MITSUBISHI ELECTRIC RESEARCH LABORATORIES
http://www.merl.com

Cranium Network Interface: Architecture and Implementation

Neil McKenzie

TR98-14 December 1998

Abstract

As the performance of networks and processors increases, the performance of the network interface becomes an increasingly significant bottleneck in the overall performance of the parallel or distributed system. The bottleneck persists because it is fundamentally difficult to design and construct network interfaces. We seek to reduce the difficulty by developing a generic network interface template architecture called Cranium. Key aspects of this architecture are hardware support for argument checking, data movement (DMA) and efficient handlers for both short messages and large messages. Cranium provides protected direct access to user-level programs. Cranium is also compatible with networks that may deliver packets out-of-order; there is zero penalty for unordered packets. We introduce the Cranium architecture and describe a related implementation project called ChaosLAN.

MERL – A MITSUBISHI ELECTRIC RESEARCH LABORATORY
http://www.merl.com

Cranium Network Interface: Architecture and Implementation

Neil R. McKenzie

TR-98-14 October 1998

Abstract

As the performance of networks and processors increases, the performance of the network interface becomes an increasingly significant bottleneck in the overall performance of the parallel or distributed system. The bottleneck persists because it is fundamentally difficult to design and construct network interfaces. We seek to reduce the difficulty by developing a generic network interface template architecture called Cranium. Key aspects of this architecture are hardware support for argument checking, data movement (DMA) and efficient handlers for both short messages and large messages. Cranium provides protected direct access to user-level programs. Cranium is also compatible with networks that may deliver packets out-of-order; there is zero penalty for unordered packets. We introduce the Cranium architecture and describe a related implementation project called ChaosLAN.

1. First printing, October 26, 1998

1 Introduction

The network interface ties together many disparate levels of a complete parallel or distributed computer system. At the hardware level, it ties together the processor, the memory and the network router. It provides services for the operating system: argument checking, protection, atomicity and restartability. It provides a medium of communication for application programs. It provides a software interface to the network for application programmers and the authors of run-time libraries and compilers. It is therefore not surprising that network interface design is difficult. The complexity of integrating all the levels of a parallel or distributed computer system makes the design of the network interface a daunting task. A lack of design skill at any of these levels will result in a network interface that delivers poor performance. Furthermore, new network interface designs often fail to build upon previous experience because earlier efforts are too closely tied to the particular implementation of the processor or network.

The intrinsic difficulty of network interface design motivated us to develop a template architecture for network interfaces, called Cranium [1, 2, 3]. We began by examining the network interface strategies in existing designs, and developing abstractions that are independent of any particular compiler, programming language, operating system, processor instruction set, network or hardware implementation technology. From these abstractions we synthesized the template network interface architecture. The goal was to create the simplest network interface design that provides all of the necessary functionality, one that provides the best possible communication performance over a wide range of possible uses and environments.

In the remainder of this introduction, we examine the techniques and benefits of hardware support in the network interface for message passing, and discuss one of the properties, *protocol processing*, in depth. We discuss the Cranium architecture in Section 2. In Section 3 we describe an implementation project that is based on the Cranium network interface architecture.

1.1 Hardware support in the network interface for message passing

To make message passing efficient, a network interface design provides direct hardware support for many aspects of the message passing system. Three important types of support that have been explored in many research and commercial network interfaces are support for argument checking, data movement and protocol processing, described as follows.

- Argument checking is a significant source of software overhead in a message passing system. If argument checking is performed only in software, it is by code that runs at supervisor level, as part of the operating system (i.e. in a device driver). All interactions between the application program and the operating system involve hundreds or thousands of processor instructions and are fundamentally slow. Support for argument checking in the network interface hardware can dramatically reduce this source of overhead.
- Hardware support for data movement is called Direct Memory Access (DMA). DMA reduces the transfer time per message word for two reasons. First, DMA can take advantage of a fast

pipelined memory bus mode called burst mode. Second, DMA allows packets to be sent or received while the processor attends to other tasks. This ability to overlap communication with computation can greatly reduce the overhead of communication. If the network interface does not provide DMA, then the data movement style is called PIO (programmed input/output) in which all data are moved using processor instructions. Some network interfaces support both DMA and PIO.

