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

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

Real-Time Computing Laboratory
The University of Michigan
{stuartd, jrexford, jdolter, kgshin}@eecs.umich.edu

Abstract

Modern parallel and distributed applications have a wide range of communication characteristics and performance requirements. This paper presents the Programmable Routing Controller *(PRC), a custom ASIC that supports flexible network policies to accommodate diverse application requirements. By dedicating a small programmable processor to each incoming link, the PRC can implement wormhole, virtual cut-through, and packet switching, as well as hybrid schemes, under a variety of unicast and multicast routing algorithms. The PRC can support several applications or traffic types* simultaneously *by implementing multiple routing-switching microcode routines.*

1 Introduction

In parallel and distributed systems, efficient communication enables fine-grained cooperation between processing nodes. Maximizing system performance requires matching application characteristics 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, parallel databases, and real-time applications generate distinct distributions for packet lengths, interarrival times, and target destinations [1, 2]. This paper introduces flexible communication hardware that can tune network policies to these diverse characteristics.

For point-to-point networks, these characteristics affect the performance of particular *routing* and *switching* schemes [3–6]. Traditional *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* [7] and *wormhole* [8], try to forward the packet directly to an idle output link; if the link is busy, virtual cut-through switching buffers the packet, whereas a wormhole packet stalls pending access to the link. Most contemporary routers utilize wormhole [8–13] or virtual cut-through [14, 15] switching for lower latency and reduced buffer space requirements.

The routing algorithm determines which links a packet traverses to reach its destination. *Oblivious* routing generates a single outgoing link for an incoming packet, whereas *adaptive* schemes can consider multiple links to balance network load and increase a packet's chance 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 [8–10], several recent designs support adaptive routing [12–15]. Multicomputer applications can also benefit from router support for multicast communication, since sending a message to multiple destinations facilitates efficient barrier synchronization and global reduction operations.

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. Opportunities for adaptive routing depend on the topology and the distance a packet must travel. Similarly, wormhole switching achieves low latency without requiring packet buffers, but virtual cut-through and packet switching may achieve better throughput at high loads. Packet size also impacts network performance, since inter-node communication often consists of large data transfers, coupled

*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.

†James Dolter is now a senior engineer at Qualcomm Inc.

1063-6404/95 $4.00 © 1995 IEEE

with small request and acknowledgement packets [2]. The network can accommodate this mixture by using wormhole switching for long packets, while allowing short packets to use virtual cut-through switching to reduce network contention [5].

Since no single routing-switching combination performs best under all conditions, a flexible network should support a range of policies. However, most existing routers implement a single routing-switching combination in hard-wired logic. This paper presents a 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 allows the PRC to support multiple routing-switching combinations *simultaneously*. Simulation studies of the PRC have demonstrated the benefits of tailoring routing-switching policies to application characteristics [6, 16–18].

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 overviews the PRC architecture, while Section 3 describes the microarchitecture of the programmable routing engines. Section 4 describes the chip's current status and avenues for future work.

2 PRC Architecture

As shown in Figure 1, the PRC manages bidirectional communication with four other nodes, with three virtual channels [19] on each unidirectional link. The controlling host processor has direct, memory-mapped access to the PRC across the VME bus; 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 the *host* interface transfers data between the buffer memory and the network.

2.1 Host Interface

To reduce the complexity of both the hardware and software protocols, the PRC coordinates data transfer in terms of *pages*. 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 processor supplies each *network interface receiver* (NIRX) with pointers to free pages in the buffer memory, for use by arriving packets.

The twelve incoming and outgoing virtual channels share access to the external buffer memory, interleaving at the word level. Since the PRC does not include internal buffers for blocked packets, packets that buffer at intermediate nodes are stored in this SRAM; the host coordinates further transmission of this 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. The host reads this event queue to perform free-list maintenance and assemble incoming packets.

Page-level data transfer facilitates scatter-gather DMA between the buffer memory and the network and allows the host to avoid unnecessary data copying by constructing packet headers on a separate page from the data. Since the PRC does not restrict packet format, the protocol software can weigh the cost-performance trade-offs for selecting packet sizes. To better accommodate different sizes, the PRC allows packets to consist of either 256-byte or 1024-byte pages; larger pages allow the PRC to operate longer without host intervention, while smaller pages reduce fragmentation.

The transmission and reception datapaths incorporate transparent error detection via *cyclic redundancy code* (CRC) generation and checking, as well as packet timestamping useful for controlling clock drift between nodes. The first page tag for a packet can specify a number of words to exclude from the CRC calculation, allowing subsequent nodes to modify a packet's routing header without invalidating the original CRC checksum. If desired, the microprogrammable routing engines can implement separate error detection or correction on the packet header.

