Two Virtual Memory Mapped Network Interface Designs

Matthias A. Blumrich, Cezary Dubnicki, Edward W. Felten,
Kai Li, Malena R. Mesarina
Department of Computer Science,
Princeton University, Princeton NJ 08544

Abstract

In existing multicomputers, software overhead dominates the message-passing latency cost. Our research on the SHRIMP project at Princeton indicates that appropriate network interface support can significantly reduce this software overhead. We have designed two network interfaces for the SHRIMP multicomputer. Both support virtual memory mapped communication allowing user processes to communicate without doing expensive buffer management, and without using system calls to cross the protection boundary separating user processes from the operating system kernel. This paper describes and compares the two network interfaces, and discusses performance tradeoffs between them.

1 Introduction

The SHRIMP (Scalable High-performance Really Inexpensive Multi-Processor) project at Princeton studies how to provide high-performance communication mechanisms in order to integrate commodity desktop computers such as PCs and workstations into inexpensive, high-performance multicomputers. Our primary performance metrics are the end-to-end latency and bandwidth available to user processes. Our goal is to provide a low-latency, high-bandwidth communication mechanism whose performance is competitive with or better than those used in specially designed multicomputers.

The network interfaces of existing multicomputers, and workstation networks, require a significant amount of software overhead to implement message-passing protocols. In fact, message-passing primitives on many multicomputers, such as the *csend/crecv* of Intel's NX/2,[1] often execute more than a thousand instructions to send and receive a message; by comparison, the hardware overhead of data transfer is negligible. For example, sending and receiving a message on the Intel Delta multicomputer requires 67 μsec, of which less than 1 μsec is due to time on the wire.[2] Other recent multicomputers such as the Intel Paragon,[3] Meiko CS-2,[4] and TMC CM-5[5] have lower message-passing latencies than Delta, but not much lower.

The main reason for such high software overheads is that these multicomputers use network interfaces that require a significant number of instructions at the operating system and user levels to provide protection, buffer management, and message-passing protocols. In these designs, communication is treated as a service of the operating system. This is expensive because it requires several crossings between user level and kernel level for each message, and also because it prevents applications from using the communication hardware in customized ways. The challenge in designing network interfaces is to provide appropriate hardware support to achieve minimal software message passing overhead, to accommodate multiprogramming under a variety of scheduling policies without sacrificing protection, and to overlap communication with computation.

As the first step of our research, we have developed an idea called *virtual memory-mapped communication* which requires network interface support for implementing virtual memory mapping between user level processes over the network. This approach allows programs to perform message passing directly between user processes without crossing the protection boundary to the operating system kernel. As a result, the software message-passing overhead is reduced significantly.

We have designed two network interfaces for the SHRIMP multicomputer which is being constructed at Princeton using Pentium PCs and an Intel Paragon routing network. In our first design, we explored how to do minimal modifications to the traditional DMA-based network interface design, while implementing virtual memory mapping in software. Our design requires a system call to initiate outgoing data transfer, but its virtual memory mapped communication can reduce the send latency overhead by up to 78%. Received messages are transferred directly to memory, reducing the receive software overhead to only a few instructions in the common case. Our second design implements virtual memory mapping completely in hardware.[6] This approach provides fully protected, user-level message passing, and it allows user programs to initiate an outgoing block data transfer with a single memory store instruction.

2 Virtual Memory Mapped Communication

The main idea of virtual memory-mapped communication is to allow applications to create a mapping between two virtual memory address spaces over the network. That is, it allows the user to map a piece of the sender's virtual memory to an equally sized piece of the receiver's virtual memory across the network, as shown in Figure 1. The mapping operation requires a system call in order to provide protection between users and processes in a multiprogrammed environment. But once the mapping has been established, the sending and receiving processes can use the mapped memory as send and receive buffers, and can communicate without any kernel involvement.

Virtual memory mapped communication has several advantages over the traditional, kernel dispatch-based message passing. One of the main advantages is that virtual memory mapped communication allows applications to perform low overhead communication since data can move between user processes without context switching and message dispatching.