- Network interfaces can also provide direct support for message protocol processing. A *message protocol* [4] is an agreement between the sender and the receiver concerning the size, format and sequence of the message. Supported protocols run efficiently. However, supporting every conceivable protocol in hardware is impossible in practice. The best strategy for the designer of the network interface is to identify the most common message protocols and support them directly, and emulate the others in software.

The challenge to the designer of the network interface hardware is to manifest all three types of support, ensure that the software layers of the message passing system are able to take advantage of these hardware support features correctly and efficiently, yet avoid designing a circuit that is needlessly complicated and difficult to construct.

Hardware support for argument checking and data movement have been covered extensively in the literature on network interfaces. However, hardware support for protocol processing has been less well documented, which motivates us to address it in the following section.

1.2 Protocol processing

To make the network interface as efficient as possible, it must support message passing protocols directly in hardware. A message passing protocol involves the interaction of data movement, notification and dispatch and management of buffers. Figure 1 describes a taxonomy of strategies for the receive interface. Under DMA there are two options. If DMA is *processor-initiated*, then it is like PIO: notification occurs first, then the data is moved (i.e. DMA is dispatched). If the DMA style is *automatic-receive*, the data movement occurs first, and the processor is optionally notified afterward.

Within automatic-receive DMA there are two styles of protocol support: buffered and unbuffered. Under the buffered protocol, incoming packets are placed into buffers managed by the receiver, such as a hardware FIFO or a ring-queue in the main memory of the processing node. Under the unbuffered protocol, data from incoming packets are passed directly into pre-allocated locations in memory, whose destinations are specified by the sender. Most network interfaces that support message passing (e.g. using the Intel NX message passing interface [5]) support a buffered protocol. Two examples of support for the unbuffered protocol are the Hamlyn network interface from HP Labs [6] and the SHRIMP network interface from Princeton [7]. The terminology used by the Hamlyn team for the unbuffered protocol is *sender-managed* communication. Network interfaces that support remote-write, such as the NI in the Cray Research T3E [8], can also be considered to support the unbuffered protocol.

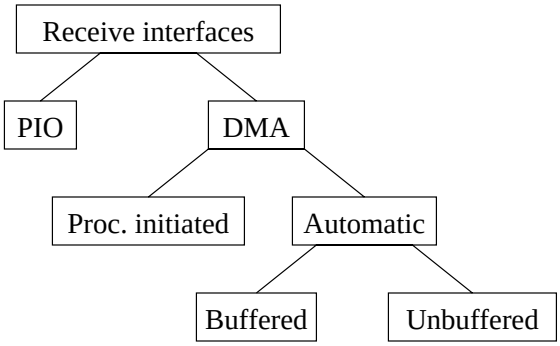

Figure 1: Taxonomy of the receive interface

The principal advantage of the unbuffered protocol is the ability to filter notification information. The network interface passes a notification to the processor only when an interesting packet has been received, such as one that denotes the end of a message. The overhead of notification can be significant, on the order of tens to hundreds of processor instructions. Therefore, any notification that is not passed on to the processor reduces the overhead of communication. The reduction in this type of overhead is the most dramatic when packets are small and messages are large.

It is possible to address the lack of proper protocol support in the network interface by using software-only techniques such as *protocol compilers* [4] and *active messages* [9]. These types of software-only techniques can provide a significant savings in overhead over previous implementations of message passing libraries. However, they cannot solve the performance problems inherent in the network interface hardware. In general, network interface designs implement only one message protocol directly. Other message protocols must be realized by software emulation. Generally speaking, the buffered protocol works well only for small messages, and the unbuffered protocol works well only for large messages. Since most programs involve some kind of mix of small and large messages, all messages of the "wrong type" will be inefficient.

