A Router Architecture for Flexible Routing and Switching in Multihop Point-To-Point Networks

Stuart W. Daniel, *Member*, *IEEE Computer Society*, Kang G. Shin, *Fellow*, *IEEE*, and Sang Kyun Yun, *Member*, *IEEE Computer Society*

Abstract—Modern parallel and distributed applications have a wide range of communication characteristics and performance requirements. These diverse characteristics affect the performance and suitability of particular *routing* and *switching* policies in multihop point-to-point networks. In this paper, we identify a core set of architectural features necessary for flexible selection and implementation of multiple routing and switching schemes. Using this, we present a *flexible router* whose routing and switching policies can be tailored to the application, allowing the network to meet these diverse needs. By dedicating a small programmable processor to each incoming link, we can implement wormhole, virtual cut-through, and packet switching, as well as hybrid switching schemes, each under a variety of unicast and multicast routing algorithms. In addition, a flexible router can support several applications or traffic types *simultaneously*, enabling better support of applications with multiple traffic classes. We have designed, implemented, and fabricated the *Programmable Routing Controller* (PRC). Cycle-level simulations of mesh-connected PRCs also demonstrate that flexible routing and switching can significantly enhance application performance.

Index Terms—Routers, cut-through switching, flexible routing and switching, switch architecture.

1 Introduction

Parallel and/or distributed applications typically employ a wide variety of communication paradigms that affect the volume and frequency of communication between nodes. Applications such as scientific computations, parallel databases, and real-time applications generate distinct distributions for packet lengths, interarrival times, and target destinations [1], [2], [3]. At the same time, applications also vary in their quality-of-service (QoS) requirements for communication: Most applications can benefit from low network latency, while others need predictable and/or high-bandwidth communication. Maximizing system performance, therefore, requires matching application characteristics and performance requirements with a suitable network design. This paper concentrates on the network policies used in message-passing multicomputers with a *point-to-point* (or direct) network topology. In a point-to-point network, every node has a direct connection with several other nodes; these nodes are then connected to others to form a network *topology*.

For point-to-point networks, application communication characteristics affect the performance of particular *routing* and *switching* schemes [4], [5], [6], [7], [8], [9]. The routing policy for a network determines the path taken by a packet between its source and destination, while the switching scheme impacts performance by determining how packets move through the intervening nodes. Numerous researchers have examined the relative performance of various routing and switching schemes and found that their relative performance varies according to the application's communication characteristics, such as the distribution of packet destinations and packet size [6], [7], [9], [10], [11]. *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* [12] and *wormhole* [13], try to forward an incoming packet directly to idle output links; if the link is busy, virtual cut-through routers buffer the packet, whereas a wormhole router stalls the packet in the network. Due to its lower communication latency, cut-through switching is preferred and used in most contemporary routers.

Most routers use *distributed* routing schemes that compute the route for a packet at every intervening node; this reduces the size of the packet header compared to *source*-routed schemes that must carry information about the entire route in the packet header. *Oblivious* routing schemes generate a single outgoing link for an incoming packet, whereas *adaptive* schemes can consider multiple links to take dynamic factor. Oblivious schemes are used by most wormhole routers due to their simplicity and also to simplify deadlock prevention. Adaptive routing algorithms generally improve performance by routing around congestion. Under certain traffic patterns, however, local decisions made by adaptive routing schemes may actually increase overall congestion by directing packets into congested regions [9]. In addition, the order in which adaptive schemes consider possible links is a major factor in determining the network latency under a particular traffic load [14]. Adaptive routing algorithms are either minimal or nonminimal. *Minimal* routing algorithms allow only shortest paths to be

- *S.W. Daniel is with Lexmark International, 740 New Circle Rd. NW, C19L/035-3, Lexington, KY 40550. E-mail: swdaniel@lexmark.com.*
- *K.G. Shin is with Real-Time Computing Laboratory, Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 481099-2122. E-mail: kgshin@eecs.umich.edu.*
- *S.K. Yun is with the Department of Computer Science, Seowon University, Cheongju, Chungbuk, 361-742, Korea. E-mail: skyun@dragon.seowon.ac.kr.*

Manuscript received 29 July 1996.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 100253.

used, while *nonminimal* routing algorithms can use nonminimal paths bypassing network congestion and faulty links. Adaptive routing algorithms can be classified as *partially adaptive* or *fully adaptive*. Partially adaptive routing algorithms use only a subset of the available physical paths between source and destination. By improving link throughput or increasing routing adaptivity, the routing algorithms can enhance communication performance, depending on the traffic pattern [9], [10], [11], [15], [16].

Many applications and systems can benefit from router support for *multicast* communication since sending a message to multiple destinations facilitates applications such as efficient barrier synchronization and global reduction operation and distributed shared memory [17], [18]. Wormhole-switched networks often employ *virtual channels* [19] for preventing deadlock and increasing network throughput, although oblivious and partially adaptive routing can also prevent deadlock. Several virtual channels share a single physical link by time multiplexing the virtual channels on the same physical link.

Since no single routing-switching combination performs best under all conditions, a router should support a range of policies. Most existing routers, however, only implement a single routing-switching scheme. Consequently, 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 for every application is impractical, we propose, implement, and evaluate a flexible router that allows network policies to be tuned to these diverse characteristics.

In this paper, we present a flexible router architecture that implements a variety of routing and switching schemes by dedicating a microprogrammable routing engine to each incoming link. In particular, we focus on the low-level routing and switching policies implemented in the router. In the next section, we overview the architecture of a flexible router. Section 3 describes the Programmable Routing Controller (PRC) implemented on a single chip, and Section 4 presents the PRC microarchitecture. Section 5 evaluates an example of flexible routing using the PRC to demonstrate the benefits of tailoring routing-switching policies to application characteristics [4], [8], [9], [20], [21], [22]. The paper concludes with Section 6.

2 Router Architectures

Contemporary routers typically implement only a single, fixed policy for routing and switching packets. The torus router [13] was the first wormhole router that implemented oblivious deadlock-free wormhole routing in a *k*-ary *n*-cube system in which each physical channel was time-multiplexed by two virtual channels. The adaptive virtual cut-through router treated in [23] is a variation of the torus router that supports a virtual cut-through switching without using any virtual channel. The router buffers blocked packets at the local node and the routing scheme is "adaptive minimal" with dimension-order selection. The CHAOS router [24] is another router using adaptive virtual cut-through switching. It provides a small *multiqueue* on-chip buffer; if this buffer fills up, packets are derouted through any available link.

These routers implement only a single, fixed (thus, inflexible) routing and switching policy, although they achieve small routing latencies with hard-wired routing engines. Moreover, most of the routing algorithms proposed and evaluated thus far have not been implemented / fabricated. It is therefore important and interesting to design, implement, and evaluate a router architecture for programmable (thus, flexible) routing and switching. In this section, we present a router architecture satisfying this requirement and its implementation.

The flexible router has the following main objectives:

1) simultaneous support of wormhole, virtual cut-through, and packet switching schemes,
2) development of a programmable routing engine for processing incoming packet headers, and
3) reduction of the complexity of higher-level protocols.

It also supports virtual channels. Its programmable feature permits multicomputer networks to handle a wide variety of header formats, routing algorithms, switching schemes at a reasonable cost. Note, however, that this flexibility comes at the expense of slightly longer routing latencies than pure hardwired routers.