Another main advantage of virtual memory mapped communication is that it moves the memory buffer management to user level. Applications or libraries can manage their communication buffers directly without having to pay the expensive overhead of unnecessary context switches and protection boundary crossings in the common cases. Recent studies and analyses indicate that moving communication buffer management out of the kernel to the user level can greatly reduce the software overhead of message passing. By using a compiled, application-tailored runtime library, the latency of multicomputer message passing can be improved by about 30%.[7]

In addition, virtual memory mapped communication takes advantage of the protection provided by virtual memory systems. Since mappings are established at the virtual memory level, virtual address translation hardware guarantees that an application can only use mappings created by itself. This eliminates the per-message software protection checking found in traditional message passing implementations.

There are several ways to implement virtual memory mapping. To achieve a simple, low-cost design, we have investigated various combinations of hardware and software. Our results are the SHRIMP-I network interface, which is designed to provide minimal hardware support, and the SHRIMP-II network interface, which is intended to provide as much hardware support as needed to minimize communication latency.

3 SHRIMP-I Network Interface

The design goal for the SHRIMP-I network interface was to start with a traditional, DMA-based network interface and add the minimal hardware support needed for implementing virtual memory mapped communication. The resulting network interface supports the traditional DMA-based model, and can optionally be used to implement virtual memory-mapped communication with some software assistance.

Figure 2 shows a block diagram of the SHRIMP-I network interface datapath. The card uses DMA transactions to interface between the EISA bus of a Pentium PC and a Network Interface Chip (NIC) connected to an Intel Paragon routing network.[8] DMA transactions are limited to the size of a memory page and cannot cross page boundaries, since pages are the unit of protection. Control is provided through a set of memory-mapped registers which device driver programs use to compose packets, initiate packet transfers, examine the status of the interface, and set up receiving memory addresses. Interrupts to the host processor can optionally be generated by incoming packets. The arbiter controls sharing of the bidirectional datapath to the NIC, giving incoming data priority over outgoing data.

The hardware supports *physical memory mapping* for incoming data. That is, each packet carries a *receive destination physical memory address* in its packet header, and the hardware automatically initiates a DMA transfer to this address upon packet arrival, without host CPU intervention.

A header of two 64-bit words is appended to the beginning of every packet. The first 64-bit word contains routing information for the Paragon network, and is stripped by the network hardware. The second 64-bit word is the SHRIMP-I packet header containing four fields: version, destination address, packet size, and action. The *version* field identifies the version of the network interface which generated the packet. The *destination address* specifies a physical base address on the destination machine to receive the packet's data. The *packet size* field specifies the number of 32-bit data words in the body of the packet. The *action* field tells the receiving network interface how to handle the packet.

A send operation is initiated by writing a packet header to the send registers. This starts the send state machine which builds a network packet and DMAs the data from memory to the NIC chip. When the packet arrives at the destination, its header will be stored in the receive registers. By default, the data of the packet will be delivered to the physical memory indicated by the *destination address* field in the packet header. Optionally, the action field of the packet can instruct the receiving logic to deliver the data to a physical address provided by the receiver (in a memory-mapped register). In addition, the action field can be used to cause an interrupt to the receiving host processor immediately after packet delivery. If there is an interrupt, the incoming datapath is frozen until the host processor explicitly restarts it by writing to a special control register.

To provide software with flexibility and to support debugging, the receiving logic can be programmed to override the actions indicated in the action field of the packet. Specifically, the receive control register can be programmed to ignore the action field and use the physical address from the special receive register as the destination address for the next incoming packet. It can also be programmed to interrupt the CPU after every packet (or to never interrupt). Finally, the receive logic can be instructed to freeze the incoming datapath after each packet arrival, which is useful for debugging.

The SHRIMP-I network interface supports both traditional message passing and virtual memory-mapped communication. In traditional message passing, the destination address is provided by the receiver and an interrupt is raised upon message arrival (as indicated by action bits in the message header). This option will allow the operating system kernel to manage memory buffers and dispatch messages.

Before using virtual memory-mapped communication, a mapping operation is needed to map a user-level send buffer to a user-level receive buffer. The mapping operation pins both buffers in physical memory. Once a mapping is established, it can be used to send messages without interrupting the receiving processor. That is, a receive operation can be performed entirely at user-level, without making a system call.