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. Each outgoing link is controlled by a module containing three *network interface transmitters* (NITXs), each implementing a single outbound virtual channel. The NITXs perform the necessary interleaving of virtual channels on the physical links, on a word-by-word basis.

The PRC treats the 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

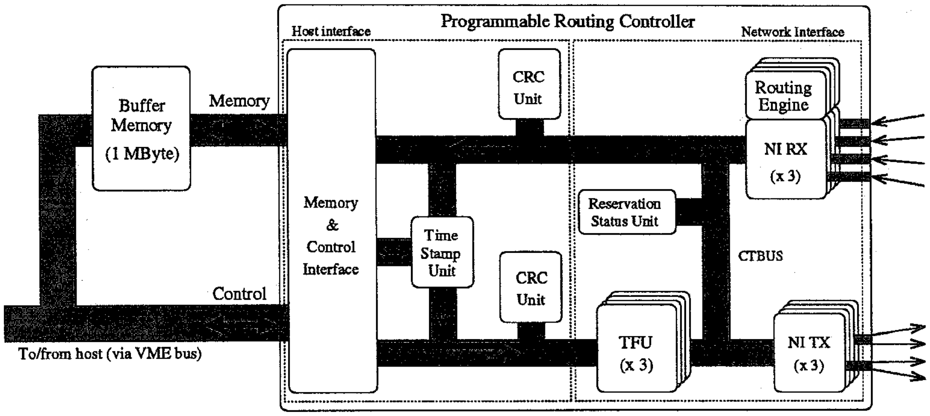

Figure 1: PRC architecture

status unit handles requests to reserve or relinquish NITXs. The network interface components communicate over the *cut-through bus* (CTBUS), a demand-slotted, time-division multiplexed bus. To match the bandwidth of the eight unidirectional links, the 32-bit CTBUS operates at twice the byte-transmission speed of a link; since the links run at half the PRC's clock frequency, a pair of outgoing links share a single set of pins, reducing the package size of the chip. The links are implemented using AMD TAXI chips operating in a synchronous mode where each link transmits 10 bits of information (8 bits data, 2 bits control) every link cycle; the control information is used by the PRC to transfer commands, flow-control acknowledgments, and virtual channel identifiers.

The CTBUS protocol includes commands to transfer data and reserve/relinquish NITXs. The DTX, MARK, and EOP commands each transfer a word of data, with an 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 mask, coupled with the bus interconnect, enables a 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 multicast algorithms.

Using the slave address mask, a CTBUS master can reserve one or more NITXs with a single RESV command; the CTBUS's bus interconnect serializes reservation requests, simplifying the design of the reservation status unit. The TFUs and NIRXs relinquish channel reservations with the FREE command; any NITX slaves forward the FREE command to the subsequent link(s) in the route to clear any downstream channel reservations. Although a FREE typically follows an EOP, separate commands allow the PRC to establish connections that outlive individual packets. The CTBUS commands, coupled with the NITX/NIRX state machines, handle the routine tasks of channel reservation, arbitration, flow control, and data transfer, allowing the routing engines to focus completely on constructing intelligent routing-switching decisions for incoming packets.

3 Receiver Architecture

While supporting multiple routing and switching schemes requires flexibility in manipulating packet headers, hardwired state machines can handle the remainder of data transfer. Since header processing time is small relative to data transfer time, multiple virtual channels can share a single routing engine. The routing engine has an instruction set designed to interpret the wide variety of header formats used in routing-switching policies. The NIRX state machines execute a simple *routing primitive* that facilitates efficient sharing of the routing engine's intelligence.

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 microprograms during system initialization and to adjust microcode operation at run-time through the notification FIFOs. 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

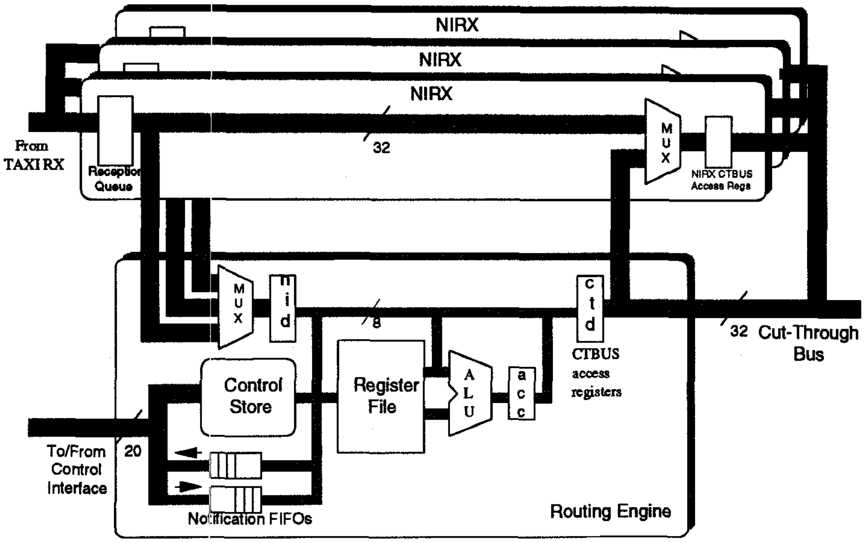

Figure 2: Receiver module

integer and boolean operations for manipulating and updating routing headers; the results of ALU operations are stored in the 8-bit accumulator (acc) 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. After receiving the routing primitive, the NIRX 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.

