A Programmable Routing Controller for Flexible Communications in Point-to-Point Networks

Submitted to the *International Conference on Computer Design*, 1995

Stuart W. Daniel, Jennifer L. Rexford, James W. Dolter,
and Kang G. Shin

Real-Time Computing Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109–2122
voice: (313) 763–0391; fax: 313-763-4617
E-mail: {stuartd, jrexford, jdolter, kgshin}@eecs.umich.edu

ABSTRACT

Modern parallel and distributed applications have a wide range of communication characteristics and performance requirements. A flexible communication subsystem can meet these diverse needs by tailoring network policies to applications. This paper describes the design and operation of the *Programmable Routing Controller* (PRC), a custom ASIC for flexible and efficient communication in point-to-point networks. By dedicating a small programmable processor to each incoming link, the PRC can implement wormhole, virtual cut-through, and packet switching, as well as hybrids of these schemes, each under a variety of unicast and multicast routing algorithms. The PRC can support several different applications or traffic types *simultaneously* by implementing multiple routing-switching microcode routines. Cycle-level simulations of mesh-connected PRCs demonstrate how flexible routing and switching can significantly enhance application performance.

1 Introduction

In parallel and distributed systems, efficient communication enables fine-grained cooperation between processing nodes. Maximizing system performance requires matching application characteristics and performance requirements with a suitable network design. However, applications employ a wide variety of communication paradigms that affect the quantity and frequency of communication between nodes. Parallel applications such as scientific computations [1, 2], parallel databases, and real-time applications [3] generate distinct distributions

The work reported in this paper was supported in part by the National Science Foundation under grant MIP–9203895. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of NSF.

for packet lengths, interarrival times, and target destinations. This paper introduces flexible communication hardware that can tune network policies to these diverse characteristics.

For point-to-point networks, these application characteristics affect the performance of particular *routing* and *switching* schemes [4–7]. The switching scheme impacts performance by determining the link and buffer resources a packet consumes at a given node in its route. Traditional *packet switching* requires incoming packets to buffer completely before transmission to a subsequent node can begin. In contrast, cut-through switching schemes, such as *virtual cut-through* [8] and *wormhole* [9], try to forward incoming packets directly to an idle output link. If the outgoing link is busy, virtual cut-through switching buffers the packet, whereas a blocked wormhole packet stalls pending access to the link. While first-generation multicomputers employed packet switching, most contemporary routers utilize wormhole [9–15] or virtual cut-through [16–19] switching for lower latency and reduced buffer space requirements.

The routing algorithm determines which nodes a packet traverses to reach its destination. *Oblivious* routing generates a single, deterministic outgoing link for an incoming packet, whereas *adaptive* routing schemes can incorporate prevailing network conditions into the routing decision. By considering multiple outgoing links, adaptive algorithms can balance network load and increase a packet's chances of cutting through intermediate nodes. Additionally, adaptive schemes may consider *nonminimal* paths in the hope of circumventing network congestion or faulty links. Although most existing routers implement oblivious routing [9–12], several recent designs support adaptive routing [14–17]. Multicomputer applications can also benefit from router support for multicast communication, since routing a single message to multiple destinations facilitates efficient barrier synchronization and global reduction operations [20].

Each routing and switching policy is best suited for traffic with particular characteristics and performance requirements. For example, adaptive routing can reduce end-to-end delay, but out-of-order packet arrival can complicate protocol processing at the receiving node [21]. Opportunities for adaptive routing vary depending on the topology, the distance a packet must travel, and network congestion. 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 internode communication often consists of large data transfers, coupled with small request and acknowledgement packets [2]. 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 [6].

Since no single routing-switching combination performs best under all conditions, a flexible router architecture should support a range of routing algorithms and switching schemes. However, most existing multicomputer routers implement a single routing-switching combination in hard-wired logic. This paper presents a custom ASIC, the Programmable Routing Controller (PRC), that implements a variety of routing and switching schemes by dedicating a microprogrammable routing engine to each incoming link. These small processors can parse the routing header of an incoming packet and construct a routing-switching decision, based on the packet header, the microcode, and prevailing network conditions. This design also

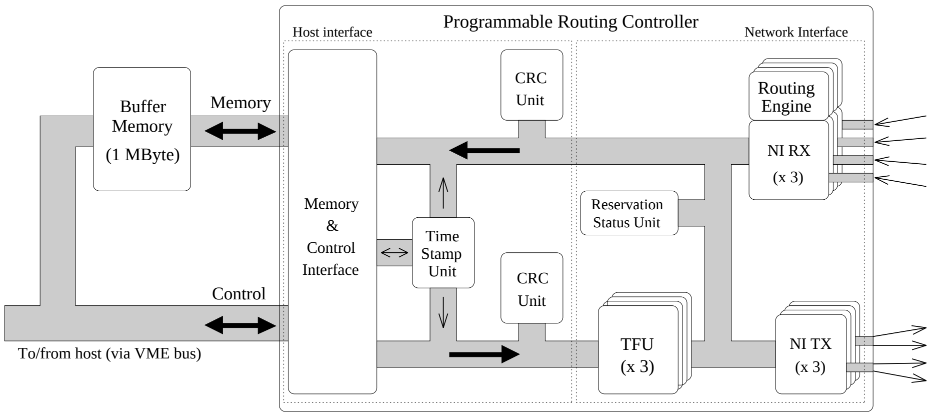


Figure 1: PRC architecture

allows the PRC to support multiple routing-switching combinations *simultaneously*.