The architecture of this flexible router is depicted in Fig. 1. The router consists of a single-chip programmable routing controller (PRC) and external buffer memory. The PRC provides four bidirectional (physical) links and three virtual channels per each physical link. The receiver module of each physical link has a *separate* routing engine in order to compensate for the larger routing latency of the programmable routing engine. A routing engine is shared by several virtual channels on a single physical link. For data switch among input links, output links, and host, a 32-bit time multiplexed bus is used instead of a crossbar switch. External buffer memory is for buffering input, output, and forwarding data. Data transfer between host and buffer memory is performed at a page level to reduce software complexity, and the PRC contains the hardware logic for the page-level data transfer.

3 The PRC Architecture

This section describes the *Programmable Routing Controller* (PRC), which is the main component of the flexible router and manages bidirectional communication with four other nodes, with three virtual channels on each unidirectional link. It has a high-performance, low-overhead interface, while implementing the switch as a high-speed, time-multiplexed bus [4], [25], [26]. Fabricated in the HP CMOS14 process, the PRC provides a single-chip solution for flexible routing in distributed and parallel networks.

The PRC architecture is shown on the right side of Fig. 1. The network interface of the PRC manages bidirectional communication with four other nodes and consists of four receiver modules, four transmitter modules, and cut-through bus (CTBUS). A transmitter module controls an

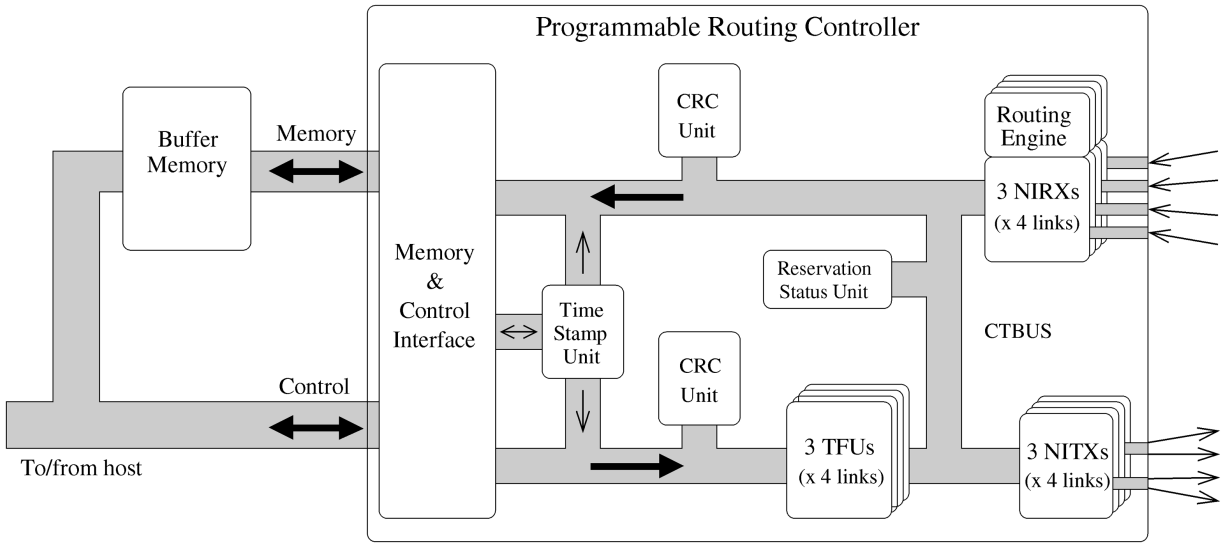

Fig. 1. Flexible router architecture.

outgoing link and contains three *network interface transmitters* (NITXs), each implementing a single transmission virtual channel. Similarly, a receiver module controls an incoming link and contains a *programmable* routing engine and three *network interface receivers* (NIRXs), each implementing a single incoming virtual channel. Data is transferred between the host interface, the NITXs, and the NIRXs via the *cut-through bus* (CTBUS), which is a 32-bit high-speed time-multiplexed bus. The host interface of the PRC provides support for reducing the complexity of both the hardware and software protocols and consists of memory interface, control interface, and transmitter fetch unit (TFU). A memory interface is used to access the buffer memory and the host accesses the PRC through a control interface to initiate packet transmission and control packet reception. TFU controls the transmission of packets from the buffer memory to the network interface.

3.1 Host Interface

The host interface of PRC provides a flexible scheme for controlling the simultaneous transmission and reception of multiple variable-length packets. It also transfers packet data between the external buffer memory and the network interface. Since the PRC does not include internal buffers for blocked packets, packets that buffer at intermediate nodes are stored in this buffer memory.

To reduce software protocol complexity, the host interface of the PRC interacts with the host 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 1,024-byte pages; larger pages allow the PRC to operate longer without host intervention. 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, avoiding unnecessary data copying for attaching and removing packet headers. Fig. 2 depicts the major components of the PRC's host interface.

A packet is transmitted in the following steps: The packet is transferred from the memory in the host to buffer memory by DMA and the packet header is constructed in a separate page of buffer memory by direct write of the host. The host then writes the page tags, specifying the address and size of pages to be transferred to the transmission page queue of the appropriate TFU. Each TFU is associated with a particular NITX. After the TFU reserved the NITX, the memory interface fetches the data from each page, one 32-bit word at a time, and transfers to the NITX. When the packet begins to arrive at an NIRX in the network interface of other nodes, it is directly forwarded to the reserved NITX or is buffered at the local node according to its destination address and switching scheme. Once the packet is buffered at the local node, the NIRX simply forwards the incoming words to buffer memory. The host supplies reception page queues with pointers to free page tags in the buffer memory for use by arriving packets. Each reception page queue is associated with a particular NIRX.

The transmission and reception data paths incorporate transparent error detection via cyclic redundancy code (CRC) generation and checking, as well as packet time-stamping useful for controlling clock drift between nodes. 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.

3.2 Network Interface

While the host manages communication at the page level, the PRC coordinates the fine-grain interaction between incoming and outbound channels. Each outbound channel is controlled by an NITX. The PRC treats the NITXs as individually reservable resources. To transmit a packet through an outgoing channel, the *master* (either a reception channel or the host interface) must first reserve the NITX channel. Once the desired NITX has been reserved, the master may forward packet data to the NITX through the switch. By providing *programmable* control over this