The tension between low latency support and high throughput support motivates the use of two different mechanisms to deal with the two cases independently. The challenge is to provide a unified approach that avoids needless complexity.

2 The Cranium network interface architecture

The initial motivation for Cranium came from a requirement to design a high-performance companion interface to the Chaos network router [1, 2, 10]. The Chaos router has two interesting attributes. It routes small fixed-size packets, using a payload the size of a processor cache-line (e.g. 32 bytes). It also uses adaptivity to improve its throughput and reduce its average latency; as a result, packets may overtake one another in the network, resulting in out-of-order arrival. The name Cranium comes from the acronym for Chaos Router Autonomous Network Interface for User-level Message

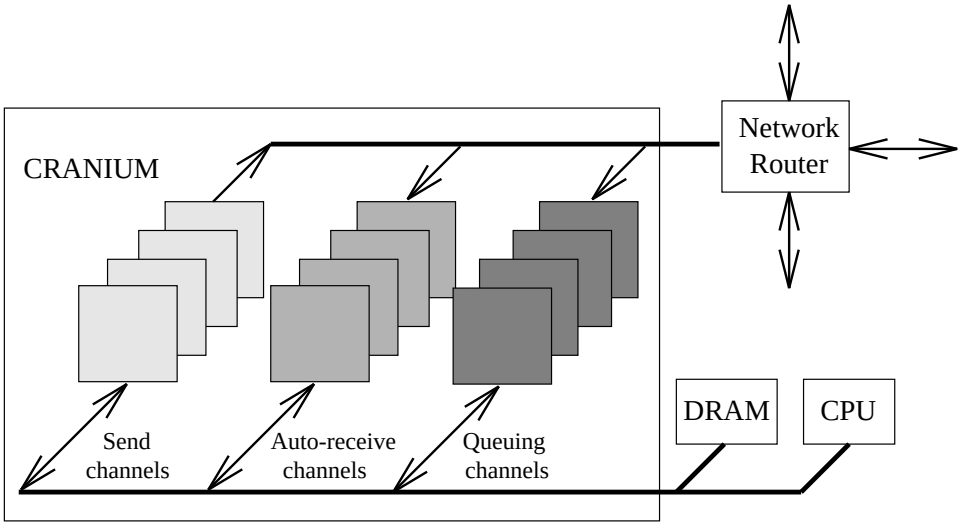

Figure 2: Cranium architecture

passing. However, the Cranium approach is broadly applicable to many different network routers.

Cranium is activated by channel commands issued by the processor. Cranium then autonomously schedules and executes its channel operations independent from and concurrent with processor execution. Cranium is multi-threaded in the sense that multiple message commands can be in progress simultaneously. Channel registers are loaded and stored directly using memory-mapped read and write operations. The overhead of sending and receiving messages is minimal, on the order of a few user-level instructions.

Figure 2 is a block diagram showing the structure of Cranium, which consists of three channel groups: send channels, auto-receive channels and queuing channels. Each channel represents a complete context for a message, including the physical address of the local message buffer, the remote node name, the number of packets to send or receive, and transfer completion status. Fields in the packet header determine which channel group and channel becomes the handler for the packet payload.

The send channel group represents the send interface. The receive interface consists of both the set of queuing channels and the set of auto-receive channels. Each queuing channel manages a separate ring-queue. Separate queues are maintained for the user and the operating system, and there is also an error queue. The send and auto-receive channels support DMA transfers up to the size of an MMU page. The send channels convert long messages into separate packets; the auto-receive channels re-assemble these packets into messages.

