet basis; if the link supports virtual channels (such as in Figure 2.3(b)), its bandwidth is allocated on a per-flit basis. Thus, multiple packets may be transiting the physical link at any one time. Virtual

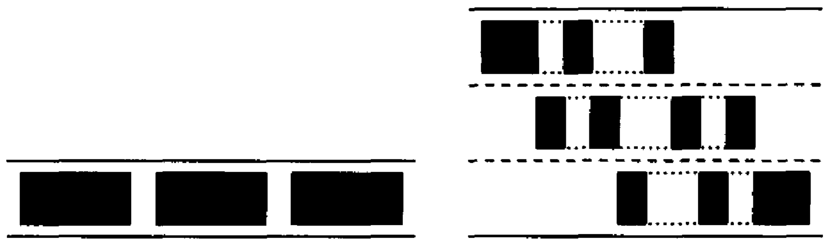


Figure 2.3: Virtual channels.

channels have been used for several purposes:

Deadlock prevention: By adding virtual channels (and thus resources) to the network, minimal-path routing schemes may create a network resource (channel) graph without cycles [11]. This is sufficient to prevent deadlock in wormhole-switched networks.

Increasing throughput: By allowing packets to use link bandwidth that would otherwise be wasted by blocked packets [12]. In Figure 2.3(a), if the first packet is stalled waiting for another channel, it idles the link and prevents the other packets from proceeding. The virtual channels in Figure 2.3(b) allow the other packets to bypass the blocked packet and thus reclaim the link bandwidth.

Increasing adaptivity: Many routing schemes use virtual channels to increase the degree of adaptivity within the network [4].

Partitioning traffic: By partitioning traffic classes with disparate characteristics or requirements onto separate virtual networks, interference between the classes can be reduced. In addition, the network policies for each virtual network can be tuned to the needs of each class [36, 58].

When a router provides multiple virtual channels over a physical link, it must also determine an *arbitration* policy that governs which channel may access the link. The simplest schemes are *first-come first-served* and *round-robin*, but these are not always preferred. Consequently, some routers implement some form of *fixed priority* arbitration, where link bandwidth can be allocated to high-priority messages at the expense of low-priority messages.

2.2 Motivation for Flexible Routing and Switching

Existing multicomputer routers employ a variety of header formats, routing-switching algorithms, and channel allocation policies, each with its own strengths and weaknesses. Table 2.1 compares the major policies (routing, switching, virtual channels, queuing, topology) of several existing multicomputer routers. From this table, we can note that existing solutions are very diverse; no one scheme exists that is always chosen over the others. In addition, existing solutions only support a single set of policies, or in some cases, a limited set of policies.

Numerous researchers have examined the relative performance of these various routing and switching schemes, and found that the best performer varies according to the application's traffic pattern [4, 13, 23, 35, 36, 70]. For example, adaptive schemes that generally improve performance by routing around congestion may actually increase congestion under certain traffic loads [23], and even when they reduce end-to-end delay out-of-order packet arrival can complicate protocol processing at the receiving node [32]. Opportunities for adaptive routing depend on the topology and the distance a packet must travel. Similarly, wormhole switching achieves low latency without requiring packet buffers, but virtual cut-through and packet switching may achieve better throughput at high loads. Packet size also impacts network performance, since inter-node communication often consists of large data transfers, coupled with small request and acknowledgement packets [9]. A flexible router can accommodate this mixture by having long packets use wormhole switching to reduce buffer space requirements, while allowing short packets to use virtual cut-through switching to reduce network contention [36]. The network topology also influences the relative value of particular routing schemes: chaotic routing, for example, has been found to be marginally better than oblivious schemes in mesh networks but significantly better in torus topologies [3].

Based on this, we wish to provide support for a wide range of routing, switching, and selection schemes, allowing the system to select which is appropriate for its needs. At the same time, however, the performance costs of this flexibility should not exceed the gains from this flexibility. Consequently, we will now examine how multicomputer networks implement these policies to decide how to provide the desired flexibility. Although multicomputer networks implement routing and switching in many different ways, every router proceeds

Router	Topology	Routing	V-chans	Arbitration
Mesh routing chip [61]	2-D mesh	e-cube	1	—
Cray T3D router [49]	3-D torus	e-cube	4	input
Message-driven processor [15]	3-D mesh	e-cube	2	priority
Torus routing chip [10]	2-D torus	e-cube	2	fair
iWarp router [53]	2-D torus	static	varies	fair
Reliable router [14]	2-D mesh	adaptive	5	fair
IMS C104 switch [68]	flexible	interval	1	—
Hnet switch [63]	flexible	table	1	—
SP1 Vulcan [65]	multistage	static	1	—
Mosaic [61]	2-D mesh	e-cube	1	—

(a) Routers that use wormhole switching

Router	Address	Topology	Routing	Queueing
Mayfly Post-office [19]	offset	hexagonal	adaptive	shared output
Chaos router [38]	offset	2-D torus	non-minimal	shared pool
DEC AN1 switch [50]	table look-up	flexible	adaptive	input
ComCoBB [66]	table	flexible	adaptive	part. input
Arctic routing chip [6]	source-list	fat-tree	static	input pools
S-connect [48]	table	variable	adaptive	central
POSTECH [40]	offset	torus	adaptive	off-chip

(b) Routers that use virtual cut-through switching

Table 2.1: Feature comparison of various multicomputer routers.

through common stages in servicing a packet. With careful inclusion of flexible hardware for each stage, multicomputer routers can support diverse network policies at a reasonable cost and speed.

2.2.1 Header Parsing

When a packet arrives from a host injection port or an incoming link, the router parses the header bytes to make a routing-switching decision. Multicomputers can employ a wide variety of header formats and sizes, depending on the routing algorithm and the underlying network topology. Under source-list routing, the header explicitly specifies the packet's entire path through the network, while distributed routing algorithms allow intermediate nodes to compute a routing decision based on the packet header and local conditions. This routing header can include the physical or relative address of the packet's destination node; for example, offset-based routing represents the destination by its distance along each dimension of the network. Alternately, the packet header can include a connection identifier that corresponds to a pre-established route, represented by tables in the intermediate routers.

To support efficient multicast routing, packet headers can specify a *set* of nodes or routes in a variety of formats, depending on the number and location of the destinations. For example, the header can represent destinations with an ordered list or a bit mask, as well as more concise encodings that identify specific regions of the network or a tree of paths from the source node [7]. In addition to destination information, the header may include the packet type or priority to distinguish between various classes of traffic, such as control or data communication, which may employ different network policies. Although most multicomputer networks limit themselves to a single header definition, flexible hardware support for logical and arithmetic operations would enable routers to manipulate almost any header format.

2.2.2 Routing Computation

After parsing the incoming header, the router generates one or more possible directions for the packet to travel, based on the header fields and the routing-switching algorithm. For example, given the current node and the packet's destination, the router could compute which outgoing links lie on a shortest-path route. When the packet can select from multiple directions, a selection function must be used to order the output links. Communication performance often depends on this ordering. Most mesh routers rank links in dimension order, giving preference to outgoing links in lower dimensions of the network. Alternately, the router could order the links according to how much farther the packet must travel in each direction, improving a packet's chance of considering multiple outgoing links at future nodes in its route [2, 19]. To reduce network congestion and balance traffic load, the router could favor links with fewer busy virtual channels [13].

Depending on the routing-switching algorithm, the router may also limit which virtual channels are available to an incoming packet. For example, many wormhole routing algorithms guarantee deadlock-avoidance properties by limiting which outgoing virtual channels a packet can consider [11, 46], based on the incoming virtual channel. For a given number of virtual channels, there is a variety of deadlock-free wormhole routing algorithms that differ in how they restrict access to outgoing links and virtual channels. By improving link throughput or increasing routing adaptivity, these various algorithms can significantly enhance communication performance, depending on the network traffic pattern [4, 56]. With flexible hardware for assigning outgoing virtual channels, multicomputer routers could tailor wormhole routing algorithms to application communication patterns. In addition, by restricting access to certain virtual channels, routers can establish virtual networks that dedicate link and buffer resources to different traffic classes [15, 36, 53, 59].

2.2.3 Switching

The router eventually commits to a routing-switching decision for each packet, based on the candidate virtual channels and prevailing network conditions. An ordered list of virtual channels encapsulates the routing options generated by the algorithm, while the switching policy determines whether the router should buffer, stall, drop, or forward the incoming packet. By viewing virtual channels as individually reservable resources, a router can implement a wide variety of routing-switching schemes through flexible control over channel reservation policies. For example, a wormhole or virtual cut-through router attempts to reserve access to the highest-ranked idle virtual channel, to immediately forward an incoming packet to the next node in its route. If all of the candidate virtual channels are reserved, virtual cut-through switching buffers the incoming packet, while a wormhole router blocks the packet until one of the channels becomes available.

2.2.4 Channel Allocation and Arbitration

When multiple packets await access to the same virtual channel, the router imposes a scheduling policy to select the next outgoing packet for transmission. To reduce implementation complexity, most existing routers assign virtual channels on a first-come first-serve basis; however, priority-based schemes have the potential to provide predictable performance and isolate certain classes of traffic [42, 44, 59, 69]. Separate from channel assignment strategy, the router arbitrates amongst the busy virtual channels to assign link bandwidth at the flit level. In addition to random or round-robin arbitration, routers can base bandwidth allocation on packet deadlines or priority to further enhance predictability [12]. Flexible support for channel allocation and arbitration schemes would allow modern routers to handle a wide variety of applications with different performance requirements.

2.3 Summary

As we have seen in this chapter, existing multicomputer routers employ a variety of different header formats, routing-switching algorithms, and channel allocation policies. Instead of hardwiring network policies in the router design, Chapter 3 introduces a router architecture that incorporates small, programmable devices for processing incoming packet headers. This architecture permits multicomputer networks to handle a wide variety of header formats, routing algorithms, and switching schemes at a reasonable cost.

CHAPTER 3

Microcontroller-Based Flexibility

I don't know if I can face this mess.

— Shoe, in Jeff MacNelly's *Shoe*

As shown in Section 2.2, current routers typically implement only a single, fixed policy for routing and switching packets. This prevents applications from tailoring the network to suit their current needs. Instead of hardwiring network policies into the router design, however, this chapter introduces a router architecture that incorporates small, programmable devices for processing incoming packet headers. This architecture permits multicomputer networks to handle a wide variety of header formats, routing algorithms, and switching schemes at a reasonable cost.

A processor-based programmable architecture offers maximum flexibility for header parsing and route computation. In addition, expressing routing and switching schemes as assembly language instructions simplifies programming new schemes. Implementing this flexibility implies that every incoming packet will require fast, efficient handling by some highly programmable and flexible *routing engine*. At the same time, however, this should be achieved at a reasonable cost, both in hardware and performance.

3.1 An Architecture for a Flexible Router

Figure 3.1 shows the architecture of a flexible router with $2L$ unidirectional links paired to form L bidirectional links and C virtual channels per link. The channels are labeled using the notation $\{\mathrm{I/O}\}_{\mathrm{link,channel}}$; thus I_{lc} denotes virtual channel c of the incoming link l. To transmit a packet through an outgoing channel, the "master" (either a reception channel

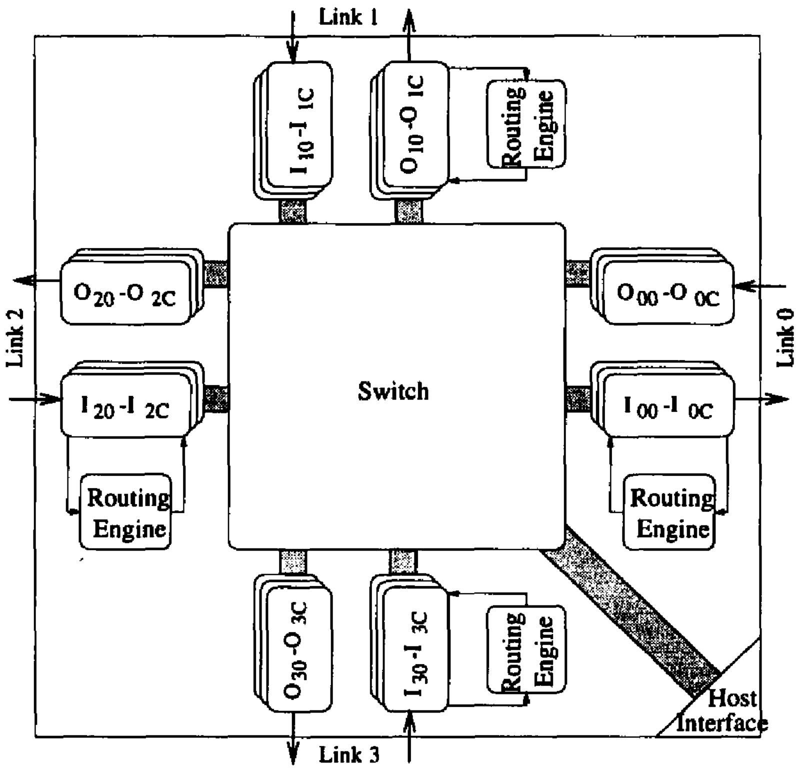


Figure 3.1: Flexible router architecture for $L = 4$, showing the location of the routing engines.

or the host interface) must first obtain a *reservation* of the "slave" channel from a central controller on the node. Once the desired slave has been reserved, the master may forward packet data to the slave through the switch. By providing programmable control over this reservation process for every packet entering the router, different routing and switching schemes may be implemented.

Packets enter the node either when injected by the local host or when received from a neighboring node by an incoming channel. For packets injected through the host interface, the routing policies can be provided by the host, which selects a transmission channel to transfer the packet to a neighboring node. When the packet is received at the neighboring node, the associated incoming channel is then responsible for parsing the header, computing a route, and forwarding the packet.

The router could dedicate a routing engine to every incoming channel to completely decouple the processing of packets from each other. This approach, however, incurs significant hardware costs for implementing $L \times C$ routing engines, each with its own control store, registers, and control logic. This replication of resources is also somewhat unnecessary, as the stages that require a routing engine usually constitute only a fraction of the time required to receive and forward a packet.

By sharing each routing engine amongst several channels, utilization of the routing engines may be increased while requiring fewer routing engines. Sharing a routing engine between channels, however, means that channels need to compete for the routing engine. As the ratio of channels to routing engines increases, the likelihood and number of channels blocked while waiting for access to the routing engine increases. A shared routing engine is also more complex, as it should be capable of implementing different schemes for each channel it is supporting.

It may be noted, however, that several of these problems are alleviated if a shared routing engine only handles packets for a single physical link. Since the link itself serializes the arrival of packet headers, the arbitration scheme for access to the routing engine does not need to take simultaneous packet arrivals into account. At the same time, there are many commonalities among routing and switching schemes written for the same physical link; these shared code segments do not need to be replicated, reducing the size of the control store necessary. Finally, in a VLSI implementation of the router, grouping a routing engine with the hardware required for handling a reception link enables creation of a single module for handling all routing and switching on a link; this module may then be replicated for every incoming link, resulting in an efficient use of space and reduction in design time.

Based on these observations, we have chosen to dedicate a single routing engine to each incoming physical link. This reduces the total number of routing engines and increases their utilization, while alleviating many of the problems of a centralized architecture.

3.2 NIRX Architecture

To share the routing engine efficiently between several virtual channels, time-consuming tasks should be hardwired into the channel functionality whenever feasible. For simplicity, we will refer to the incoming channel as the NIRX (*network interface receiver*). Thus, the NIRX is responsible for all of the mundane aspects of data forwarding, as well as intranode and internode flow control. The routing engine, on the other hand, provides header parsing, route computation, and switching for each packets before "returning" them

Field	Function
`address`	Bit mask of destination channels $\langle \text{host}, O_{L-1,C-1}..O_{00}\rangle$
`data`	Modified packet header

(a) Main components of the routing primitive

Flag	Function
`make_reservations`	Signals whether channels are already reserved.
`multicast`	Selects either multicast (all of) or unicast (only one of) semantics

(b) Routing primitive control flags

Table 3.1: Major components of the routing primitive.

to the channels. After reserving the desired outgoing channel(s), the routing engine returns control of the packet to the NIRX by issuing a *routing primitive* that configures the incoming channel's state machine for forwarding the packet. The routing primitive structure, shown in Table 3.1, contains the modified packet header and an $L \times C + 1$-bit *address mask* specifying the reserved O_{lc}.

This interaction is shown in Figure 3.2, which presents a simplified state machine implementing the NIRX. Initially, the NIRX forwards incoming data to the routing engine. This process continues until the routing engine issues a routing primitive to the NIRX. After the arrival of the routing primitive, if the `make reservation` flag is false, the NIRX will shift to the `connected/data` state and begin forwarding the body of the packet. The connection is terminated by the arrival of the last flit of the packet; forwarding this flit frees the slave NITXs and resets the NIRX to its idle state.

This interface, however, is complicated by wormhole-switched packets, which often need to stall for an indefinite period of time while waiting to reserve a channel. If the routing engine implements this stall, it will be unable to service the other NIRXs in a timely manner. To prevent this, the routing engine offloads the actual stalling to the NIRX. As pointed out in Section 2.2, routing a packet requires handling an ordered list of candidate channels, which is expensive and impractical to implement for every I_{lc}. A packet stalls, however, only when *no* candidate is free; thus, the NIRX can simply reserve the first available candidate channel. This greatly simplifies the implementation of the NIRX — the routing primitive

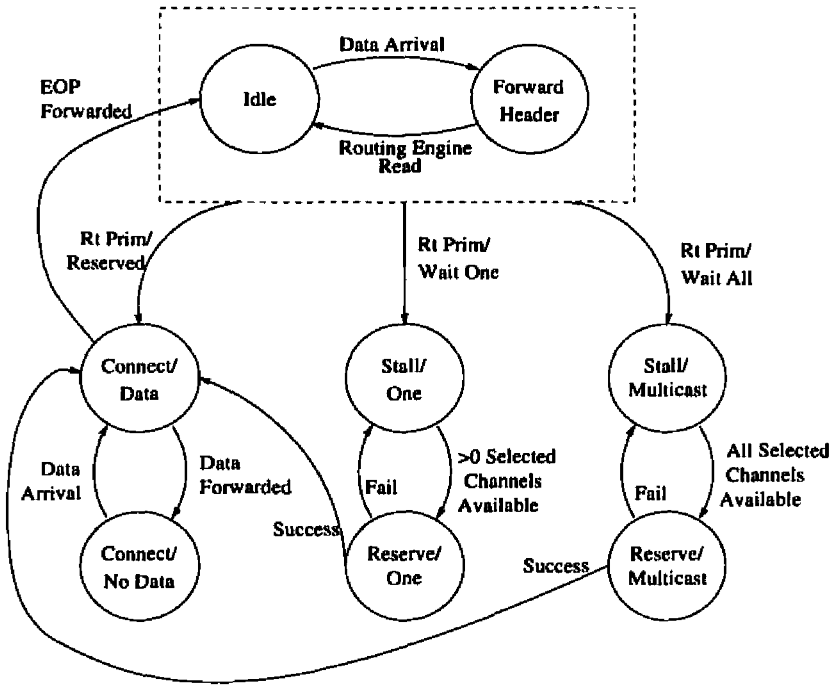


Figure 3.2: Simplified state machine for a reception channel.

simply specifies the set M of candidate virtual channels, but need not order them[1]. A single boolean flag, `make_reservation`, tells the NIRX that it needs to reserve one or more of the channels selected by the address mask. The address mask S used for switch access is defined by the function $S = \text{msb}(R \cap \overline{M})$, where R is the current reservation status mask. When the `make_reservation` flag is set in a routing primitive, the NIRX moves to its `reserve/one` state; when it detects that a selected O_{lc} is free, it attempts to reserve only that channel. A successful reservation attempt moves the NIRX to the connected state, while a failed attempt returns the NIRX to the `reserve/one` state. The NIRX checks the `multicast` flag to determine its next state.

Multicast routing schemes that utilize wormhole switching present a further complication; oftentimes, only some of the desired channels will be available while the routing engine handles the packet. Once again, having the NIRX to wait on the entire set of channels specified by the address mask allows the routing engine to offload this stalling to the NIRX at a reasonable cost. Multicast semantics are triggered by the `multicast` flag in the routing primitive, which shifts the NIRX to the `stall/multicast` state. This state operates in a similar fashion to `reserve/one`, but will only attempt a reservation when *all* selected O_{lc}

[1]In the unlikely event that multiple selected NITXs are available, a fixed priority scheme orders them.

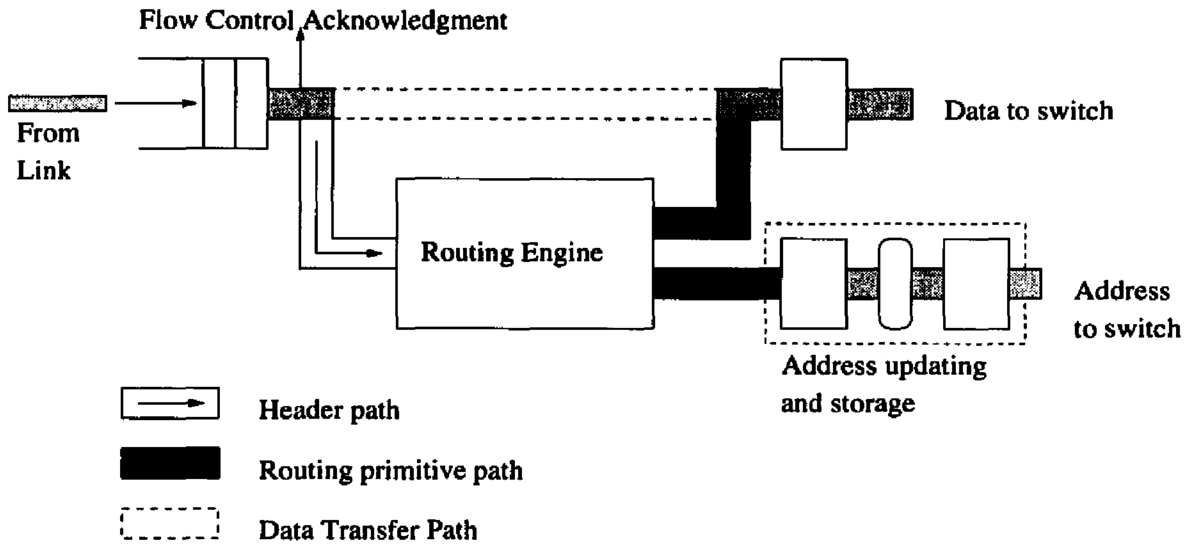


Figure 3.3: NIRX architecture.

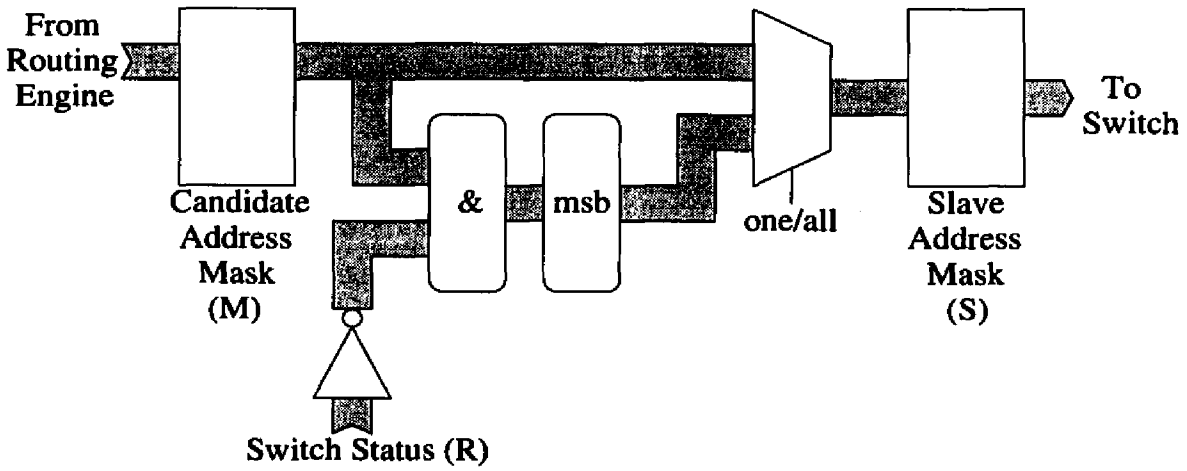


Figure 3.4: Logic for checking and updating address masks.

are available.

To summarize the operation of the NIRX, Figure 3.3 shows the paths taken by data through the NIRX and the routing engine. These paths are shown in chronological order: the header being forwarded to the routing engine is eventually followed by the return of the routing header as part of the routing primitive. Once the primitive has been issued, data is directly forwarded to the switch, bypassing the routing engine. Similarly, Figure 3.4 depicts

Command	Function
`alu`	boolean/arithmetic operation
`ldc`	load constant into register
`xfer`	copy register contents
`flag`	set, clear, and copy flags
`jump`	conditional branch
`return`	return from subroutine
`go nirx`	trigger routing primitive
`go ctbus`	trigger CTBUS access
`wait`	three-way, blocking branch

Table 3.2: Routing engine instruction set.

the logic used to compute the switch mask S from the candidate channel mask M.

3.3 Routing Engine Architecture

While the NIRX is responsible for handling data transfer, flow control, and executing some switching functions, the routing engine implements the header parsing and route computations for every incoming packet. To achieve this, the routing engine is subdivided into several major functional blocks, as shown in Figure 3.5. The data input and output modules provide the interface to the NIRXs, while the bulk of the computations are performed by the CPU block, which consists of a small, 8-bit microcontroller with a 256-instruction[2] control store for microprograms. Within the CPU block, integer-operation support is provided by an 8-bit integer ALU that implements addition, subtraction, and boolean operations; a 16-byte register file provides storage for constants and intermediate computations. The host uses the control interface to download the microprograms during system initialization and to adjust microcode operation at run time through the notification FIFOs. The switch interface, on the other hand, allows the routing engine to read the current reservation status of the NITXs without accessing the switch. Table 3.2 shows the major instructions supported by the CPU block.

The data input and output modules constitute the interface between the routing engine and the NIRXs. The interfaces are designed to move data and control instructions between the routing engine and the NIRXs under high-level software control, while requiring a minimal number of instructions and cycles. The data input module is also responsible for transparently configuring the routing engine; this configuration sets internal flags within

[2]Each instruction is 20-bits wide. Appendix B discusses the instruction format in more detail.

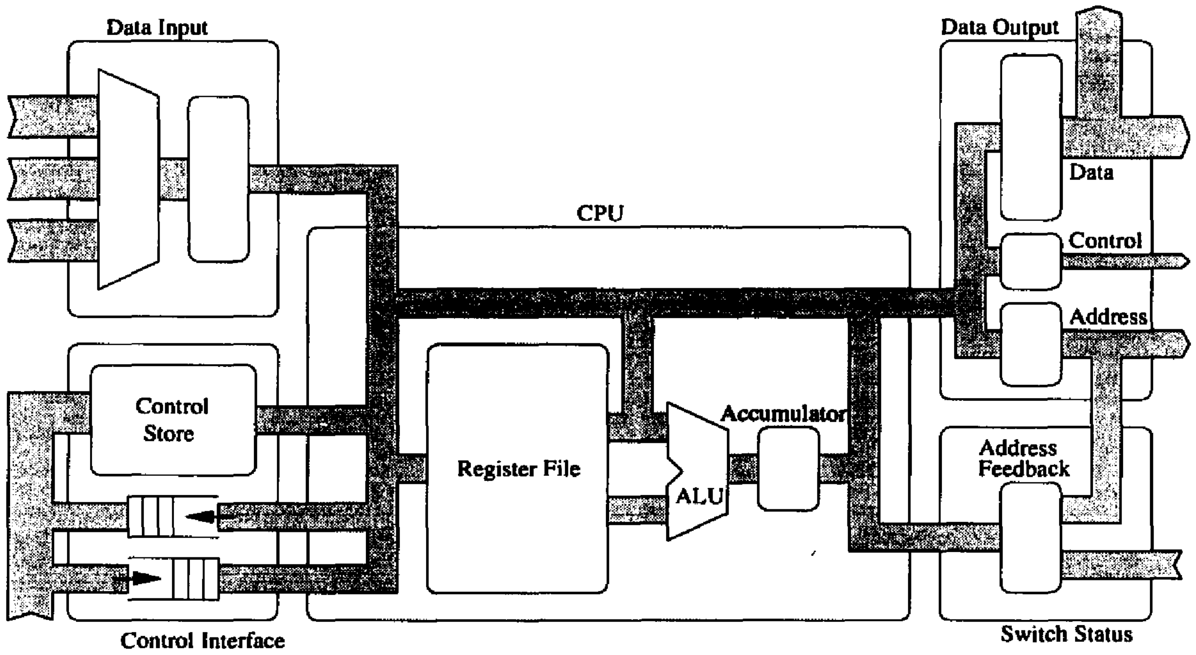


Figure 3.5: Routing engine architecture.

the routing engine that allow instructions to execute without regard to the identity of the currently selected channel, encouraging code reuse.

The data input module is controlled by a special `wait` instruction issued by the CPU block[3]. The `wait` instruction determines the priority of particular NIRXs. When a NIRX signals that it has header data, the data input module transfers the data from the NIRX to the local (`nid`) registers. Simultaneously, the CPU block will jump to the appropriate microcode for the selected NIRX, and begin executing it.

The data output module has two primary functions: issuing routing primitives to the selected NIRX and accessing the switch to reserve NITXs. After the header is processed and a routing decision has been made, the data output module is used to transfer a routing primitive (Table 3.5) to the virtual channel. Sharing these registers between the NIRX interface and the switch interface reduces the total number of registers within the routing engine and also reduces data movement; typically, the address mask used in a reservation

[3]The format of the wait instruction is as follows:

```
wait <chan>[,trap0(<chan_list>)][,trap1(<chan_list>)];
```

where priority runs from right to left. If no selected channel has data, the instruction stalls. If a channel selected by the `trap1` list has header data, the data is transferred into the inbound data registers and the routing engine jumps to the location indicated by a special `trap1` register. The other trap works similarly; if the first channel indicated is "selected", the routing engine transfers the data and proceeds to the next sequential instruction.

```
init:
  ldc c0_handler, trap0;
  ldc c1_handler, trap1;
/* waits for a packet header */
  wait c2, trap0(c1), trap1(c0);
c2_handler:
  < header parsing for channel 2 >
  < route computation >
  < channel reservation >
  ldc rtp_flags, ctctl, go rtp;
  jump true, init;

c1_handler:
  < header parsing for channel 1 >
  < route computation >
  < channel reservation >
  ldc rtp_flags, ctctl, go rtp;
  jump true, init;

c0_handler:
  < header parsing for channel 0 >
  < route computation >
  < channel reservation >
  ldc rtp_flags, ctctl, go rtp;
  jump true, init;
```

Figure 3.6: Typical microprogram structure.

Category	Functions	Comments
Obtaining header	transfer header	
	acknowledge header	
	arbitrate channels	multiple headers may be waiting
Control flow	code selection	schemes are often channel-dependent
Routing primitive	data transfer	
	signal channel	almost always preceded by transfer op
Switch access	data transfer	
	async control of access	almost always preceded by transfer op

Table 3.3: Instruction capabilities necessary for external interface control.

access to the switch is identical to that used for the routing primitive.

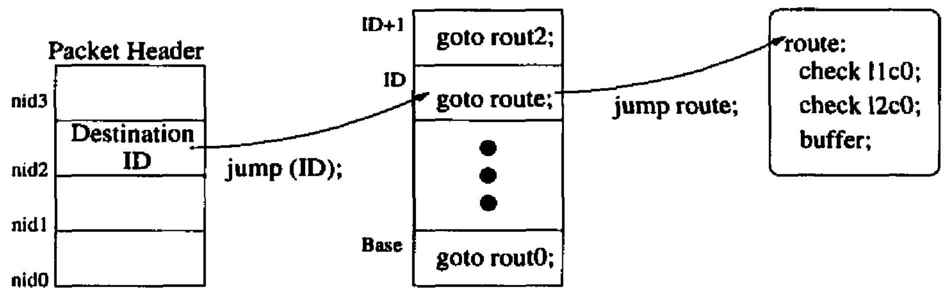
Operation of the data output module is controlled by the CPU block through special flags on data transfer instructions. This simplifies and speeds the microcode by eliminating a separate strobe instruction. The `go rtp` suffix issues the routing primitive to the NIRX, while the `go ctbus` suffix triggers a small finite state machine within the data output module that controls switch access. The remainder of the switch access is controlled by the data output module, freeing the CPU for other computations.

3.3.1 Header Parsing and Route Computation

The remaining modules of the routing engine constitute the "CPU" of the routing engine. The functionality provided by the CPU block is determined by several goals: minimizing microprogram size and execution time, coupled with the need to implement a wide range of routing and switching algorithms. Consequently, most of the operators provided by the CPU block are determined by its responsibility for implementing header parsing and route computation. Table 3.4 summarizes these functions.

One basic requirement for implementing any algorithm is support for control flow; in the CPU, this is provided by the `jump` instruction, which can alter the instruction sequence based on boolean flags such as the ALU's `zero` and `carry` bits as well as reservation status flags for the NITXs. Several user-controlled flags are also available for temporary storage of boolean conditions, and may be set, cleared, or loaded via the `flag` instruction. For example, an algorithm may save the result of a bit-mask or comparison operation on a routing header; later, a `jump` instruction could branch based on these condition bits. When a `jump` instruction includes the `save` qualifier, the routing engine stores the address of the next microinstruction, so the micro-sequencer can `return` to the main instruction flow later; this restricted implementation of subroutines reduces microprogram size by enabling code reuse.

The CPU block's first task in routing a packet is to parse the packet's routing header to determine the destination of the packet, its type, priority, and any other necessary information. As described in Section 2.2, packet headers may represent addresses in several ways. Distributed routing schemes that compute a routing decision at each node often employ offset-based addressing; updating and checking these requires integer and boolean operators, as well as the ability to check the carry and "equal-to-zero" flags after these



(a) Graphical depiction

```
          wait ni0, trap0(ni1,ni2);
          xfer nid2, trap1;
          jump (trap1);
      (nid2):
          jump true, use_these_channels;
      use_these_channels:
          < check channels in desired order >
          < else block on all suitable channels >
          jump true, init;
```

(b) Pseudo-code

Figure 3.7: Implementation of a table-lookup routing algorithm.

operations. Routing schemes that use the physical address of the destination may also use these functions to determine how to route packets.

After parsing a packet's header, the routing engine may need to select one or more channels for the packet to traverse. The functions required at this stage vary according to the addressing scheme: offset-based addressing schemes such as in Appendix 3.5 require integer addition and subtraction, along with comparison operations. The route computation may also be trivial — source-list routed schemes, for example, often carry an address mask specifying the next link(s) to traverse.

Table-based routing: Although offset-based routing algorithms are suitable for many network topologies, other topologies (especially irregular ones) may require more flexible routing schemes. To efficiently handle these topologies, routing tables are often used: each packet carries a destination address in its header. To route the packet, the routing engine simply looks up the destination in a table; the table entry instructs the routing engine on which link(s) to forward the packet. Figure 3.7 shows a table-lookup routing scheme, where a packet's destination is encoded in the nid2 field of the routing header. Rather than

Category	Function	Comments
Integer operations	addition subtraction increment decrement carry flag	neccssary for offset-based schemes
Comparison operations	less than greater than equal to check for zero	used by almost all schemes
Boolean operations	AND OR XOR NOT	bitwise and logical needed; useful for compacting info in routing header
Data transfer	load constant register copy variable storage	
Control flow	conditional branch indirect jump jump to subroutine	 allows table-lookups supports code reuse

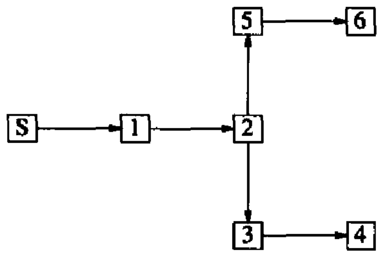
Table 3.4: Instruction capabilities for header parsing and route computation.

providing a separate memory for the routing table, however, the program uses an indirect jump (line 3) to jump into a "table" in the control store; the table entry then directs the routing engine to the appropriate routine for routing the packet. This approach has several advantages: the existing control store is used for the table, avoiding the cost of a separate RAM. In addition, having the table entries call a routing subroutine permits great flexibility in specifying channel orderings.

Multicast routing: The routing engine allows implementation of a broad spectrum of multicast routing algorithms, under both wormhole and virtual cut-through switching. Fig-

Registers	**Function**
`ctd3 — ctd0`	new header flit
`ctaddr1`	selects memory, $O_{20}..O_{32}$
`ctaddr1`	selects $O_{00}..O_{12}$
`ctctl` resvd all crcflg	boolean control flags: slaves already reserved reserve all/one of slaves include word in CRC

Table 3.5: Routing primitive as stored in registers.



Node 1	+X
Node 2	+Y,-Y
Node 3	+X
Node 4	Buffer
Node 5	+X
Node 6	Buffer

(a) Multicast path from S to nodes 4 and 6 (b) Routing header

Figure 3.8: Multicast route and routing header.

```
    < initialization code >
    ldc our_node_id, reg2;

  strip_header:
    wait ni2, trap0(ni1), trap1(ni0);
    alu nid3 - reg2;
    jump ~zero, strip_header;

  route_packet:
    alu nid2;
    jump ~zero, wh_routing;
    < code for virtual cut-through >
  wh_routing:
    < code for wormhole >
```

Figure 3.9: Example of parsing a multicast source-list header.

ure 3.9 shows a sample multicast routing algorithm that employs either wormhole or virtual cut-through switching, depending on the packet header. Each packet includes a tree of one-word routing headers to encode the nodes in the tree and the routing-switching scheme at each hop in the route, as shown in Figure 3.8. As the packet arrives at a node, the routing engine discards header words until reaching a word tagged with its node identifier in **nid3**. The next byte of the header word selects between virtual cut-through and wormhole switching, while the last two bytes determine where the receiver should forward the remainder of the incoming packet. Under wormhole switching, the receiver reserves *all* of the selected slaves devices before forwarding the packet, whereas the virtual cut-through scheme immediately directs the incoming packet to any *available* slaves. Section 3.5.2 describes this algorithm in more detail.

Category	Function	Comments
Reservation status	boolean flags	most efficient method for most algs, when code specifies destination channels
Address mask	check for conflicts update available channels check for null mask	necessary for source-list routed schemes ensure updated mask is not null

Table 3.6: Instruction capabilities for switching decisions.