Virtual memory-mapped communication is an optimization which allows software message-passing overhead to be reduced at the expense of additional mapping steps and increased consumption of physical memory caused by the pinning of send and receive buffers. If physical memory becomes scarce, virtual memory-mapped communication can be replaced with traditional message passing through kernel-allocated memory buffers.

4 SHRIMP-II Network Interface

The design goal of the SHRIMP-II network interface was to provide hardware support for protected, low-latency, user-level message-passing in order to minimize the software overhead of message passing.

This network interface design shares the main idea of the SHRIMP-I network interface design—supporting virtual memory-mapped communication. The principal difference is that the SHRIMP-II network interface implements virtual memory mapping in hardware, allowing programs to perform message passing completely at user level with full protection.

Figure 3 shows the datapath of the SHRIMP-II network interface, which connects to both the EISA bus and the Xpress memory extension connector. Outgoing data, destined for other nodes, is snooped directly off the Xpress memory bus through the memory extension connector. Since this connector does not provide the capability for mastering the Xpress bus, incoming data from other nodes is transferred to main memory through the EISA expansion bus.

The key component which allows the SHRIMP-II network interface to support virtual memory mapping in hardware is the Network Interface Page Table (NIPT). The NIPT has one entry for each page of physical memory on the node, and contains information about whether, and how, the page is mapped. Each page table entry specifies the destination node and physical page number which is mapped to, and includes various fields to control how data is sent and received.

To initiate a send operation, the source process writes to mapped memory, which takes place on the Xpress memory bus since mapped out pages are cached as write-through. It is convenient to think of the address of this write as a physical page number and an offset on that page. While the write is updating main memory, the network interface snoops it and directly indexes into the NIPT, using the page number, to obtain the mapping information. If the page is mapped out, the network interface constructs a packet header using the destination and physical mapping information from the NIPT entry, along with the original offset from the write address. The written data is appended to this header, and the now-complete packet is put into the Outgoing FIFO. When it eventually reaches the head of the FIFO, the Network Interface Chip (NIC) injects it into the network.

When the packet arrives at the destination processor, the NIC puts it into the Incoming FIFO. Once it reaches the head of this FIFO, the page number is again used to index into the NIPT to determine if that page has been mapped in. If so, the destination address from the packet is used by the EISA DMA logic to transfer the data directly to main memory. The snooping cache architecture of the Xpress PC system insures that the caches remain consistent with main memory during this transfer. Therefore, a SHRIMP system can use regular, cacheable DRAM memory as send and receive buffers for message passing without any special hardware.

The SHRIMP-II network interface supports two transfer strategies: *automatic update* and *deliberate update*. User programs select an update strategy at the time a mapping is created. Automatic update is used when the lowest transfer latency is desired. The CPU initiates data transfers as described above by simply issuing regular store instructions to mapped memory, and suffers only the local write-through cache latency. The data propagates to the destination memory while the CPU goes on with its computation.

Deliberate update is used when the highest transfer bandwidth is desired. Data written to a deliberate-update page is not automatically transferred to the destination node, but only when the user-level application issues an explicit send command. The send command initiates an EISA DMA transfer to move data from memory to the outgoing FIFO, and then to the network. Therefore, deliberate update pages need not be cached as write-through, but must be consistent with the cache at the time the send is initiated. This method allows user programs to control the point at which data is transferred from a mapped-out data structure to its destination.

In order to allow an application to issue user-level commands to control some operations of the network interface without involving the kernel, we provide a mechanism called *Virtual Memory Mapped Commands*. The network interface decodes *command memory* located in the node's physical address space, but not corresponding to actual RAM. References to command memory simply transmit information to or from the network interface at user level.

The current network interface supports one command memory space the same size as the actual physical memory, and associates a unique command page with each page of physical memory. Since both address spaces are linear and of equal size, the association is simply determined by adding or subtracting a fixed offset.

The operating system kernel gives a user-level process access to a command page by mapping that command page into the process's virtual memory space. For example, if physical page p currently holds the contents of some virtual page of process X, then the kernel can give X access to the command pages that control p. This allows X to "talk to" the network interface about p directly from user-level. If the kernel later decides to reallocate p to another process, it can revoke X's right to access the command pages corresponding to p.