Flexible control over link and buffer reservation enables the PRC to implement wormhole, virtual cut-through, and packet switching, as well as hybrid schemes, each under a variety of unicast and multicast routing algorithms. The next section of the paper presents the architecture of the PRC and describes its low-overhead interface to the controlling host processor. Section 3 describes the microarchitecture of the PRC's programmable routing engines. In Section 4, a sample microprogram illustrates how these devices can implement a routing-switching scheme that combines features of virtual cut-through and wormhole switching [22]; cycle-level simulations of a network of PRCs show how this flexible algorithm can significantly improve application performance. Section 5 describes the chip's current status and avenues for future work.

2 Architectural Overview of the PRC

As shown in Figure 1, the PRC manages bidirectional communication with up to four other nodes, with three virtual channels [23] on each unidirectional link. The controlling host processor has direct, memory-mapped access to the PRC across the VME bus; user applications may run on this host or on another processor that uses the host, coupled with the PRC, as a dedicated communication engine. The PRC's *network* interface manages the inter-node links and implements routing and switching for incoming packets, while a low-overhead *host* interface transfers data between the external buffer memory and the network links.

2.1 Host Interface

To reduce software protocol complexity, the PRC interacts with the controlling host processor in terms of *pages*, as shown in Table 1. 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 design does not include internal buffers for blocked packets, packets that buffer at intermediate nodes are stored in this 1 MByte SRAM; the host coordinates further transmission of this in-transit traffic by feeding a TFU with the packet's page tags. The PRC logs the transmission and reception of individual pages in an internal event queue. By reading this event queue, the host processor can perform free-list maintenance and assemble incoming packets.

The page-level interaction between the PRC and the host facilitates scatter-gather DMA between the buffer memory and the network interface 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 [24]. Since the PRC does not restrict packet format, 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 packet sizes, the PRC allows packets to consist of either 256-byte or 1024-byte pages; larger pages allow the PRC to operate without host intervention for longer periods.

The transmission and reception datapaths incorporate transparent error detection via *cyclic redundancy code* (CRC) generation and checking. After sending the last data word of a packet, the TFU transmits a 32-bit timestamp read from a counter on the PRC, followed by the CRC. On reception, the CRC is checked and removed, and a second timestamp added. These timestamps are useful for controlling clock drift between nodes [25], as well as for performance measurements. The outcome of the CRC check is logged in the event queue, freeing the host from checking every word of the messages it receives and transmits. The first page tag for a packet can 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 additional error detection or correction on the packet header.

The PRC control interface also includes several configuration options, as shown in Table 1. 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.

Category	Functions
Data transfer	outbound page transmission free page allocation for inbound packets
Status	event retrieval determining available slots in page queues timestamp access
Configuration	interrupt masking notification FIFOs
Initialization	microcode download

Table 1: Control interface capabilities

2.2 Network Interface

While the host manages communication at the page level, the PRC coordinates the fine-grain interaction between incoming and outgoing virtual channels. The PRC communicates with other nodes via four pairs of serial links, with each pair consisting of a transmission link and a reception link. The links are implemented using AMD TAXI chips [26] operating in a synchronous mode where each link transmits 10 bits of information (8 bits data, 2 bits control) every link cycle[1]; the control information is used by the PRC to transfer commands, flow-control acknowledgments, and virtual channel identifiers. Each outgoing link is controlled by a module containing three *network interface transmitters* (NITXs), each implementing a single outbound virtual channel. The NITXs and NIRXs perform the necessary interleaving of virtual channels to and from the physical links, on a word-by-word basis.

The PRC treats the outgoing virtual channels (NITXs) as individually reservable resources, allowing the device to support a variety of routing and switching schemes through flexible control over channel allocation policies. The reservation status unit handles requests to reserve or relinquish the NITXs, providing low-level support for both connection-oriented and connectionless transfer on each virtual channel. The network interface components communicate over the *cut-through bus* (CTBUS), a 32-bit time-division multiplexed bus. CTBUS access is controlled by a demand-slotted binary priority-tree arbiter [27, 28]. To match the bandwidth of the eight unidirectional links, the 32-bit CTBUS operates at twice the byte-transmission speed of a link.