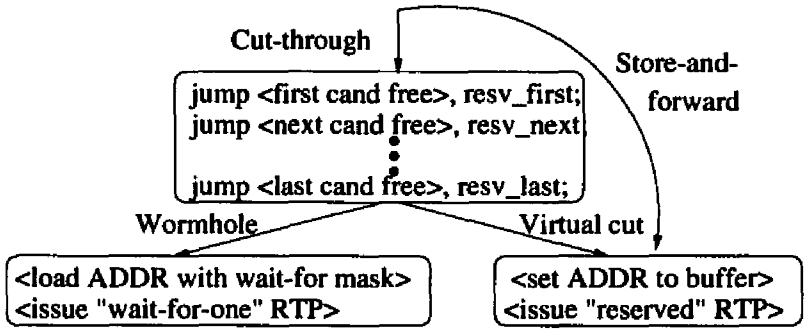

Cut-through
Store-and-forward
jump <first cand free>, resv_first;
jump <next cand free>, resv_next
jump <last cand free>, resv_last;
Wormhole
Virtual cut
<load ADDR with wait-for mask>
<issue "wait-for-one" RTP>
<set ADDR to buffer>
<issue "reserved" RTP>

Figure 3.10: Template for a "normal" switching operation.

3.3.2 Switching

Once the routing engine has committed to a routing decision for a packet, it needs to either reserve one or more NITXs or offload the packet to the NIRX. If this switching decision is combined with the route computation, the reservation status flags are typically used to trigger appropriate branches. Figure 3.10 shows a template for the switching operation, where the microprogram checks candidate channels in descending order of priority; if no candidate is available, the switching scheme determines the form of routing primitive issued to the NIRX.

Unfortunately, parsing a bit-mask of one or more channels from a source-list routing scheme is complicated and time-consuming. Accordingly, the *switch status* module allows address masks to be checked with a single operation. As shown in Table 3.6, two address mask operations are provided: a conflict check and an *as-many-of* update. Since the *as-many-of* check can potentially return a null mask, a flag for this condition is also provided. Figure 3.11 shows how the source-list multicast scheme described above employs the feedback registers to determine which selected channels are available, so that it may reserve *as many of* them as possible. By reading the feedback registers and updating the address

```
get_update:
  xfer nid1, ctaddr1;
  xfer nid0, ctaddr0;
  xfer ctfb1, reg1; // Read address
  xfer ctfb0, reg0; // feedback reg.
  xfer reg1, ctaddr1;
  xfer reg0, ctaddr0, go ctbus;
  jump ~ack, get_update;
  ldc rtp_resvd_nocrc, ctctl, go nirx;
  jump true, init;
```

Figure 3.11: Address feedback example.

registers accordingly, the entire address can be updated in just four cycles. If the reservation attempt fails because some other packet reserved one or more of the channels, the process may be repeated again. Once the routing engine successfully reserves the virtual channels, it triggers a routing primitive to relinquish control of the packet to the channel. Another sample microprogram using the address checking module is shown in Section 3.5.3.

3.3.3 Flexible Use of Virtual Channels

Although many microprograms implement the same routing-switching scheme on each virtual channel, the routing engine may have different microcode routines for each NIRX. The virtual channels provided by the receiver module design may be used to to implement a variety of deadlock-free wormhole routing algorithms. These channels may also be used to improve the throughput of static routing algorithms or to permit extra flexibility in adaptive schemes [12]. Either approach can significantly improve communication performance, depending on the network traffic pattern [56, 57]. Through the routing engines, the application (or the compiler) may configure the network to use its virtual channels for adaptive routing or to reduce contention between packets traveling on the same link.

The routing engine's flexibility also allows traffic partitioning — assigning different applications or traffic types to separate virtual networks with distinct routing-switching policies. For example, real-time messages could use packet switching and static routing for predictable performance, while best-effort packets improve their average latency through cut-through switching and adaptive routing; carrying these two types of traffic on different virtual channels allows real-time communication to coexist with best-effort packets without sacrificing the performance of either class [58]. Similarly, the router could separate short control messages and long data packets onto different virtual channels, perhaps with different switching policies [35, 36, 24]. These options enhance user control over the underlying interconnection network to improve application performance.

3.4 An Adaptive Wormhole Microprogram

This section shows an example of a complete microprogram, implementing adaptive minimal-path routing algorithm for regular networks, such as the square mesh. Unlike dimension-ordered algorithms that prefer to route packets entirely in one dimension before using others, this algorithm tries to retain adaptivity by routing packets "diagonally" through the network.

3.4.1 Algorithm Overview

To avoid deadlock, the virtual channels on each link are partitioned into the "low" and "high" channels: packets received on a low channel (channel 0) must use oblivious dimension-ordered routing. Packets using the high channels (channels 1 and 2) use adaptive, diagonal-biased routing: e.g., the routing engine tries to reduce the offset with the largest magnitude first. A pseudo-code implementation of the adaptive algorithm is shown in Figure 3.12.

3.4.2 Microcode

Figure 3.14 shows a microcode implementation of this algorithm. The complete program requires 143 instructions; no execution path, however, will actually access all of these instructions. The minimal time to route the packet is 7 instructions in 8 clock cycles for a packet arriving on channel 0 with $x < 0$. Packets using adaptive routing take longer to route; for example, a packet with $x < 0$ and $y == 0$ may be routed in 19 clock cycles, not including the time required to access the switch to reserve a transmitter.

Initialization and header wait: The `init` routine initializes internal registers and waits for a packet header to arrive. For example, line 2 sets a bit mask for checking the sign of the y offset, used later to decide if the packet should travel in the positive or negative y-direction. The `init` routine then sets the `trap` registers for the subsequent `wait` instruction; the

```
x = x + 1
if ((x != 0) && (y > 0)) then
    if (abs(x) > abs(y)) then
        route (+x, -y)
    else
        route (-y, +x)
else if ((x != 0) && (y < 0)) then
    if (abs(x) > abs(y)) then
        route (+x, +y)
    else
        route (+y, +x)
else if ((x != 0) && (y == 0)) then
    route (+x)
else if ((x == 0) && (y > 0)) then
    route (-y)
else if ((x == 0) && (y < 0)) then
    route (+y)
else /* x == 0 && y == 0 */
    buffer;
```

Figure 3.12: Pseudo-code implementation of diagonal-biased minimal-path adaptive routing algorithm

high channels (I_{21} and I_{22}) use the **adaptive** handler, while the low channel must use the dimension-ordered **dimorder** routine. Once a packet header arrives, the **wait** instruction latches the header word into the **nid** registers and branches to the appropriate routine. Finally, the y-offset is transferred to the output registers; since all later code branches eventually need this step, it is performed first to minimize code size.

Header parsing and route computation: The next step is to parse the packet's routing header (Figure 3.13) using the algorithm in Figure 3.12. The handler first increments the header's x offset to reflect the packet's previous hop before forwarding the header field to the ctd2 register in the data output module. The next step (if $x \neq 0$) is to check the y-offset. If both x and y are zero, they are compared to see which has the larger magnitude; this determines which dimension to access first.

Switching: Once the routing engine has determined which channels the packet can use, it begins checking which of these channels are free in the order of desirability; if a channel is free, it attempts to reserve it before checking the next. How this process is implemented varies according to the current needs of the program.

Packet buffering: The **buffer_packet** routine configures the NIRX to write to the memory interface. The routing engine sets the address mask to buffer the packet, and loads the ctctl field of the routing primitive. The **go nirx** qualifier in the ldc command triggers the NIRX to handle the remainder of packet reception. Consequently, the next instruction jumps to the initialization routine to reset the routing engine for the next arriving packet.

Reserving an NITX: The **get_l0c0** routine tries to acquire O_{00} by triggering a reservation command (**ctcmd_resv** was set in **init**); the other routines are similar. The routine first loads the address registers to select O_{00}, and triggers the reservation attempt (the command register was set during initialization) with the **go ctbus** directive. The success of the attempt is checked by the **return** instruction, which automatically blocks until the **RESV** attempt completes. If some other device reserves the NITX first, the **RESV** command can fail, as indicated by the **ack** flag; upon failure, the program returns to the calling routine. Otherwise, the routing engine configures the NIRX to execute the cut-through operation before jumping to init.

nid3	nid2	nid1	nid0
Pad	x offset range $-127\ldots127$	y offset range $-127\ldots127$	Pad

Figure 3.13: Adaptive minimal path routing header format

3.5 Routing-Switching Microprograms

The PRC's routing engines can be used to tailor routing-switching policies to application characteristics and performance requirements. Microprograms can parse a variety of header formats, making the routing engines more flexible than table-lookup schemes. The sample microprograms in this section are written for receiver module #2, which receives packets traveling in the $+x$ direction.

3.5.1 Table-Lookup Routing

Although offset-based routing algorithms are suitable for mesh networks, irregular network topologies require more flexible routing schemes. To efficiently handle irregular topologies, the PRC can implement table-lookup routing for networks with a limited number of nodes (less than 128 or so).

Algorithm overview: With this table-lookup scheme, every packet header contains a one-byte value specifying its destination. At every intermediate node, the routing engine uses this ID as an index into a table in its control store. Each table entry then provides a pointer to a procedure implementing the desired routing scheme.

Microcode: Figure 3.15 shows a table-lookup routing scheme, where a packet's destination is encoded in the `nid2` field of the routing header. To construct a routing decision, the routing engine uses the `wait` instruction (line 3) to "trap" to the address indicated in `nid2`; at this location, a jump instruction points to the code segment that implements the routing decision. In the sample microprogram, the `use_1011` routine considers link 0, followed by link 1, waiting for a free virtual channel if none are initially available.

Discussion: The table-lookup algorithm is quite simple and fast — the header processing time depends solely on how many channels the routing engine needs to check before finding one that is available. If the first channel checked is available, the routing engine consumes

```
init:
    ldc 0x80, reg0;
    ldc ctcmd_resv, ctctl;
    ldc dimorder, trap1;
    ldc adaptive, trap0;
/* now wait for a packet header */
handler:
    wait ni2, trap0(ni1), trap1(ni0);
    xfer nid1, ctd1;

/* check x offset */
    alu nid2 + 1;
    xfer acc, ctd2;
    jump zero, y_only;
/* check y offset */
    alu nid1;
    jump zero, x_only;
    alu nid1 & reg0;
    jump zero, y_is_neg;
/* compare magnitudes; x<0,y>0 */
    xfer nid1,reg1;
    alu nid2 + reg1;
    alu acc & reg0; /* check sign */
    jump zero, y_first;

/* check +x link */
    jump ~resvdl0c2, get_l0c2, link;
    jump ~resvdl0c1, get_l0c1, link;
/* check -y link */
    jump ~resvdl3c2, get_l3c2, link;
    jump ~resvdl3c1, get_l3c1, link;
/* check low channels */
    jump ~resvdl3c0, get_l3c0, link;
    jump ~resvdl0c0, get_l0c0, link;
block_l3l0:/* block on +x, -y links */
    ldc ctaddr_l0, ctaddr0;
    ldc ctaddr_l3, ctaddr1;
    ldc rtp_wait_one, ctctl, go rtp;
    jump true, init;

y_first:
/* check -y link */
    jump ~resvdl3c2, get_l3c2, link;
    jump ~resvdl3c1, get_l3c1, link;
/* check +x link */
    jump ~resvdl0c2, get_l0c2, link;
    jump ~resvdl0c1, get_l0c1, link;
/* check low channels */
    jump ~resvdl3c0, get_l3c0, link;
    jump ~resvdl0c0, get_l0c0, link;
    jump true, block_l3l0;
get_l0c0:
    ldc 0x0, ctaddr1;
    ldc ctaddr_l0c0, ctaddr0, go ctbus;
    return ~ack;
    ldc rtp_resvd_nocrc, ctctl, go rtp;
    jump true, init;

buffer_packet:
    ldc 0x0, ctaddr0;
    ldc  ctaddr_buff, ctaddr1;
    ldc  rtp_resvd_nocrc, ctctl, go nirx;
    jump true, init;

x_only:
    jump ~resvdl0c2, get_l0c2, link;
    jump ~resvdl0c1, get_l0c1, link;
    jump ~resvdl0c0, get_l0c0, link;
    ldc ctaddr_l0, ctaddr0; /* block l0 */
    ldc rtp_wait_one, ctctl, go rtp;
    jump true, init;

y_only:
    alu nid1;
    jump zero, buffer_packet;
    alu nid1 & reg0;
    jump zero, neg_y_only;
/* check +y link */
    jump ~resvdl1c2, get_l1c2, link;
    jump ~resvdl1c1, get_l1c1, link;
    jump ~resvdl1c0, get_l1c0, link;
    ldc ctaddr_l1, ctaddr0; /* block l1 */
    ldc rtp_wait_one, ctctl, go rtp;
    jump true, init;

dimorder:
    xfer nid1, ctd1;
    alu nid2 + 1;
    xfer acc, ctd2;
    jump zero, dim_y;
dim_x:
    ldc ctaddr_l0c0, ctaddr0;
    ldc rtp_wait_one, ctctl, go rtp;
    jump true, init;
dim_y:
    alu nid1;
    jump zero, buffer_packet;
    alu nid1 & reg0;
    jump ~zero, dim_y_neg;
    ldc ctaddr_l1c0, ctaddr0;
    ldc rtp_wait_one, ctctl, go rtp;
    jump true, init;
```

Figure 3.14: Minimal-path adaptive routing example

```
    wait ni0, trap0(ni1,ni2);
    xfer nid2, trap1;
    wait f0, trap1(true);
use_l0l1:
    jump ~resvdl0c0, get_l0c0, link;
    jump ~resvdl0c1, get_l0c1, link;
    jump ~resvdl0c2, get_l0c2, link;
    jump ~resvdl1c0, get_l1c0, link;
    jump ~resvdl1c1, get_l1c1, link;
    jump ~resvdl1c2, get_l1c2, link;
    ldc ctaddr_l0l1, ctaddr0;
    ldc 0x0, ctaddr1;
    xfer nid2, ctd2;
    ldc rtp_wait_one, ctctl, go rtp;
    jump true, init;
(nid2):
    jump true, use_l0l1;
```

Figure 3.15: Table-lookup routing microprogram.

9 cycles (just over one flit transmission time) before issuing a reservation request. This performance can be increased in some cases, however, when the desired channel ordering is directly provided by the hardware and the packet is using wait-for semantics.

While written using wait-for (wormhole) switching semantics, this scheme is easily adaptable to any switching scheme supported by the PRC. Its primary limitation is the need for a table entry for every node in the network, which limits the maximum size of the network to the available storage. This problem may be addressed by using a hierarchical addressing scheme, where packets are routed to "clusters" and then routed within the cluster, although the performance of such a scheme has not been explored.

3.5.2 Multicast Routing

In addition to unicast routing algorithms, many parallel and distributed applications can benefit from low-level support for multicast communication. The CTBUS protocol, combined with programmable header processing, allows the PRC to implement a broad spectrum of multicast routing algorithms, under both wormhole and virtual cut-through switching. Multicast header formats vary widely in size and format, depending on the set of destination nodes [7]; for example, some multicast schemes embed a "tree" of routes in

Register	Function
nid3	Node identifier
nid2	Switching scheme (1 for wormhole, 0 for virtual cut-through)
nid1	7-bit address bit mask: host, $O_{32} \ldots O_{20}$
nid0	6-bit address bit mask: $O_{12} \ldots O_{00}$

Table 3.7: Header word format for multicast routing example.

the packet header, while others use an address mask to encode destinations. Figure 3.16 shows an expanded implementation of the multicast routing algorithm discussed earlier, which employs either wormhole or virtual cut-through switching, depending on the packet header. This flexibility in switching has several benefits: since the switching mode is set on a per-node basis, certain nodes can be selected to buffer the packet if blocked even while others might be configured to stall the packet. This adds another method for breaking the deadlocks among packets.

Algorithm Overview: Each packet includes a tree of one-word routing headers to encode the nodes in the tree and the routing-switching scheme at each hop in the route; the format of a single entry is shown in Table 3.7. As the packet arrives at a node, the routing engine discards header words until reaching a word tagged with its one-byte node identifier. The second byte of the header word selects between virtual cut-through and wormhole switching, while the last two bytes determine where the receiver should forward the remainder of the incoming packet. Under wormhole switching, the receiver reserves *all* of the selected slaves devices before forwarding the packet, whereas the virtual cut-through scheme immediately directs the incoming packet to any *available* slaves.

The routing algorithm consists of the following main routines:

Initialization: The first two instructions write the identifier of the local node into two registers. Loading the local id into a general-purpose register allows an efficient comparison operation later to determine which header flits are destined for this node. The microprogram also preloads the next header flit by loading the appropriate outgoing data register, thereby ensuring that subsequent nodes ignore the first header flit.

Header stripping: After initializing internal registers, the routing engine waits for a packet header to arrive on one of the virtual channels. When a new packet arrives, lines 8–10 check the identifier field of each incoming word, discarding header information that corresponds to other nodes. After removing unneeded header words, the routing engine uses

```
init:
  ldc our_node_id, reg2; // prepare id check
  ldc our_node_id, ctd3; // initialize new header
  ldc ctcmd_resv, ctctl; // prepare for reservation attempt
  ldc strip_header, trap0; // initialize wait

strip_header:
  wait ni2, trap0(ni1,ni0);
  alu nid3 - reg2;
  jump ~zero, strip_header; // discard any flits not for us

route_packet:
  xfer nid1, ctaddr1; // extract address mask
  xfer nid0, ctaddr0;
  alu nid2;            // check switching mode
  jump zero, vc_mode;
  ldc rtp_wait_all, ctctl, go nirx; // all-of is easy
  jump true, init;

vc_mode:
  xfer ctfb1, reg1;    // Read address feedback reg.
  xfer ctfb0, reg0;
  xfer reg1, ctaddr1; // Load new address
  xfer reg0, ctaddr0, go ctbus;
  jump resv_zero, buffer_packet; // Watch out for null mask
  jump ~ack, vc_mode;
  ldc rtp_resvd_nocrc, ctctl, go nirx;
  jump true, init;
```

Figure 3.16: Multicast routing example.

the **route_packet** routine to construct the routing and switching decision for the packet.

Routing-switching decision: Then, lines 12–13 load the packet's candidate address mask into the CTBUS address registers, to prepare for a later reservation request. The routing engine selects the switching scheme by comparing `nid2` to zero (lines 14–15). For wormhole switching, the routing engine simply instructs the NIRX to reserve all of the selected virtual channels (line 20); the CTBUS protocol and the routing primitive's `all` bit allow the NIRX to completely handle channel reservation and data forwarding for the packet.

Virtual cut-through switching: Virtual cut-through switching is more complicated, since the packet does not wait for any busy outgoing channels to become available. Instead, the `vc_mode` routine employs *address feedback registers* (`ctfb0` and `ctfb1`) to check the reservation status of the selected channels[4]. By reading the `ctfb` registers and updating the `ctaddr` registers accordingly, the entire address can be updated in just five cycles, regardless of how many channels were selected. Lines 21–24 try to reserve these idle channels, branching back to `vc_mode` if the reservation request fails because some other packet reserved one or more of the channels. Before issuing a reservation request, however, we need to ensure that the new mask is not null — this is provided by the `resv_zero` flag check of line 25. Once the routing engine successfully reserves the virtual channels, line 28 relinquishes control to the NIRX to forward the remainder of the packet to the selected slave devices.

3.5.3 Source-list Routing

The microprogram in Figure 3.17 uses the switch status module to implement source-list routing with virtual cut-through switching. For each node the packet will traverse, the originating node computes a one-byte address mask (Table 3.8) specifying the channel to be used at that node; these masks are then packed together to form the routing header for the packet. As the packet traverses each node, the appropriate address mask is "stripped" off and used to route the packet. At the final destination of the packet, the address mask simply specifies the host interface to buffer the packet.

The switch status module aids this process by providing a quick means of checking the address mask — rather than decoding the mask to determine which reservation status

[4] Thus, if `ctaddr0` and `ctaddr1` select O_{30} and O_{10}, and only O_{10} is available, reading `ctfb0` and `ctfb1` returns a mask selecting only O_{10}.

Bit	Interpretation
7	Selects address register
6:0	Select destination
	$\langle \text{host}, O_{32}..O_{20} \rangle$ or $\langle O_{12}..O_{00} \rangle$

Null mask == 0x0

Table 3.8: Address mask format for source-list routing example.

flag to use, the mask is transferred to the address registers. The switch status module automatically sets the **resvok** flag to reflect whether all selected channels are currently free.

Source-list routing schemes may use several different forms of addressing. Rather than encoding a separate address mask in the header for every node the packet will traverse, *street-sign* addressing [5] uses **(node id, address mask)** pairs. Each pair indicates a node where the packet will change directions. To implement this scheme, each routing engine checks the ID value of the header; if it matches the local node ID, the routing engine will use the address mask and strip the pair from the header. If the IDs do not match, the packet is sent in the same direction it was traveling (i.e., if the packet is traveling left-right, it will be sent to the right) and the header forwarded unchanged. Street-sign addressing significantly reduces the size of the routing header if a packet needs to make only a few changes in direction.

3.6 Discussion

There are several common tasks within most routing and switching schemes. By developing an architecture for a flexible router controlled by small microcontroller-based routing engines, we have created a unique platform for investigating and comparing routing and switching schemes. Implementing these schemes as routing engine microprograms has had two benefits:

- These schemes have influenced the design of the routing engine by showing which functions were necessary and which needed hardware support.
- Highlighting operations common to one or more classes of routing and switching schemes.

```
find_mask:
  alu nid3;
  jump ~zero, got_mask_b3;
  alu nid2;
  jump ~zero, got_mask_b2;
  alu nid1;
  jump ~zero, got_mask_b1;
  alu nid0;
  jump ~zero, got_mask_b0;
  wait nidata;
  jump true, find_mask;

got_mask_b3:
  alu nid3 & reg0; /* reg0 = 0x80 */
  jump ~zero, ad1_mask_b3;

ad0_mask_b3:
  xfer reg1, ctaddr0;
  jump ~resvok, buffer_packet;
  ldc ctcmd_resv, ctaddr0, go ctbus;
  ldc 0x0, ctd3;
  xfer nid2, ctd2;
  xfer nid1, ctd1;
  xfer nid0, ctd0;
  jump ~ack, buffer_packet;
  ldc rtp_resvd_nocrc, ctctl, go nirx;
  jump true, init;
```

Figure 3.17: Source-list routing microprogram example.

For example, the `wait` instruction shows how the routing engine architecture has been influenced by writing these schemes. In its barest form, the `wait` instruction simplifies and (sometimes) speeds up the task of waiting for external events. Since all microprograms must stall while waiting for packet headers, hardware support was desirable. While this could be implemented as a sequence of individual checks, or as a single jump instruction based on a common flag, the `wait` instruction provides a more powerful and efficient implementation. Compare three potential implementations of the entry sequence:

`wait_loop:`
`    jump ni0, handle_c0;`
`    jump ni1, handle_c1;`
`    jump ~ni2, wait_loop;`
	`wait_loop:`
	`    jump ~nidata, wait\_loop;`
		`wait_loop:`
		`    wait ni0, trap0(ni1),`
		`                trap1(ni2);`

(a) Channel-flag	(b) Common-flag	(c) Wait

The first implementation, which checks individual flags for each channel to see if one has data, may take as many as five cycles to react to the arrival of header information. The common-flag and wait schemes, on the other hand, while require at most two cycles. The wait-based implementation may choose a different handler on the basis of the virtual channel, but the common-flag method cannot. The `wait` instruction also allows consistent prioritization among the virtual channels.

One significant omission from the routing engine instruction set as implemented in the PRC (Chapter 4) is a condition code flag for checking the sign bit. While this check may be implemented by a boolean AND operation and check for zero:

```
alu reg0 & x; /* reg0 contains 0x80 */
jump zero, x_is_positive;
```

which takes two instructions to execute, the same as

```
alu x;
jump ~neg, x_is_positive;
```

this structure is not as efficient when the same operand is also being checked for zero; compare:

```
alu x; /* reg0 contains 0x80 */
jump zero, x_is_zero;
alu reg0 & x; /* reg0 contains 0x80 */
jump zero, x_is_positive;
```

to:

```
alu x; /* reg0 contains 0x80 */
jump zero, x_is_zero;
jump neg, x_is_positive;
```

This structure often arises in offset-based routing algorithms such as the adaptive scheme in Section 3.4.

An alternative instruction set might include a three-way **sign** branch that would use a single instruction to cover all three possibilities:

```
alu x;
sign x_is_negative, x_is_positive;
<code for x_is_zero>
```

With this instruction, the instruction specifies the locations to jump to if x is positive and if x is negative; if x is zero, the next sequential instruction executes. While this instruction would require at a minimum 20 bits to implement (16 for the two addresses, and four bits to distinguish the major operation), it will require two separate ports from the control store for transferring the selected address to the program counter. In addition, the current routing engine instruction set does not have the "room" for an additional major operation without widening the instruction width to 21 bits. This extra cost is hard to justify for a one-cycle savings. Instead, Chapter 7 presents a high-performance routing engine that uses a different implementation of this sign function as the basis for its route computations.

CHAPTER 4

The Programmable Routing Controller

And now! Available at last! The IBM 4000 PCSr system ...
But now featuring TINT CONTROL!
— Berke Breathed's *Bloom County*

This chapter describes in detail the architecture of the *Programmable Routing Controller*, which implements one variation on the flexible router architecture presented in Chapter 3.

4.1 PRC Architectural Overview

The PRC couples the flexible network architecture of Figure 3.1 with a high-performance, low-overhead host interface. The network interface of the PRC manages bidirectional communication with four other nodes, with three virtual channels on each unidirectional link. In terms of the OSI layer model, the PRC implements the physical and data link layers, as well as parts of the network layer.

4.1.1 Network Architecture

In the PRC architecture, shown in Figure 4.1, each outgoing link is controlled by a module containing three NITXs; each NITX implements a single outbound virtual channel. Similarly, each incoming link is controlled by a receiver module (as described in Section 3.3) containing a programmable routing engine and three NIRXs. Data is transferred between the host interface, the NITXs, and the NIRXs via the *cut-through bus* (CTBUS). When the host initiates packet transmission, the appropriate *Transmission Fetch Unit* (TFU) reserves an outbound virtual channel (NITX) and forwards data to the reserved NITX across the

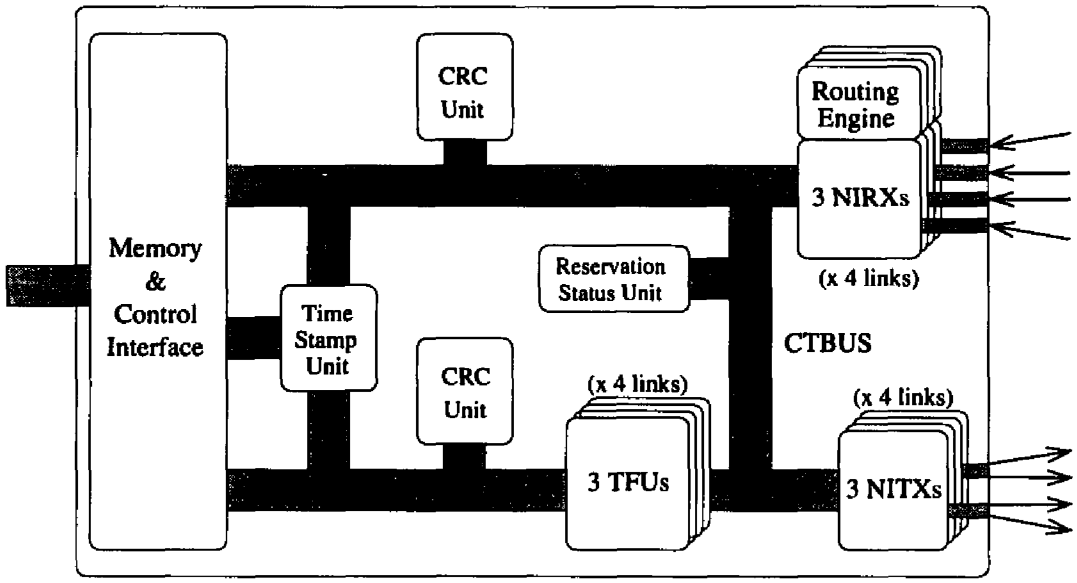


Figure 4.1: PRC Architecture.

CTBUS. Packet cut-through and reception, on the other hand, are controlled by the NIRXs within the receiver modules by transferring data either to a reserved NITX (for cut-through) or to the host interface (for reception).

4.1.2 Host Interface

The host interface of the PRC provides support for simplifying or eliminating all of the host's major communication overheads. Steenkiste [64] outlines four significant classes of operations that are associated with sending and receiving packets: transport layer protocol processing, context switching, data link protocols (dealing with the network interface), and buffer management. In addition, there are *per-byte* overheads that vary according to packet size for copying data and computing checksums. The relative importance of each overhead depends on several factors, including the maximum size of packets and the relative speeds of the CPU and memory bus. To this end, the PRC host interface provides support for simplifying each of these tasks:

Transport protocol processing: By allowing the host to select the packet sizes that best meet its needs, the PRC can reduce the total number of packets needed to transmit a message. This can also be used to eliminate the need for message reassembly by avoiding packetization entirely.

Buffer management: The PRC uses a simple *outboard* buffer [64] that is accessible to the host either through DMA or direct memory reads and writes. The paging scheme used for this buffer greatly simplifies the host's tasks in maintaining the free list of buffers.

Context switching: The PRC provides an *event queue* that logs packet-level events, allowing it to operate without host intervention. Together with a control interface that requires no handshaking, the host interface minimizes the number of interrupts and context switches required to deal with the PRC.

Data link protocol: The PRC uses a queue-based command interface that allows the host to "fire-and-forget" packet transmission commands. Also, by avoiding any handshaking and mapping all internal status registers into the VME address space, the PRC simplifies the actual host protocols that interact with it.

Data copying: The PRC's external buffer acts as an outboard buffer; since packet bodies are copied directly (via DMA) from the application memory without being accessed by the NP, data is only transferred once across the memory bus (the VME bus).

Checksumming: The PRC provides transparent error detection via end-to-end calculation and checking of a *cyclic redundancy code* (CRC) for every packet it transmits. This allows the NP to be completely bypassed during the main data transfer.

The host interface of the PRC supports a wide range of packet sizes and formats, while implementing and/or simplifying other requirements. The twelve incoming and outgoing virtual channels share access to the external buffer memory, interleaving at the word level. Since the PRC does not include internal buffers for blocked packets, packets that buffer at intermediate nodes are stored in this SRAM.

To reduce software protocol complexity, the PRC interacts with the controlling host processor in terms of *pages*, with each packet consisting of one or more (possibly non-contiguous) pages. To better accommodate different sizes, the PRC allows packets to consist of either 256-byte or 1024-byte pages; larger pages allow the PRC to operate longer without host intervention.

Figure 4.2 depicts the major components of the PRC's host interface. The primary interfaces accessed by the host are the transmitter page queues (one per outgoing virtual

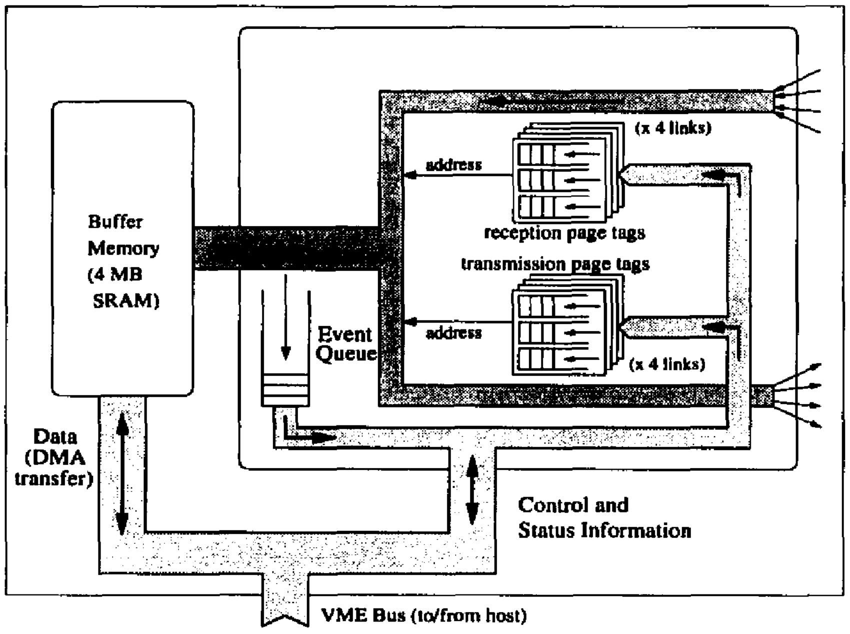


Figure 4.2: PRC host interface.

channel), the receiver page queues, and a unified *event queue*. Each page queue is associated with a particular channel, and specifies either the address and size of a page to be transmitted (transmission page queues) or the address of a page where incoming packets should be stored (reception page queues). The event queue keeps an ordered record of the pages transmitted and received by the PRC. By logging multiple events before interrupting the host for service, this reduces the host's overhead for managing the PRC.

To illustrate the simplicity of the PRC to NP interface, we can consider an example of how the NP controls the transmission and reception of a large message with over 1,024 bytes in its body. Figure 4.3 shows how the packet might be stored in the buffer memory; a large (1024-byte) page is used to store the majority of the message. To transmit this packet, the main steps are:

1. The host initiates a DMA transfer of the packet body from the AP to the buffer.

2. As the DMA transfer proceeds, the NP constructs the packet header in a separate page via direct writes from the AP.

3. The NP then writes the page tags to the transmission page queue for the desired channel if slots are available.

For reception, the interactions are also simple and fast:

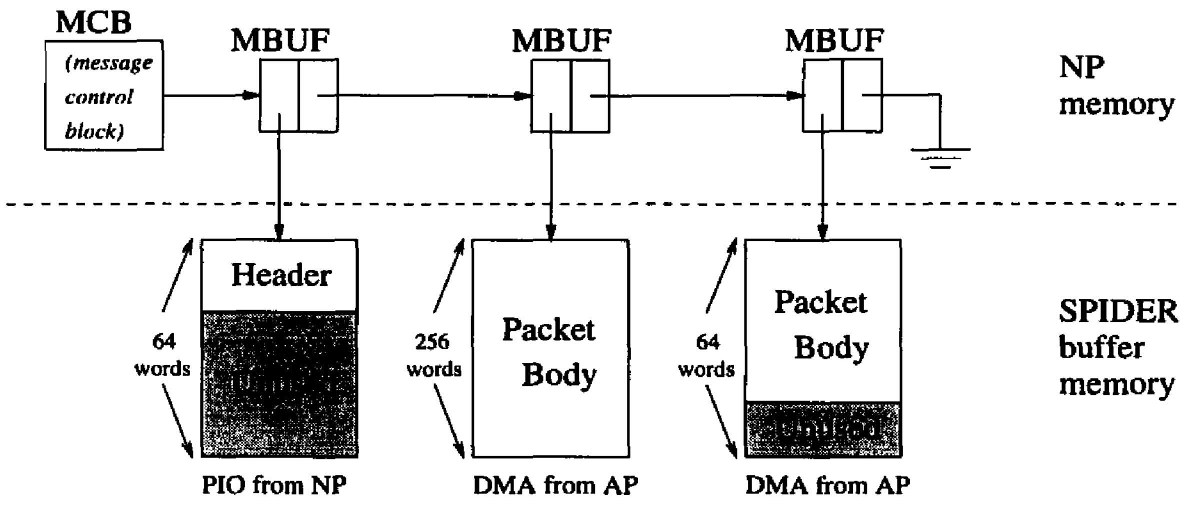


Figure 4.3: Example packet structure in memory. The shaded regions represent the unused portion of each page.

1. The reception page queue has been "primed" with page tags telling the channel where to store incoming packets.
2. After the packet has been received, the host reads the page tags from PRC event queue.
3. The host then parses the routing header to determine where the packet should go.
4. Finally, the host initiates a DMA transfer of the packet body to the application.

In this scheme, the packet body is copied over the main bus only once at its source and destination; the NP never actually examines it. The NP only interacts with the PRC via a maskable interrupt and direct reads and writes of memory-mapped registers. Thus, most of the major non-protocol operations have been entirely eliminated.

4.1.3 PRC Operation

To illustrate the operation of the PRC, consider how a message is handled during transmission from the source node, cut-through at an intermediate node, and reception at the destination node, as shown in Figure 4.4.

Transmission: When an application requests the host to transmit a message to another node, the host disassembles the message into multiple packets, where a packet consists of one or more pages. Using the control interface, the host then instructs the appropriate TFU

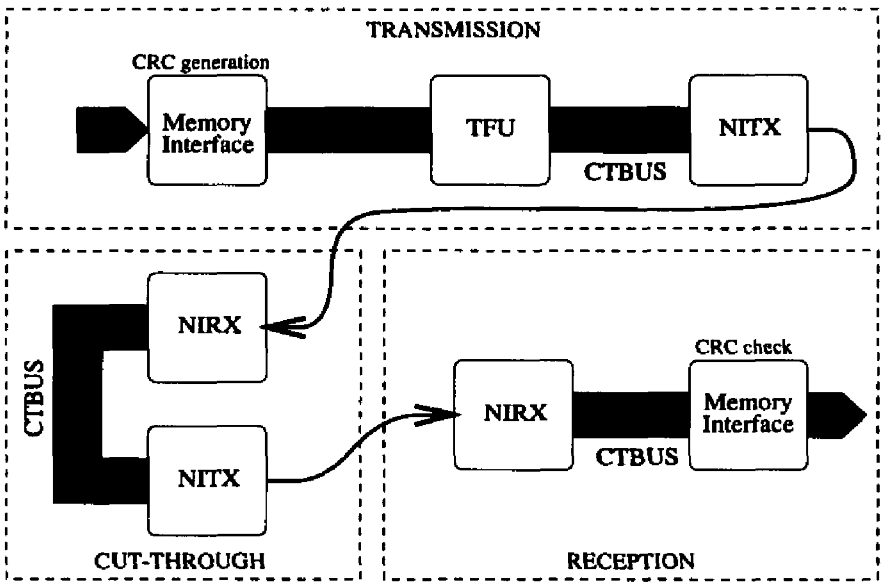


Figure 4.4: Traffic flow in the PRC.

to transmit these pages. After the TFU reserves the NITX, the memory interface fetches the data from each page, one 32-bit word at a time; during this transfer, the memory interface accumulates a 32-bit CRC. After sending the last data word of the packet, the TFU transmits a 32-bit timestamp, read from a counter on the PRC, followed by the CRC. Using the CTBUS, the TFU transfers each word to the NITX, which transmits the data as individual bytes to the subsequent node.

Cut-through: Packet reception begins when data bytes arrive at an NIRX in the network interface. Once the NIRX has received enough header data, the associated routing engine formulates a routing and switching decision for the packet. If the packet is destined for a subsequent node, the PRC can try to send the packet directly to the next node by reserving an NITX. If the packet is able to establish a cut-through, the routing engine reconfigures the NIRX to forward data directly to the reserved NITX, bypassing the rest of the PRC entirely. When the packet has cleared the node, the NIRX automatically resets itself to direct the next incoming packet header to the routing engine.

