Cranium: An Interface for Message Passing on Adaptive Packet Routing Networks*

Neil R. McKenzie, Kevin Bolding, Carl Ebeling and Lawrence Snyder

University of Washington
Department of CSE, FR-35
Seattle WA 98195, USA

Abstract. Cranium is a processor-network interface for an interconnection network based on adaptive packet routing. Adaptive networks relax the restriction that packet order is preserved; packets may be delivered to their destinations in an arbitrary sequence. Cranium uses two mechanisms: an automatic-receive interface for packet serialization and high performance, and a processor-initiated interface for flexibility. To minimize software overhead, Cranium is directly accessible by user-level programs. Protection for user-level message passing is implemented by mapping user-level handles into physical node identifiers and buffer addresses.

1 Introduction

Scalable multicomputer architectures have been converging on a standard organization with four elements: a workstation microprocessor, main memory based on dynamic RAM, a point-to-point interconnection network and a processor-network interface. Both the microprocessors and DRAM chips have become inexpensive and widely available. Multicomputer architects have therefore focused their efforts on the design of a fast network router and an efficient interface.

This report describes Cranium[1], a processor-network interface that supports *message passing* protocols on mesh networks that use *adaptive packet routing* algorithms. The advantage of explicit message passing over the implicit *non-uniform memory access* (NUMA) model is the intrinsic ability to overlap communication and computation. Augmenting the NUMA model with multithreading, scoreboarding, or prefetching makes it possible to overlap communication and computation, but at the cost of much greater hardware complexity than that of a message passing interface.

In the remainder of the introduction, we first describe the benefits and pitfalls of adaptive routing. Then we discuss the tradeoffs of different approaches to network interface design. Finally we summarize our requirements for Cranium.

* This research was supported in part by Office of Naval Research grant N00014-91-J-1007 and National Science Foundation grant MIP-92-13469.

[1] Chaos Router Autonomous Network Interface for User-level Messages

1.1 The case for adaptive routing

It is highly desirable to apply the principles of adaptive packet routing to a practical and realizable multicomputer system.

- Simulation studies show that at high network loads, adaptive routing offers higher bandwidth and lower latency than the standard dimension-order oblivious algorithm, for a given fixed network packet size [1, 2, 4].
- Adaptive routing improves *fault tolerance*, the ability to operate in the presence of failed components [3].
- It is feasible to construct fast, economically competitive adaptive routers, such as the Chaos router [4].

In the network architectures we consider in this article, network packets have bounded length. Therefore, a message that is longer than the maximum network packet size cannot be directly injected into the network; it must be converted into a sequence of packets by the sending node, which then injects them into the network. After the network delivers these packets, the original message is reassembled at the receiving node. A potential drawback to adaptive network routing is the property that packets can overtake other packets in the network. Packets can therefore arrive in an arbitrary order, even if there is only one node actively sending packets and one receiving them. This property is called *out-of-order delivery*. The problem is that there exist message passing protocols that depend on the *in-order delivery* property of the oblivious routers. One common technique is to send a number of data packets followed by a control packet to mark the end of the data. After the receiver detects the arrival of the control packet, it can safely assume that all the data packets have arrived. Under out-of-order delivery, this protocol will not work reliably.

There are two approaches for dealing with out-of-order packet delivery. One approach is to use protocols that assume in-order delivery, and augment *only* the interface hardware to convert the out-of-order stream of packets into a time-sorted stream. Such an interface is required to identify and contain early-arriving packets in a separate buffer space; in effect, it mandates double-buffering, and increases the average packet latency. This kind of interface is inherently expensive, complicated, and slow; therefore it is not practical to implement.

The other approach is to modify the low-level software protocol: instead of waiting for the arrival of the control packet, the receiver simply counts the number of arriving data packets. To accelerate this modified protocol, the network interface hardware provides a set of status bits to indicate to the software that the nth data packet has arrived. Packet counting allows reliable operation that is independent of packet arrival order, and the hardware support in the network interface can be implemented inexpensively; it is the approach we assume in the remainder of this article.