The CTBUS protocol includes commands to transfer data and reserve/relinquish NITXs, as shown in Table 2. The DTX, MARK, and EOP commands each transfer a word of data, with a EOP tag marking the last word in a packet. The MARK command tags page boundaries, allowing the destination node to reconstruct the packet's page structure upon reception; this can ensure that protocol header pages are separate from data pages on the receiving node. Each CTBUS command includes a 13-bit slave *mask* to address the memory interface and any of the 12 NITXs. The address masks, coupled with the bus interconnect, enable a

[1] To reduce the package size of the PRC, a pair of outgoing links shares a single set of pins; internally, the PRC operates at 30 MHz, twice the link speed, to serve each outgoing link at its full rate.

Command	Function	Type
DTX	Normal data transfer	Tag
MARK	End-of-page data transfer	Tag
EOP	End-of-packet data transfer	Tag
FREE	Tears down the circuit	Control
RESV	Reserves selected channels	Control

Table 2: CTBUS command summary

single CTBUS transaction to spawn transmissions on several output links simultaneously; this allows an arriving packet to generate multiple copies at each hop in its route, facilitating efficient broadcast and multicast algorithms [29, 30].

The RESV and FREE commands are used to acquire and relinquish reservations of outbound virtual channels (NITXs). While all CTBUS devices have concurrent access to the NITXs' reservation status, the bus interconnect serializes reservation requests, simplifying the design of the reservation status unit. Using the slave address mask, a CTBUS master can reserve one or more NITXs with a single RESV command. The TFUs and NIRXs relinquish channel reservations with the FREE command; any NITX slaves forward the FREE command to the subsequent node(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 CTBUS command set, coupled with the NITX/NIRX state machines, handles the routine tasks of channel reservation, arbitration, flow control, and data transfer, allowing the routing engines to focus completely on constructing routing-switching decisions for incoming packets.

3 Receiver Architecture

While supporting multiple routing-switching schemes requires flexibility in manipulating routing headers, hardwired state machines can handle the remainder of the packet transfer. Since header processing time is small relative to data transfer time, multiple virtual channels can share a single routing engine. This routing engine parses the header of each incoming packet to construct a routing-switching decision, based on the microcode, the packet header, and prevailing network conditions. The routing engine's instruction set enables the PRC to interpret a wide variety of header formats and implement different routing-switching schemes. The routing engine communicates with the NIRX state machines through a small *routing primitive* that embodies basic routing-switching operations.

3.1 Receiver Module

Each routing engine is a small, 8-bit microcontroller with a 256-word control store for microprograms, as shown in Figure 2. The host uses the control interface to download the