The command memory mechanism uses physical address space (but not physical memory) to achieve low-overhead control of the network interface. It consumes a fraction of the physical address space whose size is a small constant times the size of the local physical memory, and it consumes the same amount of virtual address space.

We are currently using the command space to implement the send command for deliberate updates.

5 System Software Support

One advantage of virtual memory mapped communication is the diversity of communication models it can support. In this section, we describe a simple communication model designed for supporting multicomputer programs. Other models, suitable for other application classes, will not be discussed here. For all models, the virtual memory-mapped communication for this network interface requires system calls to create mappings, and also primitives to send messages using mapped memory.

Our multicomputer interface uses two system calls for mapping creation: `map_send` and `map_recv`. Their arguments and costs are similar to NX/2's `csend` and `crecv` calls:

- `mapid = map_send( node_id, process_id, bind_id, mode, sendbuf, size )`
  where `node_id` is the the network address of the receiving node, `process_id` indicates the receiving process, `bind_id` is a binding identifier (whose function is similar to the message type in the NX/2 send and receive primitives), `mode` indicates whether the mapping is for automatic update or deliberate update (meaningless for the SHRIMP-I network interface which does not support automatic update), `sendbuf` is the starting address of the send buffer, and `size` is the number of words in the send buffer. This call is used on the senders side to establish a mapping. It returns a `mapid` which is used to identify this mapping for send operations. For SHRIMP-I, `mapid` is just an index into a kernel-level mapping table specific for a calling process. For SHRIMP-II, `mapid` is the virtual address in the command space corresponding to `sendbuf`.

- `map_recv( bind_id, recvbuf, size, ihandler )`

  where `bind_id` is the binding identifier to match the mapping request by the `map_send` call, `recvbuf` is the starting address of the receive buffer, and `size` is the number of words in the receive buffer. Non-NULL `ihandler` specifies a user-level interrupt handler which will be called for every message received to this mapping. The purpose of this call is to provide the mapping identified by `bind_id` with a receiving physical memory address so that the senders side can create a physical memory mapping for the virtual memory mapping.

The mapping calls will pin the memory pages of both the send and receive buffers into physical memory to create a stable physical memory mapping, enabling data transfers on both sending and receiving sides without CPU involvement on Shrimp-II, and with minimal sender-side overhead on Shrimp-I.

Every mapping is unidirectional and asymmetric, from the source (sending buffer) to the destination (receiving buffer). A mapping can be established only if the size of the receive buffer is the same as the size of the send buffer. The `mapid` can be viewed as a "handle" to select a mapping for a send operation. It is needed for SHRIMP-I because we allow multiple and overlapped mappings for the same memory. For SHRIMP-II it is needed to keep the virtual base address in the command space.

For security, we must verify that the sending process has permission to transmit data to the receiving process. In our multicomputer programming model, only objects owned by processes belonging to the same *process task group* can be mapped to each other. The membership of a process in a given task group is fully controlled by the operating system, so all processes within a task group trust each other. For example, processes cooperating on the execution of a given multicomputer program will usually belong to the same task group.

For both SHRIMP-I and SHRIMP-II, the following send operation is supported:

```
send( mapid, send_offset, size )
```

For SHRIMP-I, this operation is implemented as a system call which builds a packet for each memory page. It simply looks up the mapid in the mapping table, finds the destination physical address, builds a packet header, and initiates the outgoing data transfer. This call returns immediately after the data is sent out to the network. For SHRIMP-II, the send operation is a deliberate-update macro. If the data to be sent resides within one page, this macro executes a user-level store of `size` to the address `mapid + send_offset` in the command space. This write is decoded by the network interface as a command to initiate the requested transfer from the corresponding physical memory page. For a message spanning multiple pages, there is one store issued for each such page.

Since a destination object is allocated in user space, data can be delivered directly to the user memory without a receive interrupt for both SHRIMP-I and SHRIMP-II. The user process can observe the message delivery, for example, by polling a flag located at the end of the message buffer. In this way it can implement user-level buffer management and avoid the overhead of kernel buffer management, message dispatching, interrupts and receive system calls.