Reception/Buffering: Once the routing engine has decided to buffer a packet at the local node, however, the NIRX simply forwards the incoming words across the CTBUS to the memory interface. As each word flows to the buffer memory, the memory interface reaccumulates the CRC. The PRC event queue logs the arrival of each page, noting the address and size. At the end of the final page of the packet, the PRC appends the packet

Command	Function
DTX	Normal data transfer
MARK	End-of-page data transfer
EOP	End-of-packet data transfer
FREE	Relinquishes selected channels
RESV	Reserves selected channels
HOLD	Host-initiated override for channel allocation
CHECK	Reserves "held" channels

Table 4.1: CTBUS command set.

with a receive timestamp and logs a packet-arrival event indicating the outcome of the CRC check. If the packet has reached its destination, the host reassembles the pages into a packet and the packets into a message. Otherwise, the host coordinates further transmission of this packet by feeding a TFU with the appropriate page tags.

4.2 Network Architecture

The network interface of the PRC is centered on the CTBUS, which implements two primary functions within the PRC: data transfer and channel allocation. As shown in Table 4.1, the CTBUS protocol includes commands to transfer data and reserve/relinquish NITXs. The DTX, MARK, and EOP commands "tag" data words to denote page and packet boundaries, while the other commands control channel allocation.

As a data switch, the CTBUS combines high throughput with support for multicast operations. The 32-bit time-slotted bus operates at twice the byte-transmission speed of the internode links, matching the bandwidth of the eight unidirectional links. Since each bus transaction can address the memory interface and any of the NITXs, a single CTBUS transaction may spawn transmissions on several outbound virtual channels simultaneously; this facilitates efficient broadcast and multicast algorithms [7, 30].

4.2.1 Reservations

Access to the CTBUS, which is controlled by a demand-slotted binary priority-tree arbiter that allocates bandwidth fairly amongst the active devices [21, 39], also implicitly determines the PRC's allocation policy for reserving NITXs. Any master needing to reserve a NITX simply checks to see if the NITX is free; if so, it requests access to the CTBUS and issues a RESV command when access is granted. Since CTBUS access is pipelined, another

Cycle	9	8	7	6	5	4	3	2	1	0
0	Channel		Data Byte 3							
1	Command		Data Byte 2							
2	CRC	Ack2	Data Byte 1							
3	Ack1	Ack0	Data Byte 0							

(a) 10-bit mode

Command	1	0
Null	1	1
Ack0	1	0
Ack1	0	1
Ack2	0	0

(b) 2-bit mode

Table 4.2: Link encodings.

master may have reserved the NITX in the meantime. This is handled by the reservation status unit — upon receiving a RESV command, the reservation status unit determines if the requested NITXs are available. If all of the selected slaves are free, the NITXs are marked as reserved and a success flag (*ctack*) set. Although all CTBUS devices have concurrent access to the NITXs' reservation status, the bus interconnect implicitly serializes reservation requests, simplifying the design of the reservation status unit. Masters relinquish channel reservations with the FREE command; any NITX slaves forward the FREE command to the subsequent link(s) in the route to clear any downstream channel reservations. Although a FREE typically follows an EOP, separate commands allow the PRC to establish connections that outlive individual packets.

The HOLD command provides a simple mechanism for overriding the CTBUS access arbitration when allocating NITXs; once a HOLD command has been issued for a NITX, all subsequent RESV commands will fail. Since the HOLD may be issued while the NITX is busy, this command guarantees the next reservation for the issuing master. The CHECK command is used to reserve devices held by a HOLD command.

4.2.2 Internode Links

Besides simplifying the implementation of the reservation status unit, the data serialization provided by the CTBUS also simplifies the architecture of the NITXs controlling the outbound links. Rather than providing a separate arbitration mechanism for the link between virtual channels, data is simply transmitted in FIFO order.

The links may be implemented as either parallel or serial connections. For parallel interconnect, each PRC's output ports are directly connected to the input ports of its neighbors. The same protocol may also be used to control AMD TAXI chips [1] for serial communications over longer interconnect wires. This implementation is transparent to the

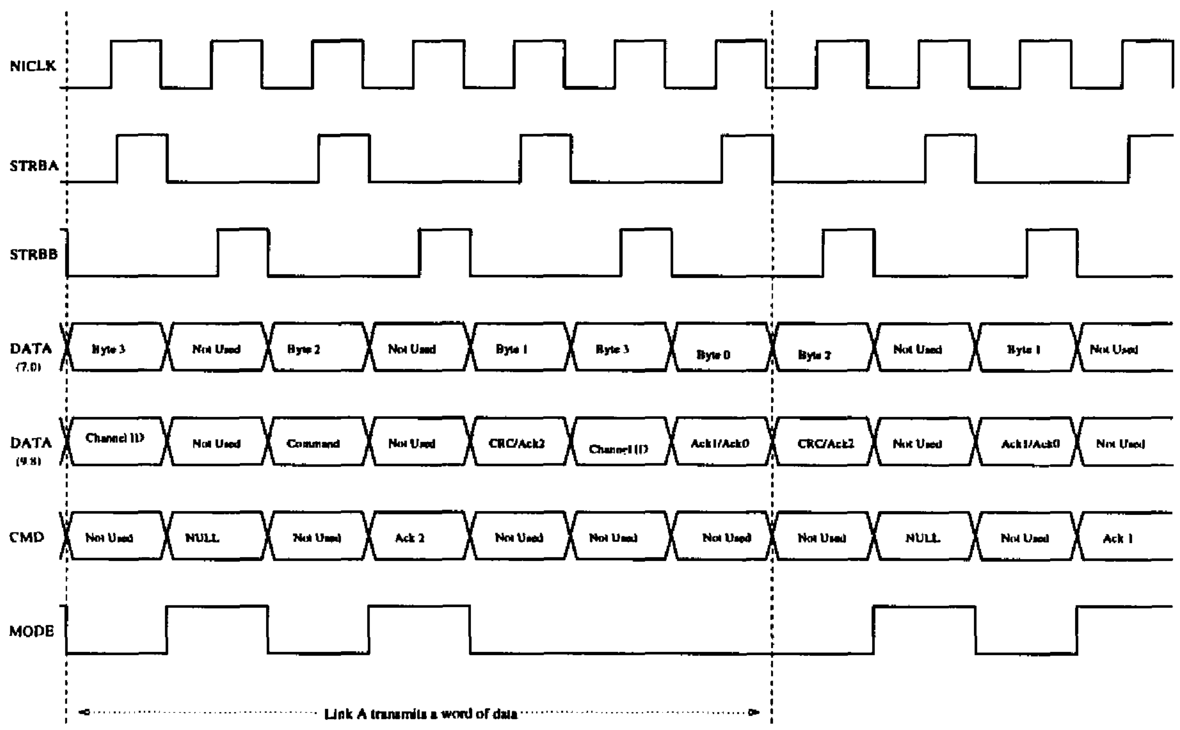


Figure 4.5: Typical link transmission cycles.

PRC internally. In both cases, either 10 bits or 2 bits of information is transmitted every link cycle, as shown in Table 4.2. Transferring a word of data consumes four 10-bit cycles to tag the data bytes with important control information.

The first data transfer identifies the outgoing virtual channel, while the second 10-bit transaction tags the data with one of the first four commands in Table 4.1. The last two 10-bit transfers encode piggy-back acknowledgements for the three virtual channels and indicate whether the outgoing word should be included in the packet's CRC computation. When no data transfers are pending, the link transmits a 2-bit flow-control acknowledgement or a null command in each cycle. Each NIRX has buffer space to store two outstanding words from an arriving packet; when the NIRX can accept another word, the paired NITX transmits a flow-control acknowledgement back to the adjacent node. The NITX/NIRX state machines, coupled with the CTBUS command set, handle the routine tasks of data transfer, flow control, arbitration and channel reservation, allowing the routing engines to focus completely on constructing intelligent routing-switching decisions for incoming packets.

4.2.3 Intranode Flow Control

Intranode flow control is one of the primary responsibilities of the CTBUS. This flow control could be handled using NACKs or tokens; using NACK-based flow control, masters optimistically forward data to their slaves. If the slave cannot accept the data, it asserts a NACK line to instruct the master to retry the transfer at some later point. This protocol, however, consumes a precious resource (CTBUS bandwidth) for failed data transfers; at heavy loads, this penalty increases at a time when it is least acceptable.

Under a token-based protocol, the master cannot forward data unless it has been given at least one token by the slave; each data transfer consumes one token. A token based flow control policy typically requires some means for transferring tokens from the slave to the appropriate master. This mechanism can be fairly complex.

Instead, the CTBUS defines a simple intranode flow control mechanism based on a variation of the token scheme. Each slave maintains a single *input ready* (IR) flag that signals its willingness to accept at least one data flit. Before forwarding data to a slave, masters are responsible for ensuring that any slave(s) are willing to accept data. Since all IR signals are "broadcast" to every master, the checking is done locally by comparing the targeted set of slaves with the set of slaves that are not willing to accept data; if this set is not empty, the master cannot request CTBUS access. This policy prevents any excess CTBUS traffic, while simplifying the design of both the masters and the slaves.

Since the CTBUS is pipelined and the slaves will require some amount of time to acknowledge a data transfer, some means must be put in place to keep masters from sending data to a slave that has not yet been able to lower its IR signal. Since the PRC already requires a master to reserve a slave before using it to transmit data, no more than one master will be monitoring a particular IR signal at any one time. By prohibiting this master from forwarding data on consecutive CTBUS cycles, we prevent these overruns. There is no performance penalty, either — each CTBUS transfer will require eight cycles to transmit on the physical link, so the second data word should arrive before the first has completely left the node.

4.2.4 Timing

To minimize the internal cycle time, the CTBUS is pipelined. Every CTBUS access is divided into three major stages: arbitration, access, and resolution. Since each stage

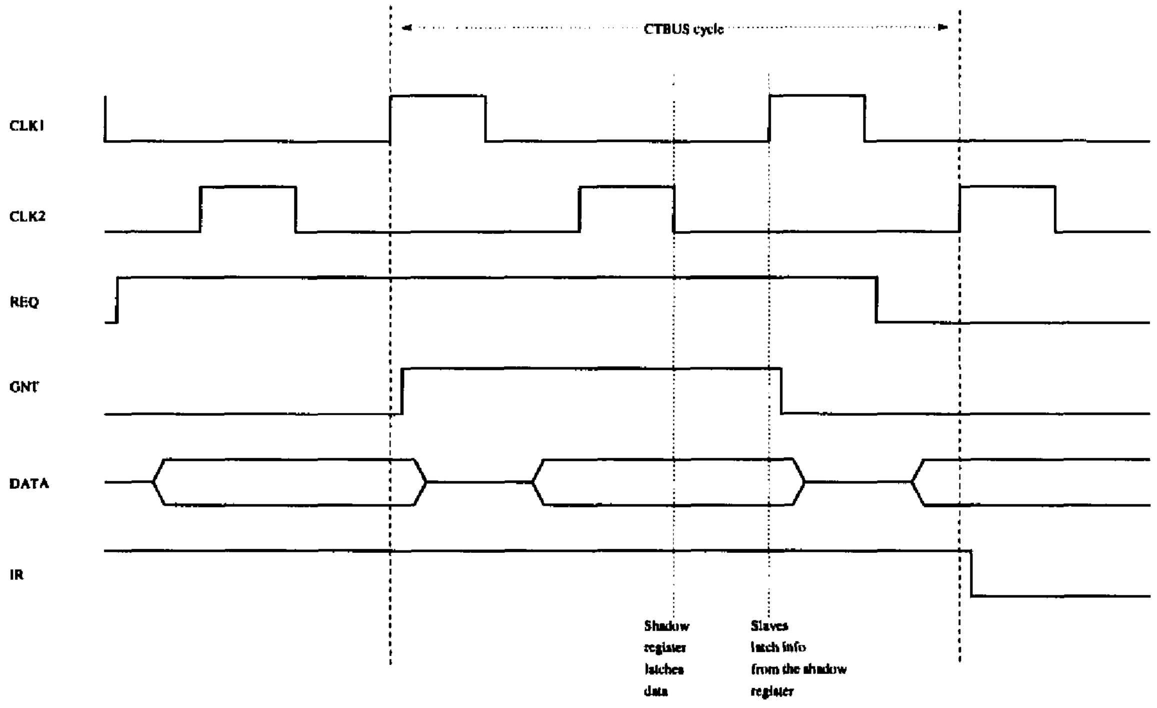

CTBUS cycle
CLK1
CLK2
REQ
GNT
DATA
IR
Shadow register latches data
Slaves latch info from the shadow register

Figure 4.6: Typical CTBUS cycle.

operates concurrently with the others, three CTBUS "cycles" are always operative and any individual access takes three cycles to complete[1].

4.3 Host Architecture

The host interface of the PRC is unique in several ways. Unlike most routers, it provides a large number of channels between the host and the network. At the same time, it is responsible for providing several services (e.g., error detection, paging, and interrupt minimization) to the host; most multicomputer routers do not provide these services.

4.3.1 Host Interface

The management of several, concurrently active, data channels has implications for protocol processing, buffer management, packet scheduling, and interrupt handling. The host must be able to handle a higher frequency of events, maintain distinct scheduling queues for each channel, and service each channel in a fair and efficient manner. Although it is possible to construct a point-to-point distributed system using replicated communication hardware at each node, cost and performance considerations necessitate an integrated, low

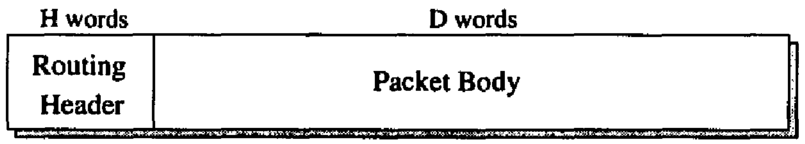
[1]Exclusive of any delays during arbitration due to competition for the bus.

overhead design for the communication adapter.

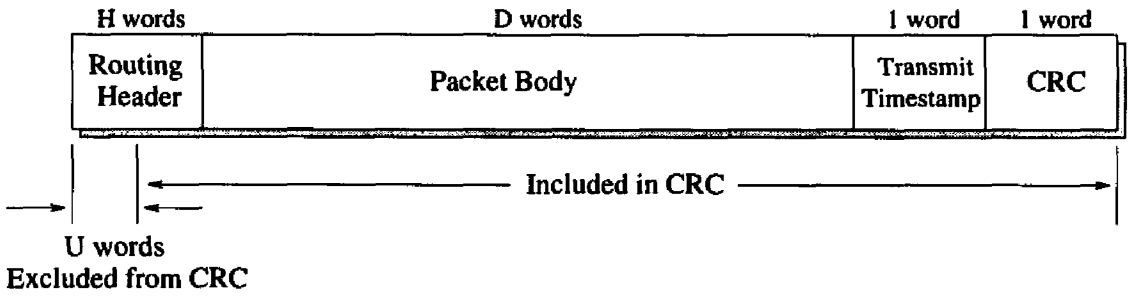
To reduce software protocol complexity, the PRC interacts with the controlling host processor in terms of *pages*, as shown in Table 4.3. Each packet consists of one or more (possibly non-contiguous) pages. The host transmits a packet by feeding page tags to a *transmitter fetch unit* (TFU); each page tag includes a memory address and the number of words to transmit. Similarly, the host supplies each *network interface receiver* (NIRX) with pointers to free pages in the buffer memory, for use by arriving packets. The twelve incoming and outgoing virtual channels share access to the external buffer memory, interleaving at the word level. Since the PRC does not include internal buffers for blocked packets, packets that buffer at intermediate nodes are stored in this SRAM.

The PRC logs the transmission and reception of individual pages in an internal event queue; the host processor reads this event queue to perform free-list maintenance and assemble incoming packets. The page-level data transfer facilitates scatter-gather DMA between the buffer memory and the network and also allows the host to construct packet headers on a separate page from the data. Generating the header on a separate page avoids unnecessary data copying, reducing the overheads of network data transfer. Since the PRC does not restrict packet length, the application or protocol software can weigh the cost-performance trade-offs for selecting packet sizes. Small packets reduce buffering delay and allow fine-grain sharing of buffer space and link bandwidth, but also incur increased costs for message fragmentation and reassembly. To better accommodate different sizes, the PRC allows packets to consist of either 256-byte or 1024-byte pages; larger pages allow the PRC to operate longer without host intervention.

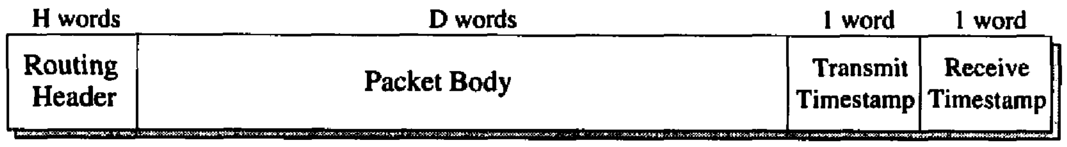
When an application requests the host to transmit a message to another node, the host disassembles the message into multiple packets, where a packet consists of one or more pages. Using the control interface, the host then instructs the appropriate TFU to transmit these pages. Each page tag includes a memory address and the number of words to transmit. After a page tag has been placed in the queue, the PRC will reserve the channel, transmit the page, and log its transmission in the event queue. The depth of the queues represents an implementation tradeoff; deeper queues allow the PRC to operate without host intervention for longer periods, but are costlier to implement. In addition, to maximize scheduling flexibility, the host often restricts the number of packets (and/or pages) that are queued in the PRC. The variable page size provides a solution to both fragmentation and



(a) As retrieved from buffer memory



(b) Transmitted over internode links



(c) Stored at destination

Figure 4.7: Packet formats.

Category	Functions
Data transfer	page transmission free page allocation
Status	event retrieval determine empty slots in page-tag queues timestamp access
Configuration	interrupt masking notification FIFOs
Initialization	microcode download

Table 4.3: Control interface capabilities.

overhead problems — larger data pages may be transmitted as a single 1024-byte page while small header pages may be stored on the 256-byte pages. The simplicity of this interface is illustrated by Figure 4.8, which shows the code necessary to transmit a single-page packet.

The reception queues provide each incoming channel with pointers to free pages in the buffer memory, for use by arriving packets. The PRC event queue logs the arrival of each page, noting the address and size. If the packet has reached its destination, the host reassembles the pages into a packet and the packets into a message. Otherwise, the host coordinates further transmission of this packet by feeding a TFU with the appropriate page tags. The host then reads this event queue to perform free-list maintenance and assemble incoming packets. During packet transmission and reception, the PRC preserves the amount of data stored on each page. This simplifies the host's buffer management and reduces data copying, since the operating system header can be maintained on a separate page and modified without affecting the main data portion of the packet.

To minimize software protocol processing overheads and increase reliability, the PRC's transmission and reception datapaths incorporate transparent error detection via *cyclic redundancy code* (CRC) generation and checking. The PRC uses a single 32-bit parallel CRC generator to compute the CRC for every outbound packet; similarly, a single checker accumulates the CRC for incoming packets en route to the memory interface. When a packet buffers at its destination or an intermediate node, the reception datapath checks the CRC and logs the outcome in the event queue. Since some routing algorithms modify packet headers at intermediate nodes, the PRC allows a packet's first page tag to specify a number of words to exclude from the CRC calculation. This allows subsequent nodes to modify a packet's routing header without invalidating the original CRC checksum; the microprogrammable routing engines can enforce separate error detection or correction on the packet header.

Many networks, such as the *Scalable Coherent Interface* (SCI) perform error-detection checks after traversing every link, "stomping" the CRC value if an error is detected. This has the advantage of earlier detection of errors, and allows noisy links to be quickly diagnosed. In some cases (especially when the interconnect is serial) this also allows a much simpler CRC generator and checker to be used. For a cut-through switched system, however, the utility of this method is not as useful, since the majority of the packet has already moved on through the network before the error is detected. By providing end-to-end error detection, on the other hand, the integrity of the body of the packet may be verified; in the PRC, the routing engine may provide limited error detection for the header if necessary.

In addition to error detection circuitry, the transmission and reception datapaths include logic for timestamping packets. These timestamp values are useful for controlling clock drift between nodes, as well as for performance measurements; using the control interface, the host can adjust the PRC's timestamp register in response to clock synchronization protocols. By affixing timestamps close to the physical links, the PRC provides an extremely accurate measure of when outgoing packets complete injection and when incoming packets finish reception. This allows the host to guarantee tight bounds on clock skew between nodes [31, 55].

The control interface also includes several configuration options, as shown in Table 4.3. The cost of managing the PRC can be reduced by selectively masking interrupts; by disabling all interrupts, the host processor can interact with the PRC through polling. Furthermore, the host can amortize the cost of servicing the PRC by reading the entire event queue during each interrupt or polling cycle. The host can influence low-level routing and switching at run-time by accessing *notification FIFOs*, which provide bidirectional information exchange with each routing engine. These FIFOs are used to diagnose and respond to dynamic conditions, such as congestion or faulty links, allowing the host to adjust the operation of the downloaded microcode.

4.3.2 Memory Interface

The *Network Processor* Bus Interface provides the PRC with a means of accessing the host's memory to store and retrieve packets. The architecture of the NPBUS interface is

```
int prc_xmit (buf, len) {
page_ptr buf;
int      len;

ULONG    status;
ULONG    tag;

tag = (buf & 0x007fff00) | CRC_MASK | LAST_PAGE | (len & 0xfc);
status = PRC_TXO_PGSTATQ->status;
status &= TXO_PG_MASK;
if (status > 0) {
    PRC_TXO_PGQ->queue = tag;
}
else
    queue_for_channel;
```

Figure 4.8: Sample code for transmitting a single-page packet.

depicted in Figure 4.9. The NPBUS interface essentially provides a single service to the various channels: this service can best be described as data storage and retrieval without need for worrying about addresses. The TXBUS and RXBUS simply present requests for service to the NPBUS interface, which then fills that request. If both buses request service simultaneously, they are serviced in a round-robin fashion; if only one bus is active, however, the other may use any available memory cycles. Appendix A gives a more detailed description of the operation of the memory interface.

Externally, the memory interface interacts with the buffer memory through a simple synchronous interface. Rather than requesting access to the buffer from the outside controller, the PRC simply waits until it is "granted" access by the controller. In turn, the buffer controller allots every idle memory cycle to the PRC — if the PRC has no data to transfer, it simply issues a read request. This one-way interface greatly simplifies the design of both the external controller and the PRC's memory interface, while allowing much faster operation. The memory controller sends three signals to the PRC: a 40 MHz clock (**np_clk**, straight from the VME bus), a synchronizing signal that divides the main clock into a two-phase, 20 MHz clock (**np_sync**), and an ownership signal (**np_owner**) that tells the PRC when it may access the memory. Figure 4.10 shows a typical memory interface

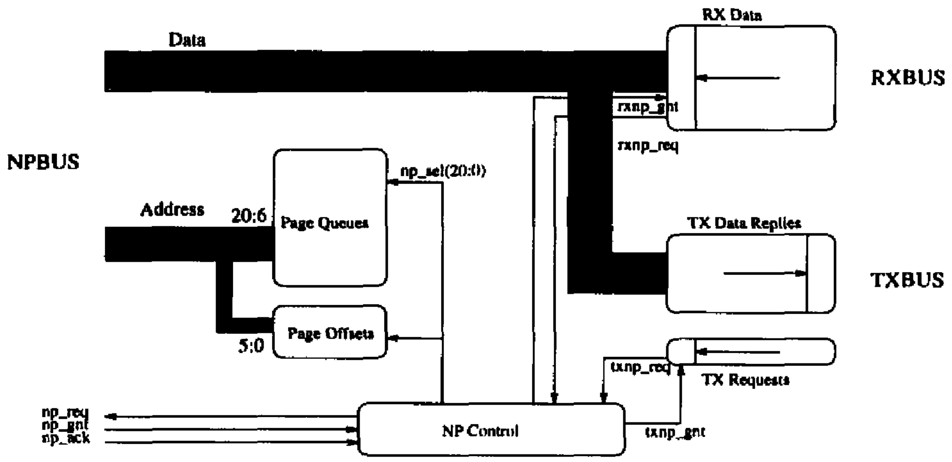


Figure 4.9: Memory interface architecture.

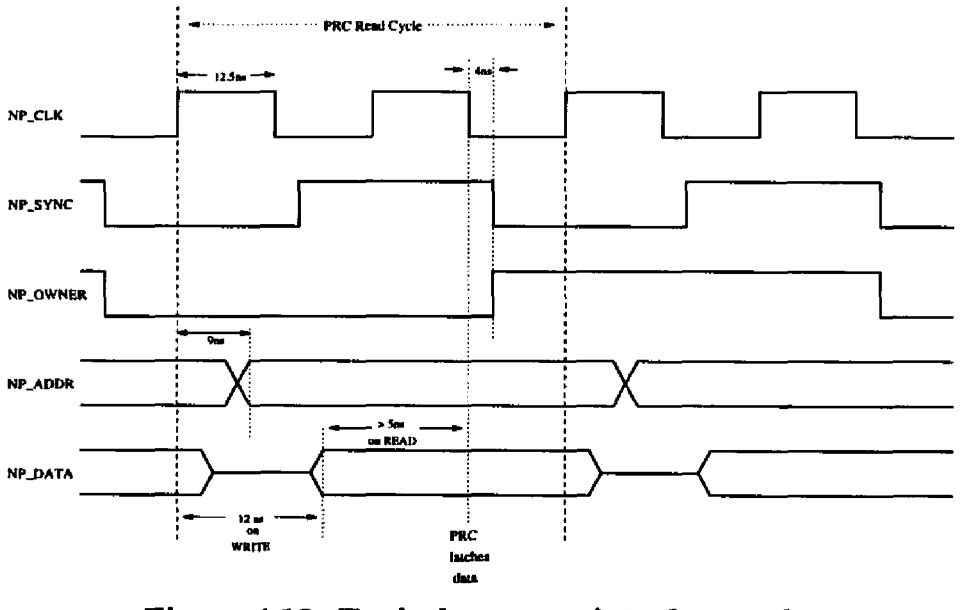


Figure 4.10: Typical memory interface cycle.

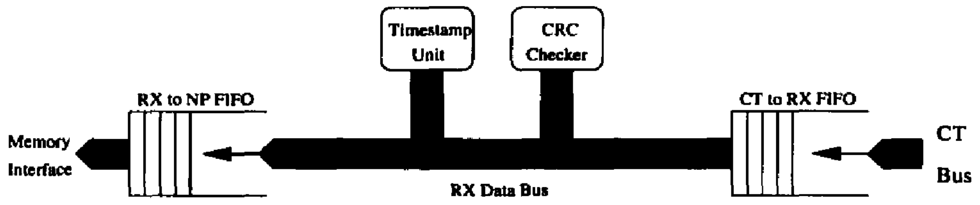


Figure 4.11: RXBUS architecture.

cycle.

4.3.3 Internal Memory Architecture

The internal interface between the memory and network interfaces is segmented into separate transmission and reception datapaths. The TXBUS transfers data between the memory interface and the TFUs, which then forward it across the CTBUS, while the RXBUS moves data from the CTBUS to the memory interface. Separating these paths simplifies the implementation of both, while increasing throughput by reducing contention.

The *Reception Bus* (RXBUS), shown in Figure 4.11, transfers data between the CTBUS and the memory interface. Its major functions include accumulating the CRC checksum for error detection, affixing the receive timestamp to incoming packets, and controlling the flow of data between the memory interface and the CTBUS. This intranode flow control prevents data overruns within the memory interface. Flow control "acknowledgments" are generated as each word is removed from the CTBUS FIFO by resetting that channel's IR flag.

The architecture and operation of the RXBUS is quite simple — it latches data and commands from the CTBUS whenever addressed, and transfers them through the FIFOs to the memory interface. The RXBUS only latches CTBUS transfers that are tagged with a DTX, MARK, or EOP command — in other words, only those transfers that actually carry data.

The Transmission Bus (TXBUS) transfers data between the memory interface and the TFUs, providing a serialization point for data requests and access to the timestamp register and the CRC generator. To allow concurrent requests and responses, the TXBUS is divided into the two major buses as shown in Figure 4.12. These buses generally operate independently: the TX Command Bus transmits requests for data from the TFUs to a request FIFO in the memory interface, while the TX Data Bus transfers data from the data FIFO

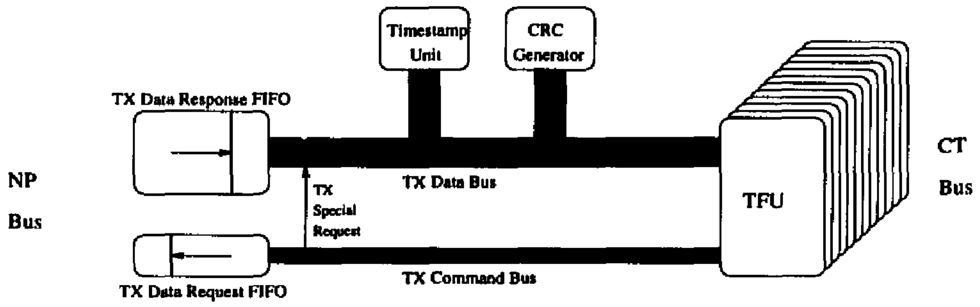


Figure 4.12: TXBUS architecture.

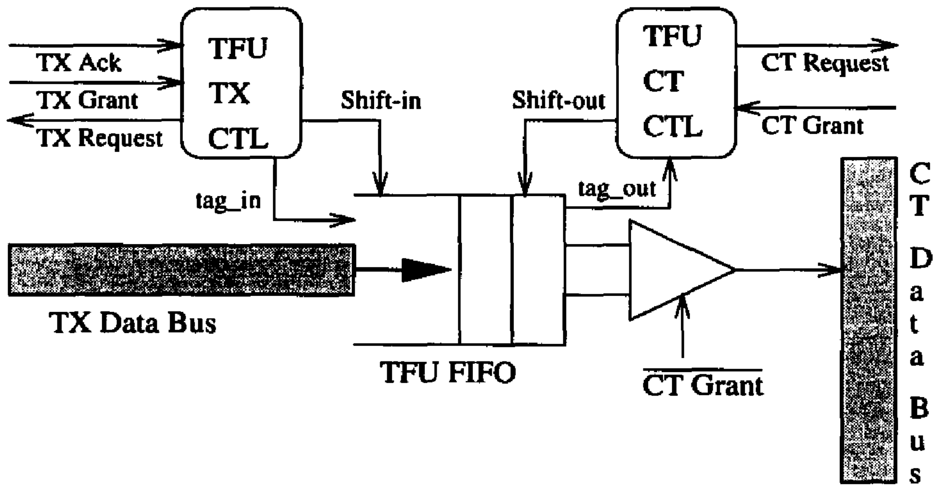


Figure 4.13: TFU architecture.

to the TFUs.

Access to the command bus is determined by a binary priority-tree arbiter, similar to that used for the CTBUS. The command bus implements three transactions: a data request (which is queued in the request FIFO), and reads of the timestamp register and the CRC register. The command bus operates independent of the data bus unless a command accesses the timestamp register or the CRC generator, in which case a *special request* signal is set. Setting this signal slaves the data bus to the command bus.

The data bus generally operates independently of the command bus: as data arrives from the memory interface it is transferred to the appropriate TFU. Each transfer is tagged as either normal data (DTX), end-of-page (MARK), or end-of-packet (EOP). These data transfers are suspended, however, whenever a timestamp or CRC read is issued on the command bus; the data bus then transfers the appropriate value to the TFUs.

The control of each of the outbound channels is centered on the *Transmitter Fetch Units*

(TFUs). Each TFU provides most of the logic necessary for controlling packet retrieval and transmission; it monitors its page queue until a page tag is detected. The presence of a page tag triggers two operations: a reservation request for the paired NITX and a data request to the memory interface. As can be seen in Figure 4.13, the TFU consists of three major components: two state machines and a FIFO for local storage of data. Each state machine controls the TFU's interactions with a specific bus. One FSM interfaces the TFU to the TXBUS, and is responsible for retrieving data from the NP Interface via the TXBUS and placing the data into the FIFO. The other FSM controls the TFU's access to the CTBUS. It reads the data tags from the output of the FIFO and then transmits data onto the CTBUS. The asynchronous FIFO provides all of the communications between the FSMs, allowing them to operate independently; if necessary, each can use a separate clock signal. Appendix A gives a more detailed description of the operation of the TFU.

4.4 PRC Status

The PRC has been fully designed using the HP CMOS14 process and Epoch design tools from Cascade Design Automation and is currently being packaged by MOSIS. The physical and timing specifications of the PRC are shown in Table 4.4. As shown in Figure 4.14, the memory interface consumes approximately one-third of the chip area, while the remaining two-thirds is used by the network interface. Within the network interface, the NIRXs and routing engines utilize two-thirds of the area, since these devices provide most of the PRC's flexibility. The memory interface, on the other hand, divides its area almost equally between the datapath (for buffering, timestamping, and verifying data) and the address/control logic.

Verilog simulations were used to test a single PRC, with the outgoing links connected to the reception ports, under random and contrived workloads. After fabrication, the PRC will be tested using an HP 82000 tester, using scan chains to access the chip's critical state machines and the bus arbitration logic. In addition, the network interface's design allows the TAXI transmission lines to directly connect to the inputs from the TAXI receivers. This allows a single PRC to transmit the packets it receives, greatly simplifying the external circuitry required for testing. The routing engines will also play an important role in testing the chip, since test microprograms can read and write various internal registers in the PRC, reporting the results to the control interface through the notification FIFOs. Microprograms can also generate test traffic on the CTBUS to verify the operation of the bus arbiter and

Parameter	Value
Size	9.0 × 7.3 mm
Transistors	490,000
Power	0.8 watts
Pins	256
Clock	40 MHz

(a) Physical specifications

Component	Clocking	Peak bandwidth
Transmitters	20 MHz, synch	200 Mbits/sec
Receivers	20 MHz, asynch	200 Mbits/sec
Control interface	10 MHz, asynch	N/A
Memory interface	20 MHz, synch	80 MBytes/sec
Internal switch	40 MHz, synch	160 MBytes/sec

(b) Timing specifications

Table 4.4: PRC specifications

the reservation status unit.

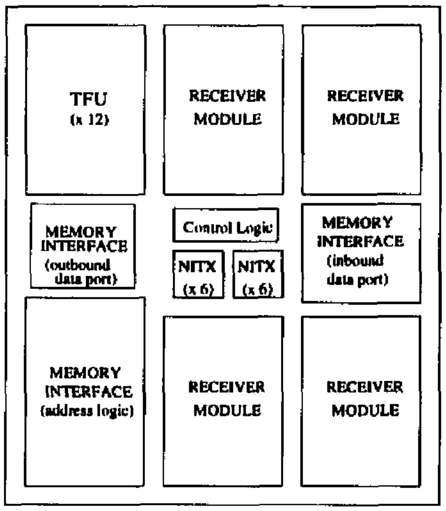
4.5 Comparison to Other Router Architectures

Due to its designed mission, the PRC architecture is unique when compared to other router architectures.

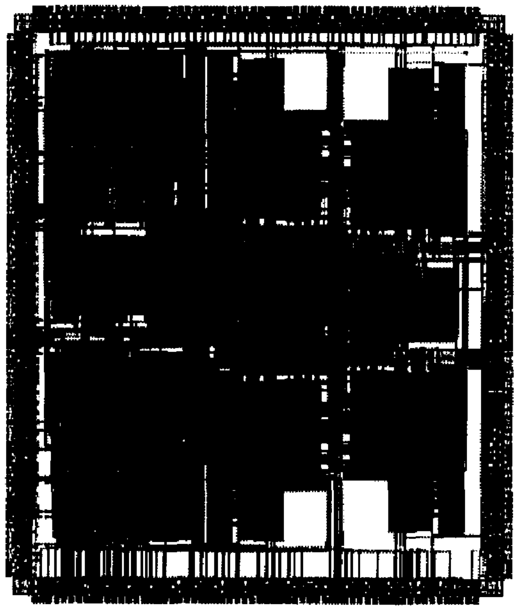
Programmable Routing Controller (James Dolter, 1993) [21]: The PRC, as presented in James Dolter's 1993 Ph.D. thesis [21], had the same design goal as the current PRC — to provide flexible routing and switching via programmable control of incoming packets. While many elements of the earlier design are present in the current PRC, many others have changed significantly.

Several major changes have been made in the architecture: the PRC now provides three virtual channels for each link (as opposed to one previously). The PRC also incorporates the entire network interface (save for the AMD TAXI transmitters) within the ASIC, instead of requiring external FPGAs to implement the link control. The additional logic required for controlling the links and doubling the number of channels necessitated several changes in the PRC architecture. The most significant of these was disassociating the routing engine from the incoming channel, and sharing each routing engine among several channels. In addition, the routing engines moved from the host side of the CTBUS to be nearer the links, shortening the critical delay path through the node. Performance was also improved by hardwiring the data transfer and decreasing the internal cycle time relative to the links.

Post Office (HP Mayfly) [19]: Perhaps the project which is closest to HARTS is the Mayfly system developed at Hewlett-Packard. This system is intended as a back end processor for LISP applications. Based on a hexagonal mesh similar to that used in HARTS,


TFU
(x 12)
RECEIVER MODULE
RECEIVER MODULE
MEMORY INTERFACE (outbound data port)
Control Logic
NITX (x 6)
NITX (x 6)
MEMORY INTERFACE (inbound data port)
MEMORY INTERFACE (address logic)
RECEIVER MODULE
RECEIVER MODULE

(a) PRC floorplan



(b) PRC layout

Figure 4.14: Floorplan of the PRC

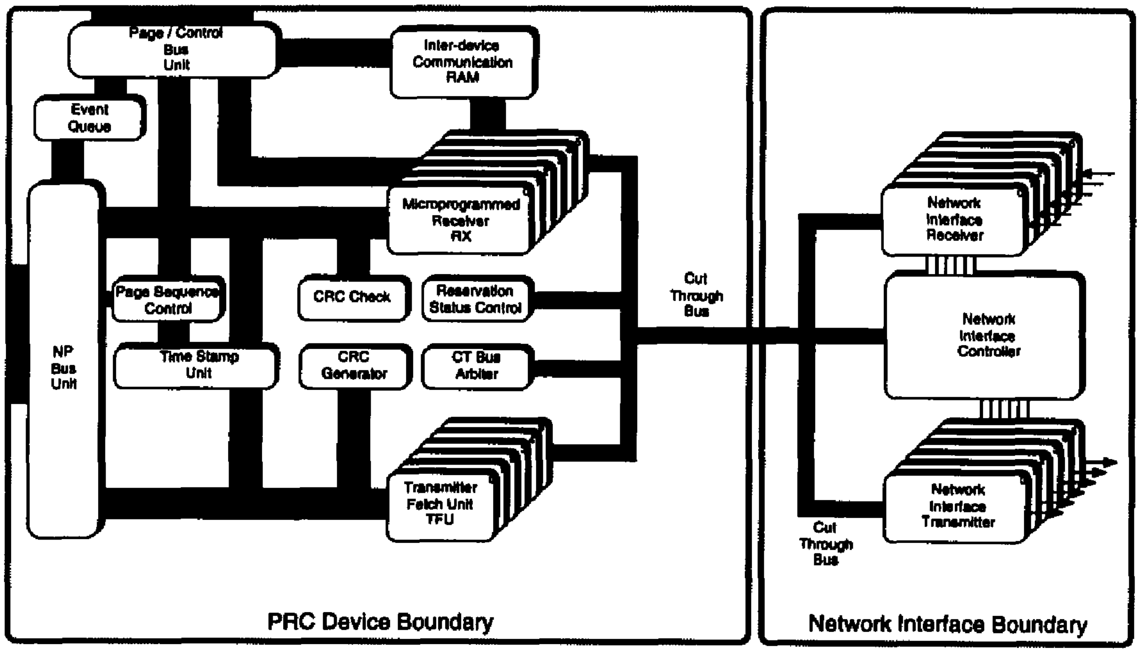

Page / Control Bus Unit
Inter-device Communication RAM
Event Queue
Microprogrammed Receiver RX
NP Bus Unit
Page Sequence Control
CRC Check
Reservation Status Control
Time Stamp Unit
CRC Generator
CT Bus Arbiter
Transmitter Fetch Unit TFU
Cut Through Bus
Network Interface Receiver
Network Interface Controller
Network Interface Transmitter
Cut Through Bus
PRC Device Boundary
Network Interface Boundary

Figure 4.15: 1993 PRC architecture.