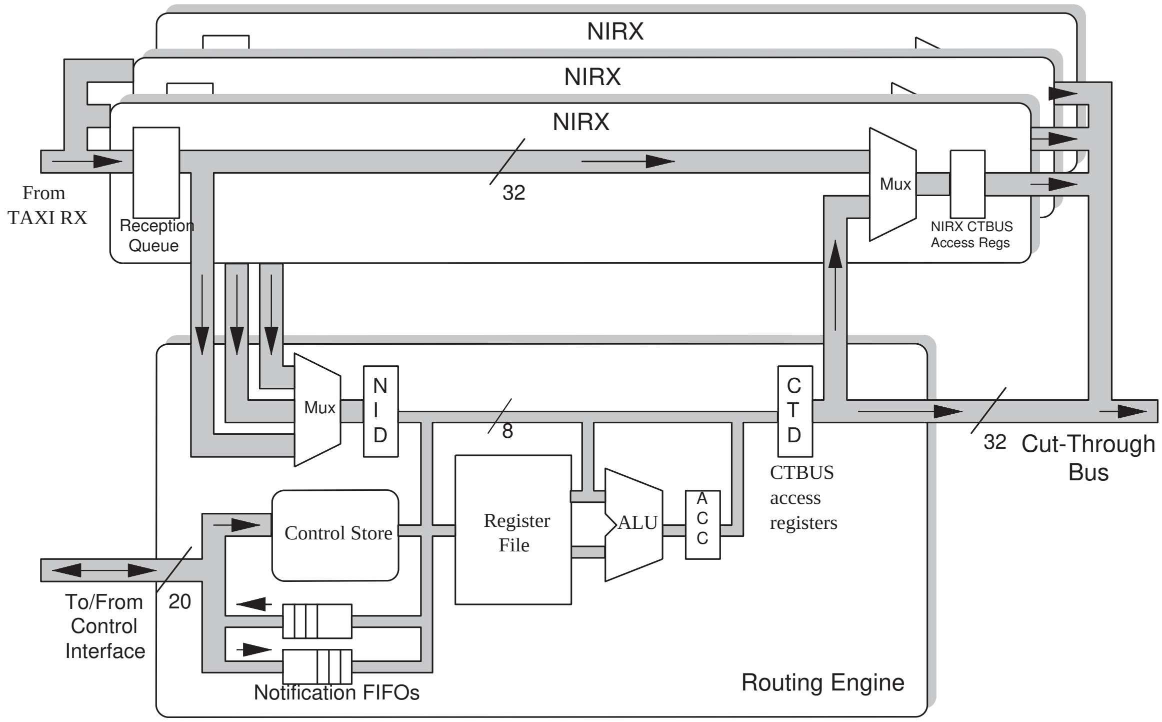


Figure 2: Receiver module

microprograms during system initialization and to adjust microcode operation at run time through the notification FIFOs, as shown in Table 1. A 16-byte register file stores an arriving packet header, as well as constants and intermediate computations. The 8-bit arithmetic logic unit (ALU) includes a variety of integer and boolean operations for manipulating and updating routing headers; the results of ALU operations are stored in the 8-bit accumulator and two boolean flags (`carry` and `zero`). The registers and flags, coupled with the ALU, enable the routing engine to parse incoming packet headers and construct a routing-switching decision.

The routing engine interacts with the NIRXs and the CTBUS through the *network interface data* registers (`nid`) and the CTBUS data registers (**`ctd`**). Arriving packet headers are initially stored in the appropriate NIRX's reception queue; upon receiving the first header word, the NIRX signals the routing engine. After selecting this NIRX for service, the routing engine moves the header data into its internal registers, triggering a flow-control acknowledgement to the adjacent node. This process repeats until the routing engine accumulates enough header words to construct a routing-switching decision. The microcode manipulates and updates the routing header, and possibly reserves outgoing virtual channels, before returning control to the NIRX. Once the NIRX receives the routing primitive from the routing engine, it acquires any necessary channel reservations and forwards the header to the CTBUS slave devices. The NIRX transfers the remainder of the packet directly from its reception queue, bypassing the routing engine entirely, until an incoming FREE command resets the NIRX.

Field	Function(s)	Register(s)
`data`	next word of data to transmit	`ctd3` — `ctd0`
`address`	selects memory interface and/or desired NITX channels	`ctaddr1, ctaddr0`
`control`	`crcflg` — If set, word is included in CRC `resvd` — slaves already reserved if set `all` — reserve all selected slaves if set else reserve at most one	`ctctl`

Table 3: Routing primitive structure

Header data exits the routing engine through the CTBUS access registers (`ctd`) as part of the routing primitive, as shown in Table 3. In addition to the header data, the routing primitive includes a 13-bit mask (stored in the `ctaddr1` and `ctaddr0` registers) to address the memory interface and any of the outgoing virtual channels; a control tag indicates how the NIRX state machine should interpret this address mask. For virtual cut-through and packet switching schemes, the routing engine can construct the packet's complete routing-switching decision and allocate any necessary NITX resources using the RESV instruction and the CTBUS interface. The routing engine buffers a packet by setting the memory-interface bit in the routing primitive.

However, wormhole switching *stalls* a blocked packet until a suitable outgoing virtual channel become available; this requires monitoring the status of the desired NITX(s) for an indefinite period of time. To free the routing engine for use by other NIRXs, the routing primitive can instruct the NIRX state machine to monitor a collection of NITXs (indicated by the `ctaddr1` and `ctaddr0` bit mask). Thus, the routing engine can reserve any *idle* virtual channels for the incoming packet, while the NIRX state machine waits on any busy channels. The NIRX reserves one, or all, of these resources as they become available, depending upon the `all` field in the routing primitives; the `all` bit differentiates between multicast routing and adaptive unicast routing schemes.

3.2 Routing Engine Instruction Set

The routing engine's instruction set includes instructions for manipulating header data, controlling external interfaces, and branching based on internal conditions, as shown in Table 4. The routing engine employs the `alu` instructions to parse and modify fields in a packet's routing header, using the ALU's `zero` and `carry` flags for conditional branches in the microcode. The `ldc` instruction loads constants, such as predetermined address masks and control tags, into registers, while the `xfer` instructions copy data between these registers. The routing engine issues routing primitives and CTBUS commands as maskable side effects of the `ldc` and `xfer` instructions; since the microcode typically writes to the CTBUS registers just before accessing the interface, this optimization reduces both microcode size and execution time.

Instruction	Function
`alu`	boolean and arithmetic operations on headers
`ldc`	load constant into register
`xfer`	copying register contents
`go nirx`[a]	triggers NIRX with a routing primitive
`go ctbus`[a]	triggers FSM controlling CTBUS access
`jump`[b]	conditional branch
`return`	return from subroutine
`flag`	setting, clearing, and transferring flag value
`wait`	three-way, blocking branch

[a] The `go` instructions are not actually separate instructions, but are side effects that can be triggered by the `ldc` and `xfer` instructions.