6 Cost and Performance

Both the SHRIMP-I and SHRIMP-II network interfaces support traditional, DMA-based message passing, and provide the option to use virtual memory mapped communication. This option can eliminate a large amount of software overhead in traditional message passing, such as buffer management and message dispatching.

Compared to the traditional network interface designs, the only addition in the SHRIMP-I network interface is to include the destination physical address of a packet in its header and to have receiving logic to deliver data accordingly. This simple addition makes the network interface very flexible. Although a system call is required to send a message, no involvement is required by the receiving node's CPU to receive and dispatch messages.

The SHRIMP-II network interface supports protected, user-level virtual memory mapped communication. The automatic update mode allows a single store instruction to initiate a send with only the local write-buffer latency. The deliberate update mode requires a few user-level instructions to send up to a page of data.

Table 1 shows the overhead components of message passing on three kinds of network interfaces: traditional, SHRIMP-I, and SHRIMP-II.[6] The SHRIMP-II network interface implements virtual memory mapping translation in hardware so that the send operation can be performed at user level. The SHRIMP-I network interface requires a system call to send a message, but it provides virtual memory-mapped communication with very little additional hardware over the traditional network interface design.

We have implemented the send operation for both the SHRIMP-I and SHRIMP-II network interfaces using Pentium-based PCs. We compared the cost of our send with that of the *csend/crecv* primitives of NX/2. For passing a small message (less than 100 bytes), the software overhead of a send for the SHRIMP-I network interface is 117 instructions plus the system call overhead, and optionally, an interrupt. For SHRIMP-II, the overhead of a send implemented as a macro is only 15 user level instructions, with no system call necessary. In contrast, the software overhead of a *csend* and a *crecv* in NX/2 is 483 instructions plus two system calls (*csend* and *crecv*) and an interrupt.

For passing a large message, the primitives for the SHRIMP-I network interface require only 26 additional instructions for each additional page transferred. For SHRIMP-II, this overhead is only 8 user-level instructions. The *csend* and *crecv* primitives of NX/2 require additional network transactions to allocate receive buffer space on the receiving side, and they must prepare data descriptors needed by the network interface to initiate a send.

The cost of mapping on both SHRIMP-I and SHRIMP-II is similar to that of passing a small message using *csend* and *crecv* in NX/2. For applications that have static communication patterns, the amortized overhead of creating a mapping can be negligible.[6]

We should point out that the semantics of the *csend/crecv* primitives of NX/2 are richer than the virtual memory-mapped communication supported by SHRIMP family of interfaces. Our comparison shows that rich semantics often comes with substantial overhead. Since both SHRIMP network interfaces support traditional message passing and virtual memory-mapped communication, they allow user programs to optimize for common cases.

7 Related Work

The traditional network interface design is based on DMA data transfer. Recent examples include the NCUBE,[9] and iPSC/860.[10] In this scheme an application sends messages by making operating system calls to initiate DMA data transfers. The network interface initiates an incoming DMA data transfer when a message arrives and interrupts the local processor when the transfer has finished so that it can dispatch the arrived message. The main disadvantage of traditional network interfaces is that message passing costs are usually thousands of CPU cycles.

One solution to the problem of software overhead is to add a separate processor on every node just for message passing.[11,12] Recent examples of this approach are the Intel Paragon[3] and Meiko CS-2.[4] The basic idea is for the "compute" processor to communicate with the "message" processor through either mailboxes in shared memory or closely-coupled datapaths. The compute and message processors can then work in parallel, to overlap communication and computation. In addition, the message processor can poll the network device, eliminating interrupt overhead. This approach, however, does not eliminate the overhead of the software protocol on the message processor, which is still hundreds of CPU instructions. In addition, the node is complex and expensive to build.

Several projects have taken the approach of lowering communication latency by bringing the network all the way into the processor and mapping the network interface FIFOs to special processor registers.[13–15] Writing and reading these registers queues and dequeues data from the FIFOs respectively. While this is efficient for fine-grain, low-latency communication, it requires the use of a non-standard CPU, and it does not support the protection of multiple contexts in a multiprogramming environment.

An alternative network interface design approach employs memory-mapped network interface FIFOs.[5] In this scheme, the controller has no DMA capability. Instead, the host processor communicates with the network interface by reading or writing special memory locations that correspond to the FIFOs. This approach results in good latency for short messages. However, for longer messages the DMA-based controller is preferable because it makes use of the bus burst mode, which is much faster than processor-generated single word transactions.