The routing engine controls NIRX operation through routing primitives, as shown in Figure 1. In addition to the header data, the routing primitive includes a 13-bit address mask to select the memory interface and any of the outgoing virtual channels. Control flags indicate how the NIRX state machine should interpret this address mask. The routing primitive's resvd flag can instruct the NIRX state machine to monitor a collection of NITXs. However, for maximum flexibility, the routing engine can also directly access the CTBUS to reserve idle channels. This allows the routing engine to consider candidate NITXs in a particular order, defined by the routing algorithm, while allowing the NIRXs to wait on a collection of busy channels. The routing engine can implement a variety of adaptive routing algorithms simply by changing what order it inspects possible outgoing links.

The resvd flag is crucial for efficiently implementing wormhole switching, since a stalled packet can require monitoring the status of candidate NITX(s) for an indefinite period of time. Instructing the NIRX to wait on busy channels frees the routing engine to process other arriving packets. If resvd is not set, the NIRX reserves one, or all, of the selected NITXs as they become available, depending upon the all bit which differentiates multicast routing algorithms from adaptive unicast routing schemes.

3.2 Instruction Set

The routing engine implements instructions for manipulating header data, controlling external interfaces, and branching based on internal conditions, as shown in Table 2. The routing engine employs the alu instructions to parse and modify fields in a packet's rout-

Register(s)	Function(s)
ctd3 — ctd0	next word to transmit
ctaddr	selects memory, NITXs
ctctl	boolean control flags:
crcflg	include word in CRC
resvd	slaves already reserved
all	reserve all/one of slaves

Table 1: Routing primitive

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

Table 2: Routing engine instruction set

ing 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 microprograms typically write to the CTBUS registers just before accessing the interface, this optimization reduces both microcode size and execution time.

Microprograms use the **jump** instruction to react to flags, including 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 save the result of a bit-mask or comparison operation on a routing header. The **jump** instruction can branch based on these condition bits. When a **jump** instruction includes the **save** qualifier, the routing engine stores the address of the next microinstruction, so the microsequencer can **return** to the main instruction flow when the subroutine completes; this restricted implementation of subroutines reduces microprogram size by enabling code reuse.

Efficiently sharing the routing engine requires a flexible method for selecting which NIRX to service. Consequently, 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; the instruction's arguments also determine which NIRX to service when multiple NIRXs have packet headers. To simplify the microcode, the **wait** instruction also sets an internal flag to identify the selected NIRX and automatically transfers the header word to the internal **nid** registers.

4 Conclusions

The PRC has been fully designed using a .8μm, three-metal process and Epoch design tools from Cascade Design Automation; the chip is scheduled for fabrication in the summer of 1995. The physical and timing specifications of the PRC are shown in Table 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.

Verilog simulations were used to test a single PRC, with the outgoing links connected to the reception ports, under random and contrived workloads. Network-level simulations were also performed with a separate simulator incorporating a cycle-level model of the PRC [20]. After fabrication, the PRC will be tested using an HP 82000 tester, using scan chains to access the chip's critical state machines. Since all traffic traverses the CTBUS, the chip includes special pins for monitoring this internal bus.

The PRC design emphasizes flexible, low-level support for routing and switching to accommodate application characteristics and performance requirements. The chip provides a unique experimental platform for investigating the subtle interplay between application characteristics and routing-switching performance. In this context, we are investigating software mechanisms for exercising router flexibility to improve application performance.

Parameter	Value
Static power diss.	0.1 W
Dynamic power diss.	1.9 W
Transistors	489,572
Size	$145mm^2$
Pins	240

(a) Physical specifications

Component	Clocking	Peak bandwidth
Transmitters	15 MHz, synch	150 Mbits/sec
Receivers	15 MHz, asynch	150 Mbits/sec
Control interface	10 MHz, asynch	N/A
Memory interface	20 MHz, synch	80 MBytes/sec
Internal switch	30 MHz, synch	120 MBytes/sec

(b) Timing specifications

Table 3: PRC specifications

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

[4] 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.

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

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

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

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

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

[14] S. Konstantinidou and L. Synder, "The Chaos router," *IEEE Trans. Computers*, vol. 43, pp. 1386–1397, December 1994.

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

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

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

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

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

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