1.2 Processor-initiated interfaces vs. automatic-receive interfaces

Broadly, the network interface has two strategies for handling incoming packets: *processor-initiated* and *automatic-receive*. In the processor-initiated strategy, the processor polls or is interrupted for each packet that arrives from the network. The processor accepts data from network via programmed I/O, optionally, it initiates a direct-memory-access (DMA) operation. In either case, the processor must *rendezvous* with the arriving packet. In the automatic-receive strategy, DMA is required; an intelligent network controller on the receiving node places the packet data in memory automatically. Importantly, interaction with the processor is removed from the critical path of placing the packet in memory. The choice of a processor-initiated interface, an automatic-receive interface, or a combination of the two approaches depends on the network architecture and the tradeoffs among generality, performance and complexity.

- *Generality: processor-initiated is more general.* Because a processor-initiated interface interacts with the processor when a packet arrives, the processor can execute an arbitrary calculation to decide where to put the packet data. An auto-receive interface places the packet directly into memory based on the packet header information alone, without involving the processor.
- *Performance: automatic-receive has better performance.* If we assume the packet-counting scheme outlined in the previous section, then the software overhead for an auto-receive interface is paid *once per message* instead of *once per packet*. The savings in overhead can be substantial if the packet size is small and the message is large. Conversely, the performance advantage may be nil for minimum size (single packet) messages, which are often used for synchronization and global combining operations.
  An auto-receive interface consumes arriving packets right away, whereas a processor-initiated interface incurs a lag time for the processor to service the network. The lag time can vary, including the time it takes the processor to process an interrupt, as well as the time it takes to execute some arbitrary code to compute a destination address for the packet data (e.g. a memory allocator). During the lag time, if there is a subsequent burst of packet traffic destined for arrival at this node, packets are likely to back up into the network. Long lag times at one destination node can actually degrade the performance of the entire network under moderate to heavy traffic loads.
- *Complexity: processor-initiated is simpler.* However, an overly simple solution will limit its performance. Subcircuits such as a DMA controller, a content addressable memory (CAM) and a hardware FIFO are common sources of complexity in the network interface. DMA is required in an auto-receive interface; it is optional in a processor-initiated interface. CAMs are commonly used in an auto-receive interface for validation of the packet header and/or address translation. CAMs are not common in a processor-initiated interface because validation and translation are usually handled in software. A hardware FIFO on the receiving side of the interface helps decouple the network link from the memory bus, and is desirable for containing bursts of packet data that would otherwise back up into the network. For good performance, the receive FIFO needs to be large if the bandwidth of the network link is larger than that of the local memory bus, or if the lag time to service the network is large. Because an auto-receive interface consumes

packets right away, it may permit the use of a much smaller FIFO.

1.3 Requirements for Cranium

The requirements for Cranium are threefold[2]. The first requirement is to provide both an *automatic-receive* interface and a *processor-initiated* interface. Automatic-receive is important for supporting a packet-counting protocol compatible with out-of-order packet delivery, serializing packet data in memory, and maintaining high-bandwidth communications. The processor-initiated interface adds flexibility for supporting synchronization, global combining and other situations where message latency is critical but bandwidth is less important.

The second requirement is to *minimize the software latency* for sending and receiving messages. Designers of the older generation of network interfaces considered message passing to be an operating system service. The latency of message passing on a system with *direct user-level access* to the network is reduced by one to two orders of magnitude compared with a similar system-level interface. Felten [5] outlines the basic strategy for safe user-level communication that Cranium supports – system partitioning, hardware validation for message destinations, gang scheduling, saving and restoring the network state, and separate user and kernel level communication.

The third requirement is the ability to *detect faults* in the network at two levels. At the low level, packets include a cyclical redundancy code (CRC) to validate the packet header and payload, to permit the interface to detect corrupted packets directly. At the high level, there must be specific support for the Chaos network. Chaotic routing has the property that delivery time of a particular packet cannot be bounded as long as the network continues to accept the injection of new packets. An absent packet may indicate either a network failure or a heavily loaded network. These two cases are distinguished with the help of a software timer. If the timer expires before all expected packets have arrived, then the software can issue a *drain* command to the network via the interface, which blocks further packet injection. A network drain ensures an upper bound on delivery time for all packets in transit. Either the tardy packets eventually arrive and computation can proceed, or network failure is detected. In the latter case, the interface issues a command to the network for self-diagnosis and possible reconfiguration. See Bolding and Yost [3] for further information on fault detection and recovery in the Chaos network.

Here is the organization of the rest of this report. Section 2 introduces the architecture of Cranium. Section 3 discusses run-time issues and high-level message passing protocols. Section 4 discusses related work on network interface design. Section 5 discusses hardware implementation issues. Section 6 shows our plan for continuing work in this area, and Section 7 is a summary.