Among commercially available MPPs, the machine with the lowest latency is the Cray T3D, which supports shared memory without caching. The T3D requires a large amount of custom hardware design, and it is not clear whether the overhead from not caching remote memories degrades the performance of message-passing applications.

8 Conclusions

This paper describes and compares two network interface designs for virtual memory-mapped communication, which can significantly reduce the software overhead for message-passing.

We have shown that with minimal additions to the traditional network interface design, we can reduce the software overhead by up to 78%. With more hardware support, the software overhead for sending a message can be reduced to a single user-level instruction. Although virtual memory-mapped communication requires `map` system calls, it can avoid a receive system call and a receive interrupt. For multicomputer programs that exhibit static communication patterns, (transfers from a given send buffer go to a fixed destination buffer), the net gain can be substantial.

Both network interfaces provide users with flexible functionality including the specification of data delivery locations, and optional receiver interrupt generation. In addition, much of the designs are in programmable logic, allowing for experimentation with hardware protocols at various stages of the data transfer.

We have built a simulator for software development, and we are in the process of constructing 16-node prototypes of SHRIMP-I and SHRIMP-II systems. We expect the SHRIMP-I system to be operational sometime during the second quarter of 1995. At that time, we also expect to have a 2-node SHRIMP-II system operational.

Acknowledgements

We would like to thank Otto Anthus, Doug Clark, Liviu Iftode, and Jonathan Sandberg for numerous discussions on the SHRIMP-I and SHRIMP-II network interface designs.

References

1. P. Pierce, "The NX/2 operating system," in *Proceedings of 3rd Conference on Hypercube Concurrent Computers and Applications*, pp. 384–390, Jan. 1988.

2. R. J. Littlefield, "Characterizing and tuning communications performance for real applications," in *Proceedings of the First Intel DELTA Applications Workshop*, pp. 179–190, Feb. 1992. Proceedings also available as Caltech Technical Report CCSF-14-92.

3. Intel Corporation, *Paragon XP/S Product Overview*, 1991.

4. M. Homewood and M. McLaren, "Meiko CS-2 interconnect elan – elite design," in *Proceedings of Hot Interconnects '93 Symposium*, Aug. 1993.

5. C. Leiserson, Z. Abuhamdeh, D. Douglas, C. Feynman, M. Ganmukhi, J. Hill, D. Hillis, B. Kuszmaul, M. S. Pierre, D. Wells, M. Wong, S. Yang, and R. Zak, "The network architecture of the connection machine CM-5," in *Proceedings of 4th ACM Symposium on Parallel Algorithms and Architectures*, pp. 272–285, June 1992.

6. M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg, "A virtual memory mapped network interface for the SHRIMP multicomputer," in *Proceedings of the 21st Annual Symposium on Computer Architecture*, pp. 142–153, Apr. 1994.

7. E. W. Felten, *Protocol Compilation: High-Performance Communication for Parallel Programs.* PhD thesis, Dept. of Computer Science and Engineering, University of Washington, Aug. 1993. Available as technical report 93-09-09.

8. R. Traylor and D. Dunning, "Routing chip set for Intel Paragon parallel supercomputer," in *Proceedings of Hot Chips '92 Symposium*, Aug. 1992.

9. J. Palmer, "The NCUBE family of high-performance parallel computer systems," in *Proceedings of 3rd Conference on Hypercube Concurrent Computers and Applications*, pp. 845–851, Jan. 1988.

10. Intel Corporation, *iPSC/860 Technical Reference Manual*, 1991.

11. R. Nikhil, G. Papadopoulos, and Arvind, "*T: A multithreaded massively parallel architecture," in *Proceedings of 19th International Symposium on Computer Architecture*, pp. 156–167, May 1992.

12. J.-M. Hsu and P. Banerjee, "A message passing coprocessor for distributed memory multicomputers," in *Proceedings of Supercomputing '90*, pp. 720–729, November 1990.