[b] A `save` option stores the address of the next microinstruction, allowing the `jump` instruction to invoke a microcode subroutine that ends with the `return` instruction.

Table 4: Routing engine instruction set summary

Conditional branches use the `jump` instruction to react to hardware and user-controlled flags. The hardware flags include the ALU's `zero` and `carry` bits, as well as reservation status flags for the NITXs. The reservation flags allow the routing engine to identify free outbound channels without accessing the CTBUS. The routing engine can also check the status of a collection of NITXs simultaneously based on the bit mask in the CTBUS address registers; this facility is especially useful for multicast routing algorithms. The microcode uses the `flag` instruction to set, clear, and load user-controlled flags to temporarily store boolean conditions; for example, an algorithm may employ the `flag` instruction to save the result of a bit-mask or comparison operation on a routing header, allowing the routing engine to update other header fields with the ALU. The `jump` instruction can branch based on these condition bits and also invoke microcode subroutines using the `save` option; when a `jump` instruction includes the `save` qualifier, the routing engine stores the address of the next microinstruction, so the microsequencer can `return` to the main instruction flow when the subroutine completes.

In addition to the `jump` instruction, the routing engine's instruction set includes a three-way blocking branch to wait for a packet header to arrive on one of the three incoming virtual channels. This `wait` instruction blocks until a header word arrives to one (or more) of the NIRX reception queues; when multiple NIRXs have packet headers, the instruction also arbitrates amongst the three virtual channels[2]. To simplify the microcode, the `wait`

[2] For example, the instruction

```
wait ni2, trap0(ni1), trap1(ni0)
```

waits for data on any of the three NIRXs, giving highest priority to channel 0 and lowest priority to channel 2. In actuality, the `wait` instruction stalls on one condition (e.g., `ni2`) but can also includes trap handlers that branch to addresses specified in two special-purpose registers (`trap0` and `trap1`) in the register file.

Register	Field	Function
nid3	h	if $h > 0$, stall and decrement
nid2	x	destination offset along x-axis
nid1	y	destination offset along y-axis
nid0	–	can be used for error-detection checksum

Table 5: Typical hybrid routing header format

instruction also sets an internal flag to identify the selected NIRX and automatically transfers the header word to the internal nid registers.

4 Example of Flexible Switching

The PRC's support for multiple routing-switching schemes can be used to tailor network performance to the current communication load. This section demonstrates the PRC's flexibility through a sample microprogram that implements hybrid switching [22]; cycle-level simulations of a mesh of PRCs illustrate how this algorithm can improve network performance.

4.1 Sample Microprogram

Figure 3 shows portions of a microprogram implementing hybrid switching [22] with dimension-ordered oblivious routing. This hybrid algorithm 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. This approach improves throughput in the network over pure wormhole switching, while buffering fewer packets than virtual cut-through switching. In addition, this threshold can be dynamically varied depending on the packet's needs or the current network load by changing the initial value of this header field.

4.1.1 Overview

The program routes packets on an unwrapped square mesh using oblivious dimension-order routing, starting with the x-axis. Since this routing algorithm requires only one virtual channel per link to prevent communication deadlocks [31], the microprogram uses the remaining virtual channels to improve network throughput. Arriving packets have a single-word routing header, as shown in Table 5. The header specifies the packet's destination in terms of x and y offset, which are decremented as the packet moves through the network; when both offsets reach 0, the packet has reached its destination node. Similarly, the h field specifies the hop-count threshold for "stalling" and is decremented at each node until it reaches 0; when a packet buffers at an intermediate node, the host processor can reinitialize the h field.

```
init:
  ldc 0x80, reg0;
  ldc 0x0, ctaddr0;
  ldc ctcmd_resv, ctctl;
  ldc handler, trap0;
  ldc handler, trap1;
/* now wait for a packet header */
  wait ni2, trap0(ni1), trap1(ni0);

handler:      /* route packet   */
  alu nid2+1;       /* update x */
  xfer acc,ctd2;
  jump ~zero,route_x_neg;
  alu nid1;          /* check y */
  jump zero, buffer_packet;
  alu nid1 & reg0;
  jump zero, route_y_pos;
  jump true, route_y_neg;

buffer_packet:
  xfer nid3, ctd3;
  xfer nid1, ctd1;
  xfer nid0, ctd0;
  ldc  ctaddr_buff, ctaddr1;
  ldc  rtp_resvd_nocrc, ctctl, go nirx;
  jump true, init;
route_x_neg:
  alu nid3;           /* check h */
  jump zero, route_x_neg_buffer;
  alu acc-1;
  xfer acc, ctd3;
  xfer nid1, ctd1;
  xfer nid0, ctd0;
  ldc ctaddr_l3, ctaddr1;
  ldc rtp_wait_1, ctctl, go nirx;
  jump true, init;

route_x_neg_buffer:
  jump ~rsvdl2c0,grab_l2c0,save;
  jump ~rsvdl2c1,grab_l2c1,save;
  jump ~rsvdl2c2,grab_l2c2,save;
  jump true, buffer_packet;

grab_l2c0:
  ldc ctaddr_l2c0, ctaddr1, go ctbus;
  xfer nid3, ctd3;
  xfer nid1, ctd1;
  xfer nid0, ctd0;
  return ~ack;
  ldc rtp_resvd_nocrc, ctctl, go nirx;
  jump true, init;
```