---

[2] Although the principles underlying the design of Cranium are presented here in the context of the Chaos router, keep in mind that they are general and apply to all routing algorithms that deliver packets out of order. Features of Cranium that are specific to Chaos are duly noted.

2 Architecture

Cranium is *autonomous*; it schedules and executes its work independently and concurrently with processor execution. Cranium is *multithreaded*; multiple message commands from a single node can be in progress simultaneously. Cranium has *user-level access*; user programs access network interface registers directly via load and store commands. The overhead of sending and receiving messages is minimal, on the order of a few user-level instructions.

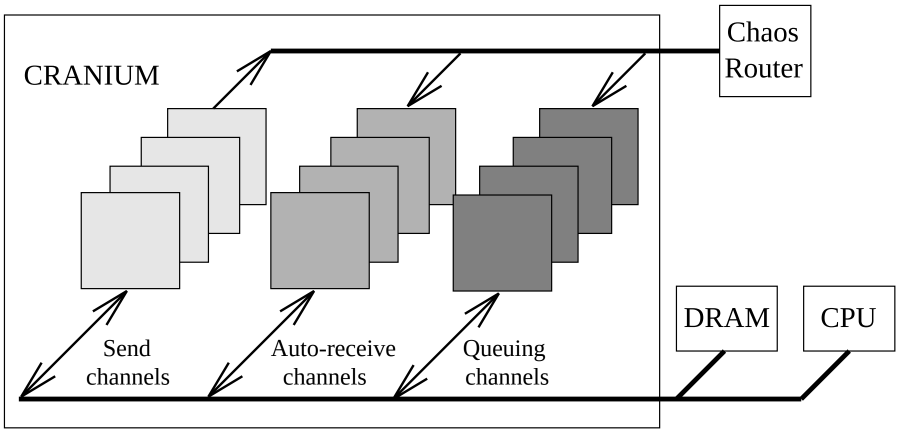

Fig. 1. Cranium architecture

The basic architecture for Cranium is shown in Figure 1. The programming model is a small fixed set of message *channels*. There are three kinds of channels. Messages are sent using the *send channels*. The *auto-receive channels* represent the automatic-receive interface, and the *queuing channels* represent the processor-initiated interface. 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.

Messages are broken up into packets by the interface at the source node and then reassembled by the interface at the destination node. A packet consists of an 8 byte header and a 32 byte field for payload (user data). The packet header consists of the destination node identifier (2 bytes), source node identifier (2 bytes), a packet sequence number (6 bits), the send channel number (4 bits), the auto-receive channel number (4 bits), the user/system flag (1 bit), the queue flag (1 bit), and two bytes for the CRC.

2.1 Overview of send channels and auto-receive channels

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. The `NI_send()` command activates a send channel; likewise, the `NI_recv()` command activates a auto-receive channel. There are 16 send channels and 16 auto-receive channels. Each channel activation represents a block transfer of up to 2K bytes (64 packets, 32 bytes per packet). Send channels tag each outgoing packet with a sequence number, which is used by the auto-receive channels to place incoming packet data into memory in the proper serial order. The processor can test the status of these two types of channels to determine completion status of the transfer. Optionally, Cranium can interrupt the processor when the transfer completes.