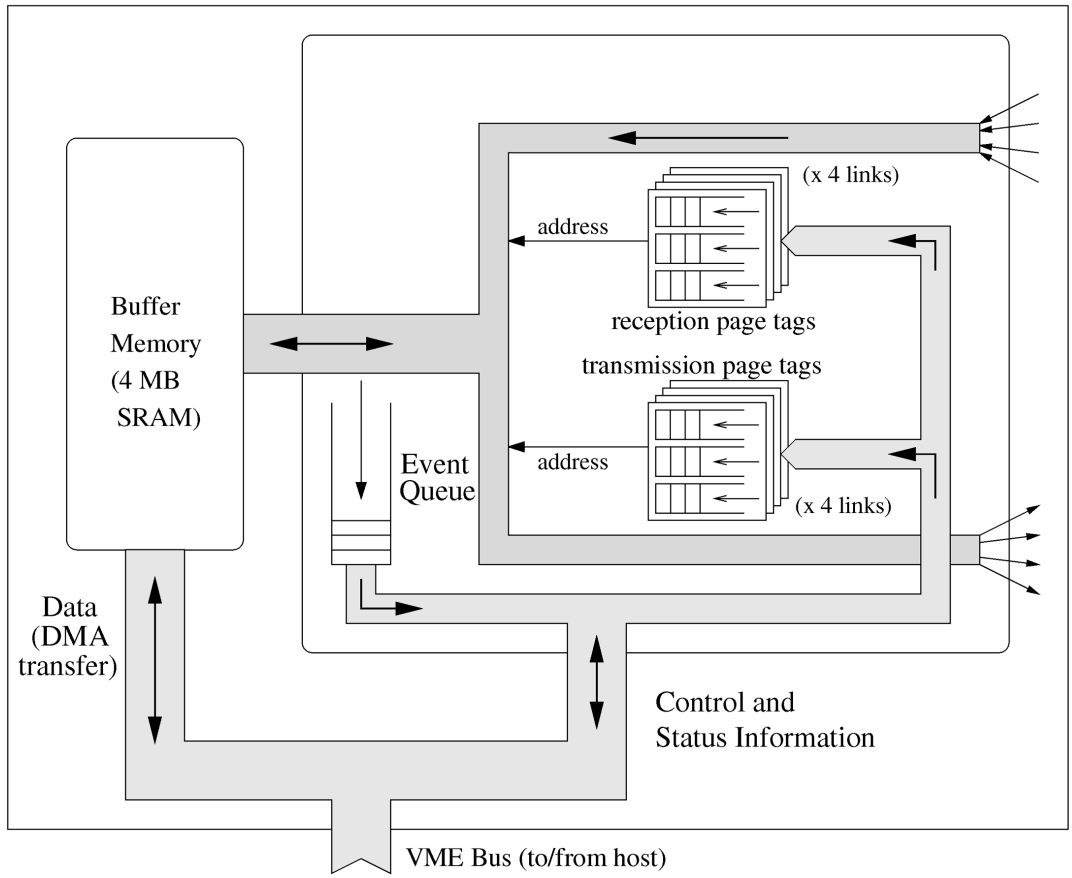

Fig. 2. PRC host interface.

reservation process for every packet entering the router, different routing and switching schemes may be implemented.

Packets are injected by the local host or from a neighboring node by an incoming channel. For packets injected through the host interface, the routing policies can be implemented by the host. When the packet is received from the neighboring node, the routing engine of the incoming link is responsible for determining the routing policy.

Each incoming channel is controlled by an NIRX. To provide flexibility in packet routing at the receivers, a programmable routing engine is necessary—it offers maximum flexibility for header parsing and route computation. In addition, expressing routing and switching schemes as assembly language instructions simplifies programming new routing schemes. This programmability of the routing engine is usually achieved at the expense of a longer routing latency than a hardwired router. In order to reduce this degradation of routing latency, each physical incoming link uses a separate routing engine which is shared by virtual channels running on the link. Since a physical link itself serializes the arrival of packet headers, the arbitration scheme for accessing the routing engine does not have to consider simultaneous packet arrivals.

To share the routing engine efficiently among several virtual channels, time-consuming functions are implemented on the incoming channels. An incoming channel is responsible for forwarding the entire packet and waiting for transmission channels when necessary. A routing engine reserves a transmission channel or determines the channels for which the packet should stall by header parsing and route computation.

Initially, the incoming channel forwards incoming data to the routing engine. This process continues until the routing engine returns a routing primitive to the incoming channel. After the arrival of the routing primitive, if a transmission channel is reserved, the incoming channel begins forwarding the body of the packet until the last flit of the packet arrives. If a transmission channel is not reserved, the incoming channel continues to check whether the selected channel is free; when a selected channel is found to be free, it reserves that channel and begins forwarding the packet (thus, without going through the routing engine). If multicast semantics are triggered in the routing primitive, it attempts a reservation only when all selected channels are available.

3.3 Switch Architecture

The network interface components of the PRC communicate over the CTBUS, a 32-bit time-division multiplexed bus. The CTBUS operates at twice the byte-transmission speed of the internode links (in our current implementation, the clock speed for PRC's internal operations is 40MHz and that for internode link is 20MHz), matching the bandwidth of the eight unidirectional links. The CTBUS implements two primary functions within the PRC: data transfer and channel allocation. As shown in Table 1, the CTBUS protocol includes commands to transfer data and

TABLE 1
CTBUS COMMAND SET

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

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. 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 [17], [27].

Access to the CTBUS, which is controlled by a demand-slotted binary priority-tree arbiter that allocates bandwidth fairly among the active devices [20], also implicitly determines the PRC's allocation policy for reserving outbound channels. Any master needing to reserve an outbound channel simply checks to see if the outbound channel is free; if so, it requests access to the CTBUS and issues an RESV command when access is granted. However, since CTBUS access is pipelined to minimize the cycle time, another master may have reserved the outbound channel in the meantime. This is handled by the reservation status unit—upon receiving an RESV command, the reservation status unit determines if the requested NITXs are available. If all of the selected NITXs are free, they are marked as reserved. Although all CTBUS devices have concurrent access to the NITX's 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 slave NITXs 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, the host can configure the PRC to establish connections that outlive individual packets.

The HOLD command provides a simple mechanism for overriding the CTBUS access arbitration when allocating outbound channels; once a HOLD command has been issued for a outbound channel, all subsequent RESV commands will fail. Since the HOLD may be issued while the outbound channel 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.

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

Input/output ports of the PRC are compatible with parallel ports of AMD TAXI (Transparent Asynchronous Xmitter-Receiver Interface) chips [28], which provide the means to establish a transparent high speed serial link between two high-performance parallel buses. The physical internode links may be implemented as either parallel or serial connections although input/output ports of the PRC are parallel. For parallel interconnects, each PRC's output ports are directly connected to the input ports of its neighbors. For serial communication over a longer interconnect through coaxial or fiber-optic media, each PRC's output ports are connected to the input ports of its neighbors through TAXI transmitter and receiver chips.

Ten bits of information, including a tag, are transmitted every two cycles and, thus, transferring a word of data consumes eight cycles. Each of the physical outgoing ports is shared or time-multiplexed by two outgoing links because of pin-limitation of the chip and can transmit data every cycle. Note that the cycle time of internode link transfers is twice that in internal operations. The extra bits tag the data bytes with control information indicating the virtual channel and CTBUS command associated with the data. In addition, this control information also carries flow-control acknowledgments for the reverse channel.

3.4 PRC Status

The PRC has been fully designed using the HP CMOS14 process and Epoch design tools from Cascade Design Automation and was packaged by MOSIS. The physical and timing specifications of the PRC are shown in Table 2. As shown in Fig. 3, 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. In addition, the network interface's design allows the transmission ports to directly connect to the reception ports. This allows a single PRC to transmit the packets it receives, greatly simplifying the external circuitry required for testing.

Verilog simulations were used to test a single PRC, with the outgoing links connected to the reception ports, under random and contrived workloads. The fabricated PRC has been tested with an HP 82000 tester using scan chains to access the chip's critical state machines and the bus arbitration logic. The PRC is found to function in complete compliance with the design specification.