Feature	1993	1996
CTBUS width	8	32
CTBUS speed	20 Mhz	40 Mhz
Network bandwidth	160 Mbit/sec	1.28 Gbit/sec
Routing engines	6	4
Connectivity	6	4
Virtual channels	no	3/link
Link control	off-chip	on-chip
Transistors	approx. 400,000	490,000
Die area	187.6 mm^2	68.4 mm^2
Feature size	1.0 μm	0.5μm

Table 4.5: Physical specification comparison.

the Mayfly system is intended for use as a computational processor for a host machine. Its communications support is realized through a device known as the *Post Office*. All communication is through 32-word packets with a fixed packet structure. This approach greatly simplifies the hardware, as does the use of a single, universal routing scheme. No provision is made in the packet format, however, for the inclusion of a checksum. Error detection and handling therefore devolves upon the application processors. Packet construction is also the task of the application processors. Thus, a significant amount of work related to communication processing is placed upon the processors which are also supposed to be executing user code.

Archetypal wormhole router: The original wormhole router was the Torus routing chip [10], which focused on deadlock-free routing in k-ary n-cube systems. By splitting each physical channel into two virtual channels, the designers were able to implement wormhole switching for packets without fear of deadlock. Data flits are transferred between the inputs and the output queues by a crossbar interconnect. Connections can be made efficiently within each switch. As with the Mayfly system, the system supports only a single routing and switching scheme. The Torus routing chip also does not provide an interface from the nodes into the interconnect.

Adaptive virtual cut-through router (POSTECH) [40]: The adaptive virtual cut-through router developed at POSTECH explores the performance tradeoffs between virtual cut-through and wormhole switching, focusing on the gains possible from adaptive routing. It is implemented as a variation of the Torus router [10], but with only one virtual channel per link. Similar to the PRC, the router buffers blocked packets at the local node. The routing scheme is adaptive minimal-path with a dimension-ordered selection scheme, and the authors show how performance is increased relative to the oblivious wormhole router at a minimal cost in hardware.

CHAOS router U. Washington [3, 37]: Another multicomputer router that uses adaptive virtual cut-through switching is the CHAOS router. Unlike the POSTECH router, however, the CHAOS router provides a small *multiqueue* buffer for packets on-chip; if this buffer fills up, packets are derouted down any available link. Its low-load performance is comparable to the wormhole routers; under heavier loads the buffer prevents link bandwidth from being wasted.

CHAPTER 5

Hybrid Switching

Scientific progress goes "Boink"?

— From Bill Watterson's *Calvin and Hobbes*

The effectiveness of a parallel or distributed system is often determined by its switching scheme. The switching scheme directly affects the internode communication latency, and is a primary factor in determining how the network's bandwidth is utilized. This chapter compares the impact of virtual cut-through and wormhole switching upon packet latency, the maximum network throughput, and the resources required for buffering packets at intermediate nodes. Based on this evaluation, this chapter then proposes and evaluate a "hybrid" switching scheme that combines the salient features of both schemes.

Virtual cut-through and wormhole switching are shown to have their strengths and weaknesses. Virtual cut-through switching provides better throughput and lower latencies at heavy loads at the cost of buffering blocked in-transit packets, while wormhole switching only requires a few small flit buffers in the router and completely isolates nodes from in-transit packets. One alternative to improving wormhole switching's performance at higher loads would be to *selectively* buffer blocked packets; this would free some network resources sooner while still isolating nodes from much of the in-transit traffic.

Virtual cut-through and wormhole switching are both cut-through switching schemes, but their performance may differ drastically under different traffic loads. For low traffic loads, the latencies of both schemes are almost identical. This is because in a lightly-loaded network the probability of blocking is very small and the latency is then determined primarily by the length of the packet and the link transmission time. As the traffic load increases, however, the probability of blocking increases, as does the likelihood of blocking other packets. Consequently, networks that use wormhole switching generally saturate from contention well before they exhaust their bandwidth [45, 12]. The effects of this contention can be reduced by increasing the number of virtual channels per physical link [12]. Since either wormhole or virtual cut-through switching may yield shorter packet latencies, depending on the network traffic and the number of hops the packet must travel, it is advantageous to support both switching schemes in order to adapt to a wider range of circumstances. Furthermore, a network which can dynamically switch from one scheme to the other can respond to the offered traffic load and the needs of the system's applications.

To address these tradeoffs, Section 5.2 introduces and evaluates a hybrid switching scheme which balances the use of network resources against the use of memory resources for storing blocked packets. This hybrid scheme decides whether to buffer or stall blocked packets based on a field within the routing header; this field identifies the number of links the packet can hold while stalling in the network. If this threshold is exceeded, the blocked packet buffers.

Section 5.1 compares the performance of virtual cut-through and wormhole switching operating on SPIDER. This comparison focuses on three metrics: the mean communication latency, the memory resources required by each scheme, and the maximum achievable throughput of the network. Section 5.2 introduces hybrid switching and evaluates it relative to both virtual cut-through and wormhole switching. The chapter concludes with Section 5.3, which summarizes the main contributions of this chapter and future directions.

5.1 Comparing Wormhole and Virtual Cut-through Switching

To more accurately compare the performance of the various routing and switching schemes, and also to evaluate the performance of SPIDER we have developed a cycle-level discrete-event simulator [21, 57]. Written in C++, this simulator accurately models the flow of the individual bytes of packets through SPIDER. This captures features such as the low-level flow control, bus arbitration delays, and microcode execution time. While the simulator does not model the actual protocol software executing on the host, it does capture the effects of these protocols on packets that buffer at intermediate nodes.

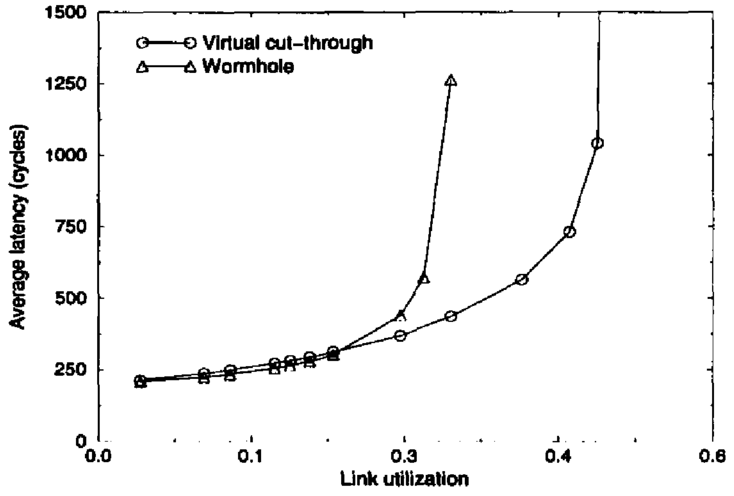


Figure 5.1: Packet delivery latencies for virtual cut-through and wormhole switching.

This section presents the results of a set of experiments that vary the packet generation rate while holding other parameters constant. At each node, the inter-arrival time of packets for transmission conformed to a negative exponential distribution. Packet destinations were uniformly distributed across all of the nodes (except where otherwise specified). The simulations also used a fixed packet size of 64 bytes.

To focus the experiments on the switching scheme, all packets use a static, dimension-ordered routing scheme [11]. Furthermore, most of the simulations use an unwrapped square mesh topology where only one virtual channel per link is required to prevent deadlock under wormhole switching. This allows the switching schemes to be compared with the same number of virtual channels.

To collect the data, the network was first placed into a steady state and data collected for 2000 packets at each node. For latency, the standard error of the mean is less than 5 cycles for the 95% confidence interval on all traffic loads. When the network is saturated, however, this steady state cannot be achieved.

5.1.1 Latency

In Figure 5.1, the mean packet latency is shown as a function of the link utilization, which

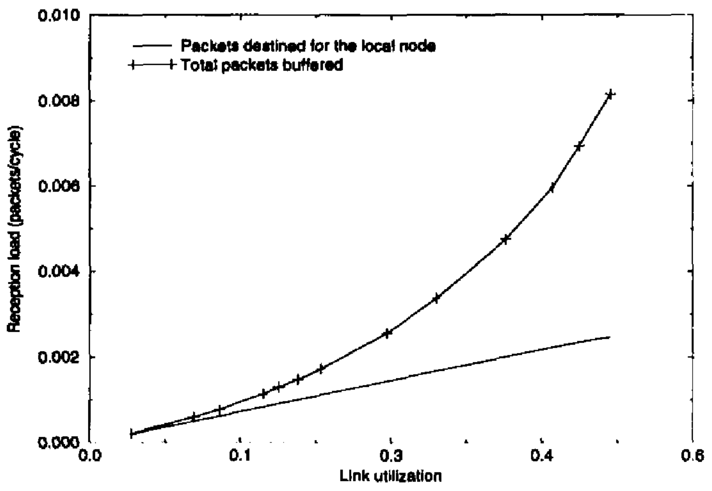


Figure 5.2: Rate of in-transit packet arrival.

is given as a percentage of the maximum capacity of the network's physical links. When the offered load is low, the average packet latency is the same under both switching schemes. Wormhole, however, reaches saturation under lighter loads than virtual cut-through due to contention for channels, resulting in a dramatic increase in the mean packet latency. Saturation occurs at a link utilization of 0.2 in this experiment. Other experiments have shown that these trends are not significantly affected by packet length or the topology of the network.

5.1.2 In-transit Load

While virtual cut-through can support a greater traffic load than wormhole, it also buffers packets at intermediate nodes. Each packet that buffers at a node consumes memory resources for its storage and control resources to process the header. If packets are buffered within the switch itself, the buffer space is necessarily limited in size. External buffers (such as those used by the PRC), on the other hand, may be much larger but are generally slower. In addition, managing these larger buffers requires either host interaction or more hardware in the router.

The relative costs of the two schemes are illustrated for a node-uniform traffic load on an unwrapped 8×8 square mesh in Figure 5.2. This figure shows the average rate (in packets per cycle, per node) of packets buffering at a node using virtual cut-through switching.

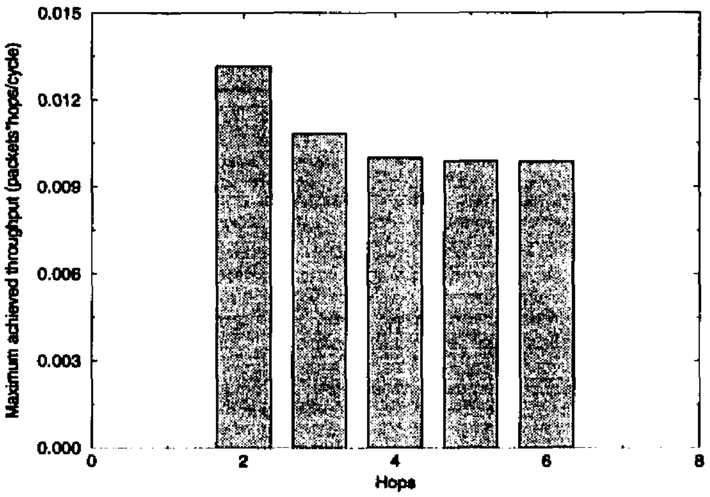


Figure 5.3: Maximum throughput for wormhole switching under a hop-uniform traffic load.

This rate is composed of two components: the "in-transit" rate and the "destination" rate. The former is the average rate of packets that are destined for other nodes buffering at a node, while the latter is the average rate of packets buffering at a node that are destined for that node. The in-transit rate is the region between the destination rate (the lower curve) and the total rate of packets buffering (the higher curve). At low loads, almost all packets successfully cut through and the in-transit arrival rate is very low. As the load increases, the probability of cut-through also drops, resulting in an increased in-transit packet arrival rate. When the network is in or near saturation, the arrival rate of in-transit packets surpasses the rate of packet generation. In this case, the load on the host for buffering and rescheduling these packets is severe.

5.1.3 Maximum Achievable Throughput

Wormhole and virtual cut-through switching are affected differently by packet distance. This can be directly shown by varying the average number of hops that packets travel. This was accomplished through a hop-uniform destination mapping, where every packet travels the same number of hops. In order to spread traffic uniformly through the network, a wrapped 8×8 square mesh (torus) is used with two virtual channels per link (the minimum to prevent deadlock under dimension-ordered routing).

Figure 5.3 shows the maximum throughput (in packets per cycle) of wormhole switching as a function of the hop count of packets. Using wormhole switching, the network saturates under a lighter link load as the packet distance increases. This is due to increased contention: packets are traveling more hops, and thus stalling more links when blocked. This has a snowball effect: blocked packets stall more links, and block other packets that may then block still other links. The overall effect, therefore, is to degrade the maximum achievable throughput. Virtual cut-through switching, on the other hand, does not exhibit this behavior, as it uses memory resources and not network resources to stall blocked packets. Its peak throughput is dependent upon the link load and not upon packet distance.

The maximum throughput of a network using wormhole switching can be increased by adding virtual channels [12], or by significantly enlarging the number of flits buffered at each node. Adding virtual channels on each link improves throughput by allowing packets to "bypass" stalled packets. The primary cost is in the increased complexity of the crossbar connecting the reception channels to the transmission channels — either the size of the crossbar must be increased, or the arbitration becomes more complex [8]. Giving each virtual channel a flit buffer large enough to hold one packet should significantly improve throughput — each blocked packet only stalls a single link. Similarly, buffers capable of holding half of a packet's flits will prevent blocked packets from stalling more than two links.

5.1.4 Wormhole Switching with Large Buffers

The previous discussions and results have assumed that packets are sufficiently long, so that their "tail" of reserved channels stretches from the current head of the packet back to the source. By increasing the portion of the packet buffered at each node, however, the length of the tail can be reduced.

Figure 5.4 shows the average packet latency for wormhole switching with up to 8 words (half of a packet) buffered at the input of each node. This limits the maximum number of links that a packet can hold while stalling to two. This reduction results in a significant increase in performance — both the average packet latency (at higher loads) and the maximum throughput of the network are increased when compared to wormhole switching. The "buffered" wormhole scheme also provides a lower average packet latency at mid-range loads than virtual cut-through. This is due to the design of the PRC — packets that buffer

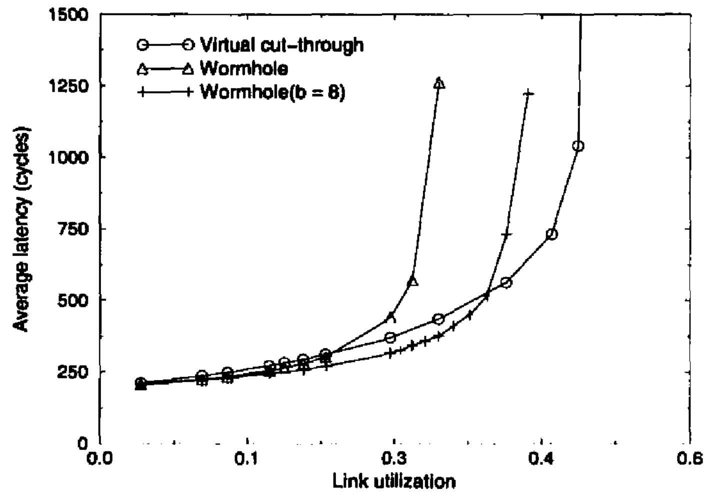


Figure 5.4: Average packet latency for "buffered" wormhole.

at an intermediate node under virtual cut-through switching must be completely buffered prior to retransmission. Since packets are still in the network with the buffered wormhole scheme, they can be forwarded to the next node as soon as the link comes free. The effect is also exaggerated by the disparate speeds of the PRC's memory and network interfaces.

One major drawback to providing such large buffers for packets at the inputs is the cost of implementing them for larger packet sizes and higher numbers of virtual channels. Since the cost is directly proportional to the largest packet size permitted in the network and the number of virtual channels on each link, the next section will introduce a hybrid switching scheme that uses a central (off-chip) buffer for packets that is cheaper to implement and can be much larger in size.

There are significant differences in the performance of wormhole and virtual cut-through switching under different traffic loads. Wormhole switching requires fewer buffers than virtual cut-through, but its maximum throughput is relatively limited, dependent on packet distance, and saturates under relatively light traffic loads. At heavy loads, virtual cut-through (as predicted) outperforms wormhole, but the cost of buffering in-transit packets can cancel out the performance gains. The following section presents a hybrid switching scheme that addresses the shortcomings of both schemes.

5.2 Evaluating Hybrid Switching

This section examines how hybrid switching provides a level of performance that bridges the gap between virtual cut-through and wormhole switching. We evaluate hybrid switching's performance relative to these schemes using the same metrics as the previous section.

5.2.1 Hybrid Switching

A "hybrid" switching scheme dynamically combines wormhole and virtual cut-through switching, using both network and memory resources to store blocked packets. There are a number of potential hybrid switching schemes that meet this requirement. To implement these schemes efficiently, however, the switching decisions should be based on information available in the packet header or at the local node.

In Section 5.1.3, we saw that increasing the number of links held by packets degraded the throughput achievable with wormhole switching. One method for improving wormhole's performance under heavier loads would be to relieve contention by buffering packets that cannot advance yet are stalling several links behind them. This scheme would avoid the long "tails" of stalled links held by blocked packets, reducing contention. Such a switching scheme would dynamically combine virtual cut-through and wormhole switching to provide improved packet latencies and a higher achievable throughput than wormhole alone, without buffering packets as often as virtual cut-through.

The hybrid algorithm used in the remainder of this paper decides whether to buffer or stall blocked packets based on a field within the routing header; this field identifies the number of links the packet can hold while stalling in the network. If this threshold is exceeded, the blocked packet buffers. The system can dynamically vary this threshold depending on the packet's needs or the current network load by changing the initial value of this header field.

Implementing the scheme is simple: a field in the routing header is set to h when the packet is generated and then decremented after every hop until it reaches 0. While $h > 0$, the packet will stall if blocked. Once $h = 0$, the packet buffers when blocked. Buffering the packet resets h to its initial value. Virtual cut-through and wormhole switching can be viewed as special cases of this algorithm: wormhole switching is equivalent to hybrid switching with $h = \infty$, while hybrid switching with $h = 0$ effectively implements virtual cut-through switching.

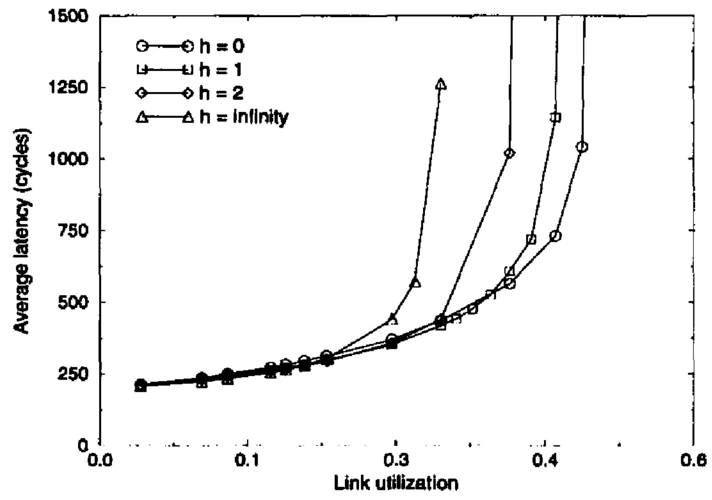


Figure 5.5: Average packet delivery latencies for hybrid switching, compared to virtual cut-through and wormhole switching.

The requirements for supporting hybrid switching are not much greater than those for supporting wormhole or virtual cut-through switching alone. When a router receives a packet, it must be able to determine how many hops the packet has traveled. If the link reservation fails, the router can then choose to buffer the packet. Due to the reduced in-transit load, the buffer requirements for hybrid switching are significantly reduced compared to virtual cut-through switching.

In the following simulations, all packets use the same dimension-order routing as in Section 5.1. As before, the simulations use a fixed packet size of 64 bytes, except where indicated otherwise.

5.2.2 Latency

In Figure 5.1, we saw that wormhole switching saturates from contention well before virtual cut-through, resulting in dramatically increased latencies. By preventing blocked packets from holding more than h links, hybrid switching decreases contention. The effects are shown in Figure 5.5, which compares the average packet latencies for wormhole switching, hybrid switching with $h = 1$, hybrid switching with $h = 2$, and virtual cut-through switching.

At very low loads, with a low probability of blocking, the mean latencies of the schemes

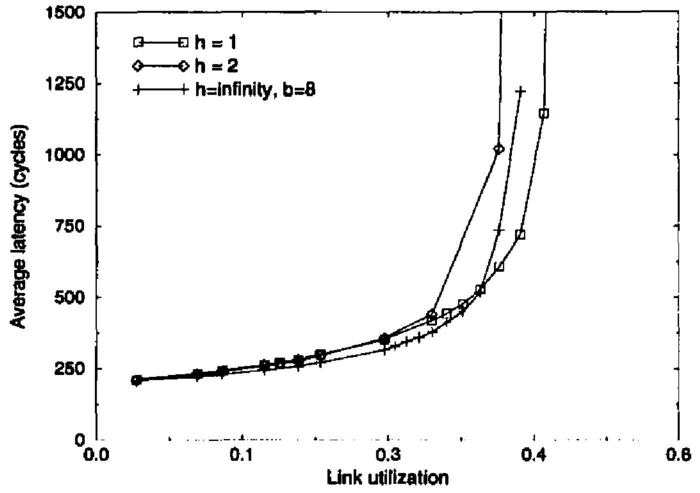


Figure 5.6: Average packet delivery latency for hybrid switching, compared to "buffered" wormhole switching.

are similar. Once this probability rises, however, hybrid switching provides lower packet latencies than wormhole switching. As h decreases, the network can handle a higher offered load without saturating. Higher values of h will resemble pure wormhole switching more closely — saturating at lower offered loads. These trends also hold over a range of packet sizes and network topologies.

The effects of buffered wormhole switching (as discussed in Section 5.1.4) are similar to hybrid switching, as both schemes limit the number of links a packet can hold while blocking in the network. They differ in one main aspect — hybrid switching may completely remove a packet from the network prior to its destination. This is both a plus and a drawback — hybrid switching can use a large external buffer for packets, allowing larger packet sizes to be supported. At the same time, use of this buffer may prevent packets from being retransmitted until they have been completely received, depending on the router's implementation.

Figure 5.6 compares the buffered wormhole scheme with hybrid switching, for $h = 1$ and $h = 2$. As expected, all three schemes exhibit similar performance, although buffered wormhole slightly outperforms both hybrid schemes at lower loads. As with virtual cut-through, this difference may be attributed to the design of the PRC, which does not allow

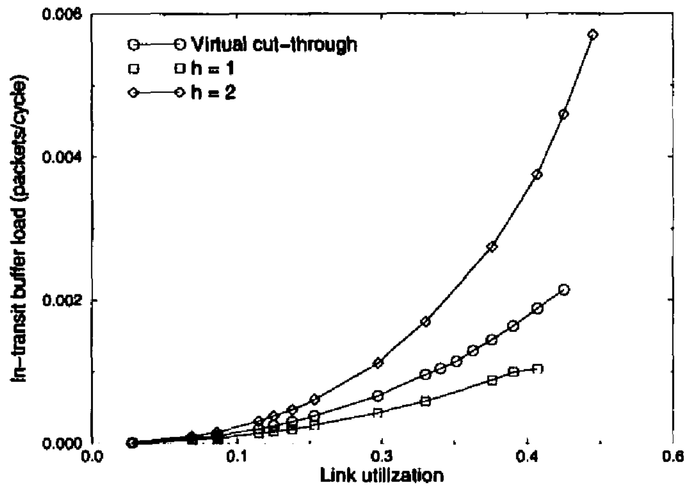


Figure 5.7: In-transit packet load for virtual cut-through and hybrid switching.

packets that buffer to perform partial cut-throughs.

5.2.3 In-transit Load

One of the primary advantages of wormhole switching is that it completely insulates nodes from in-transit traffic; the cost, however, is the consumption of network bandwidth by blocked packets. Virtual cut-through switching utilizes the network's bandwidth more efficiently, but can require nodes to handle large amounts of in-transit traffic (as shown in Section 5.1). By only buffering *some* blocked packets, hybrid switching significantly reduces this load.

A comparison of the in-transit load for hybrid switching and virtual cut-through switching is shown in Figure 5.7. This graph shows the arrival rate of in-transit packets for a range of offered loads. Even at low loads, with a very high probability of cut-through, hybrid switching significantly reduces the rate of in-transit traffic when compared to virtual cut-through. As the offered load increases, the probability of cut-through decreases and the in-transit load increases. At high loads, virtual cut-through switching uses at least $h + 1$ times more memory resources than the hybrid scheme, since the hybrid algorithm allows packets to buffer at most once every $h + 1$ hops. The actual reduction in buffering is often larger. For example, a packet traveling five hops using virtual cut-through may buffer up to four times, while hybrid with $h = 2$ will only buffer it at most once.

5.2.4 Maximum Achievable Throughput

Figure 5.8 shows the maximum achieved throughput (in packet-hops per cycle) as a function of the number of hops traveled by each packet. As in Figure 5.3, the applied traffic load is hop-uniform — every packet travels the same number of hops. The maximum throughput is only shown for those distances greater than h — when each packet travels h hops or less, hybrid switching is indistinguishable from wormhole switching.

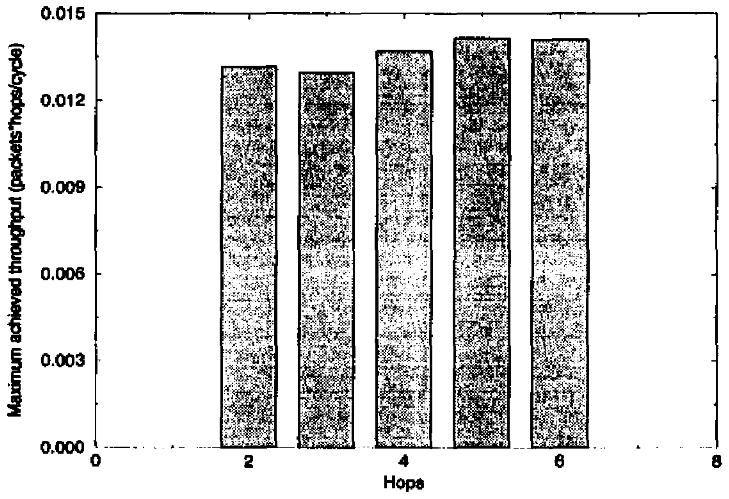
Unlike wormhole switching and virtual cut-through, however, the maximum throughput for hybrid switching *increases* with the number of hops packets travel. This phenomenon can be explained by examining the proportion of packets in each case that have traveled more than h hops without buffering. As the average number of hops traveled by each packet increases, the percentage of packets that are willing to buffer if blocked increases. This alleviates contention in the network, preventing early saturation.

5.2.5 Virtual Channels

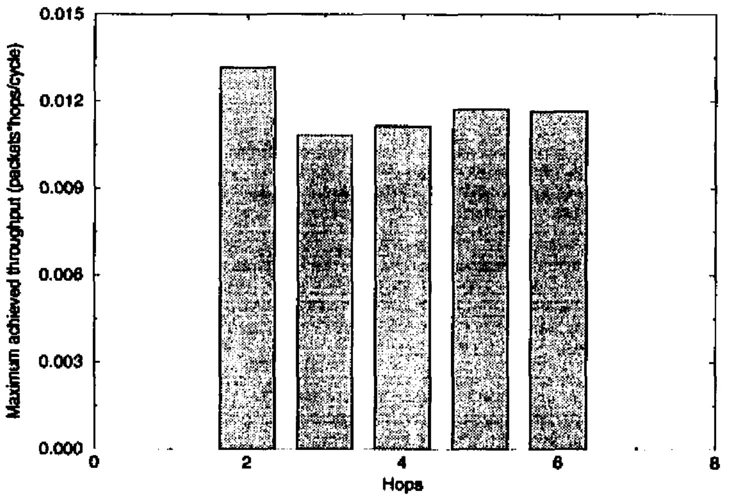
Dally [11, 12] introduced virtual channels to prevent deadlock in wormhole switched networks. Since then, virtual channels have been used to improve network throughput [12] and to partition different traffic classes to minimize interactions [58].

Virtual channels improve network throughput in wormhole-switched networks by allowing packets to bypass other blocked packets, thus utilizing otherwise idle network bandwidth. Since hybrid switching may also idle links by stalling packets in the network, it can also benefit from virtual channels. Figure 5.9 shows the effects of increasing the number of virtual channels on the average packet latency and peak throughput of hybrid switching. Under lighter loads, increasing the number of channels has little impact on the mean packet latency. The primary effect of increasing the number of channels is an increase in the maximum throughput which the network may support. The decreasing benefit of higher numbers of virtual channels is also seen for similar simulations using wormhole switching.

In wrapped topologies, many wormhole routing schemes will idle or underutilize virtual channels to prevent deadlock. While packets that will stall when blocked must utilize deadlock-free routing schemes, packets where h has reached 0 may take advantage of avail-

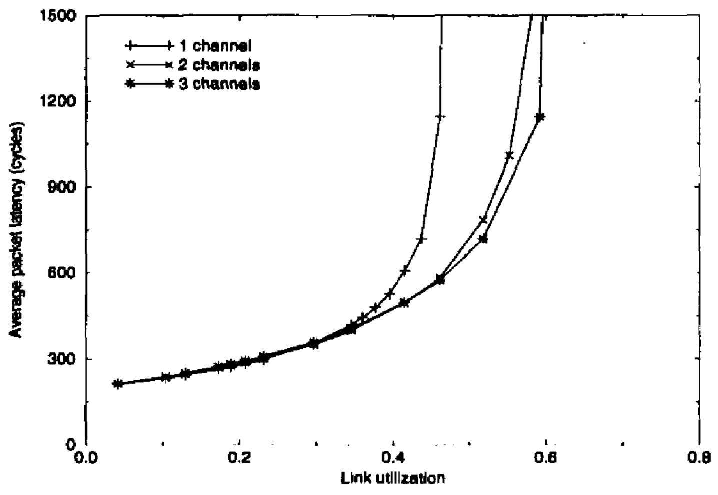


(a) Hybrid, $h = 1$

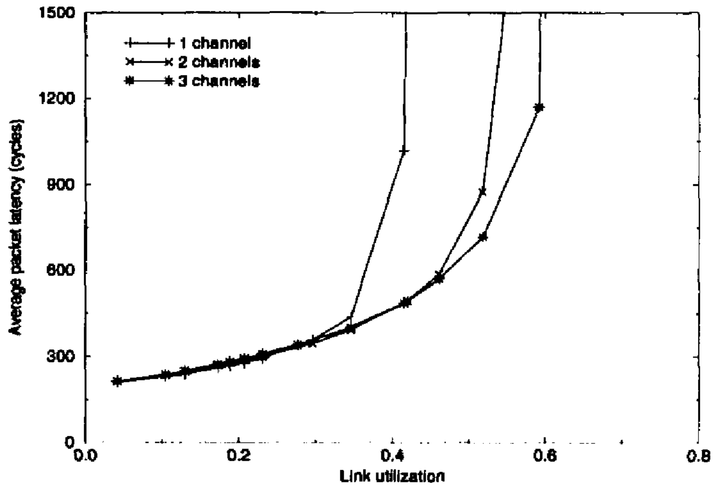


(b) Hybrid, $h = 2$

Figure 5.8: Maximum throughput under a hop-uniform traffic load.



(a) Hybrid, $h = 1$



(b) Hybrid, $h = 2$

Figure 5.9: Effects of increasing available virtual channels.

able channels without regard to preventing deadlock, since they will buffer if blocked. This increases the probability of cut-through for packets by considering channels that could not otherwise be used.

5.2.6 Discussion

The simulations in this paper did not restrict the number of buffers at each node. When the packet buffers are implemented on the same die as the router, the number and size of the buffers is restricted. By buffering fewer packets than virtual cut-through, hybrid switching reduces the buffer space needed. In addition, hybrid switching schemes can take the available buffer space into account when deciding whether to buffer or stall a blocked packet. By buffering only packets that are currently holding several links and stalling others, hybrid switching can effectively utilize limited buffers.

This section has evaluated only one variant of hybrid switching. Another promising hybrid scheme uses a "credit" scheme to determine when to buffer a blocked packet. Under this scheme, each packet header contains a field indicating the maximum number of times it can be buffered — every time the packet buffers, the field is decremented. Once this value reaches 0, the packet will stall in the network. This scheme allows packets to stall more channels, but buffering other packets should prevent network congestion. The combination of a restriction on the number of times a packet can buffer with h-hop hybrid switching also holds promise.

Hybrid switching also allows the system to dynamically determine (on a per-packet or system-wide basis) whether network or buffer resources are used to store blocked packets. This can be implemented by setting the initial value of h at the source of the packet to reflect whether the packet should consume more network or buffer resources when blocked. For example, large packets that will be traversing a large number of links may initially use larger values of h to reduce the number of times they buffer. On the other hand, systems requiring high bandwidth can use smaller values of h to shift the load to the network's buffers.

Hybrid switching uses both network and memory resources to store blocked packets, addressing the shortcomings of other cut-through switching schemes. Using network resources to store the packets can often have a snowball effect, creating contention throughout the network that limits throughput. Schemes that use memory resources, on the other hand, increase the system's communication overhead. Through hybrid switching, we attempt to balance these concerns. Potentially, the switching decision could be also based on the distance still needs to travel, or the number of buffers available at the local node. In addition, the decision could be time-based: packets could stall for some small amount of time if blocked in the hopes of being able to cut through, and then buffer. Alternately, packets that are blocked just short of their final destination could block in the network, while others that are blocked near their source would buffer. This would keep packets from blocking in the network more than once or twice.

5.3 Conclusions

The switching scheme used by a point-to-point network is a major factor in determining the latency, throughput, and overhead of communication. The various cut-through switching schemes all improve latency over store-and-forward switching (unless the network is saturated), but each has its strengths and weaknesses.

5.3.1 Related Work

One other cut-through switching scheme bears mentioning in this context. *Wormhole Intracluster Cut-through Intercluster* (WICI) switching addresses the same performance issues as hybrid switching by partitioning networks into *clusters* of nodes [52]. Within the clusters, packets use wormhole switching; between clusters, they use virtual cut-through. Thus, only nodes at the edges of clusters have buffers for blocked packets. The performance of both schemes is similar; WICI switching only requires some nodes to have packet buffers, but hybrid switching distributes the load more evenly and allows dynamic selection of switching schemes.

5.3.2 Summary

As shown in this chapter, virtual cut-through does not limit the achievable network throughput but does impose a significant load on nodes for storing and retransmitting in-transit packets. Wormhole, on the other hand, stalls blocked packets in the network and does not require large buffers for blocked packets, it is cheaper to implement. Its maximum throughput, however, is limited by contention for outgoing links.

This chapter has introduced the concept of hybrid switching, which dynamically chooses whether to buffer or stall blocked packets in order to balance resource consumption and improve network throughput. This scheme combines features of both wormhole and virtual cut-through switching by buffering a small fraction of blocked packets and limiting the number of links that blocked packets can hold. This significantly reduces the buffer requirements for in-transit packets when compared to virtual cut-through, while providing higher maximum throughput than wormhole switching. In this manner, hybrid switching bridges the performance gap between other cut-through switching schemes.

CHAPTER 6

Experiments in Flexible Routing and Switching

I'm telling everyone the world will end in year the (sic) 2000.
My compelling logic is that 2000 is a big round number.
— Dogbert, in Scott Adams' *Dilbert*

This chapter illustrates how the PRC's flexibility may be used to address different application needs. To show this, we will compare the relative performance of the different routing and switching schemes on several different traffic patterns. As in Chapter 5, the experiments were conducted with `pp-mess-sim` [21, 57] providing a cycle-level model of the PRC.

6.1 Non-uniform Traffic Loads

Adaptive routing algorithms can improve network performance by basing their routing decisions on local information. However, as shown in this section, these local decisions can increase network congestion for some communication patterns. The graphs show average latency for wormhole switching under both dimension-ordered and adaptive routing; virtual cut-through experiments showed the same qualitative trends. The adaptive routing algorithm is a fully-adaptive minimal routing scheme that requires two virtual channels per link to prevent deadlocks [22]; in these experiments, both routing algorithms employ a pair of virtual channels to enable fair performance comparisons. The dimension-ordered routing algorithm uses the extra virtual channel to reduce contention between packets traveling on the same link [12, 56].

Figure 6.1 compares the performance of the adaptive and oblivious schemes under a

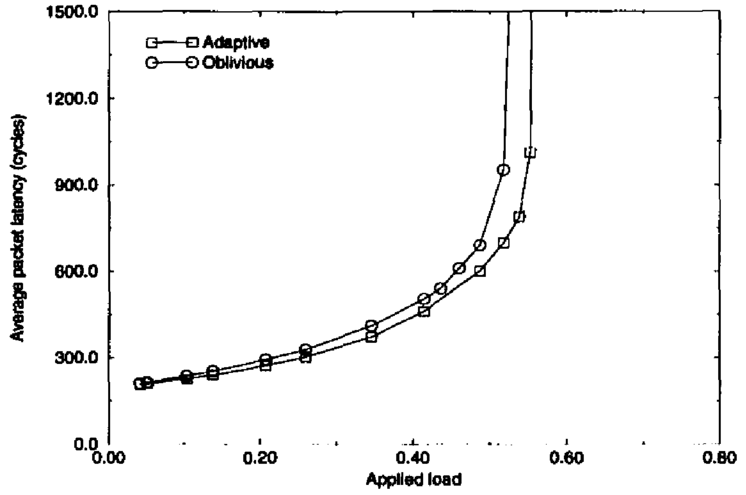


Figure 6.1: Comparison of wormhole routing algorithms under uniform destination traffic.

node-uniform destination traffic load. Due to the uniform nature of traffic, opportunities for adaptive routing are relatively limited and the adaptive scheme is only slightly better than the oblivious. Under a dimension reversal traffic load, where each node (c, d) communicates only with node (d, c), the opportunities for adaptive routing are much better. This is reflected in the improved packet latencies for adaptive routing relative to oblivious routing in Figure 6.2.