Figure 3: Microprogram routing example

The program consists of six main routines — `init`, `handler`, `route_x_neg`, `route_y_neg`, `route_y_pos`, and `buffer_packet`. The `init` routine assigns several important constants to internal registers and waits for a packet header to arrive. Upon header arrival, the `handler` code determines the next link in the packet's journey and jumps to the appropriate `route` code. Figure 3 shows the microcode for a link in the negative-x direction, so the dimension-ordered routing algorithm first considers the negative-x direction, followed by the links in the y-direction; since the three `route` segments differ only in which link they check, only `route_x_neg` is shown. The `route` routines update the h field and decide whether the packet should buffer or stall if the outgoing link has no idle virtual channels.

4.1.2 Code Description

Initialization: The `init` routine (lines 1 through 6) initializes several of the routing engine's registers. For example, line 2 sets a bit mask for checking the sign of the y offset, used later to decide if the packet needs to travel in the positive or negative y-direction. Similarly, lines 3 and 4 optimistically initialize two CTBUS registers; many later branches use these default values. Finally, the `init` routine sets the `trap` registers for the subsequent `wait` instruction; all three virtual channels (NIRXs) invoke the same handler (at line 10), since the routing-switching algorithm does not differentiate between packets on different incoming virtual channels. Once a packet header arrives, the `wait` instruction latches the header word into the NI data registers (`nid`) and sets an internal flag to indicate which NIRX received the header data.

Routing decision: The `handler` selects the next link for the packet to traverse. It begins by incrementing the header's x offset (in line 11) to reflect the hop from the previous node before forwarding the modified header field to the CTBUS data register (`ctd2`). Then, the microcode checks the x and y offsets to determine which link to traverse next; to implement dimension-ordered routing, the routing engine first inspects the x offset (still stored in the accumulator), as shown in line 13. If this x-offset is non-zero, the microcode branches to the `route_x_neg` routine; otherwise, the routing engine considers the offset in the y-direction (line 15). If the y offset is zero, too, the packet has reached its destination, and line 15 jumps to the `buffer_packet` routine. Otherwise, line 16 checks the sign bit of the y offset to determine if the packet should travel in the positive or negative y-direction, jumping to either `route_y_pos` or `route_y_neg`, as shown in lines 17 and 18.

Packet buffering: Lines 20 through 26 implement the `buffer_packet` routine, which configures the NIRX to write the packet to the memory interface. The routing engine transfers the remaining bytes of the routing header to the CTBUS data register (lines 21–23) and sets the `ctaddr1` address mask to buffer the packet (line 24); `ctaddr0` was initialized by `init`. Line 25 loads the `ctctl` field of the routing primitive, indicating that the header word should not be included in the CRC computation; the `go nirx` qualifier in the `ldc` command triggers the NIRX to handle the remainder of packet reception. Consequently, line 26 jumps to the initialization routine to reset the routing engine for the next packet.

Stalling semantics: To route a packet in the negative x-direction, the `route_x_neg` routine (lines 27–51) checks the h field in the packet header to determine whether the packet should buffer or stall if none of the link's NITXs are available, as shown in lines 28 and 29. If $h > 0$, lines 30 and 31 decrement h and transfer the modified header field to the CTBUS data register (`ctd3`) for later forwarding; lines 32 and 33 transfer the remaining header fields. The program then constructs the address and control fields of a routing primitive, so the NIRX can wait for one of the link's virtual channels to become available; in line 35, the constant `ctaddr_l3` is a bit mask identifying the three virtual channels on the outgoing link. Finally, line 36 instructs the NIRX to reserve only *one* of these NITXs by loading the `rtp_wait_1` constant into the control field of the routing primitives; this load instruction also transfers control to the NIRX (through the `go nirx` qualifier). Line 36 then jumps to the initialization routine to prepare the routing engine for another packet.