4 ROUTING ENGINE

While the incoming channel is responsible for handling data transfer, flow control, and executing some switching functions, the routing engine performs the header parsing and route computation for every incoming packet by using its programmable feature. After giving an overview of the major components of the routing engine, we will examine how the routing engine implements each of its major functions.

TABLE 2
PRC SPECIFICATIONS

Parameter	Value
Size	9.0×7.3
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

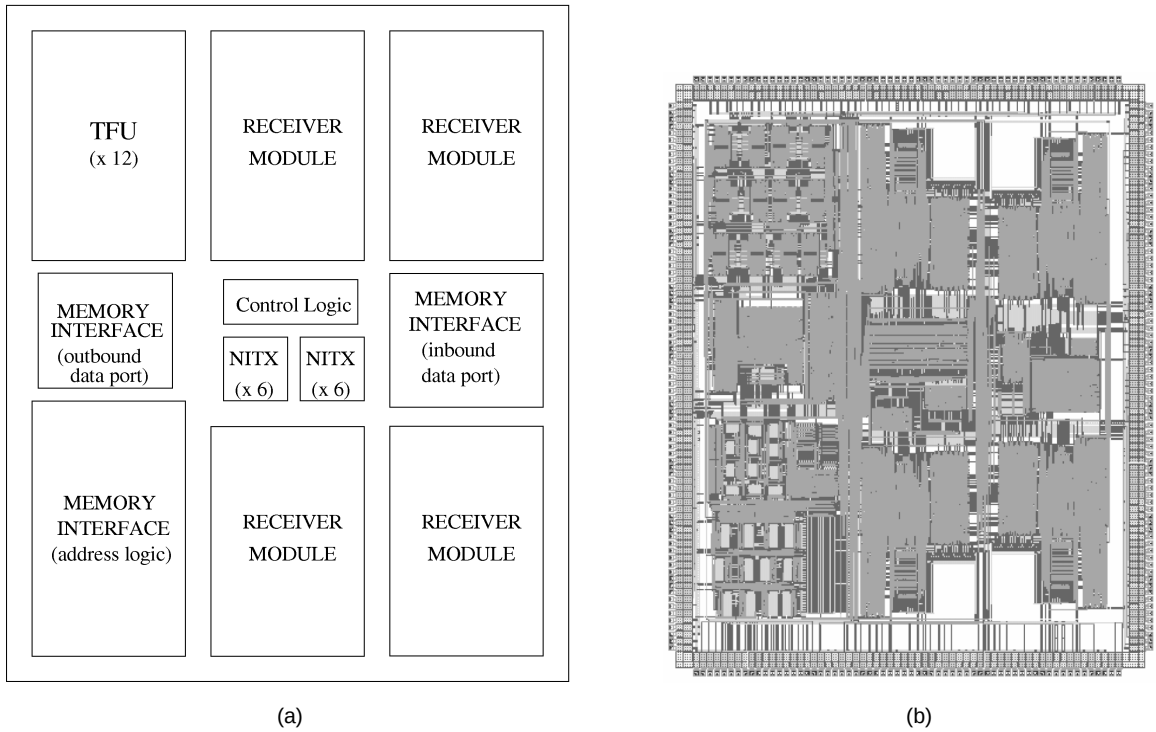

Fig. 3. Floorplan of the PRC. (a) PRC floorplan, (b) PRC layout.

4.1 Architectural Overview

The routing engine is a small, 8-bit microcontroller with 256-instruction control store for microprograms, as shown in Fig. 4. Each instruction executes in a single cycle unless a branch is required. 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 runtime through the notification FIFOs. The routing engine interacts with the NIRXs and the CTBUS through the data input module and data output module. The header word in NIRX is transferred to the network interface data register (**`nid`**) in the data input module. The microprogram manipulates and updates the routing header, and possibly reserves outbound channels. Then, the routing engine returns controls to the NIRX by sending the routing primitive through the registers in the data output modules, as shown in Table 3. The switch interface, on the other hand, allows the routing engine to read the current reservation status of the outbound channels without accessing the switch.

4.2 Instruction Set

The routing engine implements instructions for manipulating header data, controlling external interface, and branching based on internal conditions, as shown in Table 4. The routing engine employs **`alu`** instructions to parse and modify a packet header. The **`ldc`** instruction loads constants, such as predetermined address masks and control tags, into registers, while the **`xfer`** instruction copies data between these registers. The **`ldc`** and **`xfer`** instruction can issue routing primitives or CTBUS commands as side effects if it is used with **`go rtp`** or **`go ctbus`** suffix.

The support for control flow is provided by the `jump` instruction, which can react to flags such as ALU's **`zero`** and **`carry`** bits, as well as reservation status flags for 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 **`link`** qualifier, the routing engine stores the address of the next microinstruction, so the microsequencer can **`return`** to the main

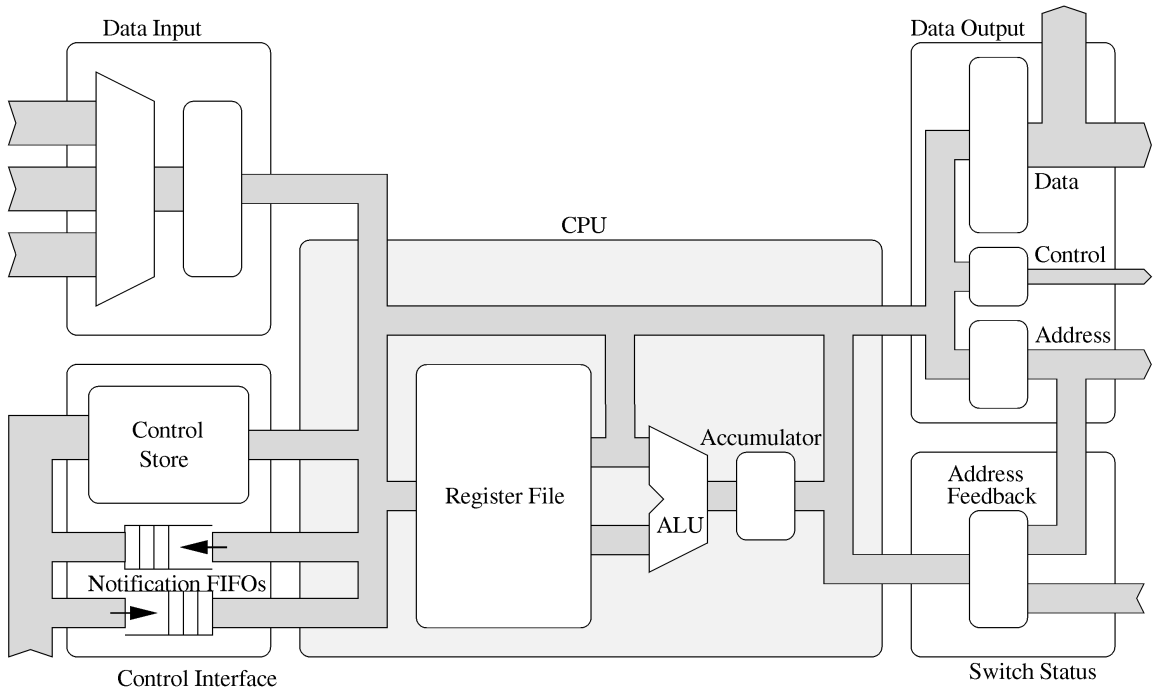

Fig. 4. Routing engine architecture.