This does not hold true for all non-uniform traffic loads, however. Figure 6.3 compares these schemes under a bit-complement traffic load. In an 8×8 square mesh, the bit-complement permutation requires source node (c, d) to communicate with node $(7-c, 7-d)$. As a result, all packets must eventually cross both the middle row and the middle column of the mesh to reach their destinations, irrespective of the routing algorithm. Because of this, many local routing decisions made by the adaptive scheme near the edges of the mesh actually move packets closer to the center of the network. Since this is already the most congested area in the network, the end-to-end latency is actually increased. Oblivious dimension-ordered routing, on the other hand, forces many packets to travel along paths that avoid the center of the network. Thus, it keeps the network traffic more uniformly spread through the mesh, improving performance. In addition, the additional routes considered by adaptive schemes source nodes to inject more packets into the network, further increasing

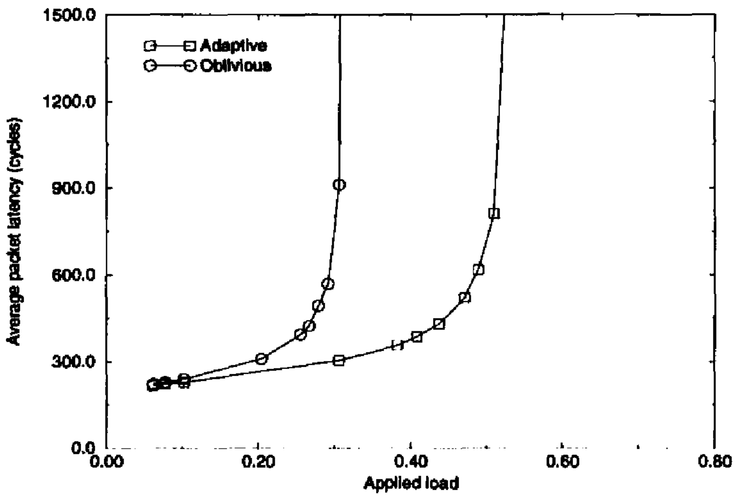


Figure 6.2: Comparison of wormhole routing algorithms under dimension reversal traffic.

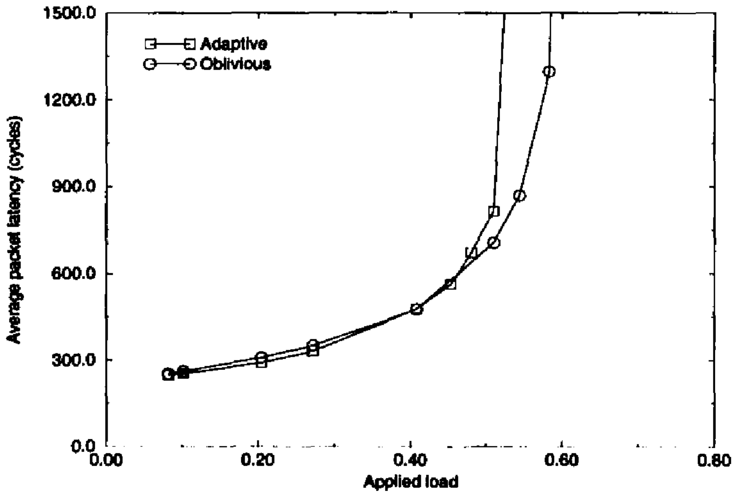


Figure 6.3: Comparison of wormhole routing algorithms under bit-complement traffic.

contention in the network. Hence, in some situations, restricted routing flexibility can effectively limit the overuse of network resources [26].

6.2 Handling Bimodal Packet Lengths

This section evaluates the use of adaptive routing, cut-through switching, and virtual channels to improve performance in a network that carries a mixture of short control packets and long data packets; such *bimodal* length distributions are common in multicomputer applications [9, 35, 36, 64]. The experiments consider an 8×8 torus network of PRCs with three virtual channels on each physical link. Network traffic consists of an even mixture of short, 64-byte packets and long, 256-byte packets; hence, small packets account for 20% of the traffic load. Each node generates traffic independently, with uniform random selection of destination nodes and exponentially-distributed interarrival times. For simplicity, we will refer to the various schemes by the shorthand notations of Table 6.1.

Many systems want to support larger packet sizes, whenever possible, to reduce the host overheads for packetization, context switching, and message reassembly. This often results in a mix of small packets and maximum-size packets that significantly impact the performance of the smaller packets. Figure 6.4 compares the average packet latency for 64-byte packets under two different packet length distributions. From this, we can observe that the average packet latency — even for the same link utilization — is much higher with the bimodal load than a uniform load of 64-byte packets. This is due to the small packets being blocked behind the larger packets, thus incurring large delays.

Several researchers have proposed methods for handling these bimodal traffic patterns. Kim and Chien [35] proposed adding virtual channels and adaptivity to eliminate the interference of long packets on short ones in wormhole-switched networks. Konstantinidou [36], as part of his Segment router architecture, proposed dividing the network into two distinct virtual networks, with distinct switching schemes for each. Short packets would use one virtual network with virtual cut-through switching, while long packets employ wormhole switching on the other virtual network. This allows short packets to avoid the long delays incurred from blocking behind much longer packets, without requiring large buffers for storing blocked data packets. We will investigate how these schemes impact performance for both traffic classes, and examine alternative methods as well.

Scheme	Routing	Selection	Switch	Channels
VCA-1	Adaptive	Dimorder	VC	1
VCA-3	Adaptive	Dimorder	VC	3
WHO-2	Oblivious	Dimorder	WH	2
WHO-3	Oblivious	Dimorder	WH	3
WHA-3	Adaptive	Dimorder	WH	3
H1O-2	Oblivious	Dimorder	Hybrid, $h = 1$	2
H2O-2	Oblivious	Dimorder	Hybrid, $h = 2$	2

Table 6.1: Notation for routing and switching schemes.

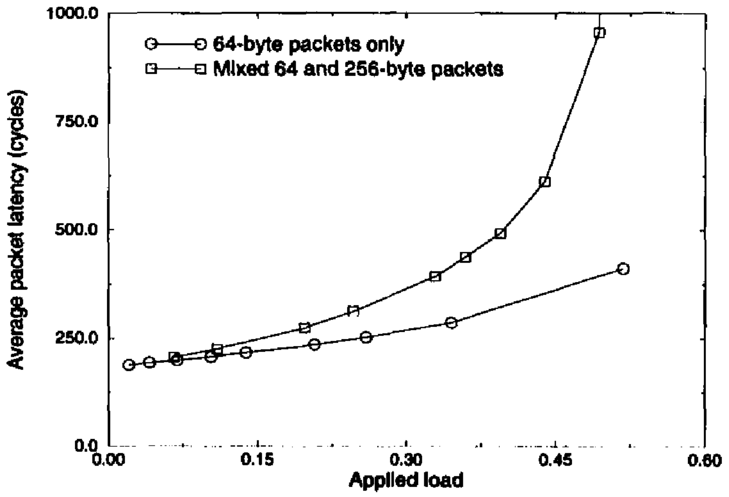


Figure 6.4: Comparison of average packet latency for 64-byte packets using WHO-3 under uniform and bimodal packet length distributions.

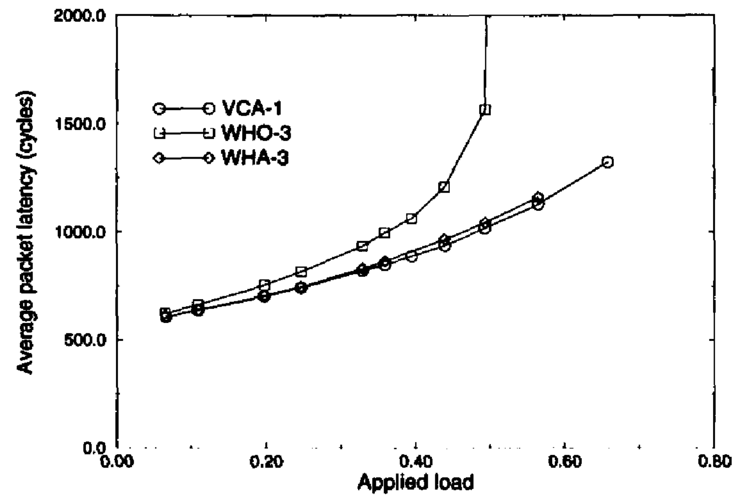
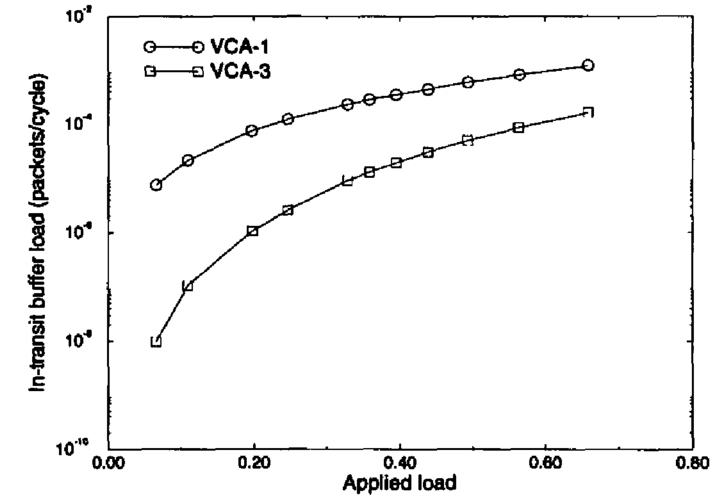


Figure 6.5: Average packet latency of 256-byte packets with a bimodal packet length distribution. The trends for 64-byte packets are similar.

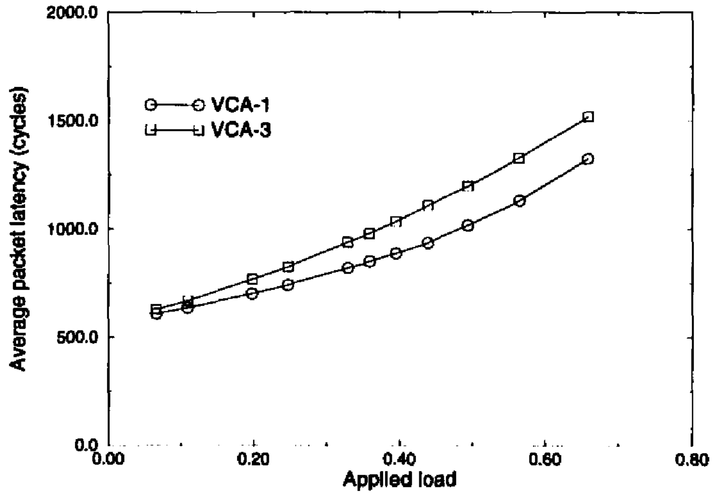
6.2.1 Uniform Routing/Switching Policies

This section examines how three different schemes perform under the bimodal traffic load, with all packets using the same scheme. Figure 6.5 compares the average packet latency for 256-byte packets for VCA-1, WHO-3, and WHA-3. Not surprisingly, the adaptive routing schemes outperform the oblivious scheme, even though the virtual cut-through scheme only uses a single virtual channel. Unfortunately, however, both of the adaptive schemes penalize the smaller packets in the same manner as the oblivious wormhole scheme.

In addition, since the PRC has three virtual channels available for each link, we might consider increasing the probability of packet cut-throughs under virtual cut-through switching by making all of the virtual channels available. The probability of cut-through has significantly increased, as shown by the rate of packets buffering in Figure 6.6(a). Surprisingly, however, the average packet latency for VCA-1 in Figure 6.6(b) is lower than for VCA-3! This result is due to the increased network congestion resulting from the additional injection ports of VCA-3; more packets can enter the network, even though the existing bandwidth



(a) Buffer load



(b) Average latency for 64-byte packets.

Figure 6.6: Comparison of VCA-1 and VCA-3.

is already taken.

6.2.2 Traffic Partitioning

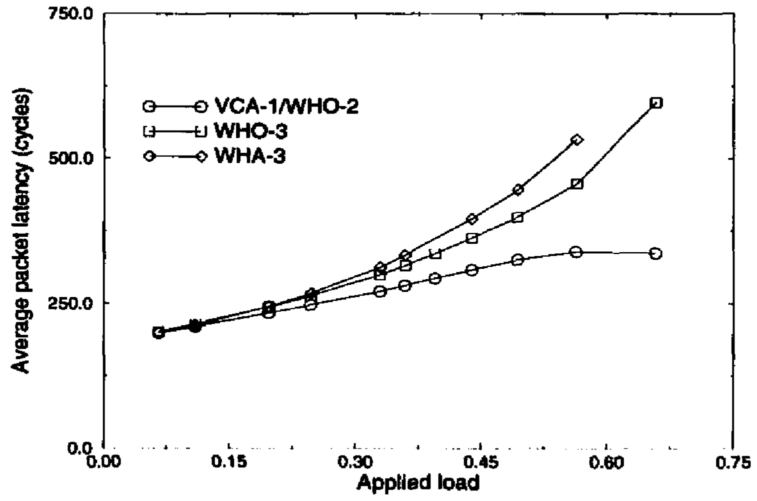
Konstantinidou's Segment router improves the performance of the short packets by segregating them onto a separate virtual network using adaptive virtual cut-through; meanwhile, the long packets still use oblivious wormhole routing [36]. Thus, the short packets use VCA-1, while the long packets use WHO-2 in the Segment router scheme. Figure 6.7 shows how the performance of this scheme compares with networks using WHO-3 or WHA-3 for all packets. The control packets obviously benefit from this partitioning: the average packet latency is significantly lower than the non-partitioned schemes. Latency actually improves somewhat when the long packets saturate their channels, freeing additional bandwidth for the short packets. The drawback to this scheme, however, is the reduced saturation point for the long packets; since these represent 80% of the applied load, many applications might benefit more from adaptive routing than traffic partitioning.

Because of the flexibility of the PRC architecture, a wide range of other traffic partitioning schemes can be contemplated and evaluated. By replacing the oblivious wormhole routing used for long packets with H1O-2 or H2O-2, as in Figure 6.8, we can raise the saturation point of the long packets. The impact on the latency of short packets is minimal unless the long packet network saturates; in these conditions, the additional buffer load from the large packets buffering increases the average latency of the short packets.

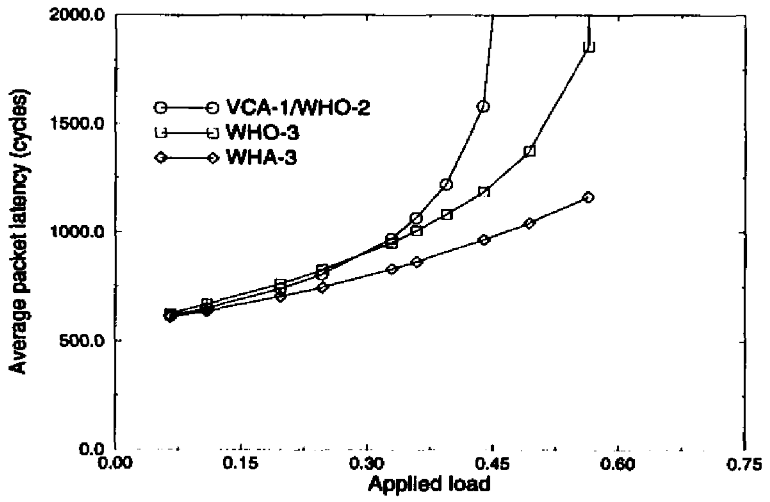
Another possible method for improving the overall system performance would be to differentiate the switching schemes used by the traffic classes without segregating them onto different virtual networks. For example, we can have short packets employ adaptive virtual cut-through on all 3 channels (VCA-3), while the long packets use WHA-3 to reduce the buffer load. Figure 6.9 shows the relative performance of this VCA-3/WHA-3 scheme to the Segment router scheme — the average packet latency for short packets is only slightly higher even under heavy loads. The longer packets, on the other hand, benefit significantly from the increased routing adaptivity and the extra virtual channel.

Other possible traffic partitioning schemes that have been evaluated include the following:

WHA-3 short, WHO-2 long: By allowing short packets to use an extra virtual channel and adaptive routing, this scheme improves their performance in a fashion similar to



(a) 64-byte packets



(b) 256-byte packets

Figure 6.7: Comparison of Segment router traffic partitioning with WHO-3 and WHA-3.

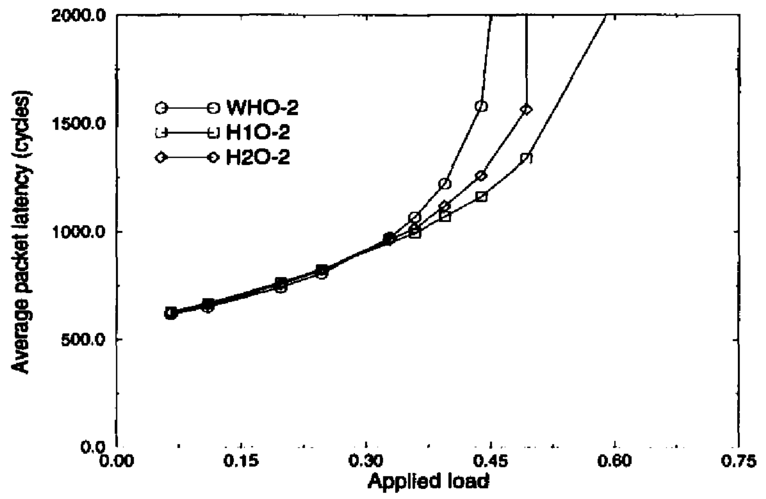


Figure 6.8: Comparison of oblivious wormhole and hybrid switching in the Segment scheme (average latency for 256-byte packets).

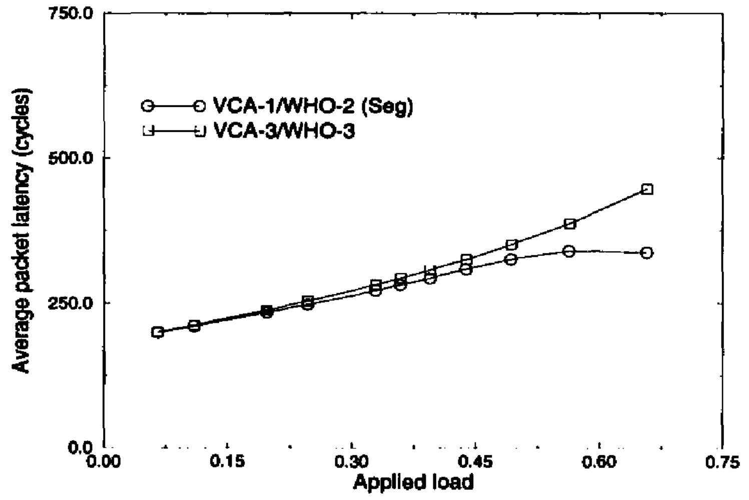
the Segment scheme. Under heavy loads, however, the short packets are unable to use the network bandwidth as well as under the Segment scheme. The long packets see only a tiny improvement in latency.

VCA-3 short, WHO-3 long: This scheme differs from the VCA-3/WHA-3 scheme described above only in using adaptive routing for the longer packets; its overall performance is not as good under a uniform destination load.

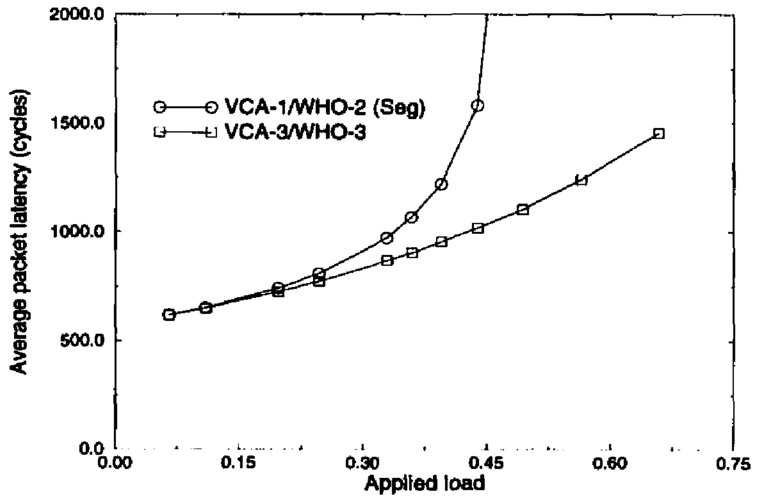
VCA-3 short, WHO-2 long: This scheme improves slightly on the Segment scheme by allowing short packets to route on any of the virtual channels; the performance of the short packets is very slightly improved, without any discernable impact on latency for long packets.

6.2.3 Summary

This section has evaluated a number of routing, switching, and traffic partitioning solutions for handling traffic loads with bimodal packet length distributions. Given the major constraint of the PRC architecture — 3 virtual channels per link — the best performance is found by using the VCA-3/WHA-3 scheme. If an additional virtual channel could be added to each link, allowing a partitioned VCA-1/WHA-3 scheme, this performance dif-



(a) 64-byte packets



(b) 256-byte packets

Figure 6.9: Comparison of Segment router traffic partitioning with VCA-3/WHA-3.

ferential might vanish. At the same time, previous results have shown that traffic loads with non-uniform destination distributions can impact the suitability of the various routing schemes. Thus, the flexibility of the PRC provides the best answer of all — the network can be adapted to suit whatever traffic load it is currently handling.

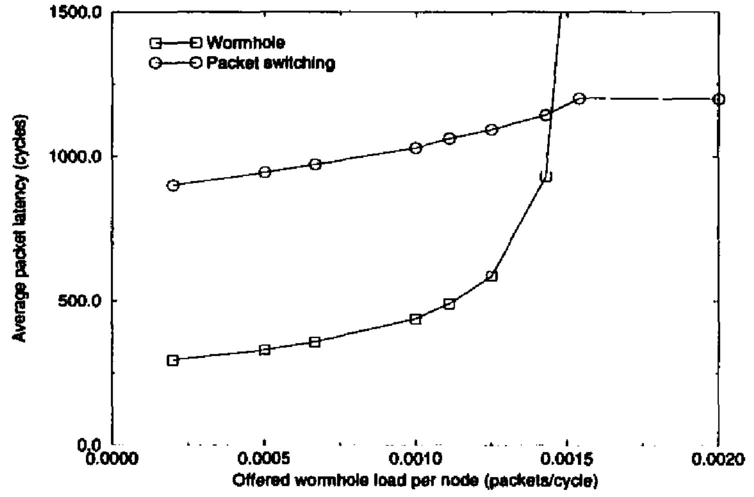
6.3 Mixing Real-Time and Best-Effort Traffic

Rexford and Shin [58] examined the interaction of traffic classes with disparate quality-of-service requirements. Many real-time systems must handle *guaranteed* traffic from applications that require predictable, if not necessarily fast, service. Simultaneously, however, they are responsible for *best-effort* packets that can tolerate variance in the quality-of-service in exchange for lower communication latencies. Packet switching is shown to have the most desirable qualities for handling guaranteed packets, while wormhole switching is better for best-effort traffic [58, 60]. This section investigates the performance of this scheme on the PRC, by simulating an 8×8 wrapped torus of PRCs.

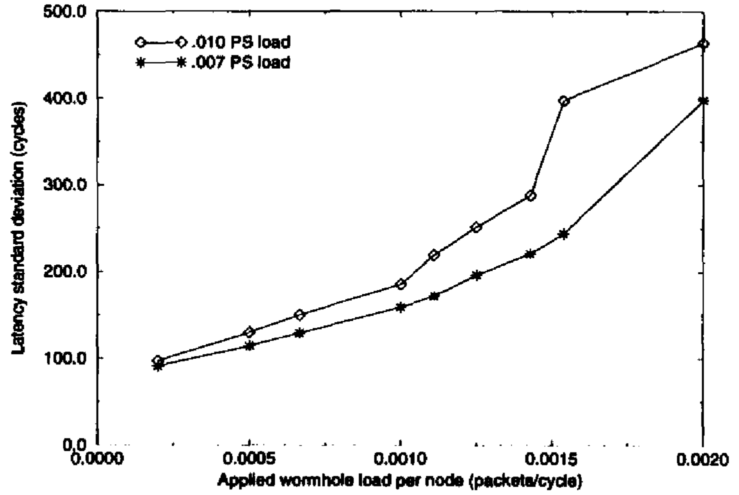
Figure 6.10 shows the interaction of these two traffic classes in the PRC where both traffic classes share a set of two virtual channels. Guaranteed packets are generated at a regular 1000-cycle interval, while the interarrival times of the best-effort packets are drawn from a Poisson distribution. The graphs show the impact of varying the best-effort load on the mean latency for both classes, and the standard deviation of the latency of the guaranteed traffic. From the rapid rise in the standard deviation of latency for the guaranteed traffic(Figure 6.10(b)), we observe that the performance of the "guaranteed" traffic is significantly affected by the best-effort traffic load.

To cope with this, Rexford and Shin partitioned the best-effort and guaranteed traffic onto separate virtual networks. This partitioning minimized the impact of the best-effort traffic on the latency and predictability of guaranteed packets [58]. Figure 6.11 repeats the experiment of Figure 6.10, but partitions the traffic onto different virtual networks. Surprisingly, however, while the standard deviation of the latency is reduced, it is still strongly impacted by the amount of best effort traffic.

This discrepancy is a direct result of changes in the PRC architecture. In the PRC architecture simulated by Rexford and Shin, which was similar to that in [21], no single virtual channel could utilize the full bandwidth of a physical link. Thus, multiple channels

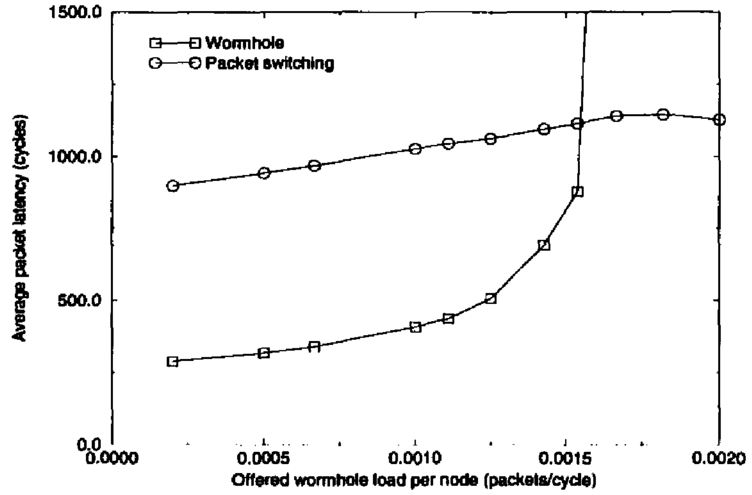


(a) Mean packet latency

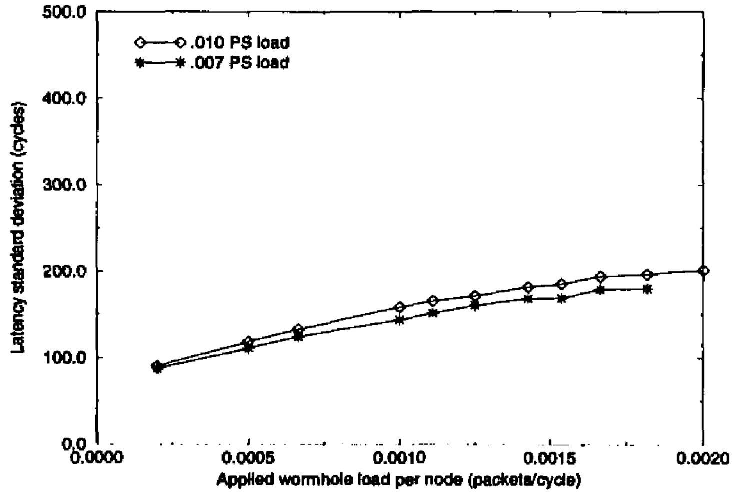


(b) Standard deviation of latency

Figure 6.10: Interaction of guaranteed and best-effort traffic on shared channels.



(a) Mean packet latency



(b) Standard deviation of latency

Figure 6.11: Interaction of guaranteed and best-effort traffic partitioned onto separate virtual networks.

could be active on a link without impacting the bandwidth each received. This is not the sole cause, however; in a later paper Rexford and Shin [60] repeated their earlier results in examining a streamlined router architecture that allowed channels to consume all available link bandwidth. Two other PRC changes, therefore, should be considered in this light. The PRC redesign greatly increased network bandwidth, but not the memory bandwidth; this disparity certainly accounts for some of the interference — the network may not be saturated, but the memory interface might be. At the same time, the CTBUS also stands out as another potential congestion point — as redesigned, it emphasizes network throughput over predictability.

6.4 Hot-Spot Routing

The flexibility of the PRC can be used to tailor routing schemes to an application's traffic pattern. This section examines how the routing scheme may be tailored to improve network performance under a *hot-spot* workload. In a hot-spot workload, a significant fraction of the packets in the network are traveling to or from a single node. These traffic patterns often arise from constructs such as global locks, centralized control, and data combining [54, 17, 18]. We will refer to such a pattern as a hot-spot of dimension n, where all nodes within n hops of the center of the hot-spot are communicating with the center, and vice versa.

From the traffic flow for this workload, we can observe that virtual cut-through switching is unlikely to provide any significant benefit over wormhole switching. Although a blocked wormhole packet may restrict other traffic from entering a node, this traffic must ultimately traverse the same links as the stalled packet. Buffering the blocked packet cannot alleviate this contention. Consequently, we will restrict our attention to the routing and selection schemes used for a wormhole-switched network.

Figure 6.12 compares the paths taken by packets in a dimension-3 hot-spot under oblivious and adaptive routing schemes. From Figure 6.12(a), we observe that the oblivious scheme will not uniformly distribute traffic among the links into the center node; the links along the y-axis receive three times as much traffic as the x-axis links. However, a minimal-path adaptive scheme can utilize alternate links to better distribute this traffic flow, as shown in Figure 6.12(b).

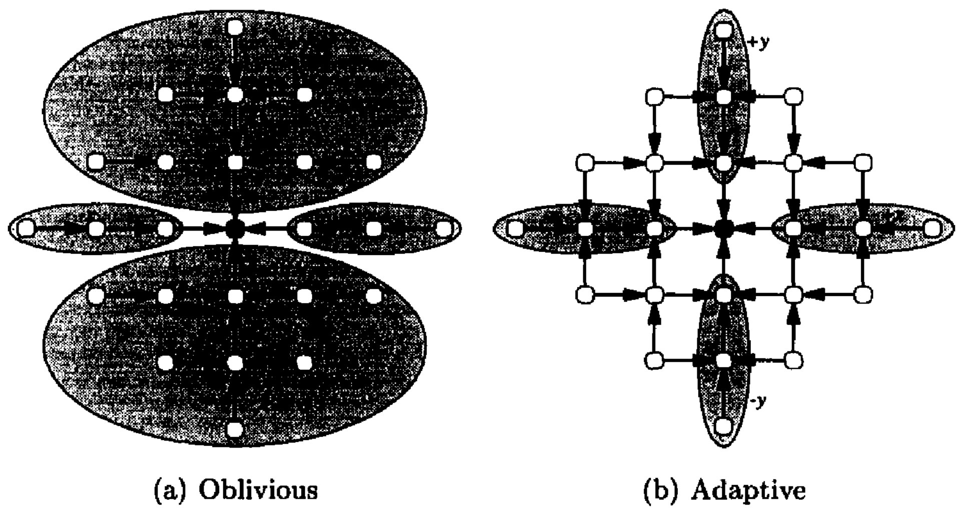


Figure 6.12: Possible routing paths for dimension-ordered oblivious and adaptive routing in the hot-spot. Each shaded region indicates the nodes that share a a single link to the center.

One important feature of the traffic flow in Figure 6.12(b) is that it is acyclic: no cycles of dependencies can exist among packets from this application. This has a major implication: packets should be able to use any minimal-path channel without restriction. Although a fixed-policy wormhole router will have already restricted the channel selections to prevent deadlock, the PRC's flexibility allows use of a *tailored*, hot-spot specific routing scheme for hot-spots that removes all deadlock restrictions in computing the candidate links and channels.

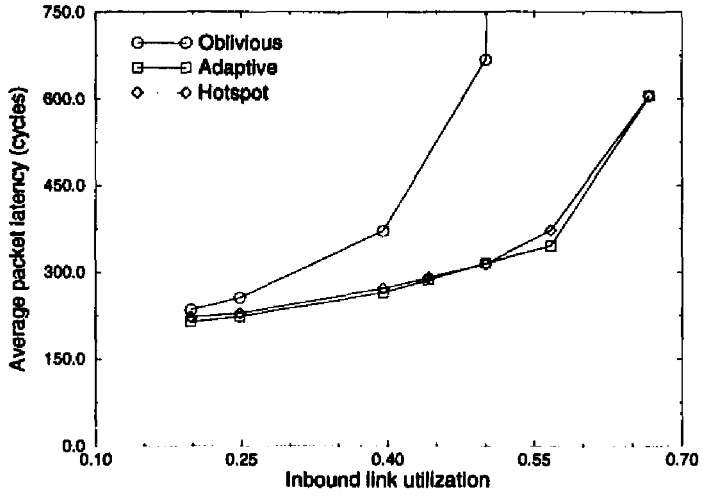
To illustrate this, we will examine the average packet latency for packets in an 8×8, unwrapped square mesh network of PRCs under a hot-spot traffic load and a mixed load with packets from both hot-spot and uniform destination distributions. All network traffic consists of fixed size, 64-byte packets; each node generates traffic independently, with exponentially-distributed interarrival times. Each of the routing schemes uses 3 virtual channels: the oblivious scheme uses all three for increasing throughput, while the adaptive scheme has two adaptive (high) channels and a deadlock-free (low) channel. The tailored hot-spot scheme uses all three virtual channels for adaptive routing; a random selection function is used to prioritize the minimal-path links.

6.4.1 Performance Comparison

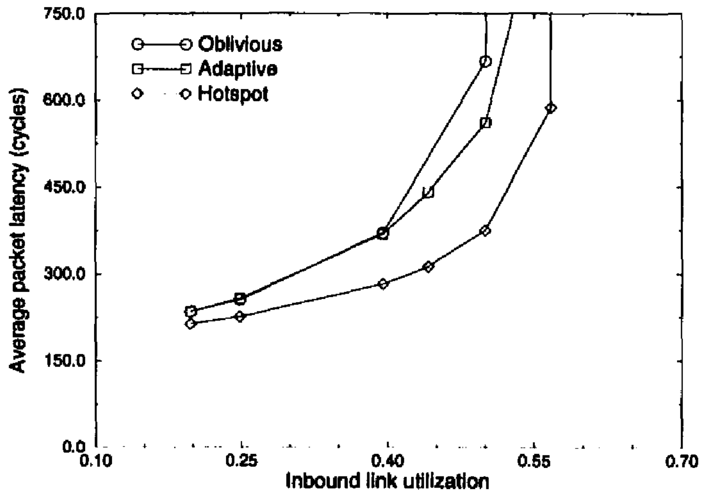
Figure 6.13 shows the latency for packets under three different routing schemes when the only traffic is from a dimension-5 hot-spot (all nodes within 5 hops of the center). The disparate link loads of oblivious routing causes the high latencies and early saturation of packets trying to exit the hot-spot node in Figure 6.13(a). Since both of the other schemes are adaptive, their performance is almost identical — the worst congestion for outbound traffic occurs during the first hop. If we examine the average latency for packets traveling to the hot-spot node (Figure 6.13(b)), oblivious routing again performs the worst, with the network saturating before either of the adaptive schemes. The tailored hot-spot scheme significantly outperforms both of the others, however, due to its better distribution of the traffic and improved utilization of virtual channels.

In general, applications will not generate hot-spot traffic in isolation. Typically, some percentage of the network traffic will be directed to the hot-spot node, while the remainder of the traffic destinations are uniformly distributed [54]. Using the PRC, we can tailor the routing scheme to suit each class of traffic: hot-spot packets use the tailored routing scheme described above, while the background traffic uses a dimension-ordered routing scheme that orders channels to prevent deadlock. To prevent deadlock, the hot-spot routing scheme is restricted to use only two of the three available virtual channels.

Figure 6.14 compares the performance of the background and hot-spot traffic under these schemes. For simplicity, the background load is varied while the hot-spot load is held constant at a rate that utilizes 45% of the hot-spot node's network bandwidth. Oblivious routing performs quite poorly: packets are unable to avoid the congested hot-spot, which further increases congestion and results in a low network saturation point. Adaptive routing performs significantly better. The tailored hot-spot scheme, however, has an unexpected benefit: by restricting hot-spot traffic from an entire virtual network, at least one channel at every node (even in the hot-spot) is always available for use by background traffic. This results in the relatively slow rise in latency for background traffic with the tailored hot-spot schemes, without adversely impacting the latency of the hot-spot traffic.

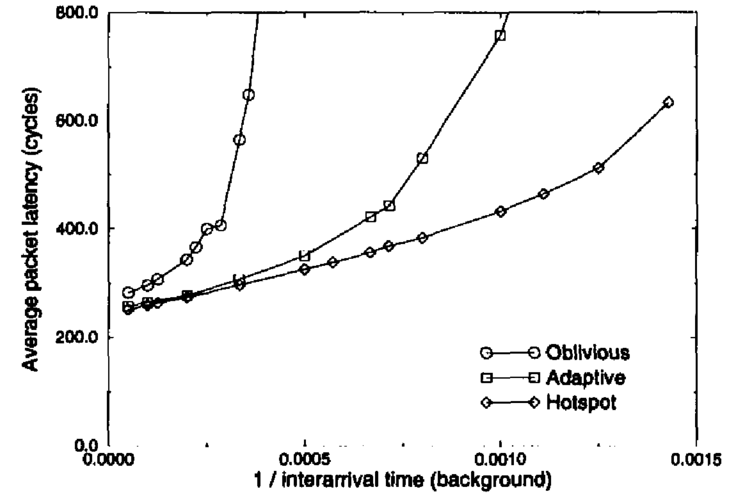


(a) Traffic out of the center

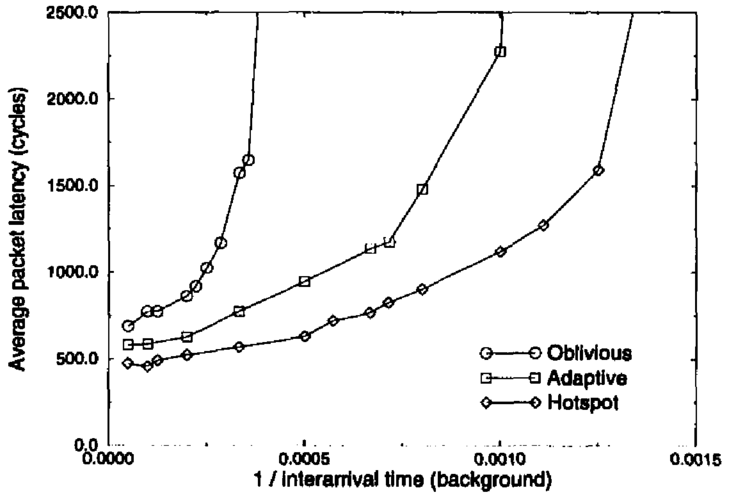


(b) Traffic into the center

Figure 6.13: Average latency comparison for hot-spot traffic only, with a dimension 5 hot-spot.