2.2 Buffer mapping and node mapping

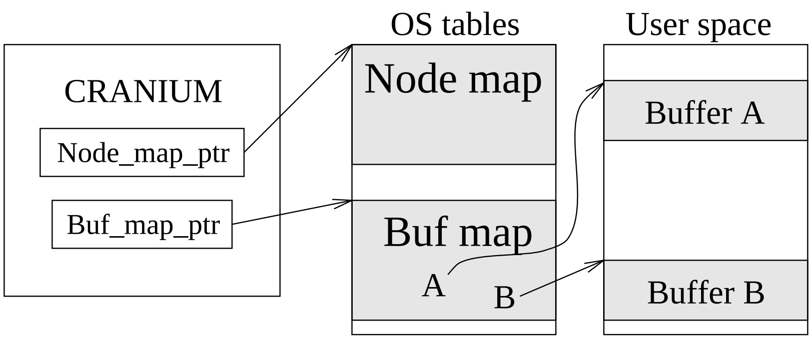

Fig. 2. Protection for safe user-level access via mapping tables

Because Cranium's channels are directly available to the user, there are hardware mechanisms for ensuring safe user-level access. Otherwise, a user program might accidentally (or maliciously) interfere with network operation, by spoofing or intercepting messages meant for the operating system, or sending or receiving messages at memory locations outside the user's protection domain. Cranium accomplishes this goal using *mapping tables* that are shared with (and privileged to) the operating system. Figure 2 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, called `Node_Map_Ptr` and `Buf_Map_Ptr`. 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. an offset from the base address of the table) is called a *node handle*. An entry in the buffer map is the physical address of a 2 kilobyte user-space buffer. An index into the buffer map is called a *buffer handle*, which is returned to the user process at the completion of a `NI_buffer_map()` operating system call. When a buffer is mapped into the network interface, its physical page is also pinned in memory. It makes sense for the programmer to locate all buffer mapping system calls at the beginning of the user program, so that all subsequent network commands are performed at user level.

The buffer mapping feature is very flexible. There is no distinction between a send buffer and a receive buffer; the same buffer can be used to collect a message from one node and then pass it to another node without copying. A single buffer can be used for *multicast*, to send several messages at once to a set of nodes.

Figure 2 shows two user buffers, A and B, that the user program has registered with the operating system 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. An advantage of using a handle for protection and mapping is that it reduces the amount of data that the user program needs to pass to Cranium. There are 4096 entries for each table, enough to map 8 MB of message buffer space. Handles are only 12 bits instead of 32 or more bits usually required for the address of a user buffer.

2.3 Queuing channels

The queuing channels implement the processor-initiated interface. Packets in the queue are not serialized by the interface; they appear 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. There are several advantages to storing packets in main memory rather than using a deep hardware FIFO[3].

- Main memory is constructed using inexpensive dynamic RAM chips; hardware FIFOs tend to use expensive static RAM.
- The processor is notified *after* the interface places the packet data into main memory, rather than before. The user program can access packet data immediately after notification instead of waiting for the transfer to complete.
- The operating system can change the sizes of the queues dynamically at user program load time, rather than requiring a hardware reconfiguration. The operating system needs to ensure that queue overflow rarely occurs, by making the frequently accessed queues of sufficient size to handle even large bursts of traffic.

There are four queuing channels: the user queue, the user error queue, the system queue and the hardware error queue. Packets that have the queue bit in the packet header enabled are routed to the user queue. The user error queue is used for packets that signal a protocol (soft) error (see Section 2.4). Both the user queue and the user error queue buffers are mapped into user space.

Figure 3 shows the organization of queuing channel memory. For each queuing channel, the channel context is a set of pointers into main memory, called the start pointer (SP), the end pointer (EP), and a pointer to the current (most

[3] This use of system memory helps reduce the size of the receiver's hardware FIFO, but it is still necessary for decoupling the network from the memory bus.

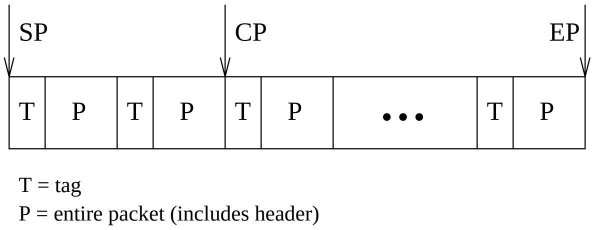

Fig. 3. Organization of circular buffer in DRAM for queuing channel

recently arrived) packet buffer (CP). Queue memory is organized as a circular buffer that uses a simple producer-consumer protocol, with Cranium as producer and the processor (i.e. the user program) as consumer.

At initialization, CP and SP both point to the start of the queue buffer. When a packet arrives, Cranium performs a read-modify-write cycle on the tag field (T). If the tag field was 0, then Cranium atomically writes a nonzero value[4] to the tag field, writes the entire packet (including header) into the packet field (P), and advances CP to point to the next tag field. If the tag field was nonzero, then Cranium signals queue overflow and interrupts the processor (see Section 2.4). The user program detects the presence of a packet by polling the tag field. If the packet is present, the user program reads the packet information, processes it, then sets the tag field to zero to release the space back to Cranium.

2.4 Error handling in Cranium

When Cranium encounters a run-time error, it sends an interrupt to the processor. There are two kinds of errors: command errors and packet errors. Command errors are caused by an illegal access to Cranium by the user program, such as trying to initiate a new command on an already active channel, passing the handle of an unmapped node or buffer, or accessing a privileged channel. Packet errors arise when a packet arrives from the network, but Cranium cannot process it normally, so the operating system must intervene. There are several different kinds of packet errors and corresponding error handling mechanisms.