TABLE 3
ROUTING PRIMITIVE AS STORED IN REGISTERS

Registers	Function
`ctd3-ctd0`	next word to transmit
`ctaddr1, ctaddr0`	selects host, NITXs
`ctctl`	Boolean control flags
resvd	slaves already reserved
all	reserve all of slaves
crcflg	include word in CRC

TABLE 4
ROUTING ENGINE INSTRUCTION SET

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
`wait`	three-way, blocking branch
`go rtp`	trigger routing primitive
`go ctbus`	trigger CTBUS access

instruction flow later; this restricted implementation of subroutines reduces microprogram size by enabling code reuse.

The special **`wait`** instruction controls the data input module. This instruction blocks until a header word arrives at one of incoming channels. Then it transfers data from the incoming channel to the **`nid`** registers and begins executing the appropriate microcode for the selected incoming channel.

4.3 Routing and Switching Microprogram

The PRC has programmable routing engines and can implement various routing and switching schemes. Fig. 5 shows a typical microprogram structure.

4.3.1 Offset-Based Routing

Distributed routing schemes often employ offset-based addressing. Fig. 6 shows an example of parsing a packet header in offset-based routing. This program uses an adaptive, diagonal-biased routing scheme, e.g., the routing engine tries to reduce the offset with the largest magnitude first. In this way, packets will have multiple minimal-length paths to their destination as long as possible. The example program uses the ALU to examine the signs of the offsets, and to compare their magnitudes. Based on this information, the routing engine can jump to the proper code for routing the packet.

```
1    init:
2      ldc c0_handler, trap0;
3      ldc c1_handler, trap1;
4    /* waits for a packet header */
5      wait c2, trap0(c1), trap1(c0);
6    c2_handler:
7      < header parsing for channel 2 >
8      < route computation >
9      < channel reservation >
10     ldc rtp_flags, ctctl, go rtp;
11     jump true, init;
12   c1_handler:
13     < header parsing for channel 1 >
14     < route computation >
15     < channel reservation >
16     ldc rtp_flags, ctctl, go rtp;
17     jump true, init;
18   c0_handler:
19     < header parsing for channel 0 >
20     < route computation >
21     < channel reservation >
22     ldc rtp_flags, ctctl, go rtp;
23     jump true, init;
```

Fig. 5. Typical microprogram structure.

```
1    handler:
2        wait ni2, trap0(ni1), trap1(ni0);
3        xfer nid1, ctd1;
4    /* check x offset */
5        alu nid2 + 1;
6        xfer acc, ctd2;
7        jump zero, y_only;
8    /* check y offset */
9        alu nid1;
10       jump zero, x_only;
11       alu nid1 & reg0;
12       jump zero, y_is_neg;
13   /* compare magnitudes; x<0,y>0 */
14       xfer nid1,reg1;
15       alu nid2 + reg1;
16       alu acc & reg0; /* check sign */
17       jump zero, y_first;
```

Fig. 6. Header parsing example.

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, e.g., offset-based addressing, such as in the Appendix, 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.

4.3.2 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 and the table entry instructs the routing engine on which link(s) to forward the packet. Rather than providing a separate memory for the routing table, however, the routing engine may use an indirect jump into a "table" stored in its control store. Each table entry then directs the routing engine to the appropriate routine for routing the packet. This approach has several advantages:

1) The existing control store may be used for the table, avoiding the cost of a separate RAM;
2) Having the table entries call a routing subroutine permits great flexibility in specifying channel orderings.

4.3.3 Multicast Routing

The routing engine allows implementation of a broad spectrum of multicast routing algorithms under both wormhole and virtual cut-through switching. Fig. 7 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 Fig. 7a. 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 slave devices before forwarding the packet, whereas the virtual cut-through scheme immediately directs the incoming packet to any *available* slaves.

Register	Function
nid3	Node identifier
nid2	Switching scheme 1 = wormhole
nid1	7-bit address bit mask {host, $O_{32}..O_{20}$}
nid0	6-bit address bit mask {$O_{12}..O_{00}$}

(a)

```
1    init:
2      < initialization code >
3      ldc our_node_id, reg2;
4
5    strip_header:
6      wait ni2, trap0(ni1), trap1(ni0);
7      alu nid3 - reg2;
8      jump ~zero, strip_header;
9
10   route_packet:
11     alu nid2;
12     jump ~zero, wh_routing;
13     < code for virtual cut-through >
14   wh_routing:
15     < code for wormhole >
```

(b)

Fig. 7. Example of parsing a multicast source-list header. (a) Header format, (b) pseudocode implementation.

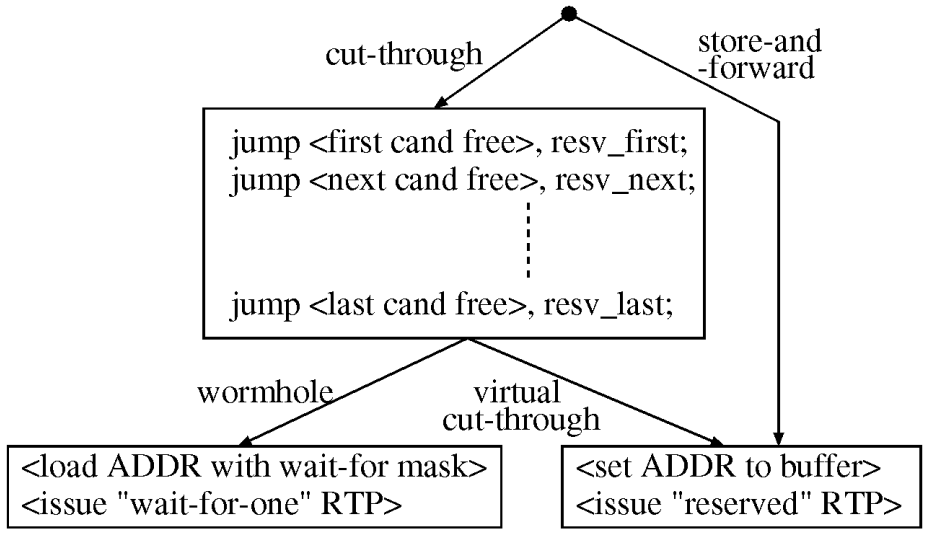

Fig. 8. Template for a switching operation.

4.3.4 Switching

Once the routing engine has committed to a routing decision for a packet, it needs to either reserve one or more outbound channels, or return control to the incoming channel (for buffering or a stall). During this stage, the reservation status flags are typically used to trigger appropriate microcode branches. Fig. 8 shows a template for the switching operation, where the microprogram checks candidate outbound channels in descending order of priority; if no candidate is available, the switching scheme determines the form of routing primitive issued to the incoming channel.

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. 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.