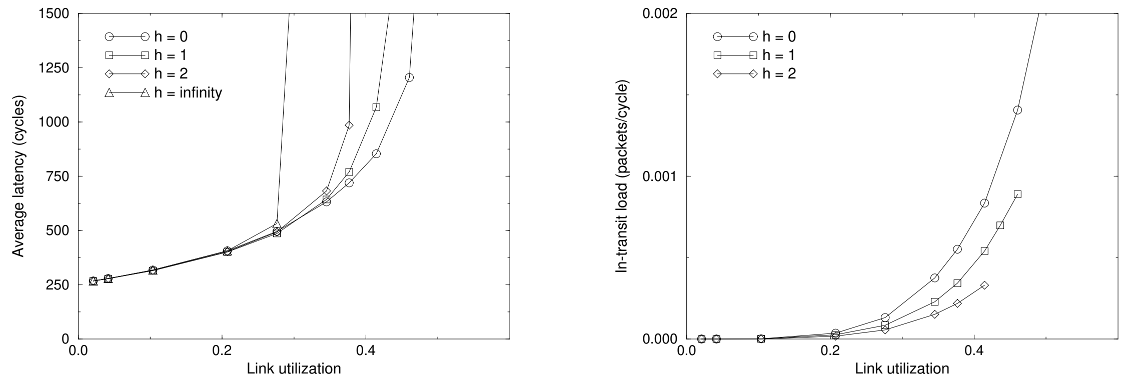
Buffering semantics: When $h < 0$, the packet should buffer if the outgoing link has no free NITXs; thus, the `route_x_neg_buffer` routine (lines 38–42) directly checks the reservation status of the three channels of the link. For example, line 39 checks the status of virtual channel 0 (identified by the `rsvdl3c0` bit mask) and jumps to the `get_l3c0` subroutine, which tries to reserve the channel if it is free; the `save` at the end of the instruction saves the address of the next instruction, allowing the code to return from the subroutine (if the subroutine cannot reserve the NITX). If all three NITXs are already reserved, the blocked packet is buffered locally by jumping to the `buffer_packet` routine.

Reserving an NITX: Lines 44 through 51 show one of these reservation subroutines. In line 45, the microcode loads the NITX's address into the CTBUS address register and triggers the state machine that controls CTBUS access. The CTBUS access takes at least three cycles to complete, so the routing engine uses this time to transfer the rest of the routing header to the CTBUS data registers, as shown in lines 46–48. Then the `return` instruction (line 49) automatically blocks until the `RESV` attempt completes. The CTBUS `ack` flag indicates if the reservation attempt was successful. If the RESV request fails (i.e., `ack` is false), the `grab_l2c0` subroutine returns to the instruction indicated by the `save` register to allow the `route_x_neg_buffer` routine to consider other NITXs. Otherwise, line 50 directs the NIRX to execute the cut-through operation; after transferring control to the NIRX, the microcode jumps to the initialization routine to await the next incoming packet.

4.2 Simulation Experiments

This hybrid switching algorithm balances the use of channel and buffer resources for "storing" blocked packets. By initializing h to some sufficiently large value, the hybrid algorithm implements wormhole switching by always stalling blocked packets; similarly, setting h to 0 effectively implements virtual cut-through switching. Figure 4 compares hybrid switching for several values of h using two performance metrics — average packet latency and in-transit load; the in-transit load measures the average rate that packets buffer at intermediate nodes, and represents the buffer and control resources required to service these packets.

These simulations were run using the `pp-mess-sim`, an object-oriented discrete-event simulator for evaluating multicomputer routers [32]; `pp-mess-sim` includes a cycle-level model of the PRC that captures the details of flow control, resource arbitration, and microcode execution. These simulations evaluate an 8×8 square mesh carrying 64-byte packets; similar



(a) Average latency (b) In-transit load

Figure 4: Comparison of various switching schemes

performance trends occur for other network topologies and packet sizes. Each node independently generates packets with exponentially-distributed inter-arrival times and uniform random selection of destination nodes.

As shown in Figure 4(a), selectively buffering blocked packets improves achievable network throughput; as h increases, the network can effectively handle a larger traffic load. However, larger values of h allow more packets to buffer at intermediate nodes, consuming buffer and processing resources. At high loads, virtual cut-through switching uses more than three times more memory resources than the $h = 2$ hybrid scheme, since the 2-hop hybrid algorithm allows packets to buffer at most once every three hops. The system can exploit the differences between these schemes by dynamically varying the hop threshold, depending on the system's current load and on the characteristics of the current packet.

This particular routing-switching example employs dimension-ordered oblivious routing, using virtual channels solely to improve network throughput. The PRC can also use its three virtual channels to implement deadlock-free adaptive routing algorithms, which improve performance for some traffic patterns [33]. The PRC's programmable routing engines allow the network to use its virtual channels for adaptive routing or to reduce contention between packets traveling on the same link, depending on the application's traffic patterns. Alternately, the network can assign virtual channels to different applications or traffic types with distinct network policies.