- *Network hardware error.* Symptom: CRC mismatch on incoming packet. Action: the packet is sent to the hardware error queue.
- *Protocol error.* Symptom: either the auto-receive channel for the packet is inactive, or the packet header does not match the expected information in the channel context[5]. Action: the packet is sent to the user error queue.

[4] In our implementation, we write a 64-bit timestamp into the tag field to represent the nonzero value (the least significant bit is always 1). Timestamp information can be used by the user program for performance analysis.

[5] For instance, the source node ID in the packet header does not match the source node ID of the auto-receive channel.

- *Queue overflow.* Symptom: the tag field of the current slot of the indicated queue was nonzero (see Section 2.3). Action: A control bit in the NI selects one of two possible actions: *either* hold packet data and wait for an operating system command to remove the packet, *or* simply drop the packet.

3 Run-time support for message passing

The user queue and the auto-receive channels give the user program a wide degree of flexibility in the implementation of efficient message passing primitives. The user queue is the proper mechanism for funneling packets that arrive without prior arrangement; it provides the best support for synchronization, small data transfers and active message handlers [6]. After two nodes synchronize, the auto-receive channels can be used for high bandwidth communication. The restriction with the auto-receive channels is that the processor on the receiving node must initiate an auto-receive channel transfer *before* the processor on the sending node initiates its corresponding send channel transfer. The user queue and user error queue, like the send and auto-receive channels, are available to the programmer at user level. Because these queues are just shared memory, they are accessed directly by loads and stores without touching network interface registers.

Traditionally, run-time support for message passing primitives has been inefficient for two reasons: the architecture of the network interface makes it necessary for user programs to call the operating system to access the network, and messages arriving at destination nodes are stored first into system buffers and later copied into user-space buffers. Cranium enables the user program to eliminate both sources of overhead. First, since Cranium has direct user-level access, there is no system call overhead. Second, the auto-receive channels can be used to place the data directly in user-space buffers in the proper order without copying the data.

3.1 Anatomy of a block transfer

A block transfer of data consists of the following sequence of operations. We assume that the message buffer addresses have already been entered into the buffer map.

- On the destination node, the processor issues an `NI_recv()` command to a auto-receive channel. The channel is initialized with a node handle and a buffer handle. The network interface performs two table lookups: one on the node handle to get the source node ID, and another on the buffer handle to get the physical destination address for the incoming data.
- On the source node, the processor issues an `NI_send()` command to a send channel to initialize it with a node handle, a buffer handle, and the packet count. The network interface performs table lookups on the node and buffer handles to get the destination node ID and the physical source address for the outgoing data.

– The source interface creates and injects packets into the network, using the send channel context to create the proper packet header for each packet. The channel's sequence number field is incremented after each packet is sent.
– For each arriving packet, the destination interface validates the packet for correct CRC. Next, the interface confirms that the auto-receive channel selected in the packet header is in the active state. The interface then computes the memory address from the base physical address of the channel, offset by the packet sequence number times the payload size (32 bytes). Finally it places the payload data in memory at this address.

Status information is maintained by the interfaces on both the source and destination nodes. As soon as the send channel has finished injecting all its packets into the network, it goes idle. When all these packets arrive at the destination node, the destination's auto-receive channel goes idle.