2.1 Send channels and auto-receive channels

The auto-receive channels implement the unbuffered protocol. The send channels and the auto-receive channels are symmetric; the programmer model for both sets of channels is nearly identical. At any time, a send channel or auto-receive channel is either in the idle state or in an active state where it is transmitting or receiving packets. Each activation of a channel initiates a block transfer of up to an MMU page. If a cache line is 32 bytes and an MMU page is 8K bytes, then there are up to 256 packets per channel command. Each send channel and auto-receive channel maintains a packet count, a node ID and a physical buffer base address. The packet counter in the send channel is copied into the sequence number field in the packet header. For each packet that is sent, the physical address of the payload data is computed by adding the physical base address to the packet counter times the size of a cache line. The auto-receive channels place incoming packet data into memory in the proper location by the same offset from the receiver's physical base address, thereby providing compatibility with networks that may deliver packets out-of-order. The processor can load the packet counter value from the send and auto-receive channels to determine completion status of the transfer. Optionally, Cranium can interrupt the processor when the counter reaches zero, indicating that the transfer has completed. Note that any send channel can send to any auto-receive channel at any node. A restriction is that an auto-receive channel must be activated before the send channel starts sending packets to it. A protocol error is signaled when a packet is handled by an inactive auto-receive channel, or if one or more of the fields in the packet header do not match what is expected by the auto-receive channel. The culprit packets are then deposited into the error queue.

2.2 Queuing channels

The queuing channels implement the buffered protocol. Packets destined for a particular queuing channel are placed into the corresponding ring-buffer in memory in the order they are ejected from the network. Queue memory is implemented using main memory; queue buffers are locked into the physical memory map. User programs access queue memory directly using load and store operations. By using main memory to hold the ring-queues, it becomes possible to reduce the size of the hardware FIFO in the receive interface. DRAM memory is much cheaper and lower power than comparably sized SRAM needed in the FIFO. Main memory storage also allows the operating system to change the size of queue memory when the user program is started, rather than requiring the network interface hardware to be reconfigured. Like the auto-receive channels, data are moved first by DMA and the processor is notified afterward. Unlike the auto-receive channels, every packet that arrives into the queuing channels sends a notification to the processor.

2.3 Protection

Cranium provides the necessary features to implement protected, safe user-level access: address mapping, logical node identifiers, atomic packet injection, network drain and guaranteed delivery of operating system messages. Cranium does not allow the user program to write the physical base

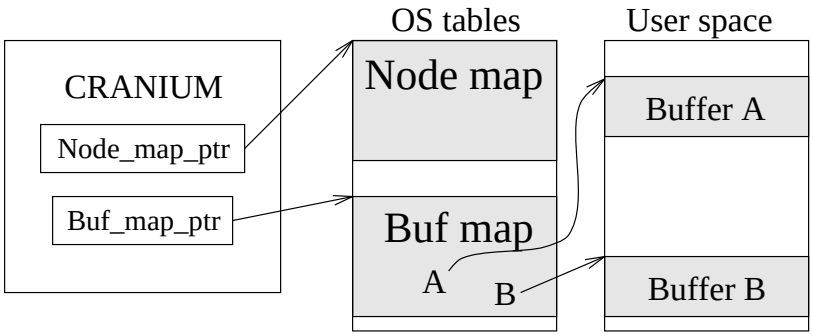

Figure 3: Protection for safe user-level access via mapping tables

address of the send or auto-receive channel directly. Similarly, the user program cannot load the node ID directly, because there may be destination nodes that the user should not be able to access. In each case there is a level of indirection provided by mapping tables, one for node IDs and one for buffer addresses. Each mapping table occupies the node's DRAM in a protected location, accessible only to the operating system and pinned into physical memory. User programs can specify only the indices into these tables. Cranium performs a table lookup in each case. If the table entry contains a valid value, then the translation succeeds. If the table entry contains an invalid value, then the translation fails and the transfer is canceled. It is therefore impossible for the user program to cause Cranium to send packets that reference ill-formed buffer addresses or node IDs.