5 An Example of Flexible Routing

The PRC's flexibility can support a variety of routing and switching schemes. In particular, it can implement hybrid switching [22], [29], which dynamically combines both virtual cut-through and wormhole switching to provide higher achievable throughput than wormhole or virtual cut-through alone, and flexible routing which tailors routing schemes to the underlying application traffic pattern. Applications are known to generate bimodal traffic loads with diverse quality of service (QoS) requirements, such as a mixture of short control packets and long data packets, or a mixture of real-time and best-effort packets. Flexible routing in the PRC enables selection of routing and switching schemes on a per-packet or per-channel basis to adapt its performance to the traffic load and the application QoS requirement.

To evaluate the PRC architecture in a variety of network configurations, we have developed a cycle-level simulation model that captures the details of flow control, resource

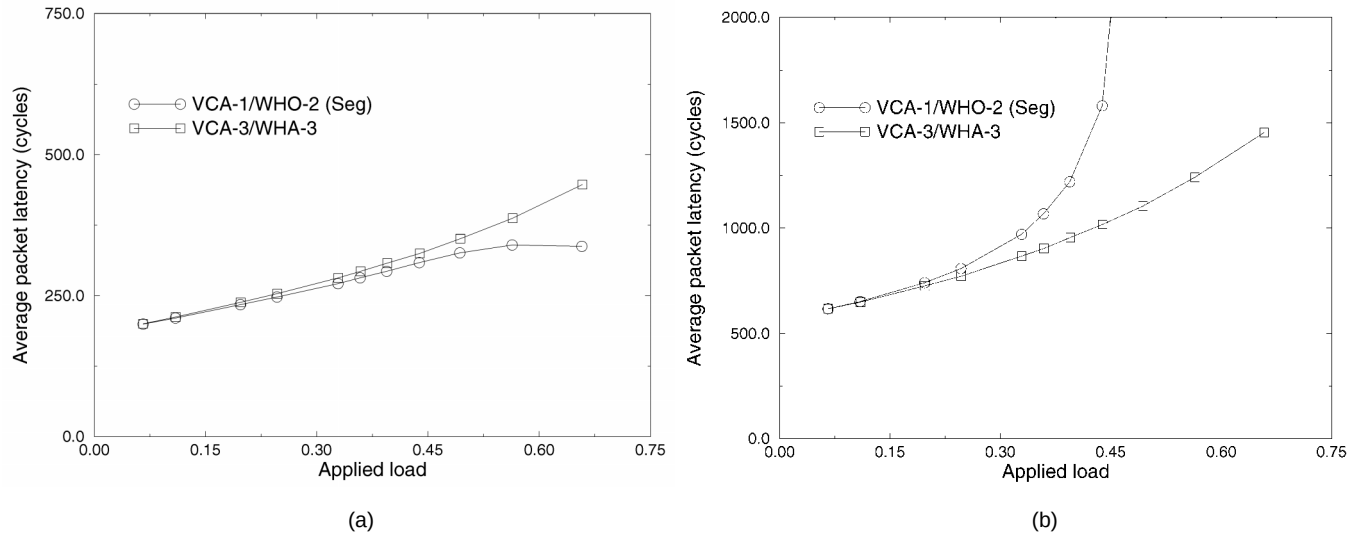

Fig. 9. Comparison of Segment router traffic partitioning with VCA-3/WHA-3. (a) 64-byte packets, (b) 256-byte packets.

arbitration, and microcode execution. Implemented in the **pp-mess-sim** simulation environment [20], [30], [31], this cycle-level model captures the details of flow control, resource arbitration and microcode execution. We have employed **pp-mess-sim** to study the PRC under a wide range of network topologies, application workloads, and routing-switching schemes [8], [9], [20], [21], [22], [29]. Most of these studies have focused on the benefits of hybrid switching and flexible routing, but this section presents an example showing how applications can benefit from tailored routing schemes.

Specifically, we evaluate 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 [2], [6], [7]. Several researchers have proposed methods for handling these bimodal traffic patterns. Kim and Chien [6] proposed adding virtual channels and adaptivity to eliminate the interference of long packets on short ones in wormhole-switched networks. Konstantinidou [7], as part of his Segment Router Architecture, proposed dividing the network into two distinct virtual networks, with a distinct switching scheme 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 longer packets, without requiring large buffers for storing blocked data packets. Thanks to the PRC's flexibility, one can partition traffic into a wide range of (instead of just two) traffic classes.

Konstantinidou's Segment Router improves the performance of short packets by segregating them onto a separate virtual network using adaptive virtual cut-through, but long packets still use oblivious wormhole routing [7]. Thus, short packets use adaptive virtual cut-through on one channel (VCA-1), while long packets use oblivious wormhole routing on other two channels (WHO-2) in the Segment Router. This partitioned VCA-1/WHO-2 scheme can also tailor routing and switching on a per-channel basis or per-message basis in the PRC.

A possible way of improving the overall system performance would be to differentiate the switching schemes used by different traffic classes without segregating them onto different virtual networks. For example, we can have short packets employ adaptive virtual cut-through on all three channels (VCA-3), while having long packets use adaptive wormhole routing on all three channels (WHA-3) to reduce the buffer requirement. This VCA-3/WHA-3 scheme can tailor routing and switching on a per-message basis in the PRC.

Let's investigate how these two schemes using PRC impact performance for both traffic classes. The experiments are done on 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 percent of the traffic load. Each node generates traffic independently, with uniform random selection of destinations and exponentially-distributed interarrival times.

Fig. 9 shows the relative performance of the VCA-3/WHA-3 scheme to the Segment Router scheme—the average latency for short packets is only slightly higher, even under heavy loads. Longer packets, on the other hand, benefit significantly from the increased routing adaptivity and the extra virtual channel. Thus, given the major constraint of the PRC architecture—three virtual channels per link—the VCA-3/WHA-3 scheme outperforms the partitioned VCA-1/WHA-2 scheme.

6 Conclusion

We have examined how network performance may be improved through hardware support for multiple routing and switching schemes that may be selected on a per-packet or per-channel basis. We demonstrated several key benefits of a flexible router architecture. First, no routing and/or switching scheme performs best for all workloads—a flexible router architecture allows the router policies to be tailored to the

underlying application needs. Second, since applications are known to generate bimodal workloads with diverse quality of service requirements, traffic partitioning combined with flexible policy selection is crucial to meeting the needs of all applications. Unlike the PRC, most existing routers do not offer this capability.

To this end, 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. We decomposed the general problem of packet routing into several steps: header parsing, route computation and selection, and switching. 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. Finally, we have designed, implemented and fabricated the PRC to further explore and utilize this flexible architecture.

```
1    x = x + 1
2    if ((x != 0) && (y > 0)) then
3        if (abs(x) > abs(y)) then
4            route (+x, -y)
5        else
6            route (-y, +x)
7    else if ((x != 0) && (y < 0)) then
8        if (abs(x) > abs(y)) then
9            route (+x, +y)
10       else
11           route (+y, +x)
12   else if ((x != 0) && (y == 0)) then
13       route (+x)
14   else if ((x == 0) && (y > 0)) then
15       route (-y)
16   else if ((x == 0) && (y < 0)) then
17       route (+y)
18   else /* x == 0 && y == 0 */
19       buffer;
```

Fig. 10. Pseudocode implementation of diagonal-biased minimal-path adaptive routing algorithm.