(a) Background traffic



(b) Inbound hot-spot traffic

Figure 6.14: Average packet latency for several routing schemes with a dimension 3 hot-spot traffic pattern, with background traffic.

6.5 Discussion

This chapter has examined how the PRC, through its support of multiple routing and switching schemes that are selected on a per-packet or per-channel basis, can adapt its performance to the traffic load. These examples have demonstrated several key benefits of a flexible router architecture:

- No routing and/or switching scheme performed best for all workloads. Thus, a flexible router architecture allows the router policies to be tailored to the underlying application.
- Since many applications generate bimodal workloads with different quality of service requirements, traffic partitioning combined with flexible policy selection is crucial to meeting the needs of all the traffic.

CHAPTER 7

The z-channel: A High-Performance Routing Engine

I feel a need ... a need for speed.
— from *Top Gun*

7.1 Introduction

The preceding chapters have shown the desirability of hardware support for flexible routing and switching. While microcontroller-based routing engines provide an extremely high degree of flexibility in selecting network policies, the cost in performance may outweigh some of these benefits. In particular, the sequential nature of operations within these routing engines leads to increased routing latencies. Consequently, this chapter addresses the issue of raw performance by developing a new receiver architecture that preserves flexible selection of routing and switching policies without compromising performance.

7.1.1 Motivation

The microcontroller-based routing engine provides almost unlimited flexibility in routing and switching packets by executing a sequential set of relatively primitive operations. As a result, the lower bound on the number of cycles required to route a packet, reserve a transmitter, and forward the packet header is approximately 16 cycles. In the PRC, this routing latency (l_r) corresponds to the time required to transmit two flits across a link. If the microprogram requires more instructions to route the packet, the routing latency increases accordingly. Using the example of the adaptive microprogram of Section 3.4, the routing engine can take up to 19 cycles to issue a routing primitive; accessing the switch to reserve a channel and forward the packet header will take at least another 8 cycles. Thus, the routing latency for a packet is often closer to 3 or 4 flit cycles. By way of contrast, routers that use fixed routing and switching policies typically route the packet in a single flit cycle [37]. This routing latency is a major component of the overall network latency in cut-through switching networks; the total network latency for a P-flit packet traversing D nodes is $l_r D + P$ in the absence of contention. Thus, if a 16-flit packet traverses 10 nodes, its total network latency is approximately $10 \times l_r + 16$ flit cycles; if $l_r = 1$, the network latency is 26 flit cycles. If $l_r = 3$, as often occurs with the PRC, the network latency would be 46 flit cycles. This is a large performance gap that cannot always be overcome via flexible selection of network policies.

In reinventing the routing engine, therefore, our primary goal is to eliminate this performance gap while providing relatively flexible routing, selection, and switching policies and support for traffic partitioning. This means that our goal is a routing latency of one flit cycle (8 cycles) or less. To achieve these goals, certain restrictions on the supported network topologies, routing-switching algorithms, and packet formats will be necessary. Drawing on insights from the preceding work, this chapter develops the z-channel routing engine to meet this goal.

7.1.2 Overview of the z-channel

Instead of implementing low-level primitives in microcode, the z-channel provides a programmable method for selecting *routing instructions* from a table downloaded by the host. Each routing instruction encapsulates the entire routing and switching decision for a packet; Section 7.3 discusses the format and implementation of these instructions. Figure 7.1 shows a simplified view of the z-channel architecture: the *routing engine* parses the packet header to determine the address of the desired routing instruction in the table; the switching control module then executes this routing instruction to reserve the desired channel.

7.2 Header Parsing and Route Computation

The routing engine provides three of the four major phases of packet routing: header parsing, route computation, and route selection. For efficiency in both time and implementation cost, the methods for selecting the appropriate routing instruction are necessarily limited. Consequently, the z-channel architecture restricts the header and addressing for-

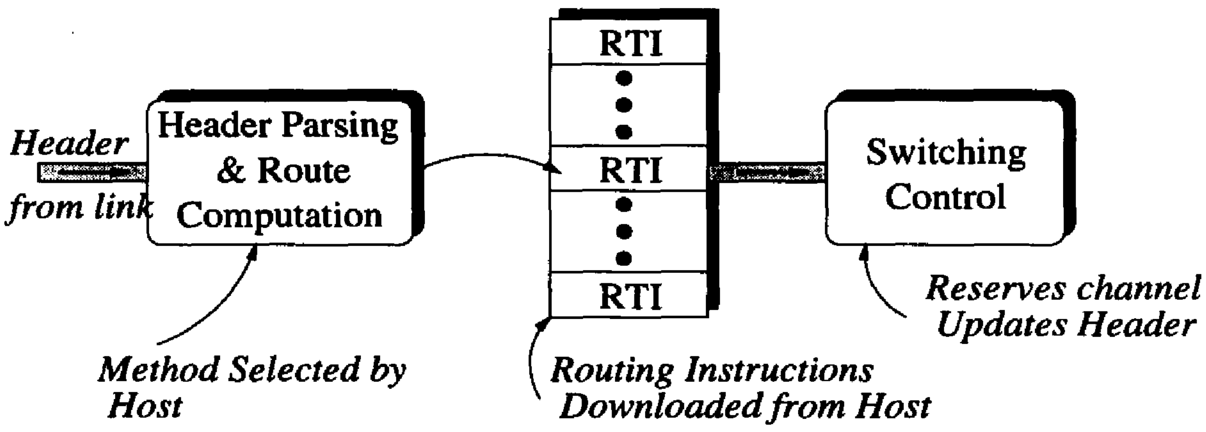


Figure 7.1: A simplified diagram of the z-channel.

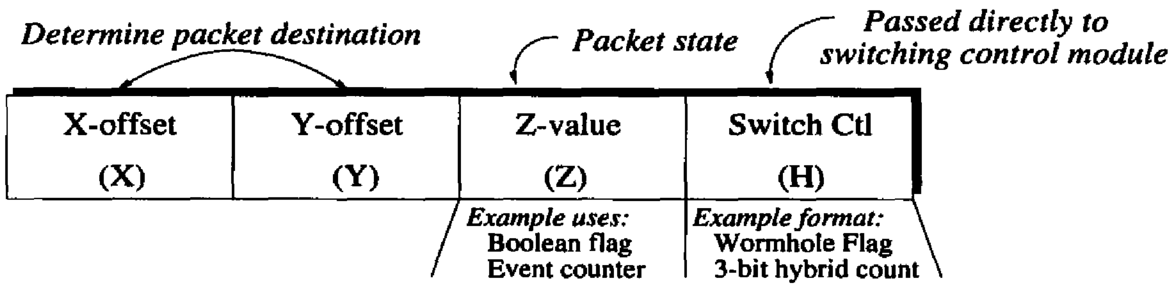


Figure 7.2: Header flit format.

mat so that packet headers can be efficiently processed by a hardwired state machine. Since the z-channel fixes the addressing scheme, one should be chosen that scales well with network size, and does not require lengthy calculations.

The traditional approach to table-lookup routing is to encode a numeric destination ID in the header; to find the routing instruction for a packet going to the node with id N, the router simply reads the routing table at address N [48, 63]. There are several drawbacks to this approach, however. With the destination table-lookup, each router requires a routing table with an entry for every node in the network. This limits the number of nodes in the network to the size of the lookup tables. In addition, routing flexibility is conditioned solely upon the current node and destination of the packet, and not on other factors such as the current link(s) the packet holds, its current channel, and so forth. Adding support for any of these would greatly increase the size of the table.

The addressing scheme that best meets our requirements for efficient implementation and scalability is offset-based addressing. Using an offset-based addressing scheme, we can use two 8-bit offsets to specify the packet destination in a 128 $\times$ 128 2-D topology; after each hop, only one offset needs to be incremented or decremented. The remainder of this section shows how we can develop an efficient, programmable architecture for mapping an offset-based header (such as in Figure 7.2) into the correct routing instruction.

7.2.1 xy-sign Table

If we compare the pseudo-code for two minimal-path routing algorithms,[1] a common element is noted:

```
x = x + 1
if (x != 0) then
    route (+x)
else if (y < 0) then
    route (+y)
else if (y > 0) then
    route (-y)
else
    buffer
```

Oblivious dimension-ordered

```
x = x + 1
if ((x != 0) && (y > 0)) then
    route (+x, -y)
else if ((x != 0) && (y < 0)) then
    route (+x, +y)
else if ((x != 0) && (y == 0)) then
    route (+x)
else if ((x == 0) && (y > 0)) then
    route (-y)
else if ((x == 0) && (y < 0)) then
    route (+y)
else /* x == 0 && y == 0 */
    buffer;
```

Adaptive minimal path

Essentially, the route computation for both algorithms can be viewed as a **case** statement predicated on the signs of x and y; the signs of the offsets indicate the general direction in which the packet should travel.

To implement this **case** statement efficiently, we can define a function that maps the signs of the offsets into a numeric index into the routing instruction table. Thus, if we envision each table entry in Figure 7.3 as containing a routing instruction downloaded from the host, we can implement each of these schemes by simply computing the signs of x and y and reading the routing instruction. Support for traffic partitioning and virtual channels is added by simply providing a separate table for every virtual channel at the link; since each table contains only 8 entries[2] the replication is certainly feasible.

7.2.2 z-checks

However, if we consider a more complicated routing algorithm, this scheme breaks down. Consider the adaptive algorithm of Section 3.4, which uses a diagonal selection function to

[1] Each implementation is written for routing a packet traveling along the $+x$ axis.

[2] There are 9 possible combinations of the x and y sign values. We will assume the $(x = 0, y = 0)$ entry is hardwired to buffer the packet; thus, only 8 table entries are required.

		y		
		$=0$	<0	>0
	$=0$	Buff	O_{+y}	O_{-y}
x	<0	O_{+x}	O_{+x}	O_{+x}
	>0	O_{-x}	O_{-x}	O_{-x}

(a) Oblivious

		y		
		$=0$	<0	>0
	$=0$	Buff	O_{+y}	O_{-y}
x	<0	O_{+x}	$O_{+x}O_{+y}$	$O_{+x}O_{-y}$
	>0	O_{-x}	$O_{-x}O_{+y}$	$O_{-x}O_{-y}$

(b) Adaptive

Figure 7.3: Routing tables for dimension-ordered minimal path algorithms, indexed by the signs of x and y.

retain adaptivity by reducing the larger offset first:

		y		
		$=0$	<0	>0
	$=0$	Buff	O_{+y}	O_{-y}
x	<0	O_{+x}	$\begin{cases} O_{+x}O_{+y} & \text{if } \lvert x\rvert > \lvert y\rvert, \\ O_{+y}O_{+x} & \text{otherwise.} \end{cases}$	$\begin{cases} O_{+x}O_{-y} & \text{if } \lvert x\rvert > \lvert y\rvert, \\ O_{-y}O_{+x} & \text{otherwise.} \end{cases}$
	>0	O_{-x}	$\begin{cases} O_{-x}O_{+y} & \text{if } \lvert x\rvert > \lvert y\rvert, \\ O_{+y}O_{-x} & \text{otherwise.} \end{cases}$	$\begin{cases} O_{-x}O_{-y} & \text{if } \lvert x\rvert > \lvert y\rvert, \\ O_{-y}O_{-x} & \text{otherwise.} \end{cases}$

Essentially, we need to perform an additional check on the packet header before arriving at the proper routing instruction. To handle this, the xy-sign table can indicate a function (called the *z-check*) that computes the index into the routing instruction table. This z-check has two main functions: selection functions for link ordering, and/or an additional header parsing stage. Many of the z-checks center around a z value in the routing header format of Figure 7.2, which is typically used as a boolean flag or an event counter. Table 7.1 shows a number of potential z-checks, along with their suggested *outcomes*. To implement the diagonal selection function described above, we would use the $x - y$ magnitude comparison z-check, which will return a different integer value for each possible outcome: $|x| > |y|$, $|x| = |y|$, or $|x| < |y|$.

With the addition of the z-check, we can implement any offset-based routing scheme that can be written in the following form, where the elements in angle brackets (< and >) are downloaded by the host:

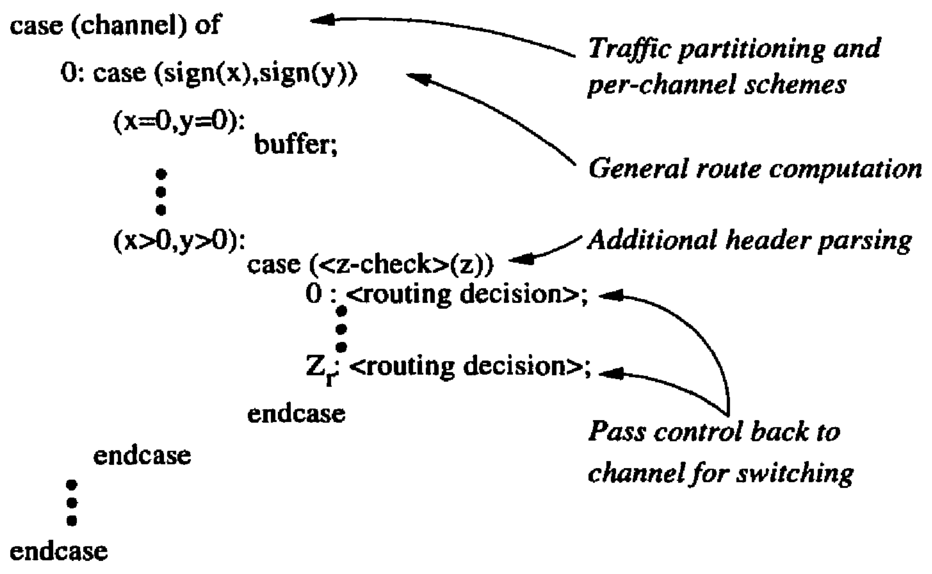
Check	Outcomes
z sign check	$z < 0$
	$z = 0$
z as event counter	$z > 0$
z boolean	$z \neq 0$
boolean flag	$z = 0$
null	
random	varies
x-y comparison	$\|x\| > \|y\|$
	$\|x\| = \|y\|$
diagonal sel. func.	$\|x\| < \|y\|$
z-c comparison	$\|z\| > \|c\|$
	$\|z\| = \|c\|$
useful for z as counter	$\|z\| < \|c\|$
Minimum congestion	Prefer $+x$
	Prefer $-x$
	Prefer $+y$
another sel. func.	Prefer $-y$

(a) Single-valued z

Check	Outcomes
Diagonal	$z_b\&(\|x\| > \|y\|)$
	$\overline{z_b}\&(\|x\| > \|y\|)$
	$z_b\&(\|x\| \leq \|y\|)$
	$\overline{z_b}\&(\|x\| \leq \|y\|)$
Sign	$z_b\&(z_c > 0)$
	$\overline{z_b}\&(z_c > 0)$
	$z_b\&(z_c = 0)$
	$\overline{z_b}\&(z_c = 0)$
Constant	$z_b\&(z_c > c)$
	$\overline{z_b}\&(z_c > c)$
	$z_b\&(z_c \leq c)$
	$\overline{z_b}\&(z_c \leq c)$
Counter/Diagonal	$(z_c > c)\&(\|x\| > \|y\|)$
	$(z_c > c)\&(\|x\| \leq \|y\|)$
	$(z_c \leq c)\&(\|x\| > \|y\|)$
	$(z_c \leq c)\&(\|x\| \leq \|y\|)$

(b) Double-valued z

Table 7.1: z-check examples.



```
case (channel) of
    0: case (sign(x),sign(y))
        (x=0,y=0):
                    buffer;
        •
        •
        •
        (x>0,y>0):
                   case (<z-check>(z))
                       0 : <routing decision>;
                       •
                       •
                       •
                       Z_r: <routing decision>;
                   endcase
        endcase
    •
    •
    •
endcase
```

Traffic partitioning and per-channel schemes
General route computation
Additional header parsing
Pass control back to channel for switching

Essentially, the z-channel implements a set of layered **case** statements. The first case statement provides channel-specific routing, while the second parses the x and y offsets to extract the general direction for the packet to travel. Further route selection and header parsing may then be selected by the host, by specifying a z-check function that checks the packet header to determine index of the final routing instruction.

More complex routing algorithms can be implemented by decomposing z into two separate values: a single-bit boolean flag (z_b) and an unsigned integer counter (z_c). This ability is particularly useful for implementing multiple routing algorithms on the same virtual channel when one (or both) of the schemes requires some state information in the header. Using this "two-valued" z, z_b distinguishes which routing scheme to use for the packet, leaving z_c free for use as a counter or boolean flag by both schemes.

To summarize, the z-channel routing engine uses a sign check function on x and y as its primary criterion in determining the routing instruction for an incoming packet. In addition, a downloaded z-check further specifies which routing instruction to use. Thus, we can now specify a general algorithm for the z-channel routing engine:

1. Update the x or y offset:

$$\begin{cases} x = x - 1 & \text{if link 0,} \\ y = y - 1 & \text{if link 1,} \\ x = x + 1 & \text{if link 2,} \\ y = y + 1 & \text{if link 3,} \end{cases}$$

2. Compute $\text{sign}(x)$ and $\text{sign}(y)$, where

$$\text{sign}(x) = \begin{cases} 0 & \text{if } x = 0, \\ 1 & \text{if } x < 0, \\ 2 & \text{if } x > 0. \end{cases}$$

3. If $x = 0$ and $y = 0$, buffer the packet at the local node.

4. Read the z-check from table entry at address $Z_{xy} = 3 \times \text{sign}(x) + \text{sign}(y) - 1)$.

5. Compute the base address $Z_b = Z_{xy} \times Z_r$, where Z_r is the maximum number of outcomes for a z-check.

6. The z-check returns a value Z_{chk} from $0, \ldots, Z_r - 1$.

7. Read the routing instruction from the routing instruction table at address $Z_{rti} = Z_b + Z_{chk}$

Despite the apparent complexity of some of these steps, they may be executed quite efficiently. For example, if Z_r is a power of 2, computing Z_b and Z_{rti} requires only concatenation

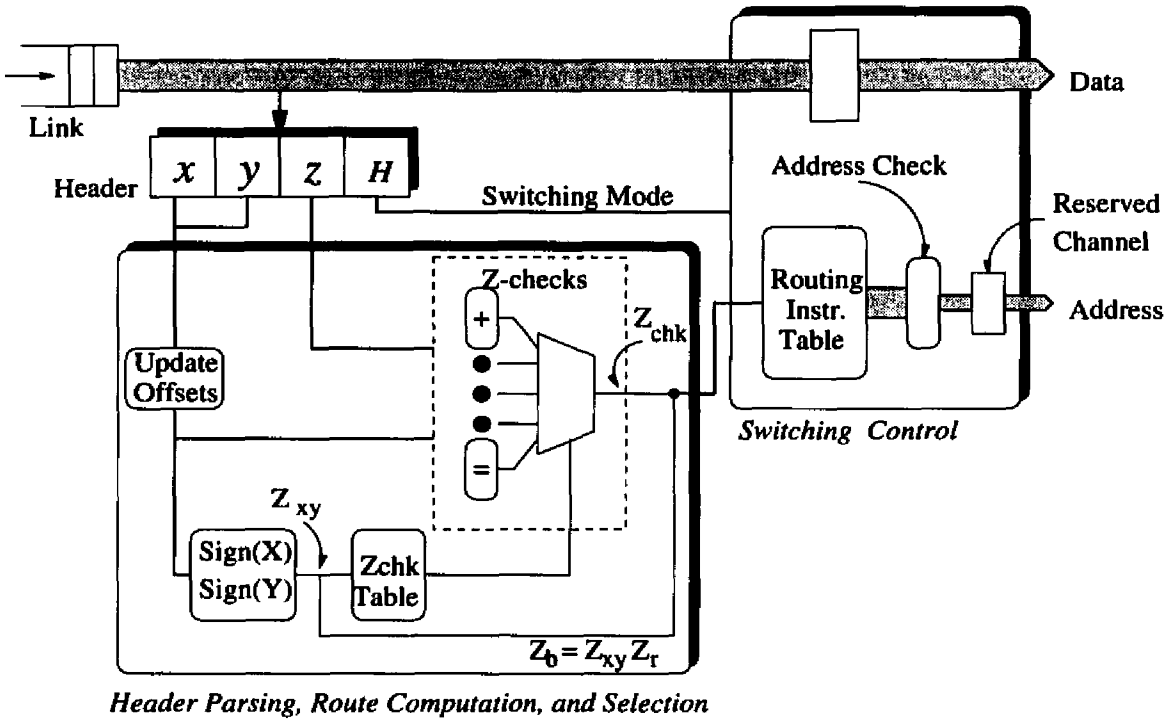


Figure 7.4: *z*-channel architecture.

operators with no actual logic. The other functions are somewhat more complex, but each can easily be computed in a single cycle. Figure 7.4 shows the *z*-channel architecture, which directly implements the above series of **case** statements. To reduce the implementation cost, all of the logic outside the *z*-check and routing instruction tables is shared amongst the channels at the link.

7.2.3 *z*-operations

The final element needed to support flexible routing is the *z-operation*, which updates the *z* value in the header to reflect the state of the packet after a routing primitive succeeds. Many routing schemes (especially non-minimal wormhole routing schemes) maintain a limited state for the packet in the header. For example, this state might indicate whether a packet has traversed any channels from a particular set or indicate the number of times a packet has been misrouted. Since updating *z* for each of these applications requires knowledge of *which* channel is reserved during the switching phase, we need to update *z* *after* the switching control module has reserved the next channel for the packet.

Figure 7.5 shows how one such scheme, Dally and Aoki's static dimension reversal routing algorithm [13], can be supported when *z* is updated *after* the switching control module

Operation
Increment z
Decrement z
Pass z
Set z
Clear z

(a) Single-valued z

Operation	
z_b	z_c
Set	Increment
Set	Decrement
Set	Pass
Clear	Increment
Clear	Decrement
Clear	Pass
Pass	Increment
Pass	Decrement
Pass	Pass

(b) Double-valued z

Table 7.2: z-operation examples.

			y	
		$= 0$	< 0	> 0
	$= 0$	Buffer	O_{1z} / pass z O_{0z+1} / pass z O_{3z+1} / pass z	O_{3z+1} / pass z O_{0z+1} / pass z O_{1z} / pass z
x	< 0	O_{0z+1} / pass z O_{1z} / pass z O_{3z+1} / pass z	O_{0z+1} / pass z O_{1z} / pass z O_{3z+1} / pass z	O_{0z+1} / pass z O_{3z+1} / pass z O_{1z} / pass z
	> 0	N/A	O_{1z} / pass z O_{0z+1} / pass z O_{3z+1} / pass z	O_{3z+1} / pass z O_{0z+1} / pass z O_{1z} / pass z

(a) Routing table for channel I_{2z}, where $z < r$.

			y	
		$= 0$	< 0	> 0
	$= 0$	Buffer	O_{1z} / pass z O_{0z+1} / inc z O_{2z+1} / inc z	N/A
x	< 0	O_{0z+1} / inc z O_{1z} / pass z O_{2z+1} / inc z	O_{1z} / pass z O_{0z+1} / inc z O_{2z+1} / inc z	O_{0z+1} / inc z O_{1z} / pass z O_{2z+1} / inc z
	> 0	O_{2z+1} / inc z O_{1z} / pass z O_{0z+1} / inc z	O_{1z} / pass z O_{2z+1} / inc z O_{0z+1} / inc z	O_{2z+1} / inc z O_{1z} / pass z O_{0z+1} / inc z

(b) Routing table for channel I_{3z}, where $z < r$.

Figure 7.5: Dally and Aoki's static dimension reversal routing algorithm, with no backtracking allowed.

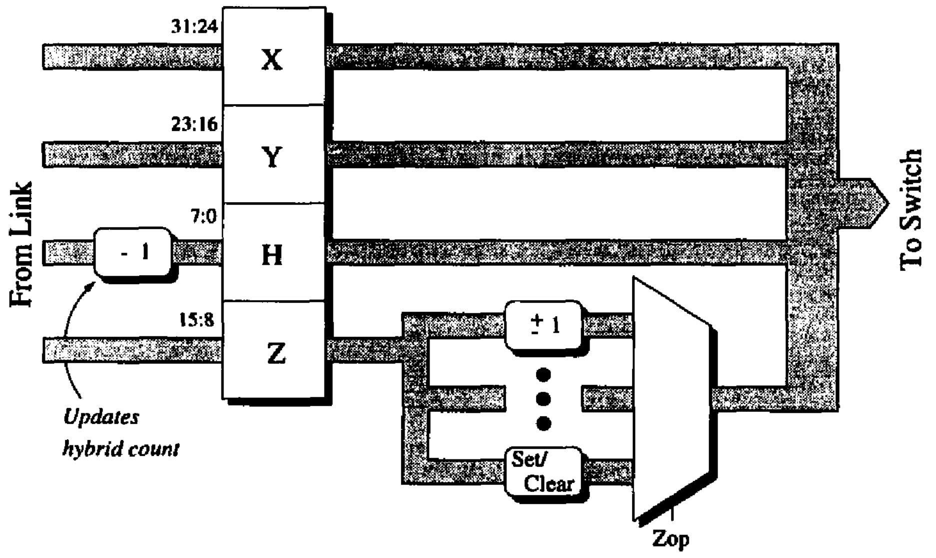


Figure 7.6: z-operation architecture.

has reserved a channel. In this scheme, the virtual channels of the network are partitioned into r separate virtual networks. Packets are initially injected on the class-0 network with the state variable z set to 0. They may be routed freely on any virtual channel in the class-0 network. If the packet is routed contrary to the dimension-ordering (i.e., from a channel on the y-axis to a channel on the x-axis), z is incremented and the packet moves to the class-z network. By making the class-r network strictly dimension-ordered and non-adaptive, cycles (and thus deadlock) are prevented. To implement this scheme, we need to adjust the value of z based on which channel is actually reserved when the routing instruction executes, as in Figure 7.5.

Thus, for every candidate channel in the routing instruction, we also need to specify a z-operation to execute if that channel is successfully reserved. Table 7.2 shows the z-operations necessary to use z as a counter, boolean flag, or both. As seen in Figure 7.6, z-operations are simple to implement, requiring only minimal changes to the switching control module.

7.3 Switching Control

To make the z-channel feasible, however, we need to develop a routing instruction that can compactly express almost any routing decision, *and* a simple hardware design capable of executing this routing instruction. At a minimum, each routing instruction must be capable of conveying both a *candidate set* of the desired channels, and some *ordering* of these channels. In addition, every candidate channel will have some z-operation associated with it that will update the packet state.

Typically, a routing decision takes a form similar to this example from the adaptive wormhole microprogram of Section 3.4:

```
/* check -y high */
    jump ~resvdl3c2, get_l3c2, link;
    jump ~resvdl3c1, get_l3c1, link;
/* check +x high */
    jump ~resvdl0c2, get_l0c2, link;
    jump ~resvdl0c1, get_l0c1, link;
/* check low channels */
    jump ~resvdl3c0, get_l3c0, link;
    jump ~resvdl0c0, get_l0c0, link;
    jump true, block_1310;
```

In this example, the ordered candidate set of channels is $\langle O_{32}, O_{31}, O_{02}, O_{01}, O_{30}, O_{00}\rangle$. While a bit-mask could easily convey the members of the set, expressing the ordering is much more difficult. One approach would be to assign a unique ID to each channel, and encode the candidate set as a bit string of these IDs. This approach offers the maximum degree of flexibility, and is simple to encode and parse in hardware; the drawback, however, is that it does not scale well. The PRC architecture, for example, includes 12 separate transmission channels at each node; if we wished to express any ordering of these channels, we would need 48 bits (4 bits per channel for 12 channels). Since we will need several bits per channel to specify the z-operation, the actual instruction will be much larger.

Fortunately, however, most routing schemes partition the available virtual channels into *sets*, where the channels in a set share common routing schemes. This notion of sets appears in the switching code used in the microprograms presented in Chapter 3, where microprograms often check several channels that are considered "equivalent" before proceeding to check another set of channels. In the example above, the channels are partitioned into four sets: the "high" channels on link 3 (O_{32} and O_{31}), the high channels on link 0 (O_{02} and O_{01}), and a set for each low channel (O_{30} and O_{00}). The order among channels within each set is not important, but the sequence of the sets themselves may be crucial.[3] Consequently, we can simplify the problem by representing the candidate set as a sequence of routing primitives, each specifying a set of *equivalent* channels.

[3]I.E. the routing algorithm doesn't care whether O_{32} or O_{31} is checked first, as long as both are checked before O_{02} and O_{01}.

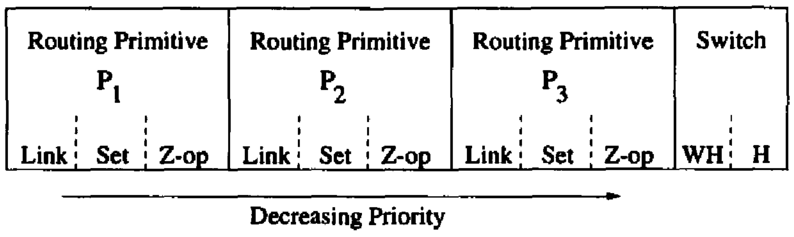


Figure 7.7: Routing instruction format.

7.3.1 Routing Instructions

Formally, we will define a routing instruction as an ordered sequence $P_1P_2\ldots P_{d_r}H$, where each P_i is a routing primitive and H is some switching control flag. Each routing primitive P_i specifies a set of candidate channels, while H indicates what action to take if no routing primitive succeeds. A routing primitive *succeeds* if at least one of its candidate channels is available and successfully reserved; if no selected channel is free or the reservation fails, the routing primitive is deemed to have *failed*. To differentiate between the various switching schemes, H specifies how to react when no channel can be reserved. For wormhole switching, this requires that the routing instruction be reset and executed again; virtual cut-through buffers the packet. Using this notation, the switching statement examined earlier could be written as: $P_1 = O_{32}O_{31}, P_2 = O_{02}O_{01}, P_3 = O_{30}, P_4 = O_{00}, H =$ block. Using a routing instruction, we can express any possible routing decision that does not require more than d_r sets. For almost any routing scheme in a two-dimensional topology, $d_r = 4$ is sufficient.

7.3.2 Implementing Routing Instructions

Figure 7.7 shows one possible format of a routing instruction, where each routing primitive specifies a *composite* channel mask and a z-operation. The composite address mask consists of two components: an L-bit *Link* mask selecting the desired links, and a C-bit mask (*Set*) specifying which channels to use on those links. For example, with a composite address mask, if Link = 0110_2 and Set = 101_2, channels O_{22}, O_{20}, O_{12}, and O_{10} would be selected. Since this expansion can be implemented with a single level of logic, hardware costs are minimal. While a full mask is simpler to process and allows the greatest flexibility in selecting channels, a composite mask requires fewer bits to store. Since each routing instruction contains several routing primitives, and storage for many routing instructions

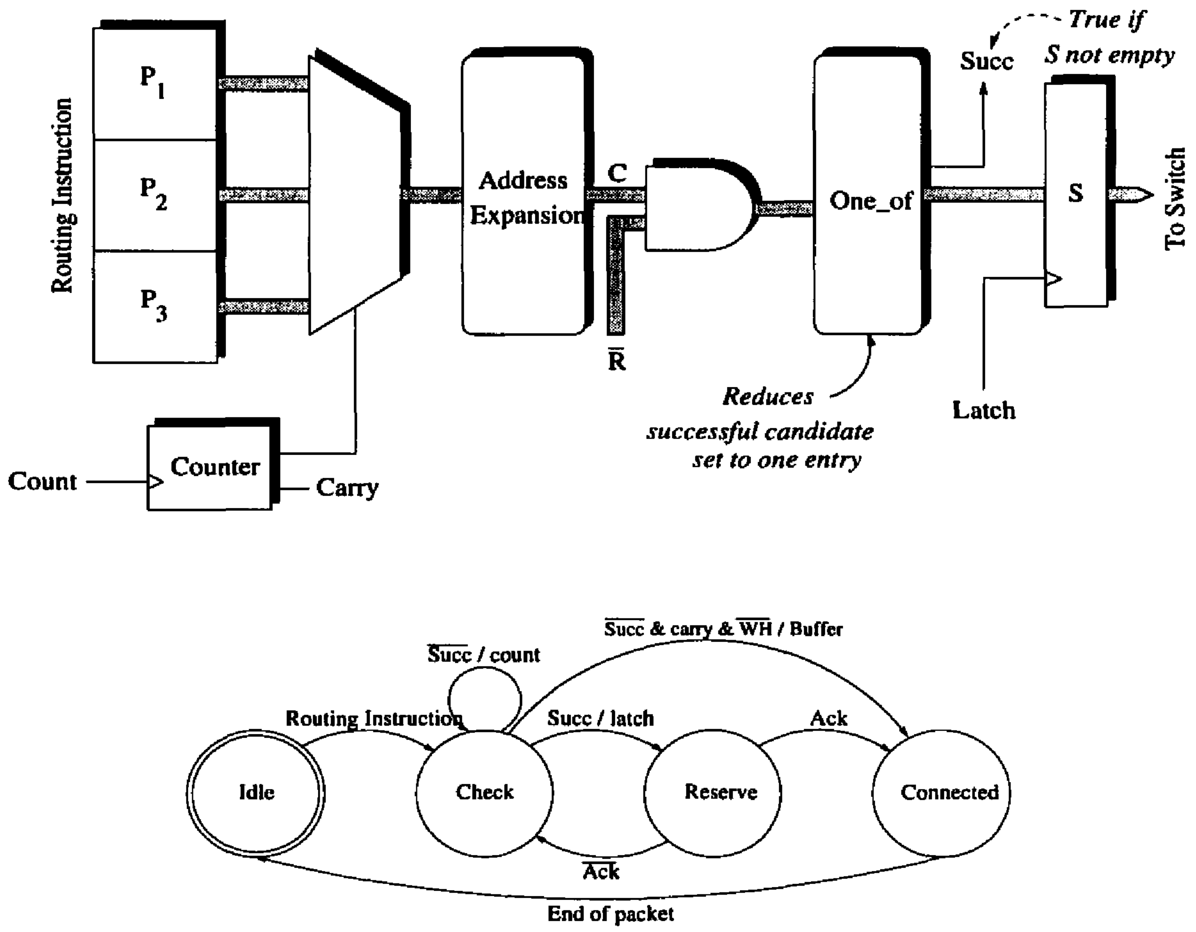


Figure 7.8: Address logic and controlling state machine.

may be necessary, space considerations may often outweigh the slight performance gain of a full mask.

In the z-channel, the switching control phit H is taken directly from the routing header of the packet. It consists of the current value of h (the hybrid switching counter) and a wormhole-override flag w. If w is set, or $h > 0$, the packet will stall if blocked. Otherwise, the packet buffers when blocked. Using this format of H, we can select wormhole, virtual cut-through, or hybrid switching for any network size. Other formats of H are easily possible.

Figure 7.8 shows the architecture and a simplified state machine for the switching control module. A counter controls the select inputs for a multiplexor; this selects the current routing primitive to execute. After a routing primitive is selected, the address mask is expanded and compared to the current reservation status to determine which (if any) selected channels are available. If any channels are found, the success flag (`succ`) is set and a hardwired function (`one_of`) is used to reduce the candidate channel set to only one entry. The appropriate channel address is then latched and used for the reservation attempt. If all routing primitives fail, the carry flag is asserted when the counter overruns and the switching control tag checked to determine whether to execute the instruction again or "give up" and buffer the packet.

7.4 Evaluation

The z-channel illustrates the themes mentioned at the beginning of this thesis: there are significant tradeoffs between the amount of flexibility provided and the utility of this flexibility, and the implementation and performance costs of this flexibility. The microcontroller-based routing engine provides a maximal amount of flexibility with a reasonable implementation cost, but at a performance cost under certain conditions. The z-channel, however, restricts its application domain to certain network topologies and addressing schemes to improve performance. The remainder of this section shows how the z-channel actually provides a useful degree of flexibility without paying a penalty in raw performance.

The performance of the z-channel in routing a packet is shown by the simplified timing diagram in Figure 7.9. Once the x and y phits have arrived from the link, the sign functions and the initial offset can be computed in less than a single clock cycle. The z-check can then be read from the table in the next cycle, so that it will be stable by the time z arrives from the link. The next cycle the z-check executes, determining the final index (Z_{rti}) into the routing instruction table. The routing instruction is then read out of the table and is stable by the time the switching tag H arrives. Thus, the header parsing, route computation, and initial selection stages are completely finished by the time the routing header has arrived.

The only delay experienced by the packet is for the switching decision; i.e., the time necessary to execute the routing instruction and reserve a channel. In the most common case, the first routing primitive succeeds — this requires only a single cycle before a reservation request is issued. In the PRC, the reservation request requires 3 cycles to complete; thus, the minimum routing latency for the z-channel architecture in the PRC is 3 cycles, or 3/8 of a flit cycle. This compares extremely well with other router architectures, which typically aim for a single flit delay for routing, and comes in well under our goal of a minimum

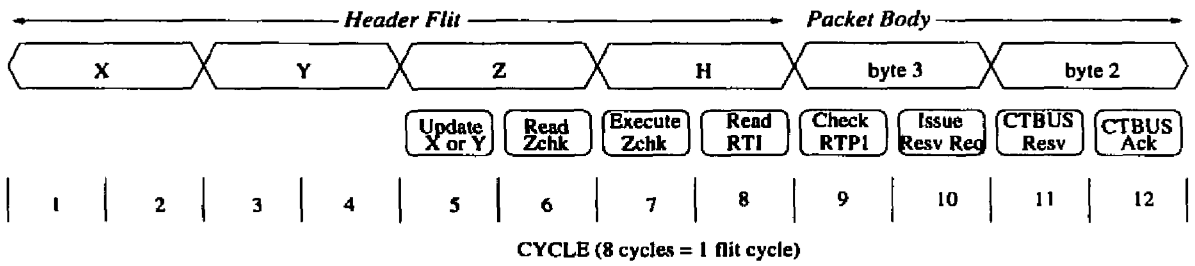


Figure 7.9: z-channel routing timeline, showing how the routing decision is made before the entire header flit has arrived.