3.2 Standard message passing example

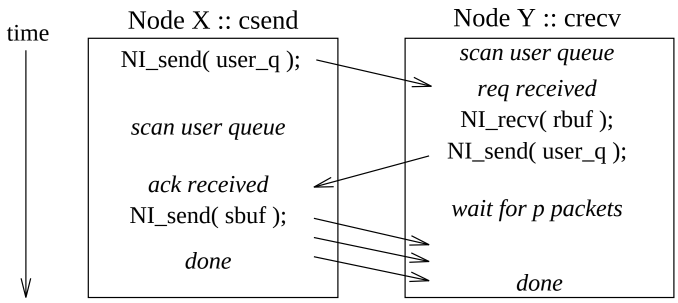

Fig. 4. NX-style message passing example

Most parallel message-passing codes use a standard interface for message passing, such as NX, Express, PVM, and many others [7]. Standard interfaces improve program portability and readability, compared with using the specific message passing primitives provided by the architecture. We now show a simple example of run-time support for Cranium using the semantics of `csend()` and `crecv()`, the most basic method of communication under NX [8]. Figure 4 illustrates how the NX commands are implemented in terms of Cranium commands, for the case of medium length messages (between 32 and 32K bytes). In the figure, italics represent pseudocode, and roman type represents NI commands. In this example, X is the source node and Y is the destination node. We wish to transfer up to $32p$ bytes from buffer `sbuf` on X to buffer `rbuf` on Y, where $1 \leq p \leq 1024$. Node X executes `csend()`; at the same time, node Y executes `crecv()`. The first step is for X to send a packet to Y's user queue to request the block transfer. After Y receives the packet and the message type is matched, Y sets up one or more auto-receive channels, then sends an acknowledge packet back to X's user queue. After X receives the ack packet, X initiates the corresponding number of send channels for the block transfer to Y. After p packets arrive at Y, the message transfer is complete.

For messages smaller than 32 bytes, the block transfer phase is not necessary; the receiver simply copies the small message from queue memory to buffer memory. For messages larger than 32K bytes, there are not enough channels to initiate the entire transfer at once. There must be additional acknowledge packets from Y back to X to denote that it is safe to reuse channels. The loss of performance due to these extra packets is small, because the fraction of bandwidth consumed is only 1/64, and the latency can be hidden by the use of multiple channels.

4 Related work

4.1 Tightly-coupled interfaces vs. bus-based interfaces

A tightly-coupled processor-network interface is one that is integrated onto the same silicon chip as the processor; the instruction set architecture of the processor includes specific instructions for sending and receiving messages. Examples include iWarp [9], MDP (J-machine) [10], and *T [11]. The advantage of tightly-coupled systems is an order of magnitude lower message latency than a comparable bus-based interface. However, there are several disadvantages to this strategy. By placing the interface onto the processor chip, some other functionality must be eliminated or moved off-chip. In the *T design, about 15 per cent of the processor's chip area is dedicated to the network interface [12]. The resulting single-processor performance is less than the fastest possible. Another disadvantage to the tight coupling is the extra amount of engineering effort needed to design and build the interface. Unlike the case for a bus-based interface, this effort cannot be easily amortized across a family of different processors. For these reasons, tightly-coupled interfaces are less able to compete economically and are not as widely used in commercial multicomputer systems as bus-based interfaces. Continuing research in this area may eventually solve these problems. In the meantime, bus-based interfacing remains the most economically viable strategy for implementing multicomputers, using standard workstation processors and memories. Examples of systems using bus-based network interfaces with direct user-level access are CM-5 [13], Shrimp [14], Hamlyn [15] and Cranium.

4.2 Comparison of Cranium with Shrimp and Hamlyn

Cranium is highly similar to the Shrimp processor-network interface [14]. In both systems, messages are sent and received at the cost of only a few user-level instructions. Both systems use a separate phase to map user-space buffers with the operating system and the network interface. Both systems use DMA, rather than the programmed-I/O approach used by the CM-5 interface. Both systems use an automatic receive interface for high throughput. However, Cranium and Shrimp have somewhat different goals. Here are the significant differences between the two designs.

- The automatic receive channels in Cranium are designed to work with a network that delivers packets out-of-order. Shrimp requires in-order packet arrival.
- Cranium's auto-receive channels count the number of packets that arrive. The Shrimp interface does not count packets; furthermore, this information is not even visible at the user level, making it difficult to detect if packets are dropped in the interconnect. Programs running under Cranium can always determine the number of packets that arrive and therefore detect dropped (or late arriving) packets in the Chaos interconnect.
- Once a logical link between two nodes is established under Shrimp, it persists until it is explicitly terminated by the user program. Logical links under Cranium are implicitly broken at the point where all packets regarding that logical link have been sent and received. Under Shrimp, creating and removing a logical link each require an operating system call, whereas Cranium requires only user-mode instructions. However, Cranium requires additional synchronization packets to support long transfers (i.e. the connections need to be created again and again). Thus, their relative performance depends on the behavior of the user program; Shrimp is likely to be more efficient for large transfers where the configuration of links is relatively static, and Cranium is likely to be more efficient for smaller transfers or if the configuration is highly dynamic.
- Cranium is better suited for multicasting than Shrimp is, for two reasons. Cranium has multiple channels to send a set of messages concurrently, whereas Shrimp has only one outgoing DMA context. Shrimp's mapping phase binds a local buffer to a [remote node, remote buffer] pair; Cranium's buffer mapping and node mapping steps are independent and orthogonal. Cranium can send the contents of a single buffer to a set of nodes without copying data or invoking the operating system, which is not possible under the current design of Shrimp.