Figure 3 shows the two mapping tables: the node map and the buffer map. The network interface contains a pair of hardware registers that contain the base physical addresses for each mapping table. `Node_Map_Ptr` points to the base of the node map, and `Buf_Map_Ptr` points to the base of the buffer map. An entry in the node map is the physical identifier for a remote node, or zero to indicate an unmapped node entry. An index into the node map (i.e. the offset from the base address of the table) is called a *node handle*. An index into the buffer map is called a *buffer handle*. In order to construct entries in the node map or the buffer map, the user program must call the operating system and pass a user virtual address or node identifier as an argument. The OS performs the mapping and returns the handle to the user program. When a buffer is mapped, the OS must also pin its page into physical memory. For reasons of efficiency, it makes sense for the programmer to map all message buffers during an initialization phase of the user program, so that all subsequent network commands are performed at user level. Figure 3 shows two user buffers, A and B, that the user program has registered with the OS and thereby entered into the buffer map. When the user program wishes to send a message to node X using buffer A, it passes X's node handle and A's buffer handle to a Cranium send channel. The interface performs a table lookup on each handle to verify that their corresponding table entries contain valid data, and places the results of the lookup operations into the send channel's node ID and physical buffer address registers.

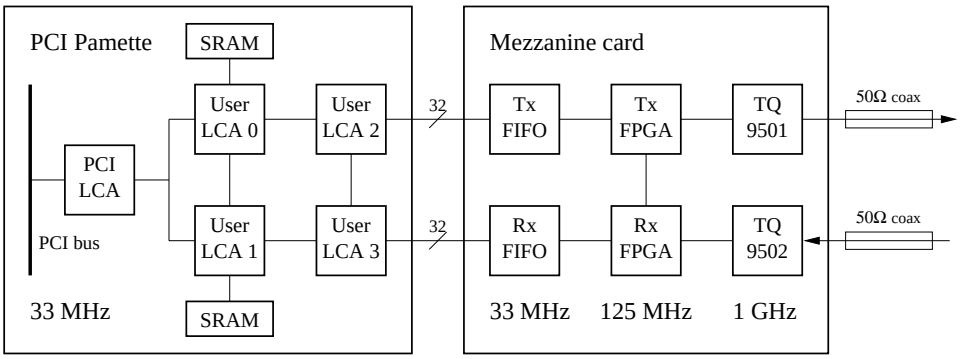

Figure 4: Organization of the PCI Pamette card and the Fibre Channel mezzanine card

3 ChaosLAN implementation project

Extensive software simulation studies demonstrate the efficacy of the Cranium approach [2]. However, the best simulation is realized by a physical implementation, because it accounts for artifacts that are very difficult to simulate, such as operating system interrupt response overhead. The ChaosLAN project [3] represents a prototype implementation of a gigabit LAN using chaotic routing and the Cranium network interface architecture.

The goals of the ChaosLAN network interface are to demonstrate the high performance capability using only readily available subsystems and components; no custom silicon is involved. The ChaosLAN network interface is based on a generic PCI card called the PCI Pamette [11], a product of Digital's System Research Center. The Pamette contains five Xilinx Logic Cell Arrays (LCAs); one LCA contains the PCI interface and the other four are user-defined. All of the Cranium circuitry is realized in the Pamette's four user LCAs and its on-board SRAM.

The gigabit LAN linkage is based on the Fibre Channel standard; we are using commodity Fibre Channel transmitter and receiver chips from TriQuint (TQ9501 and TQ9502, respectively). The TQ9501, TQ9502, four FIFOs and three Altera FPGAs occupy a small mezzanine card that plugs into the Pamette. The Altera FPGAs execute the 8b/10b encode and decode operations according to the Fibre Channel specification. Figure 4 is a block diagram that shows the structure of the Pamette and mezzanine card combination.