(a) Microcontroller

x \ y	$=0$	<0	>0
$=0$	12	15 - 17	14 - 16
<0	12 - 14	18 - 22	17 - 22
>0	—	—	—

(b) z-channel

x \ y	$=0$	<0	>0
$=0$	0	1 - 2	1 - 2
<0	1 - 2	1 - 4	1 - 4
>0	—	—	—

Table 7.3: Comparison of routing latency (in cycles) for a packet arriving on a high channel using WHA(Diag)-3.

routing latency of one flit cycle.

To illustrate the performance difference between the z-channel and the microcontroller routing engines, Table 7.3 compares the routing latency for the adaptive diagonal-biased wormhole routing scheme. Each entry gives the time (in cycles) required before issuing a reservation request for a free channel; the first entry is for a channel in the first set checked (highest priority) and the second is for a channel in the last set checked. For the z-channel, the actual routing decision is made before the header has completely arrived; the only delay encountered is that required to actually execute the routing instruction.

7.4.1 Implementation Cost

The implementation cost of the z-channel routing engine is largely determined by the size of the routing instruction tables. In turn, the size of these tables is largely dictated by the number of virtual channels on each link and the number of outcomes of each z-check. The following parameters determine the size of the z-check and routing instruction tables

Table	Entries		Bits/Entry
z-check Table	$S(3^D - 1)$	$\times$	$\lceil \log_2 Z_n \rceil$
Routing Ins. Table	$S(3^D - 1)\lceil \log_2 Z_r \rceil$	$\times$	$d_r(L + S + \lceil \log_2 p_n \rceil)$
Indirection Table	$S(3^D - 1)\lceil \log_2 Z_r \rceil$	$\times$	$\lceil \log_2 R \rceil$

Table 7.4: Formulas for determining z-channel RAM sizes.

in a k-ary n-cube topology.

Z_n	number of z-checks
Z_r	maximum number of outcomes for a z-check
L	number of internode links
D	number of dimensions in the topology (equal to n)
C	number of virtual channels per link
S	maximum number of channel sets. If $S = C$, every channel is a distinct set.
d_r	number of routing primitives in each outcome list
p_n	number of z-operations

Table 7.4 shows the size of the z-channel tables as a function of these parameters. Notice that the cost of implementing the tables is dependent primarily on three main factors: the maximum number of z-check outcomes Z_r, the number of dimensions D in the topology (which is not easily altered) and S, the maximum number of channel sets. The cost of the tables may be dominated by the S^2 factor; if $S = C$, traffic routing and partitioning flexibility is maximized, but at the cost of large tables.

There are several methods that could be used to reduce the size of these tables. The first method divides the C channels at each link into S sets, where $S < C$. Each channel within a set is treated identically by the actual routing engine; the switching control module, however, can reserve individual channels from within a set. Thus, the additional virtual channels can only be used to decrease the likelihood of blocking within a set. Table 7.5 shows the memory size of several possible configurations of the z-channel routing engine as a function of Z_r and S.

If we examine the routing algorithms implemented so far, we can observe that many of the routing instructions are duplicated in several places. By sharing each routing instruction between several entries in the routing table, we can reduce the implementation cost. This

	$S = 3$	$S = 8$	$C = 8, S = 4, R = 32$
$Z_r = 2$	48 + 1920	128 + 7680	64 + 1568
$Z_r = 4$	48 + 3840	128 + 15360	64 + 1728
PRC microcontroller routing engine: 5120 bit control store			

Table 7.5: Sample memory sizes (in bits) for various z-channel configurations, with $Z_n = 4, D = 2, d_r = 4, p_n = 5$

can be done by introducing an *indirection table* between the z-check and routing instruction tables. The z-check phase would execute normally, but the results would be used as the index to the indirection table; this entry then points to the actual routing instruction in a separate table with R entries. Table 7.4 gives the cost of the indirection table by itself.

7.4.2 A New PRC Routing Engine?

One implementation of the z-channel router has been designed as a drop-in replacement for the PRC's microcontroller-based routing engines. This version provides the z-checks and z-operations necessary to use z as a single-valued unsigned integer counter or as a boolean flag, which is sufficient for any of the offset-based schemes discussed in this work. More details on this routing engine are given in Appendix D.

The routing engine is combined with a modified version of the PRC NIRX to create a new receiver module that performs roughly the same functions as the existing PRC receiver module. The z-channel routing engine has three routing instruction tables (one per channel), each holding sixteen 36-bit routing instructions. The combined z-check table has twenty-four 2-bit entries, for a total of $1,776$ bits of RAM. In contrast to the PRC, the tables are implemented using dual port RAMs, which increases their cost but allows the host to dynamically modify their contents without interrupting router operation.

Designed under Epoch with the same parameters used for the PRC, the z-channel receiver module requires $45,721$ transistors in 3.39 square millimeters. The PRC receiver module, on the other hand, has $65,639$ transistors in 4.58 square millimeters. These figures are also somewhat misleading, as they include a significant amount of logic in the NIRX data queues themselves that is common to both the PRC receiver module and the z-channel.

7.5 Discussion

This chapter has introduced two innovations: the routing instruction and the z-channel routing engine. The routing instruction allows routing decisions to be expressed and executed efficiently through a simple state machine, without restricting the switching techniques. While this chapter has concentrated on supporting hybrid switching, the concept can easily be extended to include other switching schemes or constrained to use only a single switching scheme.

The degree of flexibility offered by the z-channel routing has several important benefits: applications with knowledge of their communication patterns can tailor the routing and switching schemes to distribute traffic evenly throughout the network. At the same time, this flexibility also allows faulty links to be avoided and dynamic changes in the low-level policies to adjust to the system's current needs. The z-channel is capable of implementing any distributed routing algorithm that meets certain criteria:

- Offset-based addressing in a regular or nearly regular topology
- Bases the routing decision only on the destination of the packet, the current channel, the reservation status of local channels, and a state variable (z) in the header.

At the same time, the z-channel provides a unique level of performance: its routing latency is equivalent to most fixed-scheme designs, while supporting a large number of hardware-efficient unicast schemes.

There are several ways in which the z-channel architecture can be extended:

z-channel: Small irregular networks could bypass the z-channel routing engine by adding a flag (either in the packet header or at each node) that would bypass the routing engine and z-checks to directly access the routing instruction table with an index carried in the header. This would provide a simple way to extend the capabilities of a single router implementation. Other extensions to the z-channel architecture could be easily added in the same manner. One example of this would be source-list routing: flagged packets would carry a routing primitive directly within the header.

Further improvements in performance and cost may be gained at a minor loss in flexibility by not specifying a separate z-check for every possible combination of sign(x) and sign(y). Instead, only one z-check would be specified for each channel. This would limit the routing engine to those schemes that require only one z-check other than null. The benefits, however, include eliminating one of the two RAM reads. In a system with very fast links, this might make the difference between a routing latency $l_r = 1$ and $l_r = 2$.

Routing instructions: The switching decision time can be improved by executing several routing primitives in parallel during each cycle. While an expanded architecture is costlier to implement and may requires a longer clock cycle, it provides several benefits. When routing primitives are checked sequentially, the potential exists for a low-priority channel to be reserved even though a higher-priority channel has just been freed. Implementing the routing primitive checks in parallel also allows the controller to reach low-priority channels faster. The architecture also allows other potential organizations; for example, if each routing instruction contained 4 primitives, 2 could be checked each cycle. This would decrease the time required to cycle through the primitives while also requiring less logic and allowing a faster cycle time than a fully parallel architecture. The actual method chosen for executing routing instructions represents a tradeoff between performance and implementation cost; the actual design may be optimized for any particular implementation.

Deadlock recovery may be added by specifying a maximum number of cycles through the routing instruction. If this threshold is exceeded, the packet may be buffered locally, misrouted, or aborted if the necessary reverse channel signaling is provided. Source-directed deadlock recovery schemes (such as compressionless routing [34]) could also be supported if the forward channel signalling is added.

CHAPTER 8

Conclusions

This is the way the dissertation ends.
Not with a bang, but with future work.
— apologies to T.S. Eliot

In this work, we have explored both the benefits and costs of providing flexibility in low-level network policies. While we have focused on multicomputer networks, many of the results are applicable in other network domains, such as ATM and multistage networks. In Chapter 2, we reviewed the major routing and switching schemes, and examined their common properties. This analysis decomposed the general problem of packet routing into four major steps:

Header parsing: Extracting the packet destination from its routing header.

Route computation: Determining which channels will bring a packet closer to its destination.

Selection: Ordering the candidate channels to reflect the routing scheme.

Switching: Executing the routing decision from the previous phases, and how to deal with the case where a packet cannot immediately cut through.

Using this decomposition, we then proposed an architecture for a flexible router that incorporates small, programmable devices for processing incoming packet headers. To make this scheme cost-effective, we provided programmable control of the shorter, more complex steps that is shared amongst several channels; this also necessitated developing mechanisms for offloading the time-consuming steps to channel-specific state machines.

In developing this architecture, we have made several contributions:

- **Routing instructions.** Section 3.2 introduced the routing primitive as a method for communicating routing and switching decisions from an expensive shared resource (the routing engine) to a much simpler, dedicated resource (the NIRX). This concept was extended to the routing instruction, which incorporates ordered sets of channels in Section 7.3; this allows the NIRX to implement the entire switching operation.

- **Microcontroller-based routing engine.** Chapter 3 presented a microcontroller-based routing engine architecture that offers maximal flexibility for selecting routing algorithms, link selection functions, and switching schemes. Due to the relatively high implementation cost of providing a routing engine for every incoming packet, we devised a method for sharing this resource among several incoming packets without significantly impacting performance.

- The **Programmable Routing Controller** combines the microcontroller-based routing engines with a powerful host interface, providing a uniquely adaptable platform for point-to-point communications in distributed systems.

- The **z-channel receiver** addresses the issue of providing flexibility without compromising performance. Rather than supporting as many routing and switching schemes as possible, the z-channel focuses on providing flexible routing and switching for offset-based routing schemes in regular topologies.

- **Hybrid switching,** a novel cut-through switching scheme that draws on a comparative analysis of the performance of wormhole and virtual cut-through switching to emphasize the better aspects of both.

To make a flexible network successful, support for accessing this flexibility must be provided at any of several points:

- Allowing applications to directly specify network policies.

- Compile-time determination of the appropriate routing-switching scheme.

- Dynamic operating system support for recognizing situations and altering the network schemes appropriately.

In addition, many topics still need further research within the area of providing flexible point-to-point communication networks, including:

- **Routing-Switching Scheme Performance.** To fully utilize the flexibility of the PRC and/or the z-channel, we need to explore the performance of various routing-switching schemes for a variety of application workloads. In particular, an emphasis should be placed on examining realistic workloads.

- **Multicast routing instructions.** Extending the routing instruction format to include support for multicast schemes would allow the design of a simple, source-list routed multicast network. Each header flit could carry a single routing instruction; the routing engine itself would not require any logic other than that to execute the instruction.

- **Multicast z-channel.** Alternately, it would be interesting to consider the implementation of a path-based multicast router using the z-channel concept. Rather than branching packets down multiple outgoing links, the only multicast operation permitted would be a simultaneous cut and buffer.

- **z-channel hybrid router.** While Chapter 7 addressed many of the issues required to provide a high-performance flexible router, many others still remain. Of particular interest should be studies that assess the number of injection ports required (more is not necessarily better), an appropriate queueing policy for on-chip buffers, and a crossbar switch.

APPENDICES

APPENDIX A

PRC Internals

A.1 Introduction

This appendix documents the internal architecture and details of the PRC, which was presented in Chapter 4.

A.2 Transmitter Fetch Units

A.2.1 TXBUS Interface

The FSM *tfu_tx_cntl* (referred to as the **TXFSM** hereafter) controls the TFU's interactions with the TXBUS. The **TXFSM** is responsible for monitoring the status of the channel's page queue, retrieving data from the NPBUS whenever the local TFU FIFO has space and there is a valid page tag in the page queue. The **TXFSM** is also responsible for retrieving the timestamp and finally obtaining the CRC.

Normally, the **TXFSM** spins until it detects a page tag on the output of the page queue. Upon seeing this, it places an *SOP* tag in the FIFO. After placing the *SOP* tag in the FIFO, the **TXFSM** waits for the FIFO's input ready to return to true. It then requests the first longword of the message from the TXBUS. **TX_REQ** is dropped when **TX_GNT** is seen. Data will be latched into the FIFO when **TX_ACK** rises. The **TXFSM** spins in this mode until it receives a **TX_ACK** while **TX_EOP** is true. This signals that the packet has finished. The **TXFSM** masks the tag such that a *DATA* tag goes into the FIFO instead of an *EOP* tag. When space is again available in the FIFO, the **TXFSM** accesses the TXBUS and retrieves the current timestamp. The next TXBUS access will retrieve

tag(1)	tag(0)	Meaning
0	0	Data
0	1	Mark
1	0	SOP
1	1	EOP

Table A.1: TFU FIFO Tag Encodings.

the CRC. This longword is then marked with the *EOP* tag, signalling the CTBUS interface that the packet is now complete. The **TXFSM** then returns to its normal wait state.

In order to avoid generating data requests to the TXBUS when there is not a valid tag in the page queue, the **TXFSM** monitors the **qrdy** line, and uses this in combination with the input ready of the local FIFO to decide when to request additional data.

A.2.2 TFU FIFO

The FIFO in the TFU has two main functions: it provides local data storage for up to four longwords at a time, and it also isolates the data retrieval functions of the TFU from the data transfer functions. Thus, the TXBUS and CTBUS clocks need not be synchronized. The area cost (versus a TFU implemented using two sets of transparent latches) is not significant (on the order of 10 to 15 percent).

A.2.3 CTBUS Interface

The primary component of the CT Bus interface for the TFU is the **FSM** contained in *tfu_ct_ctl* (hereafter referred to as the **CTFSM**. This **CTFSM** is responsible for monitoring the output of the TFU FIFO and taking appropriate actions to deal with that output.

The **CTFSM** normally spins until an *SOP* tag appears on the output of the FIFO. At this point, the **CTFSM** will reserve the associated **CTTX**. During the reservation process, the **CTFSM** will shift out the *SOP* tag. Once the **CTTX** has been reserved, the **CTFSM** will wait until a valid tag again appears at the output of the FIFO. If the tag is a *DATA* tag, the **CTFSM** will simply access the CTBUS to the data. It will then shift the data and tag out of the FIFO and wait until **IRDATA** returns to true before transmitting more data (if the data is available). If a *MARK* tag appears on the output of the FIFO, the **CTFSM** will first access the CTBUS to transmit a *MARK*. This will then be followed by the remainder of the data. Finally, the *EOP* tag is handled in a manner similar to the

pc_data(1)	pc_data(0)
use_hold	keep_resv

Table A.2: PCBUS Mode Writes.

MARK tag, except that an *EOP* is transmitted on the CTBUS and the **CTFSM** will break out of the main loop after transmitting the last byte. Once **IRDATA** returns to true, the **CTFSM** transmits a *FREE* command and waits for another *SOP* tag.

A.2.4 PC Bus Interface

The TFU also possesses a simple interface from the PCBUS. This is a simple, write-only interface whereby the IMU may write two flops within each TFU. These flops control how the TFU obtains reservations and whether the TFU will keep a reservation after transmitting a packet. These two variables are referred to as **use_hold** and **keep_resv** within the CTBUS interface (the TXBUS interface never looks at them). When **use_hold** is set, the TFU will use the *HOLD* command to obtain reservations. If **keep_resv** is set, the TFU will keep reservations after obtaining them. In addition, if the TFU is currently idle and does not have a reservation, it will use *HOLD* to obtain one if both signals are set. Table A.2 describes the mapping between the bits of **PC_DATA** and the above signals.

A.3 Transmission Bus (TXBUS)

This section describes the TXBUS, shown in Figure 4.12. The TXBUS is divided into two major buses: the TX Command Bus and the TX Data Bus. These buses may operate independently. The TX Command Bus (TXCBUS) transmits requests for data from the TFUs to the Command FIFO in the TX2NP interface. The TX Data Bus (TXDBUS) then transfers data from the Data FIFO in the **TX2NP** interface to the TFUs.

A.3.1 The TX Command bus

The **TFUs** arbitrate for the TX Command Bus in the normal fashion. A binary priority tree arbiter is implemented in *txarbcntl.fin*. Requests are expected to change with the rising edge of **CLK2**, while the grant lines will change on the rising edge of **CLK1**. Along with the normal request line associated with each **TFU**, there is also a special request line coming

tx_sp_req	tx_cmd	Command
0	-	Data Request
1	0	Timestamp read, accum CRC
1	1	CRC read, clear CRC

Table A.3: TX Command Bus Commands.

crc_cmd(1)	crc_cmd(0)	Command
0	0	No Operation
0	1	Accumulate CRC
1	0	Clear CRC
1	1	Drive and Clear CRC

Table A.4: CRC Generator Commands.

from each **TFU** to the arbiter. A **TFU** that sets its special request line (**tx_sp_req**) is indicating that it wishes to have control of both the Command and Data Buses during the next cycle. This allows the **TFU** to execute timestamp and checksum reads during a single bus cycle, without generating traffic in the NP Bus interface. Table A.3 describes the commands available on the TX Command Bus. While other encodings are possible, they are not used by the **TFUs**.

A.3.2 The TX Data bus

The TX Data Bus is responsible for removing data from the data FIFO and placing it in the appropriate **TFU**. It has been designed to use a shadow register that reads out the FIFO's contents and then places these into the appropriate **TFU**. This allows the TXBUS to potentially use every cycle available on the Data Bus. When **tx_sp_req** is set, however, the TX Data Bus does not transfer data out of the FIFO. Instead, the TX Data Bus becomes a slave to the Command Bus, allowing a **TFU** to read a timestamp or CRC in a single bus cycle.

The TX Data Bus is controlled by the logic block in *tx2np_tx_cntl*. This block decodes the channel id returned from the NPBUS interface through the data FIFO, sets the **TXD_ACK** lines appropriately, and also generates the appropriate command to the CRC generator. It also passes on the *MARK* and *EOP* signals from the data FIFO to the **TFUs**. The CRC commands are shown in Table A.4.

A.3.3 NPBUS Interface

The NPBUS interface is fairly simple. The output ready of the command FIFO is used as the NPBUS request. During the first phase of the NPBUS clock, the command FIFO is shifted out, and the command stored in a shadow register. The stored **crc_accum** flag is then saved into the data FIFO, along with the **mark** and **eop** flags from the NPBUS. Note that the channel ID reported to the NPBUS with a request is not that stored in the shadow latch, but that coming from the command FIFO. The stored channel id is only used for tagging the data as it goes into the data FIFO. The data FIFO is clocked off **CLK2** from the NPBUS.

A.3.4 Design Notes

The TXBUS interface takes advantage of several properties of the TFU to simplify its performance. It limits the **TFUs** to having only one outstanding request at a time. This allows the TXBUS to ignore the input readies on the command and data FIFOs, since both FIFOs contain a slot for each TFU. This does not affect the overall performance of the PRC, since the CTBUS and NPBUS interfaces are much slower than the TXBUS. In simulations, the TXBUS fills a **TFU**'s FIFO faster than the CTBUS can empty it.

The TXBUS data FIFO's output has probably been overdesigned. Since the NPBUS cannot place a longword in the FIFO every cycle, it is unlikely that the capability of using every cycle on the data bus will be used.

A.4 Reception Bus (RXBUS)

The RXBUS is responsible for carrying data between the PRCRXs and the NPBUS interface. The RXBUS is conceptually quite simple, as it consists of a CRC checker, logic for accessing the timestamp register, and two FIFOs for storing data. The RXBUS unit latches data and commands from the CTBUS whenever it is addressed, and transfers it through the FIFOs to the memory interface. Table A.5 shows how the RXBUS command lines are interpreted.

RXCMD		
1	0	Function
0	0	Data (normal)
0	1	Mark
1	0	No operation
1	1	EOP

Table A.5: RXBUS Command Encodings.

A.5 Memory Interface

The *Network Processor* Bus Interface provides the PRC with a means of accessing the host's memory to store and retrieve packets. The architecture of the NPBUS interface is depicted in Figure 4.9. The NPBUS interface essentially provides a single service to the various channels: this service can best be described as data storage and retrieval without need for worrying about addresses. The routing engine's and TFU's simply present requests for service to the NPBUS interface, which then fills that request. The only break in this is the inclusion of some control information for demarcating pages and packets.

Network Interface Operation: The network interface of the PRC interfaces to eight AMD TAXI (four each transmitters and receivers). To minimize the pin count of the PRC, the transmitters are operated in a synchronous mode where data is transmitted each cycle. Figure 4.10 shows a typical memory interface cycle. The external lines of the NPBUS interface are summarized below:

np_clk: A 40 MHz clock used to clock the memory interface of the PRC.

np_owner: Active-low grant line to PRC. This line is assumed to change with the falling edge of *np_clk*, while *np_sync* is active. Simulations assume the external signal changes after some small (typically 4ns) delay from the falling edge of *np_clk*.

np_sync: This active-high signal, when not active, denotes the first clock signal of a two-cycle access. It should change on the falling edge of *np_clk* every clock cycle.

np_data(31:0): Bidirectional data bus. PRC will drive it on writes when it is the bus owner. This bus takes less than 10 nsec to stabilize. Signals must be stable for 5 ns before and after rising edge of *np_clk* and *np_sync*. Pads are tristated the PRC is not the bus master.

np_addr(20:0): Unidirectional address bus. The PRC will drive it whenever it is the bus owner. Stabilizes within 10 ns of the rise of *np_clk*. The pads are tristated when the PRC is not the bus master.

np_rw: Indicates a read or a write. (1 implies a write). This signal is driven to read during idle periods, and is tristated when the PRC does not own the bus. Takes approximately 5ns to stabilize.

Internal Interface Operation: Internally, the NPBUS interface is based on a request-grant paradigm with an implied two-phase clock. The RXBUS and TXBUS can make requests independent of this clock, but the NPBUS will grant them coincident with the rise of the first clock phase.

Page Tag Queues

The PRC maintains its page tags in this block. Page tags are written into the appropriate queues by the host processor using the PCBUS. This unit provides bits 20 to 6 of the **np_addr**for small pages, bits 20 to 8 otherwise.

Page Offset Registers

This unit consists of registers for each channel, incrementers, and length checks. It provides bits 5 to 0 of the **np_addr**for small pages and bits 7 to 0 for large pages, and also generates the **mark** signal. The unit looks at *np_sel*and drives the selected value into a hold register at the rise of the first clock phase. The incremented value is latched back on *np_clk2*.

A.6 Cut-Through Bus (CTBUS)

The PRC and the Network Interface interact through the CTBUS. Access to the CTBUS is dynamically determined by an arbitration unit in the PRC using a binary priority-tree arbitration. The major lines of the bus include:

ct_data(31:0): The data lines. These are driven by the current bus master. The value is latched on the falling edge of CLK2 by the shadow registers.

Command	Encoding	Function
DTX	0x0	Data transfer
MARK	0x1	Indicates page fault following next data word
FREE	0x2	Reservation release
EOP	0x3	Indicates end-of-packet following next data work
RESV	0x4	Reservation request
HOLD	0x5	Place a "hold" on the next reservation of the addressed device(s)
CHK	0x6	Obtain reservation of "held" device(s)
NOOP	0x7	No operation

Table A.6: CTBUS commands.

ct_addr(12:0): These lines determine which slaves are being addressed during the current bus cycle. These lines are driven by the bus master as soon as possible after the grant lines change.

ct_mst(3:0): These lines identify the current bus master. See Table A.7 for their interpretation.

ext_req(11:0): Active-high signals from the NIRXs requesting "mastery" over the CTBUS.

ext_gnt(11:0): Active-high grant signals to the NIRX channels.

tfu_req(11:0): Active-high CTBUS requests from the TFUs.

tfu_gnt(11:0): Active-high CTBUS grant lines to the TFUs.

rx_req(3:0): Active-high CTBUS requests from the routing engines.

rx_gnt(3:0): Active-high CTBUS grants to the routing engines.

ct_ir(23:0): Active-high signals from the slaves indicating they can accept data. These lines are clocked by CLK2.

ct_ctl(2:0): Driven by the current bus master, these lines identify the type of transaction being performed. Table A.6 shows the various command encodings. Similar to the data lines, the control bus should be stable by the falling edge of CLK2.

ct_ack: Active-high acknowledgment of reservations; value changes on falling CLK1, is read on rising CLK2 of the cycle following CTBUS access.

Master	Encoding
NIRX Link 0 Channel 0	0000
NIRX Link 0 Channel 1	0001
NIRX Link 0 Channel 2	0010
NIRX Link 1 Channel 0	0100
NIRX Link 1 Channel 1	0101
NIRX Link 1 Channel 2	0110
NIRX Link 2 Channel 0	1000
NIRX Link 2 Channel 1	1001
NIRX Link 2 Channel 2	1010
NIRX Link 3 Channel 0	1100
NIRX Link 3 Channel 1	1101
NIRX Link 3 Channel 2	1110

Table A.7: CTBUS Master encodings (*ctmst.*)

A.7 Network Interface

The Network Interface provides parallel access from the PRC to eight AMD TAXI chips, four each receivers and transmitters. To conserve on interface pins, two time-multiplexed buses are used to communicate with the transmitters. To further reduce the pincount, the strobe lines to the transmitters are also shared. The interface from the TAXI receivers to the PRC is entirely asynchronous and driven by the receivers. The PRC simply latches the command and data lines whenever receivers drive their strobe line, and then read them.

A.7.1 External Interface

The network interface of the PRC has the following external pins. For all data, mode, and command lines, values are stable for 10ns before and after the rise of the strobe. All lines are active-high.

taxiclk: A 20 MHz, 33 percent duty cycle clock output by the PRC to drive the CLK input of the TAXI transmitters and receivers.

taxi_rx_strb(3:0): STRB0 lines from the TAXI RXs.

taxi_rx_mode(3:0): S1 lines from TAXI RXs

taxi_rx0_data(9:0): Data lines from TAXI RX0 (+X)

taxi_rx1_data(9:0): Data lines from TAXI RX1 (+Y)

Data Transfer Mode (*mode* = 1)										
Cycle	9	8	7	6	5	4	3	2	1	0
0	CHN		Data Byte 3							
1	CMD		Data Byte 2							
2	CRC	Ack2	Data Byte 1							
3	Ack1	Ack0	Data Byte 0							

Channel	9	8
Channel 0	0	0
Channel 1	0	1
Channel 2	1	0

Command	9	8
DTX	0	0
MARK	0	1
FREE	1	0
EOP	1	1

Command mode (*mode* = 0)		
Command	1	0
Null	1	1
Ack0	1	0
Ack1	0	1
Ack2	0	0

Table A.8: Transmission order for data through a TAXI TX.

taxi_rx2_data(9:0): Data lines from TAXI RX2 (-X)

taxi_rx3_data(9:0): Data lines from TAXI RX3 (-Y)

taxi_rx0_cmd(1:0): Command lines from TAXI RX0 (+X)

taxi_rx1_cmd(1:0): Command lines from TAXI RX1 (+Y)

taxi_rx2_cmd(1:0): Command lines from TAXI RX2 (-X)

taxi_rx3_cmd(1:0): Command lines from TAXI RX3 (-Y)

taxi_txa_mode: S1 line to TAXI TX0 (+X) and TX1 (+Y)

taxi_txb_mode: S1 line to TAXI TX2 (-X) and TX3 (-Y)

taxi_txa_data(9:0): data lines to TAXI TX0 (+X) and TX1 (+Y)

taxi_txb_data(9:0): data lines to TAXI TX2 (-X) and TX3 (-Y)

taxi_txa_cmd(1:0): command lines to TAXI TX0 (+X) and TX1 (+Y)

taxi_txb_cmd(1:0): command lines to TAXI TX2 (-X) and TX3 (-Y)

taxi_tx_strb(1:0): STRB0 to TAXI TXs, strb(0) goes to TX0 and TX2, strb(1) goes to TX1 and TX3

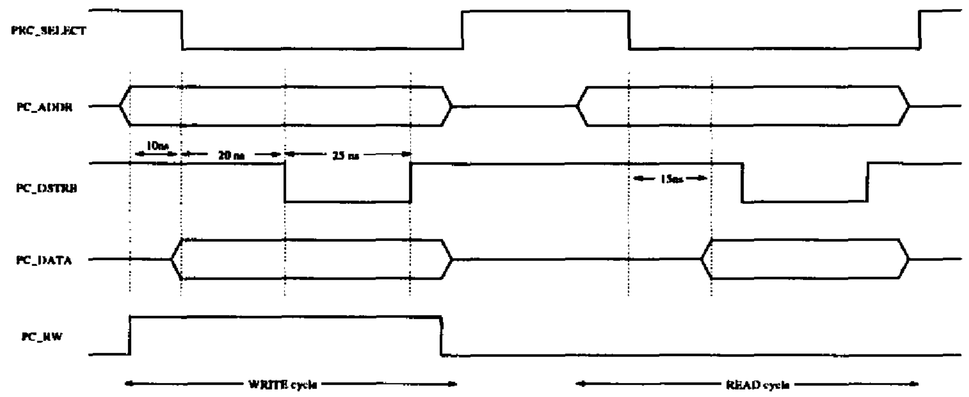


Figure A.1: Control interface read and write cycles.

A.8 Control Interface

The control interface of the PRC is a slave-only interface, with the external master initiating all accesses. External should drive the *prc_select* line low whenever the PRC is not being accessed.

pc_addr(10:2): Address selects for the control interface. The bus is 9 bits wide, and assumed to use bits 10 through 2 of the host interface (to avoid problems with unaligned accesses). The six most significant bits are used to select an internal component, while the other three carry a command. These lines must be stable at least 20ns before the rise of *pc_dstrb*.

pc_data(31:0): Active-high data bus. These must be stable at least 20ns before *pc_dstrb* during a write and held stable throughout that period. On a read, they will be stable within 20ns of the address and select line stabilizing.

pc_dstrb: Active-low data strobe. It must be held active for at least 25 ns during both read and write cycles.

prc_select: Active-low chip select. Must stabilize 20ns before the rise of *pc_dstrb*.

pc_rw: Signals a read or write access. A high (vdd) value signals a write. Must be stable at least 20ns before the rise of *pc_dstrb*.

prc_intrpt: Active-low interrupt request line. Held low until cleared by a write to the **intctl** register.

The control interface may also be used to monitor the CTBUS shadow registers by reading the **pcctdt and pcctad** registers. These registers are located at hex addresses `0x1a8` and `0x1a9` respectively. The registers are not clocked, so *pc_dstrb* is not required. This capability is provided solely for testing purposes, and is not recommended for general consumption — i.e., don't try this at home!

A.9 Testing Logic

The following is an ordered list of components on the scan chain, beginning from the *scanin* input. Thus, the last component in the list will be driving *scanout*.

1. RXBUS
   (a) *state(0:1)*: state of the **ct2rx_ctl** FSM.
2. NPINT
   (a) *state(0:2)*: state of the **np_ctl** FSM.
   (b) *rx_req_l*: latched RXBUS request for NP access
   (c) *clk1_en*: latched NPCLK1 enable
   (d) *clk2_en*: latched NPCLK2 enable
   (e) *txd_si_en*: enables shift-in signal to TXD fifo
   (f) *nphalt_l*: latched NPHALT
   (g) *tx_req_l*: called SCANOUT, latched TXBUS NP request
3. TXBUS arbiter
   (a) *cycle(0:3)*: Used to determine current priority levels for the arbiter.
   (b) *tx_gnt(0:11)*: The active-high grant signals to each TFU.
   (c) *txc_chan(0:3)*: The encoded channel id of the current bus master that will be latched into the command FIFO.
   (d) *sp_req*: An active-high signal that indicates the current cycle is a "special request" (either a timestamp or CRC read).
   (e) *tx_cmd_lat*: Latched CMD bus signal
   (f) *tx_idle*: An active high signal indicating that the current cycle is idle.

4. TFUSUPERBLK (TFUs 0-11) Each TFU has:

   (a) *fifo_fout_l*: Stabilized IR for next slot from the FIFO.

   (b) *fifo_ir_l*: Stabilized input ready from the FIFO.

   (c) *qrdy_l*: Stabilized output ready from the page queue.

   (d) *eop_tag*: This flag used to drive an EOP tag into the FIFO, and is set by the **TXFSM**.

   (e) *sop_si*: This flag is used to generate a shift-in pulse into the FIFO for the SOP page tag. It is set by the **TXFSM**.

   (f) *tx_ctl_state(0:2)*: The current state of the **TXFSM**.

   (g) *tx_req*: The current request from the TFU for the TXBUS.

   (h) *fifo_or_l*: Stabilized output ready from the FIFO.

   (i) *ct_state(0:3)*: Current state for the **CTFSM**.

   (j) *so_flg*: This flag is used to drive the shift-out signal to the FIFO.

   (k) *ct_req*: CTBUS request signal.

5. Reservation unit

   (a) *held(0:11)*: Current held devices

   (b) *resvd(0:11)*: Currently reserved devices

6. NITX0 and NITX1

   (a) *state0(0:2)*: current state for NITX FSM 0

   (b) *state1(0:2)*: current state for NITX FSM 1

7. CTBUS arbiter

   (a) *cycle(0:4)*: This value, which is the output from a counter, is used to determine the priority of each channel.

   (b) *ct_msti(2:3)*: These lines indicate the current CTBUS master if it is an NIRX or routing engine, but are inverted from the values shown in Table A.7.

   (c) *txct_en_bar*: True when a TFU is granted

(d) *nirx_wrt*: True when a NIRX is granted

(e) *ext_gnt(0:11)*: The active-high grant lines for the NIRXs, these are sent to the routing engines to allow them to drive the address lines.

(f) *rx_gnt(0:3)*: The active-high grant lines for the routing engines.

(g) *tfu_gnt(0:11)*: The active-high grant lines sent to the TFUs.

(h) *ct_idle*: An active-high signal denoting the current bus cycle as idle. This signal is only used within the arbiter.

8. NIRX blocks 0-3

   (a) *rxct_req*: CTBUS request signal for the routing engine.

   (b) *ext_req(0:2)*: CTBUS request signal for the NIRXs.

APPENDIX B

Routing Engine Internals

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

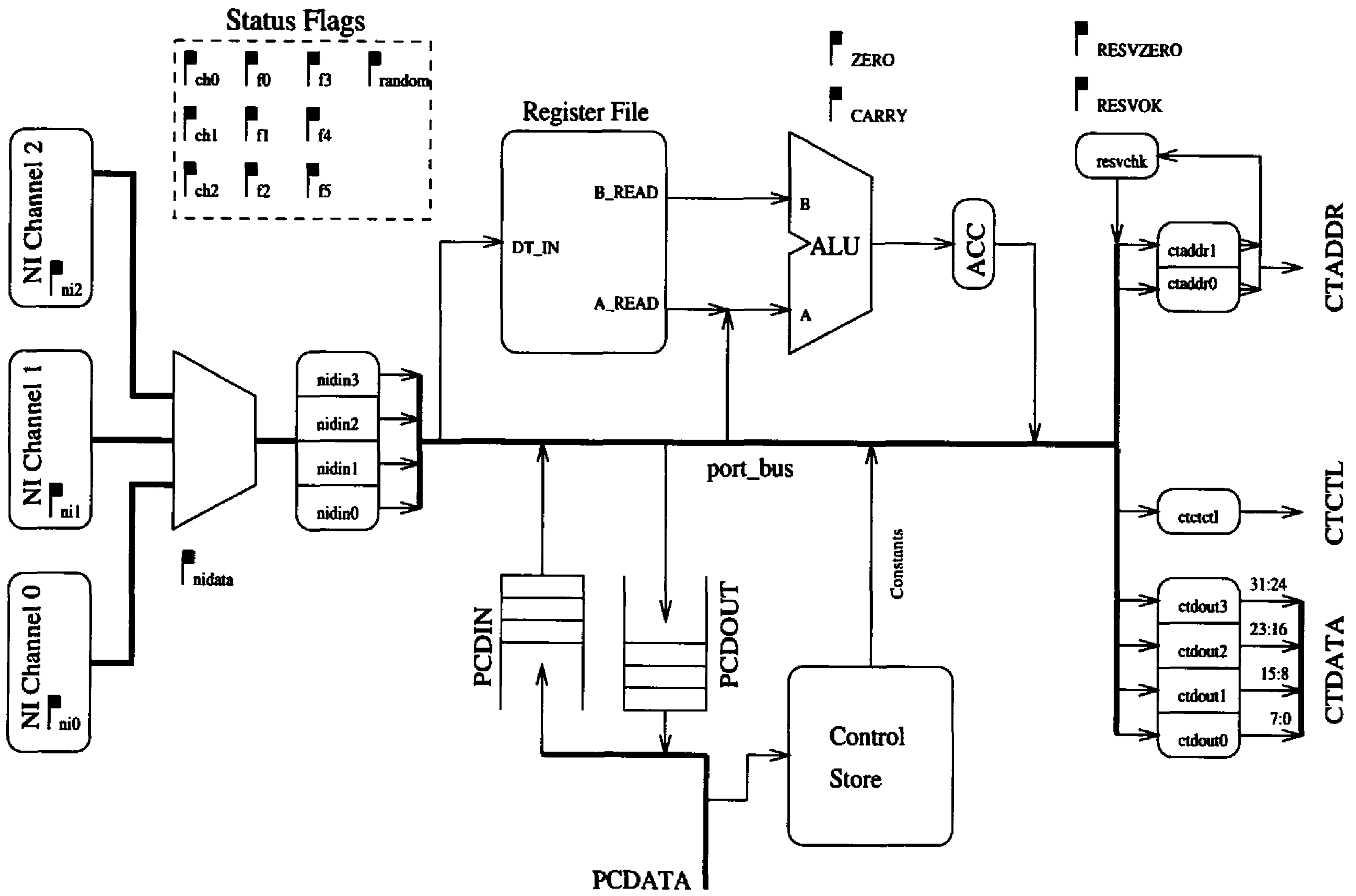


Figure B.1: PRC routing engine internal architecture.