5 Current Status and Future Work

The PRC has been fully designed using a .8μm, two-metal process and Epoch design tools from Cascade Design Automation; the chip is scheduled for fabrication in April 1995. The ASIC has a total of 484,412 transistors and fits in a die 12.3mm by 16.4mm. As shown

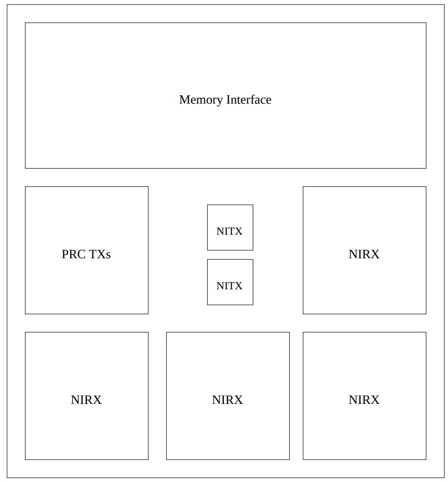


Figure 5: Floorplan of the PRC

in Figure 5, 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.

Internally, most of the chip uses a two-phase, 30 MHz clock; externally, the TAXI transmitters operate in a synchronous mode, sending data every 66ns (15 MHz). The memory interface uses a 40MHz clock and consumes two clock cycles for each memory access; since the PRC shares the buffer memory with the host processor, the available bandwidth to the PRC ranges from 40 MB/sec to 80 MB/sec, depending on the host. The network interface, by comparison, provides up to 120 MB/sec of data bandwidth on both the CTBUS and the physical links; by implementing cut-through switching schemes, the PRC can exploit the network interface's bandwidth without overwhelming the memory interface. When one virtual channel operates in isolation, the PRC can drive the outgoing link at approximately 10.2 MB/sec; when the link multiplexes two or three virtual channels on a link, the PRC divides the available 15 MB/sec bandwidth evenly amongst the active NITXs.

The PRC design emphasizes flexible, low-level support for packet routing and switching to accommodate application characteristics and performance requirements. Instead of implementing programmable routing engines, the design could have included on-chip buffer space for blocked packets. This buffer would necessarily be limited in size. Using a switching technique such as hybrid switching, on the other hand, enables the network to reduce packet buffering requirements, while also limiting channel contention. In addition, the PRC's routing engines provide a unique experimental platform for investigating the interplay between application characteristics and routing-switching performance.

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," in *Proc. Int'l Symposium on Computer Architecture*, pp. 2–13, May 1993.

[3] 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.

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

[5] 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.

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

[7] 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*, (Cancun, Mexico), pp. 188–193, April 1994.

[8] P. Kermani and L. Kleinrock, "Virtual cut-through: A new computer communication switching technique," *Computer Networks*, vol. 3, pp. 267–286, September 1979.

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

[10] W. J. Dally, J. A. S. 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.

[11] 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.

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

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

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

[15] 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.

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

[17] A. L. Davis, "Mayfly: A general-purpose, scalable, parallel processing architecture," *Lisp and Symbolic Computation*, vol. 5, pp. 7–47, May 1992.

[18] 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.

[19] Y. Tamir and G. Frazier, "Dynamically-allocated multi-queue buffers for VLSI communication switches," *IEEE Trans. Computers*, vol. 41, pp. 725–737, June 1992.

[20] X. Lin and L. M. Ni, "Multicast communication in multicomputer networks," *IEEE Trans. Parallel and Distributed Systems*, vol. 4, pp. 1105–1117, October 1993.

[21] 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.

[22] K. G. Shin and S. W. Daniel, "Analysis and implementation of hybrid switching." to appear in *Proc. International Symposium on Computer Architecture*, June 1995.

[23] W. Dally, "Virtual-channel flow control," *IEEE Trans. Parallel and Distributed Systems*, vol. 3, pp. 194–205, March 1992.

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

[25] P. Ramanathan, D. D. Kandlur, and K. G. Shin, "Hardware assisted software clock synchronization for homogeneous distributed systems," *IEEE Trans. Computers*, vol. C-39, pp. 514–524, April 1990.

[26] Advanced Micro Devices, 901 Thompson Place, P.O. Box 3453, Sunnyvale CA 94088-3453, *Am79168/Am79169 TAXI-275 Technical Manual*, ban-0.1m-1/93/0 17490a ed.

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

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

[29] D. D. Kandlur and K. G. Shin, "Reliable broadcast algorithms for HARTS," *ACM Trans. Computer Systems*, vol. 9, pp. 374–398, November 1991.

[30] C.-M. Chiang and L. M. Ni, "Multi-address encoding for multicast," in *Proc. Parallel Computer Routing and Communication Workshop* (K. Bolding and L. Snyder, eds.), no. 853 in Lecture Notes in Computer Science, (Seattle, Washington), pp. 146–160, Springer-Verlag, May 1994.

[31] 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.

[32] J. Rexford, J. Dolter, W. Feng, and K. G. Shin, "PP-MESS-SIM: A simulator for evaluating multicomputer interconnection networks." To appear in *Proc. Annual Simulation Symposium*, April 1995.

[33] 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.