Cranium is also similar to Hamlyn [15]. Both interfaces are designed to work with a network that deliver packets out-of-order. Both interfaces make use of multiple DMA contexts, called mapping table slots in Hamlyn, similar to Cranium's channels. A proposed implementation of Hamlyn has thousands of slots, compared with the small number of channels we propose for Cranium. All of Hamlyn's communication uses an automatic receive interface; there is no rendezvous interface. However, Hamlyn must still provide the possibility for any node to communicate with any other node. Therefore, $N-1$ slots per node must be preallocated for any-to-any communication, where N is the number of nodes in the multicomputer. Hamlyn therefore requires the system to "burn" a potentially large number of slots for communication that *might* occur during a computation. For these reasons, we believe that Hamlyn is comparable in features and performance to Cranium, but it is relatively complicated and expensive to implement.

5 Implementation issues

There are a number of implementation issues that space does not permit us to cover in depth. Here, we describe a few of these pertinent issues briefly.

The first issue is to select a processor architecture for the multicomputer node. Ideally the processor should have a 64-bit data bus, to let the user process issue a `NI_send()` or `NI_recv()` command in a single memory store operation[6]. It should also have multiprocessor support for cache coherency, with the option of write-update[7]. The cache line size should be 32 bytes to match the size of the packet payload. When a packet arrives, the processor snoops the bus to see if the cache line is mapped into first level cache. The physical address comes from the packet's auto-receive channel in the interface. If snooping on the arriving packet results in a cache hit, then the interface performs a write-update of the packet payload into the first level cache in the processor.

The second issue is a rough estimate of the complexity of an initial implementation of Cranium. The finite state control decomposes into four communicating state machines: a processor-bus interface, a DMA engine, a network-link interface, and a channel scheduler. Here are the approximate memory requirements for each component in Cranium: 100 bits per send channel, 150 bits per auto-receive channel, 100 bits per queuing channel, 200 bits for the channel scheduler, and 320 bits per FIFO stage. Assuming 16 send channels, 16 auto-receive channels, 4 queuing channels, one channel scheduler, and two FIFO stages, the total is about 5K bits. Also required are a 16-bit adder for address calculation, a 64-bit timestamp counter, and CRC logic. An advantage of the channel paradigm is that it eliminates the need for a content-addressable memory, because channels are indexed directly by number.

Cranium and Shrimp have many aspects in common (see Section 4.2), and we anticipate that their relative performance will be similar in many cases. For instance, packet latency under Shrimp (the time to transfer a single packet between two nodes) is on the order of a few microseconds. Actual performance numbers for Cranium will depend on the performance of the network, and the available technology and the amount of optimization in the network interface hardware. An advantage of the mapping table strategy for user-level protection is that a range of implementations for Cranium are possible, such as using regular system DRAM for low cost, or dedicated SRAM to reduce latency.

6 Status and future work

We have completed a preliminary paper design of Cranium. We have also created a visualization tool called `visNI` to assist in further understanding of the dynamic properties of Cranium, such as bus contention.

---

[6] Each command requires passing approximately 50 data bits from the processor to the interface.

[7] Older processor architectures that do not support multiprocessor cache coherence must flush the entire data cache before sending and after receiving every network packet, causing a loss of performance.

The next phase is evaluation of the Cranium design. Evaluation has two parts: simulation and prototype construction. For simulation, we can take advantage of two existing behavioral simulation tools: a network simulator and a processor simulator. The Chaos network simulator is a very flexible tool for modeling a wide variety of routing algorithms, including chaotic routing, oblivious routing and minimal adaptive routing. The Meerkat processor simulator [16] is a functional simulator for multiprocessors, and is enhanced with timing models for caches, TLBs, register scoreboarding and other processor resources. The simulator is functionally equivalent to a physical multiprocessor prototype system that uses a grid of busses for its interconnect. The simulator estimates execution time to within 5 to 8 per cent of the reference hardware. We are replacing the simulator's bus model with the mesh network simulator. Using the combined simulation strategy, we can begin evaluating simple parallel workloads such as Discrete Fourier Transform and Jacobi iteration. The simulator provides a sanity check for the ideas contained within this article, and will help us decide whether the initial values chosen for the architecture (number of channels, depth of channels, size of mapping tables) represent the best tradeoff between hardware complexity and performance. After the simulator has been constructed, the next step is to construct a hardware prototype. An initial pass at the network interface implementation may involve an inexpensive embedded microcontroller for the channel scheduler and field programmable logic technology such as PALs and FPGAs for the DMA engine, bus interface and network link interface. A higher-speed version may involve a semi-custom solution using sea-of-gates technology.