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		B Port Select					A 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 (REG0)
0	0	0	0	1	Reg 1 (REG1)
0	0	0	1	0	Reg 2 (REG2)
0	0	0	1	1	Reg 3 (REG3)
0	0	1	0	0	Reg 4 (REG4)
0	0	1	0	1	Reg 5 (REG5)
0	0	1	1	0	Reg 6 (REG6)
0	0	1	1	1	Reg 7 (REG7)
0	1	0	0	0	Reg 8 (REG8)
0	1	0	0	1	Reg 9 (REG9)
0	1	0	1	0	Reg 10 (REG10)
0	1	0	1	1	Reg 11 (REG11)
0	1	1	0	0	Reg 12 (REG12)
0	1	1	0	1	Reg 13 (REG13)
0	1	1	1	0	Trap Reg 0 (TRAP0)
0	1	1	1	1	Trap Reg 1 (TRAP1)
1	0	1	0	0	PCBus FIFO In (PCDIN)
1	0	1	0	1	NI Data In 0 (NID0)
1	0	1	1	0	NI Data In 1 (NID1)
1	0	1	1	1	NI Data In 2 (NID2)
1	1	0	0	0	NI Data In 3 (NID3)
1	1	0	0	1	CTBus Resv Feedback 0 (CTFB0)
1	1	0	1	0	CTBus Data Feedback 1 (CTFB1)
1	1	1	1	0	Accumulator (ACC)

ALU Port B Select				
M(13)	M(12)	M(11)	M(10)	Device Selected
0	0	0	0	Reg 0 (REG0)
0	0	0	1	Reg 1 (REG1)
0	0	1	0	Reg 2 (REG2)
0	0	1	1	Reg 3 (REG3)
0	1	0	0	Reg 4 (REG4)
0	1	0	1	Reg 5 (REG5)
0	1	1	0	Reg 6 (REG6)
0	1	1	1	Reg 7 (REG7)
1	0	0	0	Reg 8 (REG8)
1	0	0	1	Reg 9 (REG9)
1	0	1	0	Reg 10 (REG10)
1	0	1	1	Reg 11 (REG11)
1	1	0	0	Reg 12 (REG12)
1	1	0	1	Reg 13 (REG13)
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								x
			0	Do not trigger controller															
			1	Trigger controller															
				0	Do not trigger CTBUS interface														
				1	Trigger CTBUS interface														

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	x
			0	Do not trigger controller															
			1	Trigger controller															
				0	Do not trigger CTBUS interface														
				1	Trigger CTBUS interface														

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 (REG0)
0	0	0	0	1	Reg 1 (REG1)
0	0	0	1	0	Reg 2 (REG2)
0	0	0	1	1	Reg 3 (REG3)
0	0	1	0	0	Reg 4 (REG4)
0	0	1	0	1	Reg 5 (REG5)
0	0	1	1	0	Reg 6 (REG6)
0	0	1	1	1	Reg 7 (REG7)
0	1	0	0	0	Reg 8 (REG8)
0	1	0	0	1	Reg 9 (REG9)
0	1	0	1	0	Reg 10 (REG10)
0	1	0	1	1	Reg 11 (REG11)
0	1	1	0	0	Reg 12 (REG12)
0	1	1	0	1	Reg 13 (REG13)
0	1	1	1	0	Trap Reg 0 (TRAP0)
0	1	1	1	1	Trap Reg 1 (TRAP1)
1	0	1	0	0	PCBus FIFO In (PCDIN)
1	0	1	0	1	NI Data In 0 (NID0)
1	0	1	1	0	NI Data In 1 (NID1)
1	0	1	1	1	NI Data In 2 (NID2)
1	1	0	0	0	NI Data In 3 (NID3)
1	1	0	0	1	CTBus Resv Feedback 0 (CTFB0)
1	1	0	1	0	CTBus Resv Feedback 1 (CTFB1)
1	1	1	1	0	Accumulator (ACC)

Destination Select Field					
M(14)	M(13)	M(12)	M(11)	M(10)	Device Selected
0	0	0	0	0	Reg 0 (REG0)
0	0	0	0	1	Reg 1 (REG1)
0	0	0	1	0	Reg 2 (REG2)
0	0	0	1	1	Reg 3 (REG3)
0	0	1	0	0	Reg 4 (REG4)
0	0	1	0	1	Reg 5 (REG5)
0	0	1	1	0	Reg 6 (REG6)
0	0	1	1	1	Reg 7 (REG7)
0	1	0	0	0	Reg 8 (REG8)
0	1	0	0	1	Reg 9 (REG9)
0	1	0	1	0	Reg 10 (REG10)
0	1	0	1	1	Reg 11 (REG11)
0	1	1	0	0	Reg 12 (REG12)
0	1	1	0	1	Reg 13 (REG13)
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 Out Reg 0 (CTD0)
1	0	1	1	0	CTBUS Data Out Reg 1 (CTD1)
1	0	1	1	1	CTBUS Data Out Reg 2 (CTD2)
1	1	0	0	0	CTBUS Data Out Reg 3 (CTD3)
1	1	0	0	1	CTBUS Address Reg 0 (CTADDR0)
1	1	0	1	0	CTBUS Address Reg 1 (CTADDR1)
1	1	0	1	1	CTBUS Control Reg (CTCTL)
1	1	1	0	0	PCBUS FIFO out (PCDOUT)

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	Flag Mask													

Flag Select Mask														
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)	Flag Modified
-	-	-	-	-	-	-	-	-	-	-	-	-	1	User Flag 0
-	-	-	-	-	-	-	-	-	-	-	-	1	-	User Flag 1
-	-	-	-	-	-	-	-	-	-	-	1	-	-	User Flag 2
-	-	-	-	-	-	-	-	-	-	1	-	-	-	User Flag 3
-	-	-	-	-	-	-	-	-	1	-	-	-	-	User Flag 4
-	-	-	-	-	-	-	-	1	-	-	-	-	-	User Flag 5
-	-	-	-	-	-	-	1	-	-	-	-	-	-	Read NI0
-	-	-	-	-	-	1	-	-	-	-	-	-	-	Read NI1
-	-	-	-	-	1	-	-	-	-	-	-	-	-	Read NI2
-	-	-	-	1	-	-	-	-	-	-	-	-	-	Read selected NI
-	-	-	1	-	-	-	-	-	-	-	-	-	-	Set channel to NI0
-	-	1	-	-	-	-	-	-	-	-	-	-	-	Set channel to NI1
-	1	-	-	-	-	-	-	-	-	-	-	-	-	Set channel to NI2
1	-	-	-	-	-	-	-	-	-	-	-	-	-	Set channel to null

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	Trap	Condition Select								Trap 0 Mask				Trap 1 Mask				Link Enb
		0	Evaluate conditions normally																
		1	Automatically jump to TRAP1 address																
																		Do not store return Link	0
																		Store return Link	1

Trap 0 Select Mask					
M(8)	M(7)	M(6)	M(5)	Trap Condition	Prty
-	-	-	1	CT Interface Busy	3
-	-	1	-	NI 0 data ready flag	0
-	1	-	-	NI 1 data ready flag	1
1	-	-	-	NI 2 data ready flag	2

Trap 1 Select Mask					
M(4)	M(3)	M(2)	M(1)	Trap Condition	Prty
-	-	-	1	NI 0 data ready flag	1
-	-	1	-	NI 1 data ready flag	2
-	1	-	-	NI 2 data ready flag	3
1	-	-	-	PCDIN OR flag	0

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)
0	1	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
-	-	0	0	0	0	0	0	Resv Stat Link 0 Chan 0
-	-	0	0	0	0	0	1	Resv Stat Link 0 Chan 1
-	-	0	0	0	0	1	0	Resv Stat Link 0 Chan 2
-	-	0	0	0	0	1	1	Resv Stat Link 1 Chan 0
-	-	0	0	0	1	0	0	Resv Stat Link 1 Chan 1
-	-	0	0	0	1	0	1	Resv Stat Link 1 Chan 2
-	-	0	0	0	1	1	0	Resv Stat Link 2 Chan 0
-	-	0	0	0	1	1	1	Resv Stat Link 2 Chan 1
-	-	0	0	1	0	0	0	Resv Stat Link 2 Chan 2
-	-	0	0	1	0	0	1	Resv Stat Link 3 Chan 0
-	-	0	0	1	0	1	0	Resv Stat Link 3 Chan 1
-	-	0	0	1	0	1	1	Resv Stat Link 3 Chan 2
-	-	0	0	1	1	0	0	Ack Flag
-	-	0	0	1	1	0	1	PCDIN output ready
-	-	0	0	1	1	1	0	PCDOUT input ready
-	-	0	0	1	1	1	1	Random Bit
-	-	0	1	0	0	0	0	User Flag 0
-	-	0	1	0	0	0	1	User Flag 1
-	-	0	1	0	0	1	0	User Flag 2
-	-	0	1	0	0	1	1	User Flag 3
-	-	0	1	0	1	0	0	User Flag 4
-	-	0	1	0	1	0	1	User Flag 5
-	-	0	1	0	1	1	0	Resv Feedback Zero Flag
-	-	0	1	0	1	1	1	NI 0 data ready flag
-	-	0	1	1	0	0	0	NI 1 data ready flag
-	-	0	1	1	0	0	1	NI 2 data ready flag
-	-	0	1	1	0	1	0	Reservation OK Flag
-	-	0	1	1	0	1	1	selected NI DR flag
-	-	0	1	1	1	0	0	CT Interface Busy
-	-	0	1	1	1	0	1	NI CRC flag
0	0	0	1	1	1	1	0	Channel0 Flag
0	1	0	1	1	1	1	0	Channel1 Flag
1	0	0	1	1	1	1	0	Channel2 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
-	-	1	0	0	0	0	0	Resv Stat Link 0 Chan 0
-	-	1	0	0	0	0	1	Resv Stat Link 0 Chan 1
-	-	1	0	0	0	1	0	Resv Stat Link 0 Chan 2
-	-	1	0	0	0	1	1	Resv Stat Link 1 Chan 0
-	-	1	0	0	1	0	0	Resv Stat Link 1 Chan 1
-	-	1	0	0	1	0	1	Resv Stat Link 1 Chan 2
-	-	1	0	0	1	1	0	Resv Stat Link 2 Chan 0
-	-	1	0	0	1	1	1	Resv Stat Link 2 Chan 1
-	-	1	0	1	0	0	0	Resv Stat Link 2 Chan 2
-	-	1	0	1	0	0	1	Resv Stat Link 3 Chan 0
-	-	1	0	1	0	1	0	Resv Stat Link 3 Chan 1
-	-	1	0	1	0	1	1	Resv Stat Link 3 Chan 2
-	-	1	0	1	1	0	0	Ack Flag
-	-	1	0	1	1	0	1	PCDIN output ready
-	-	1	0	1	1	1	0	PCDOUT input ready
-	-	1	0	1	1	1	1	Random Bit
-	-	1	1	0	0	0	0	User Flag 0
-	-	1	1	0	0	0	1	User Flag 1
-	-	1	0	1	1	1	0	User Flag 2
-	-	1	1	0	0	1	1	User Flag 3
-	-	1	1	0	1	0	0	User Flag 4
-	-	1	1	0	1	0	1	User Flag 5
-	-	1	1	0	1	1	0	Resv Feedback Zero Flag
-	-	1	1	0	1	1	1	NI 0 DR Flag
-	-	1	1	1	0	0	0	NI 1 DR Flag
-	-	1	1	1	0	0	1	NI 2 DR Flag
-	-	1	1	1	0	1	0	Reservation OK Flag
-	-	1	1	1	0	1	1	NIsel DR Flag
-	-	1	1	1	1	0	0	CT Interface Busy
-	-	1	1	1	1	0	1	NI CRC Flag
0	0	1	1	1	1	1	0	Channel0 Flag
0	1	1	1	1	1	1	0	Channel1 Flag
1	0	1	1	1	1	1	0	Channel2 Flag
0	0	1	1	1	1	1	1	False
0	1	1	1	1	1	1	1	Carry Flag
1	0	1	1	1	1	1	1	Zero Flag
1	1	1	1	1	1	1	1	True

Table B.8: Condition Code Selection Coding.

B.1 Internal Architecture of the RX

The operation of the RX is managed by several cooperating state machines. Most of these state machines are dedicated to control and operation of the external interfaces: the inbound NI interface and the outbound CTBUS interface. The last state machine is dedicated to instruction sequencing and the maintenance of user and status flags.

In describing the implementation of the RX we will first discuss the operation/control of the major interface units and then describe the operation of the microsequencer which controls all of these units under the direction of the downloaded microcode.

B.1.1 Data output module

The outbound CTBUS interface is intended to isolate the μsequencer from the details of the CTBUS. The operations are complicated by the fact that the interface has to implement the byte-level flow-control associated with the semantics of the **IRDATA** lines on the CTBUS transmitters when they are selected as slaves.

The outbound CTBUS interface contains four 8 bit registers that are addressable as the target of a transfer. The **ctd** register stores the data that will drive the *ctdata* lines of the CTBUS during the next CTBUS transaction. The **ctaddr0** and **ctaddr1** register store the address mask used to select slaves during the next CTBUS transaction. The interpretation of the values can be found in Table B.9. The **ctctl** stores the CTBUS command that will be issued.

This interface is also used to transfer routing primitives to the state machine that will control the bulk of a packet's movements.

Again, similar to the RXBUS, the CTBUS interface conveys its status with the **ctbusy** flags and is triggered with the **CTGO** line. The **ACK** flag will contain the value of the **CTBUSACK** as driven by the slave device or reservation status unit as appropriate.

B.1.2 Instruction sequencing

Microinstruction sequencing and decoding is controlled by the state **useq** machine implemented in the VSYN file **$PROJECT_HOME/verilog/useq.v** and the registers shown in Figure B.2.

ctaddrl							ctaddr0						Device addressed
6	5	4	3	2	1	0	5	4	3	2	1	0	
1	-	-	-	-	-	-	-	-	-	-	-	-	Local node (to buffer)
-	1	-	-	-	-	-	-	-	-	-	-	-	Link 3, channel 2
-	-	1	-	-	-	-	-	-	-	-	-	-	Link 3, channel 1
-	-	-	1	-	-	-	-	-	-	-	-	-	Link 3, channel 0
-	-	-	-	1	-	-	-	-	-	-	-	-	Link 2, channel 2
-	-	-	-	-	1	-	-	-	-	-	-	-	Link 2, channel 1
-	-	-	-	-	-	1	-	-	-	-	-	-	Link 2, channel 0
-	-	-	-	-	-	-	1	-	-	-	-	-	Link 1, channel 2
-	-	-	-	-	-	-	-	1	-	-	-	-	Link 1, channel 1
-	-	-	-	-	-	-	-	-	1	-	-	-	Link 1, channel 0
-	-	-	-	-	-	-	-	-	-	1	-	-	Link 0, channel 2
-	-	-	-	-	-	-	-	-	-	-	1	-	Link 0, channel 1
-	-	-	-	-	-	-	-	-	-	-	-	1	Link 0, channel 0

Table B.9: ctaddrl and ctaddr0 device mapping.

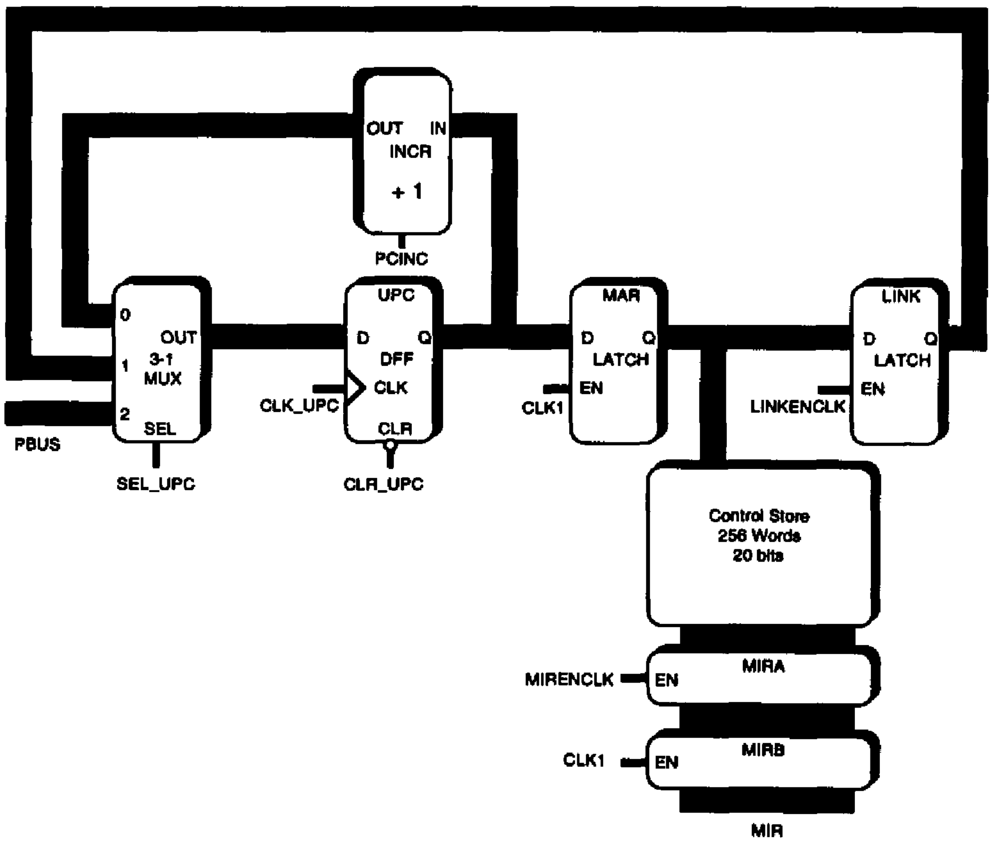


Figure B.2: Partial structure of the address sequencing hardware of the RX.

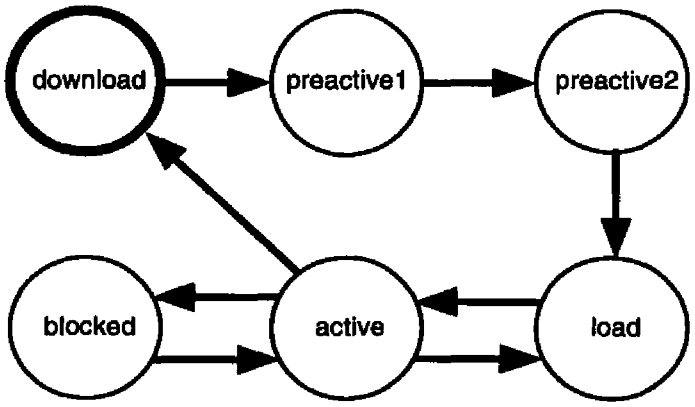


Figure B.3: Simplified state diagram for microsequencer controller.

APPENDIX C

The PRC Micro-Assembler

C.1 Introduction

This appendix describes the input language grammar for the PRC's micro-assembler (`prcmasm`). This micro-assembler was originally written by James Dolter [21] and has been updated to support the new structure of the PRC. The assembler is written to use the GNU C++ compiler g++, GNU Bison, and GNU Flex.

C.2 Object file format

The object file produced by `prcmasm` is a simple ASCII test file separated into two sections, the preamble and the code. The preamble section, lines identified by having a "#" in column 1, provides information to assist the user in identifying the source of the object code. The code section consists of a series of address-data number pairs separated by a "/". There is only one pair per line and only addresses containing data will be placed in the code section. The numbers are in hexadecimal without a proceeding "0x" and each line is terminated with a ";".

C.3 Input Grammar

Below is a pseudo BNF grammar abstracted from the bison input file used to produce the parser section of the `prcmasm`. All non-terminals are bracketed by ⟨/⟩ and alternatives are separated by |. This notation may cause a little confusion since | is also used in the boolean 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⟩ , operation ⟨int⟩ |
    alu ⟨alu_op⟩

⟨mux_select⟩ → false | carry | zero | true

⟨alu_op⟩ →
    ⟨source⟩ + ⟨alu_b⟩ |
    ⟨source⟩ + ⟨alu_b⟩ + 1 |
    ⟨source⟩ - ⟨alu_b⟩ - 1 |
    ⟨source⟩ - ⟨alu_b⟩ |
    ⟨source⟩ |
    ⟨source⟩ + 1 |
    ⟨source⟩ - 1 |
    ∧ ⟨source⟩ |
    ⟨source⟩ | ⟨alu_b⟩ |
    ⟨source⟩ | ∧ ⟨alu_b⟩ |
    ∧ ⟨source⟩ | ⟨alu_b⟩ |
    ⟨source⟩ & ⟨alu_b⟩ |
    ⟨source⟩ & ∧ ⟨alu_b⟩ |
    ∧ ⟨source⟩ & ⟨alu_b⟩ |
    ⟨source⟩ exor ⟨alu_b⟩ |
    ⟨source⟩ exnor ⟨alu_b⟩ |
    -1 |
    0 |
    1

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

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

⟨dest⟩ →
    reg0 |
    reg1 |
    reg2 |
    reg3 |
    reg4 |
    reg5 |
    reg6 |
    reg7 |
    reg8 |
    reg9 |
    reg10 |
    reg11 |
    reg12 |
    reg13 |
    trap0 |
    trap1 |
    ctd0 |
    ctd1 |
    ctd2 |
    ctd3 |
    ctaddr0 |
    ctaddr1 |
    ctctl |
    pcdout

⟨source⟩ →
    reg0 |
    reg1 |
    reg2 |
    reg3 |
    reg4 |
    reg5 |
    reg6 |
    reg7 |
    reg8 |
    reg9 |
    reg10 |
    reg11 |
    reg12 |
    reg13 |
    ctfb0 |
    ctfb1 |
    nid0 |
    nid1 |
    nid2 |
    nid3 |
    acc |
    pcdin |
    trap0 |
    trap1
```

⟨alu_b⟩ →
reg0 |
reg1 |
reg2 |
reg3 |
reg4 |
reg5 |
reg6 |
reg7 |
reg8 |
reg9 |
reg10 |
reg11 |
reg12 |
reg13 |
trap0 |
trap1

⟨xfer_instruct⟩ → **xfer** ⟨source⟩ , ⟨dest⟩ ⟨goop⟩

⟨wait_instruct⟩ → **wait** ⟨polarity⟩ ⟨condition⟩ ⟨trap0_handler⟩ ⟨trap1_handler⟩ ⟨link_control⟩

polarity → ~ | ϵ

⟨trap0_handler⟩ → , **trap0** **(** ⟨trap0_conds⟩ **)** | ϵ

⟨trap1_handler⟩ → , **trap1** **(** ⟨trap1_conds⟩ **)** | ϵ

⟨trap0_conds⟩ → ⟨trap0_conds⟩ , ⟨trap0_cond⟩ | ⟨trap0_cond⟩

⟨trap0_cond⟩ → ∧ **ctbusy** | **ni0** | **ni1** | **ni2**

⟨trap1_conds⟩ → ⟨trap1_conds⟩ , ⟨trap1_cond⟩ | ⟨trap1_cond⟩

⟨trap1_cond⟩ → **pcinor** | **ni0** | **ni1** | **ni2**

⟨condition⟩ →
ack |
ctbusy |
f0 |
f1 |
f2 |
f3 |
f4 |
f5 |
rsvstat0 |
rsvstat1 |

rsvstat2 |
rsvstat3 |
rsvstat4 |
rsvstat5 |
rsvstat6 |
rsvstat7 |
rsvstat8 |
rsvstat9 |
rsvstat10 |
rsvstat11 |
ni0 |
ni1 |
ni2 |
ch0 |
ch1 |
ch2 |
nidata |
nicrc |
resvok |
resvzero |
random |
false |
carry |
pcinor |
pcoutir |
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⟩

⟨valid_flags⟩ → ⟨valid_flags⟩ , ⟨valid_flag⟩ | ⟨valid_flag⟩

⟨valid_flag⟩ →
f0 |
f1 |
f2 |
f3 |
f4 |
f5 |

```
rdni0 |
rdni1 |
rdni2 |
rdnidata |
ch0 |
ch1 |
ch2 |
chnull |
all
```

APPENDIX D

A z-channel Routing Engine

This appendix describes the version of the z-channel that was designed as a possible replacement for the microcontroller-based routing engine in the PRC. This implementation provides the z-checks and z-operations necessary to use z as a single-valued unsigned integer counter or as a boolean flag, which is sufficient for any of the schemes presented in Chapter 7. If necessary, the architecture could be easily modified to support a double-valued scheme; this allows two routing schemes to coexist on the same channel when one or more of them needs access to a separate state variable.

Several z-checks have been eliminated by subsuming them with other operations. For example, the boolean z-check is identical to the z sign check for an unsigned value of z,

z-check	Outcomes	Encoding	Offset
z sign check	$z > 0$	01	1
	$z = 0$	01	0
x-y comparison	$\lvert x\rvert > \lvert y\rvert$	00	1
	$\lvert x\rvert \leq \lvert y\rvert$	00	0
z-c comparison	$\lvert z\rvert > \lvert c\rvert$	10	1
	$\lvert z\rvert \leq \lvert c\rvert$	10	0
random		11	1
		11	0

Table D.1: z-checks in the example implementation.

z-operation	Encoding
Decrement z	00
Increment z	01
Pass z	10
Clear z	11

Table D.2: z-operations in the example implementation.

Num z-checks	$Z_n = 4$
Max outcomes	$Z_r = 2$
Num links	$L = 4$
Num dimensions	$D = 2$
Channels per link	$C = 3$
Channel sets	$S = 3$
RTPs/RTI	$d_r = 4$
Num z-ops	$p_n = 4$

Table D.3: General z-channel parameters.

no additional operator is necessary. In addition, no null check is required; this can be implemented by using z-check and simply using the same routing instruction for every outcome.

Table D.2 shows the necessary z-operations. One notable omission, at first glance, is the *set* operation. This operation, however, is provided by the *increment* operator; for any value of z save 255, incrementing it will result in a non-zero z. In addition, if we are setting z at some node, we already know whether it is non-zero or not from the earlier z-check, and can choose not to set it again if it is already set. Eliminating the *set* operation reduces the number of bits required to specify the z-operator from 3 to 2, which reduces the overall size of each routing instruction from 40 bits to 36 bits, a savings of 10 percent.

D.1 Operation

The example z-channel closely follows the architecture presented in Figure 7.4, although there are several minor changes. The PRC links have an internode flow control window of 2 flits; consequently, each receiver must be able to buffer two unacknowledged flits at any one time. Thus, the data register in the switching control module is implemented as a two-deep FIFO (for unacknowledged flits) and a separate data register for the (acknowledged) flit that is waiting for the CTBUS. The actual z-channel routing engine operates almost entirely asynchronously; the same state machine that removes data from the link FIFO issues the routing instruction during the same cycle that it forwards the header to the flow-control FIFO. To eliminate excess registers from the switching control module, each channel's module includes its own RAM for the routing instruction table; the routing instruction issue simply reloads the address register for the RAM and sets a single buffer/cut flag. If the buffer flag is set, the packet is automatically buffered.

A host-controlled *run_load* signal is used to trigger or suspend full operation of the z-channel routing engine; when it is cleared, any incoming packet will be buffered automatically. This signal has two main uses: ensuring that packets will not be forwarded until the routing tables have been initialized, and temporarily halting packet routing to download a new routing-switching scheme(s).

BIBLIOGRAPHY

BIBLIOGRAPHY

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

[2] H. G. Badr and S. Podar, "An optimal shortest-path routing policy for network computers with regular mesh-connected topologies," *IEEE Trans. Computers*, vol. C-38, no. 10, pp. 1362–1370, October 1989.

[3] K. Bolding, S.-C. Cheung, S.-E. Choi, C. Ebeling, S. Hassoun, T. A. Ngo, and R. Wille, "The Chaos router chip: Design and implementation of an adaptive router," in *Proc. VLSI*, September 1993.

[4] R. Boppana and S. Chalasani, "A comparison of adaptive wormhole routing algorithms," in *Proc. Int'l Symposium on Computer Architecture*, pp. 351–360, 1993.

[5] S. Borkar, R. Cohn, et al., "Supporting systolic and memory communication in iWarp," in *Proc. Int'l Symposium on Computer Architecture*, pp. 70–81, 1990.

[6] G. A. Boughton, "Arctic routing chip," in *Proc. Parallel Computer Routing and Communication Workshop*, pp. 310–317, June 1994.

[7] C.-M. Chiang and L. M. Ni, "Multi-address encoding for multicast," in *Proc. Parallel Computer Routing and Communication Workshop*, pp. 146–160, May 1994.

[8] A. A. Chien, "A cost and speed model for k-ary n-cube wormhole routers," in *Proc. Hot Interconnects*, August 1993.

[9] R. Cypher, A. Ho, S. Konstantinidou, and P. Messina, "Architectural requirements of parallel scientific applications with explicit communication," in *Proc. Int'l Symposium on Computer Architecture*, pp. 2–13, May 1993.

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

[11] 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.

[12] W. Dally, "Virtual-channel flow control," *IEEE Trans. Parallel and Distributed Systems*, vol. 3, no. 2, pp. 194–205, March 1992.

[13] W. Dally and H. Aoki, "Deadlock-free adaptive routing in multicomputer networks using virtual channels," *IEEE Trans. Parallel and Distributed Systems*, vol. 4, no. 4, pp. 466–475, April 1993.

[14] W. J. Dally, L. R. Dennison, D. Harris, K. Kan, and T. Zanthopoulos, "The reliable router: A reliable and high-performance communication substrate for parallel computers," in *Proc. Parallel Computer Routing and Communication Workshop*, pp. 241–255, June 1994.

[15] W. J. Dally, J. A. Fiske, J. S. Keen, R. A. Lethin, M. D. Noakes, P. R. Nuth, R. E. Davison, and G. A. Fyler, "The Message-Driven Processor: A multicomputer processing node with efficient mechanisms," *IEEE Micro*, pp. 23–39, April 1992.

[16] C. Dalton, G. Watson, D. Banks, C. Calamvokis, A. Edwards, and J. Lumley, "Afterburner," *IEEE Network Magazine*, pp. 36–43, July 1993.

[17] S. P. Dandamudi and D. L. Eager, "The effectiveness of combining in reducing hot-Spot contention in hypercube multicomputers," in *Proc. International Conference on Parallel Processing*, pp. I–291 - I–295, 1990.

[18] S. P. Dandamudi and D. L. Eager, "Hot-spot contention in binary hypercube networks," *IEEE Trans. Computers*, pp. 239 - 244, February 1992.

[19] 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.

[20] 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.

[21] J. Dolter, *A Programmable Routing Controller Supporting Multi-Mode Routing and Switching in Distributed Real-Time Systems*, PhD thesis, University of Michigan, September 1993.

[22] J. Duato, "A new theory of deadlock-free adaptive routing in wormhole networks," *IEEE Trans. Parallel and Distributed Systems*, pp. 1320–1331, December 1993.

[23] W. Feng, J. Rexford, S. Daniel, A. Mehra, and K. Shin, "Tailoring routing and switching schemes to application workloads in multicomputer networks," Computer Science and Engineering Technical Report CSE-TR-239-95, University of Michigan, May 1995.

[24] W. Feng, J. Rexford, A. Mehra, S. Daniel, J. Dolter, and K. Shin, "Architectural support for managing communication in point-to-point distributed systems," Technical Report CSE-TR-197-94, University of Michigan, March 1994.

[25] D. Ferrari, "Client requirements for real-time communication services," *IEEE Communications Magazine*, pp. 65–72, November 1990.

[26] F. Hady and D. Smitley, "Adaptive vs. non-adaptive routing: An application driven case study," Technical Report SRC-TR-93-099, Supercomputing Research Center, Bowie, Maryland, March 1993.

[27] J.-M. Hsu and P. Banerjee, "Performance measurement and trace driven simulation of parallel CAD and numeric applications on a hypercube multicomputer," *IEEE Trans. Parallel and Distributed Systems*, vol. 3, no. 4, pp. 451–464, July 1992.

[28] N. C. Hutchinson and L. L. Peterson, "The x-Kernel: An architecture for implementing network protocols," *IEEE Trans. Software Engineering*, vol. 17, no. 1, pp. 1–13, January 1991.

[29] H. Kanakia and D. R. Cheriton, "The VMP network adapter board (NAB): high-performance network communication for multiprocessors," *Proceedings of the SIGCOMM Symposium*, pp. 175–187, August 1988.

[30] 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.

[31] D. D. Kandlur, K. G. Shin, and D. Ferrari, "Real-time communication in multi-hop networks," in *Proc. Int. Conf. on Distributed Computer Systems*, pp. 300–307, May 1991.

[32] V. Karamcheti and A. A. Chien, "Software overhead in messaging layers: Where does the time go?," in *Proc. Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 51–60, October 1994.

[33] 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.

[34] J. Kim, Z. Liu, and A. Chien, "Compressionless routing: A framework for adaptive and fault-tolerant routing," in *Proc. Int'l Symposium on Computer Architecture*, pp. 289–300, April 1994.

[35] J. H. Kim and A. A. Chien, "Evaluation of wormhole routed networks under hybrid traffic loads," in *Proc. Hawaii Int'l Conf. on System Sciences*, pp. 276–285, January 1993.

[36] S. Konstantinidou, "Segment router: A novel router design for parallel computers," in *Symposium on Parallel Algorithms and Architectures*, June 1994.

[37] S. Konstantinidou and L. Snyder, "Chaos router: Architecture and performance," in *Proc. Int'l Symposium on Computer Architecture*, pp. 212–221, May 1991.

[38] S. Konstantinidou and L. Snyder, "The Chaos router," *IEEE Trans. Computers*, vol. 43, no. 12, pp. 1386–1397, December 1994.

[39] 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.

[40] H. S. Lee, H. W. Kim, J. Kim, and S. Lee, "Adaptive virtual cut-through as an alternative to wormhole routing," in *Proc. International Conference on Parallel Processing*, pp. I–68 – I–75, 1995.

[41] C. Leiserson, Z. Abuhamdeh, D. Douglas, C. Feynman, M. Ganmukhi, J. Hill, W. D. Hillis, B. Kuszmaul, M. St. Pierre, D. Wells, M. Wong, S.-W. Yang, and R. Zak, "The network architecture of the connection machine CM-5," in *Symposium on Parallel Algorithms and Architectures*, pp. 272–285, June 1992.

[42] J.-P. Li and M. W. Mutka, "Priority based real-time communication for large scale wormhole networks," in *Proc. International Parallel Processing Symposium*, pp. 433–438, April 1994.

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

[44] M. W. Mutka, "Using rate monotonic scheduling technology for real-time communications in a wormhole network," in *Proc. Workshop on Parallel and Distributed Real-Time Systems*, April 1994.

[45] J. Ngai and C. Seitz, "A framework for adaptive routing in multicomputer networks," in *Symposium on Parallel Algorithms and Architectures*, pp. 1–9, June 1989.

[46] L. Ni and P. McKinley, "A survey of wormhole routing techniques in direct networks," *IEEE Computer*, pp. 62–76, February 1993.

[47] L. M. Ni, "Should scalable parallel computers support efficient hardware multicast?," Technical Report MSU-CPS-ACS-107, Michigan State University, Lansing, Michigan, April 1995.

[48] A. G. Nowatzyk, M. C. Browne, E. J. Kelly, and M. Parkin, "S-Connect: from networks of workstations to supercomputer performance," in *Proc. Int'l Symposium on Computer Architecture*, pp. 71–82, June 1995.

[49] W. Oed, *The Cray Research Massively Parallel Processor System: Cray T3D*, November 1993.

[50] S. S. Owicki and A. R. Karlin, "Factors in the performance of the AN1 computer network," in *Proc. ACM SIGMETRICS*, pp. 167–180, June 1992.

[51] D. K. Panda, "Issues in designing efficient and practical algorithms for collective communication on wormhole-routed systems," in *Proc. of the 1995 ICPP Workshop on Challenges for Parallel Processing*, pp. 8 – 15, 1995.

[52] H. Park and D. P. Agrawal, "WICI: An efficient switching scheme for large scalable networks," in *Proc. IEEE Symposium on Parallel and Distributed Processing*, pp. 385 – 392, 1994.

[53] C. Peterson, J. Sutton, and P. Wiley, "iWarp: A 100-MOPS LIW microprocessor for multicomputers," *IEEE Micro*, pp. 26–29,81–87, June 1991.

[54] G. F. Pfister and V. A. Norton, "'Hot-spot' contention and combining in multistage interconnection networks," *IEEE Trans. Computers*, pp. 943 – 948, October 1985.

[55] P. Ramanathan, K. G. Shin, and R. W. Butler, "Fault-tolerant clock synchronization in distributed systems," *IEEE Computer*, pp. 33–42, October 1990.

[56] S. Ramany and D. Eager, "The interaction between virtual channel flow control and adaptive routing in wormhole networks," in *Proc. International Conference on Supercomputing*, pp. 136–145, July 1994.

[57] J. Rexford, J. Dolter, W. Feng, and K. G. Shin, "PP-MESS-SIM: A simulator for evaluating multicomputer interconnection networks," in *Proc. Simulation Symposium*, pp. 84–93, April 1995.

[58] J. Rexford, J. Dolter, and K. G. Shin, "Hardware support for controlled interaction of guaranteed and best-effort communication," in *Proc. Workshop on Parallel and Distributed Real-Time Systems*, pp. 188–193, April 1994.

[59] J. Rexford, J. Hall, and K. G. Shin, "A router architecture for real-time point-to-point networks," To appear in *Proc. Int'l Symposium on Computer Architecture*, May 1996.

[60] J. Rexford and K. G. Shin, "Support for multiple classes of traffic in multicomputer routers," in *Proc. Parallel Computer Routing and Communication Workshop*, pp. 116–130, May 1994.

[61] C. L. Seitz and W. Su, "A family of routing and communication chips based on the Mosaic," in *Symp. on Integrated Systems: Proc. of the Washington Conf.*, 1993.

[62] K. G. Shin, "HARTS: A distributed real-time architecture," *IEEE Computer*, vol. 24, no. 5, pp. 25–35, May 1991.

[63] D. Smitley, F. Hady, and D. Burns, "Hnet: A high-performance network evaluation testbed," Technical Report SRC-TR-91-049, Supercomputing Research Center, Institute for Defense Analyses, December 1991.

[64] P. Steenkiste, "A systematic approach to host interface design for high-speed networks," *IEEE Computer*, 1994.

[65] C. B. Stunkel, D. G. Shea, B. Abali, M. M. Denneau, P. H. Hochschild, D. J. Joseph, B. J. Nathanson, M. Tsao, and P. R. Varker, "Architecture and implementation of Vulcan," in *Proc. International Parallel Processing Symposium*, pp. 268–274, April 1994.

[66] Y. Tamir and G. Frazier, "Dynamically-allocated multi-queue buffers for VLSI communication switches," *IEEE Trans. Computers*, vol. 41, no. 6, pp. 725–737, June 1992.

[67] L. M. Thompson, "Using pSOS$^+$ for embedded real-time computing," in *Proc. COMPCON*, pp. 282–288, 1990.

[68] P. Thompson, "Concurrent interconnect for parallel systems," *The Computer Journal*, vol. 36, no. 8, pp. 778–784, 1993.

[69] K. Toda, K. Nishida, E. Takahashi, N. Michell, and Y. Yamaguchi, "Design and implementation of a priority forwarding router chip for real-time interconnection networks," *International Journal of Mini and Microcomputers*, vol. 17, no. 1, pp. 42–51, 1995.

[70] Anjan K. V. and T. M. Pinkston, "An efficient, fully adaptive deadlock recovery scheme: *DISHA*," in *Proc. Int'l Symposium on Computer Architecture*, pp. 201–210, June 1995.

[71] X. Zhang, "System effects of interprocessor communication latency in multicomputers," *IEEE Micro*, pp. 12–15,52–55, April 1991.