13. S. Borkar, R. Cohn, G. Cox, T. Gross, H. Kung, M. Lam, M. Levine, B. Moore, W. Moore, C. Peterson, J. Susman, J. Sutton, J. Urbanski, and J. Webb, "Supporting systolic and memory communication in iWarp," in *Proceedings of the 17th Annual Symposium on Computer Architecture*, June 1990.

14. D. S. Henry and C. F. Joerg, "A tightly-coupled processor-network interface," in *Proceedings of 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 111–122, Oct. 1992.

15. W. J. Dally, "The J-Machine system," in *Artificial Intelligence at MIT: Expanding Frontiers* (P. Winston and S. Shellard, eds.), pp. 550–580, MIT Press, 1990.

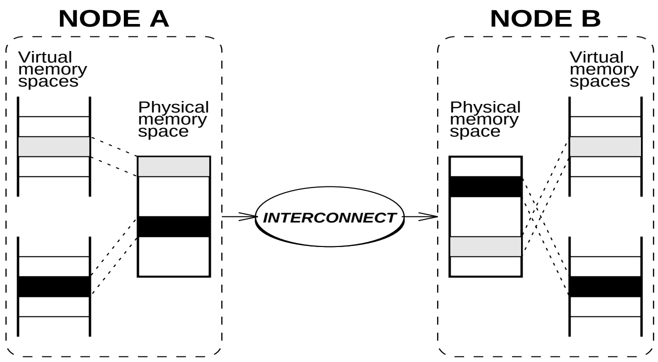

Figure 1: *Virtual memory mapping*

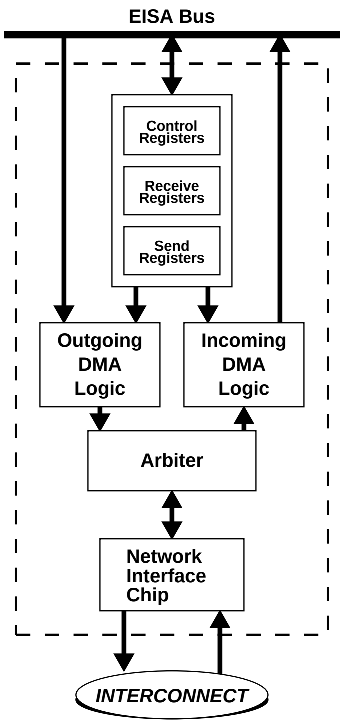

Figure 2: *SHRIMP-I network interface datapath*

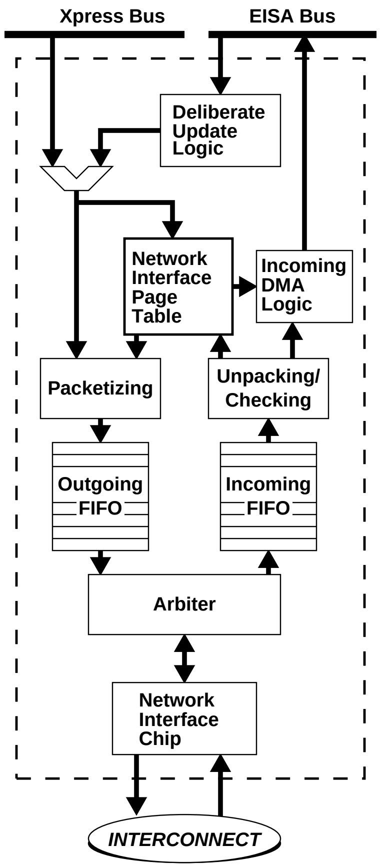

Figure 3: *SHRIMP-II network interface datapath*

	Message passing overheads	Traditional	SHRIMP-I	SHRIMP-II
Sender Software	Send system call	x	x	
	Send argument processing	x	x	
	Verify/allocate receive buffer	x		
	Preparing packet descriptors	x	x	
	Initiation of send	x	x	x
Hardware	DMA data via I/O bus	x	x	x
	Data transfer over the network	x	x	x
	Data transfer via I/O bus	x	x	x
Receiver Software	Interrupt service	x	optional	optional
	Receive buffer management	x		
	Message dispatch	x		
	Copy data to user space (small)	x		
	Receive system call	x		

Table 1: *Message passing overheads: breakdown of three kinds of network interface designs*