With two network interfaces housed in two workstations, it is possible to construct a simple point-to-point network. To construct networks with more than two nodes, the ChaosLAN team is also developing a network switch based on a custom chip called the Chaos router chip. Our prototype switch uses 16 Chaos chips to connect up to 16 workstations. Larger networks can be constructed by cascading multiple switches.

The ChaosLAN mezzanine card is fully functional. We will provide performance measurement data on the network interface in a follow-up document. Unfortunately, the development of the router board (see [3]) has been halted due to a lack of funding.

4 Summary

We motivate and describe a template network interface architecture called Cranium. The architecture was developed by studying a large number of existing network interfaces in both research and commercial systems. The key observations were that hardware support was needed in three critical areas: argument checking to reduce OS overhead, data movement (DMA) and support for both the buffered and unbuffered message protocols. Traffic in networks tends to be bi-modal: most messages are small, but a substantial fraction of packets are associated with a few large messages. The buffered message protocol works best with small messages and the unbuffered message protocol is most efficient with large messages. We also introduce a low-cost prototype hardware implementation of Cranium called the ChaosLAN network interface. Our goal is to demonstrate that the concepts in Cranium can be implemented inexpensively and deliver excellent performance for both large and small messages.

Acknowledgments

Thanks go to my colleagues on the ChaosLAN implementation team: Kevin Bolding, Chris Fisher, Carl Ebeling and Larry Snyder.

References

[1] Neil McKenzie, Kevin Bolding, Carl Ebeling and Lawrence Snyder. Cranium: an interface for message passing on adaptive packet routing networks. *Proc. of Parallel Computer Routing and Communication Workshop*, Seattle WA, May 1994, Springer-Verlag, pp. 266-280.

[2] Neil R. McKenzie. *The Cranium network interface architecture: support for message passing on adaptive packet routing networks.* PhD dissertation, technical report UW-CSE-TR 97-02-04, University of Washington, February 1997.

[3] Neil McKenzie, Kevin Bolding, Carl Ebeling and Lawrence Snyder. ChaosLAN: design and implementation of a gigabit LAN using chaotic routing. *Proc. of Parallel Computer Routing and Communication Workshop*, Atlanta GA, June 1997, Springer-Verlag, pp. 211-223.

[4] Edward W. Felten. Protocol compilation: high-performance communication for parallel programs. PhD dissertation, University of Washington, Dept. of CSE, Sept. 1993, UW-CSE-TR 93-09-09.

[5] Paul Pierce. The NX message passing interface. *Parallel Computing* 20(4), April 1994, pp. 463-80.

[6] Greg Buzzard, David Jacobson, Scott Marovich and John Wilkes. Hamlyn: an high-performance network interface for sender-based memory management. *Proc. of Hot Interconnects III Symposium*, Stanford University, Palo Alto, CA, August 1995. Also available as technical report HPL-95-86, Hewlett-Packard Company, HP Labs, Computer Systems Laboratory, July 1995.

[7] Mattias A. Blumrich, Kai Li, R. Alpert, Cezary Dubnicki, Edward W. Felten and J. Sandberg. A virtual memory-mapped network interface for the SHRIMP multicomputer. *Proc. of the 21st International Symposium on Computer Architecture*, Chicago IL, April 1994, pp. 142-153.

[8] Steve Scott. Synchronization and communication in the T3E multiprocessor. *Proc. of ASPLOS VII*, Cambridge MA, October 1996, pp. 26-36.

[9] Thorsten von Eicken, David E. Culler, Seth C. Goldstein and Klaus E. Schauser. Active messages: a mechanism for integrated communication and computation. *19th Annual International Symposium on Computer Architecture,* May 1992, pp. 256-266.

[10] Kevin Bolding. *Chaotic routing: design and implementation of an adaptive multicomputer network router.* PhD dissertation, University of Washington, Dept. of CSE, Seattle WA, July 1993.

[11] Mark Shand et al. The PCI Pamette V1. World Wide Web site, `http://www.research.digital.com:80/SRC/pamette/`.