APPENDIX
ROUTING-SWITCHING MICROPROGRAMS

The 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 microprogram in this section is written for receiver module #2, which receives packets traveling in the $+x$ direction.

Fig. 11 shows an example of an adaptive, minimal-path routing algorithm for regular networks, such as the square mesh. 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 an adaptive, diagonal-biased routing scheme, e.g., the routing engine tries to reduce the offset with the largest magnitude first. A pseudocode implementation of the adaptive algorithm is shown in Fig. 10.

Fig. 11 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 seven instructions in eight 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.

A1 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 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.

A2 Header Parsing and Route Computation

The next step is to parse the packet's routing header using the algorithm in Fig. 10. The handler first increments the header's x offset (received in **nid2**) 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 (in **nid1**). If both x and y are zero, they are checked to see which has the larger magnitude; this determines which dimension to access first.

A3 Switching

Once the routing engine has determined which channels the packet can use, it begins checking which of these channels are free. The order in which the channels are checked indicates their priority—e.g., high channels are generally preferred to low channels. 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.

A4 Packet Buffering

The **buffer_packet** routine configures the NIRX to write to the memory interface. The routing engine sets the address

```
1   init:
2       ldc 0x80, reg0;
3       ldc ctcmd_resv, ctctl;
4       ldc dimorder, trap1;
5       ldc adaptive, trap0;
6   /* now wait for a packet header */
7   handler:
8       wait ni2, trap0(ni1), trap1(ni0);
9       xfer nid1, ctd1;
10
11  /* check x offset */
12      alu nid2 + 1;
13      xfer acc, ctd2;
14      jump zero, y_only;
15  /* check y offset */
16      alu nid1;
17      jump zero, x_only;
18      alu nid1 & reg0;
19      jump zero, y_is_neg;
20  /* compare magnitudes; x<0,y>0 */
21      xfer nid1,reg1;
22      alu nid2 + reg1;
23      alu acc & reg0; /* check sign */
24      jump zero, y_first;
25
26  /* check +x link */
27      jump ~resvdl0c2, get_l0c2, link;
28      jump ~resvdl0c1, get_l0c1, link;
29  /* check -y link */
30      jump ~resvdl3c2, get_l3c2, link;
31      jump ~resvdl3c1, get_l3c1, link;
32  /* check low channels */
33      jump ~resvdl3c0, get_l3c0, link;
34      jump ~resvdl0c0, get_l0c0, link;
35  block_l3l0:/* block on +x, -y links */
36      ldc ctaddr_l0, ctaddr0;
37      ldc ctaddr_l3, ctaddr1;
38      ldc rtp_wait_one, ctctl, go rtp;
39      jump true, init;
40
41  y_first:
42  /* check -y link */
43      jump ~resvdl3c2, get_l3c2, link;
44      jump ~resvdl3c1, get_l3c1, link;
45  /* check +x link */
46      jump ~resvdl0c2, get_l0c2, link;
47      jump ~resvdl0c1, get_l0c1, link;
48  /* check low channels */
49      jump ~resvdl3c0, get_l3c0, link;
50      jump ~resvdl0c0, get_l0c0, link;
51      jump true, block_l3l0;
52  get_l0c0:
53      ldc 0x0, ctaddr1, go ctbus;
54      ldc ctaddr_l0c0, ctaddr0, go ctbus;
55      return ~ack;
56      ldc rtp_resvd_nocrc, ctctl, go rtp;
57      jump true, init;
58
59  buffer_packet:
60      ldc 0x0, ctaddr0;
61      ldc  ctaddr_buff, ctaddr1;
62      ldc  rtp_resvd_nocrc, ctctl, go rtp;
63      jump true, init;
64
65  x_only:
66      jump ~resvdl0c2, get_l0c2, link;
67      jump ~resvdl0c1, get_l0c1, link;
68      jump ~resvdl0c0, get_l0c0, link;
69      ldc ctaddr_l0, ctaddr0; /* block l0 */
70      ldc rtp_wait_one, ctctl, go rtp;
71      jump true, init;
72
73  y_only:
74      alu nid1;
75      jump zero, buffer_packet;
76      alu nid1 & reg0;
77      jump zero, neg_y_only;
78  /* check +y link */
79      jump ~resvdl1c2, get_l1c2, link;
80      jump ~resvdl1c1, get_l1c1, link;
81      jump ~resvdl1c0, get_l1c0, link;
82      ldc ctaddr_l1, ctaddr0; /* block l1 */
83      ldc rtp_wait_one, ctctl, go rtp;
84      jump true, init;
85
86  dimorder:
87      xfer nid1, ctd1;
88      alu nid2 + 1;
89      xfer acc, ctd2;
90      jump zero, dim_y;
91  dim_x:
92      ldc ctaddr_l0c0, ctaddr0;
93      ldc rtp_wait_one, ctctl, go rtp;
94      jump true, init;
95  dim_y:
96      alu nid1;
97      jump zero, buffer_packet;
98      alu nid1 & reg0;
99      jump ~zero, dim_y_neg;
100     ldc ctaddr_l1c0, ctaddr0;
101     ldc rtp_wait_one, ctctl, go rtp;
102     jump true, init;
```

Fig. 11. Minimal-path adaptive routing example.

mask to buffer the packet, and loads the `ctctl` field of the routing primitive. The `go rtp` 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.

A5 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`**.

ACKNOWLEDGMENTS

The work reported in this paper was supported in part by the U.S. National Science Foundation under grant MIP–9203895 and by the KOSEF under the postdoctoral fellowship. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the U.S. National Science Foundation. We would like to thank James Dolter (one of the original PRC architects) and Jennifer Rexford for their invaluable contributions to this work, and Sung-Whan Moon for testing the fabricated PRC.

REFERENCES

[1] 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, pp. 451–464, July 1992.

[2] R. Cypher, A. Ho, S. Konstantinidou, and P. Messina, "Architectural Requirements of Parallel Scientific Applications with Explicit Communication," *Proc. Int'l Symp. Computer Architecture*, pp. 2–13, May 1993.

[3] D. Ferrari, "Client Requirements for Real-Time Communication Services," *IEEE Comm. Magazine*, pp. 65–72, Nov. 1990.

[4] S.W. Daniel, "Flexible Router Architectures for Point-to-Point Networks," PhD thesis, Univ. of Michigan, May 1996.

[5] 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, Md., Mar. 1993.

[6] J.H. Kim and A.A. Chien, "Evaluation of Wormhole Routed Networks under Hybrid Traffic Loads," *Proc. Hawaii Int'l Conf. System Sciences*, pp. 276–285, Jan. 1993.

[7] S. Konstantinidou, "Segment Router: A Novel Router Design for Parallel Computers," *Proc. Symp. Parallel Algorithms and Architectures*, June 1994.

[8] J. Rexford, J. Dolter, and K.G. Shin, "Hardware Support for Controlled Interaction of Guaranteed and Best-Effort Communication," *Proc. Workshop Parallel and Distributed Real-Time Systems*, pp. 188–193, Apr. 1994.

[9] 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 Eng. Technical Report CSE-TR-239-95, Univ. of Michigan, May 1995.

[10] R. Boppana and S. Chalasani, "A Comparison of Adaptive Wormhole Routing Algorithms," *Proc. Int'l Symp. Computer Architecture*, pp. 351–360, 1993.