The current design of Cranium uses the standard gang-scheduling model for safe user-level access [5, 13]. Recently, there has been interest in studying general processor scheduling on multicomputers that have user-level support for message passing [11, 14]. The idea is to let multiple user processes execute at the same time inside the same machine partition. The interaction between adaptive packet routing and general processor scheduling is interesting, and we intend to address this issue in our continuing work with Cranium.

7 Summary

We present a design for a network interface called Cranium. Its architecture meets three design goals: to use both an automatic-receive and a processor-initiated style interface, to support fault detection, and to allow direct user-level access to the network. The automatic-receive interface allows Cranium to accept an out-of-order incoming packet stream, for compatibility with adaptive packet routing networks. Automatic-receive also allows Cranium to handle large numbers of small packets efficiently. A set of queuing channels implements the processor-initiated interface, for added flexibility to support a wide variety of higher level message passing protocols. Protection is implemented by mapping user-level handles into physical node identifiers and buffer addresses. We intend to further evaluate the Cranium design, first by using simulation tools and later by constructing a hardware prototype.

Acknowledgements: we thank our colleagues at the University of Washington for their constructive criticism and input, especially Rob Bedichek, Alex Klaiber, Dave Keppel and the Chaos research team. We also thank Jim Peek at Pyramid Technology, John Wilkes, Greg Buzzard, David Jacobsen and Scott Marovich at HP Labs, Ed Felten at Princeton, and the anonymous reviewers.

References

1. John Y. Ngai and Charles L. Seitz. A Framework for Adaptive Routing in Multicomputer Networks. *Proc. of the Symposium on Parallel Architectures and Algorithms,* May 1989.
2. Kevin Bolding and Lawrence Snyder. Mesh and Torus Chaotic Routing. *Advanced Research in VLSI and Parallel Systems; Proc. of the 1992 Brown/MIT Conference,* March 1992, pp. 333-347.
3. Kevin Bolding and William Yost. Design of a Router for Fault-Tolerant Networks. *Proc. of the 1994 Parallel Computer Routing and Communication Workshop*, Springer-Verlag, Seattle WA, May 1994.
4. 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.
5. Edward W. Felten. *Protocol Compilation: High-Performance Communication for Parallel Programs.* PhD dissertation, University of Washington, Dept. of CSE, Sept. 1993. Available as UW CSE technical report TR 93-09-09.
6. Thorsten von Eicken et al. Active Messages: A Mechanism for Integrated Communication and Computation. *19th Annual International Symposium on Computer Architecture,* May 1992, pp. 256-266.
7. Oliver A. McBryan. An Overview of Message Passing Environments. *Parallel Computing* 20(4), April 1994, pp. 417-444.
8. Paul Pierce. The NX Message Passing Interface. *Parallel Computing* 20(4), April 1994, pp. 463-480.
9. Shekhar Borkar et al. Supporting Systolic and Memory Communication in iWarp. *Proc. of the 17th Annual International Symposium on Computer Architecture,* Seattle WA, May 1990, pp. 70-81.
10. William J. Dally et al. The Message-Driven Processor: A Multicomputer Processing Node with Efficient Mechanisms. *IEEE Micro,* April 1992, pp. 23-39.
11. Greg M. Papadopoulos et al. *T: Integrated Building Blocks for Parallel Computing. *Proc. of Supercomputing '93,* Portland OR, November 1993, pp. 624-635.
12. Greg M. Papadopoulos. Personal communication. Supercomputing '93, Portland OR, November 1993.
13. Charles Leiserson et al. The Network Architecture of the CM-5. *Symposium on Parallel Algorithms and Architectures*, 1992, pp. 272-285.
14. Mattias A. Blumrich et al. 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.
15. John Wilkes. Hamlyn – an Interface for Sender-Based Communications. Technical Report HPL-OSR-92-13, Hewlett-Packard Company, HP Labs, Operating System Research Dept., November 1992.
16. Robert Bedichek. *The Meerkat Multicomputer.* PhD dissertation, University of Washington, Dept. of CSE, Seattle WA, June 1994.