[11] 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, Apr. 1993.

[12] P. Kermani and L. Kleinrock, "Virtual Cut-Through: A New Computer Communication Switching Technique," *Computer Networks*, vol. 3, pp. 267–286, Sept. 1979.

[13] W.J. Dally and C.L. Seitz, "The Torus Routing Chip," *J. Distributed Computing*, vol. 1, no. 3, pp. 187–196, 1986.

[14] W. Feng and K. Shin, "Impact of Selection Functions on Routing Algorithm Performance in Multicomputer Networks," Technical Report CSE-TR-287-96, Univ. of Michigan, Mar. 1996.

[15] S. Ramany and D. Eager, "The Interaction Between Virtual Channel Flow Control and Adaptive Routing in Wormhole Networks," *Proc. Int'l Conf. Supercomputing*, pp. 136–145, July 1994.

[16] S. Konstantinidou and L. Snyder, "The Chaos Router," *IEEE Trans. Computers*, vol. 43, no. 12, pp. 1,386–1,397, Dec. 1994.

[17] D.D. Kandlur and K.G. Shin, "Reliable Broadcast Algorithms for HARTS," *ACM Trans. Computer Systems*, vol. 9, pp. 374–398, Nov. 1991.

[18] L.M. Ni, "Should Scalable Parallel Computers Support Efficient Hardware Multicast?" Technical Report MSU-CPS-ACS-107, Michigan State Univ., Lansing, Mich., Apr. 1995.

[19] W.J. Dally and C.L. Seitz, "Deadlock-Free Message Routing in Multiprocessor Interconnection Networks," *IEEE Trans. Computers*, vol. 36, no. 5, pp. 547–553, May 1987.

[20] J. Dolter, "A Programmable Routing Controller Supporting Multi-Mode Routing and Switching in Distributed Real-Time Systems," PhD thesis, Univ. of Michigan, Sept. 1993.

[21] 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, Univ. of Michigan, Mar. 1994.

[22] K.G. Shin and S.W. Daniel, "Analysis and Implementation of Hybrid Switching," *Proc. Int'l Symp. Computer Architecture*, pp. 211–219, June 1995.

[23] H.S. Lee, H.W. Kim, J. Kim, and S. Lee, "Adaptive Virtual Cut-Through as an Alternative to Wormhole Routing," *Proc. Int'l Conf. Parallel Processing*, pp. I–68–I–75, 1995.

[24] 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," *Proc. VLSI*, Sept. 1993.

[25] J. Dolter, S. Daniel, A. Mehra, J. Rexford, W. Feng, and K. Shin, "SPIDER: Flexible and Efficient Communication Support for Point-To-Point Distributed Systems," *Proc. Int'l Conf. Distributed Computer Systems*, pp. 574–580, June 1994.

[26] S. Daniel, J. Rexford, J. Dolter, and K. Shin, "A Programmable Routing Controller for Flexible Communications in Point-To-Point Networks," *Proc. IEEE Int'l Conf. Computer Design*, pp. 320–325, Oct. 1995.

[27] C.-M. Chiang and L.M. Ni, "Multi-Address Encoding for Multicast," *Proc. Parallel Computer Routing and Comm. Workshop*, pp. 146–160, May 1994.

[28] *Am79168/Am79169 TAXI-275 Technical Manual*, ban-0.1m-1/93/0 17490a ed. Sunnyvale, Calif.: Advanced Micro Devices, 1993.

[29] K.G. Shin and S.W. Daniel, "Analysis and Implementation of Hybrid Switching," *IEEE Trans. Computers*, vol. 45, no. 6, pp. 684-692, June 1996.

[30] J. Rexford, J. Dolter, W. Feng, and K.G. Shin, "PP-MESS-SIM: A Simulator for Evaluating Multicomputer Interconnection Networks," *Proc. Simulation Symp.*, pp. 84–93, Apr. 1995.

[31] J. Rexford, J. Hall, and K.G. Shin, "A Router Architecture for Real-Time Point-To-Point Networks," *Proc. Int'l Symp. Computer Architecture*, pp. 237-246, May 1996.

Stuart W. Daniel received the BE degree in electrical engineering and computer science from Vanderbilt University in 1989 and the ME and PhD degrees in computer engineering from the University of Michigan in 1994 and 1996. He is an engineer at Lexmark International, where he is working in software research and development.

Kang G. Shin received the BS degree in electronics engineering from Seoul National University, Seoul, Korea in 1970, and the MS and PhD degrees in electrical engineering from Cornell University, Ithaca, New York, in 1976 and 1978, respectively. From 1978 to 1982, he was on the faculty of Rensselaer Polytechnic Institute, Troy, New York. He has held visiting positions at the U.S. Airforce Flight Dynamics Laboratory, AT&T Bell Laboratories, the Computer Science Division within the Department of Electrical Engineering and Computer Science at U.C. Berkeley, and the International Computer Science Institute, Berkeley, California, IBM T.J. Watson Research Center, and Software Engineering Institute at Carnegie Mellon University. He also chaired the Computer Science and Engineering Division, EECS Department, The University of Michigan for three years beginning in January 1991. He is a professor and director of the Real-Time Computing Laboratory, Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, Michigan.

He has authored/coauthored more than 600 technical papers and numerous book chapters in the areas of distributed real-time computing and control, computer networking, fault-tolerant computing, and intelligent manufacturing. He has coauthored (jointly with C.M. Krishna) a textbook, *Real-Time Systems* (McGraw Hill, 1997). In 1987, he received the Outstanding IEEE Transactions on Automatic Control Paper Award and, in 1989, Research Excellence Award from The University of Michigan. In 1985, he founded the Real-Time Computing Laboratory, where he and his colleagues are investigating various issues related to real-time and fault-tolerant computing.

His current research focuses on Quality of Service (QoS) sensitive computing and networking with emphasis on timeliness and dependability. He has also been applying the basic research results to telecommunication and multimedia systems, intelligent transportation systems, embedded systems, and manufacturing applications.

He is an IEEE fellow, was the program chairman of the 1986 IEEE Real-Time Systems Symposium (RTSS), the general chairman of the 1987 RTSS, the guest editor of the 1987 August special issue of *IEEE Transactions on Computers* on *Real-Time Systems*, a program co-chair for the 1992 International Conference on Parallel Processing, and served on numerous technical program committees. He also chaired the IEEE Technical Committee on Real-Time Systems during 1991-1993, was a distinguished visitor of the Computer Society of the IEEE, an editor of *IEEE Transactions on Parallel and Distributed Computing*, and an area editor of *International Journal of Time-Critical Computing Systems*.

Sang Kyun Yun received the BS degree in electronics engineering from the Seoul National University, Korea, in 1984, and the MS and PhD degrees in electrical engineering from Korea Advanced Institute of Science and Technology (KAIST), Korea, in 1986 and 1995, respectively. He is currently an associate professor in the Department of Computer Science, Seowon University, Cheongju, Korea. He worked at Hyundai Electronics Industries, Korea, from 1986 to 1990. He was a visiting researcher at the Real-Time Computing Laboratory in the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, during 1998. His interests include interconnection network, parallel processing, computer architecture, and computer network.