Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer

Matthias A. Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, and Edward W. Felten
Department of Computer Science, Princeton University, Princeton NJ 08544

Jonathan Sandberg
Panasonic Technologies, Inc., 2 Research Way, Princeton, NJ 08540

Abstract

The network interfaces of existing multicomputers require a significant amount of software overhead at the operating system and user levels to provide protection and to implement message passing protocols. This paper describes the design of a low-latency, high-bandwidth, virtual memory-mapped network interface for the SHRIMP multicomputer project at Princeton University. Without sacrificing protection, the network interface achieves low latency by using virtual memory mapping and write-latency hiding techniques, and obtains high bandwidth by providing a user-level block data transfer mechanism. We have implemented several message passing primitives in an experimental environment, demonstrating that our approach can reduce the message passing overhead to a few user-level instructions.

1 Introduction

The new trend of constructing parallel systems from commodity, off-the-shelf components offers a great cost/performance advantage over traditional custom-designed parallel systems. The SHRIMP multicomputer project at Princeton University studies the use of commodity PCs or workstations and commercially available routing backplanes to construct scalable, high-performance multicomputers. Our focus is on building a low-cost communication mechanism whose latency and bandwidth are comparable to or better than existing custom-designed multicomputers.

The design challenges for such a high-performance communication mechanism are to minimize message passing overhead, to accommodate multiprogramming under a variety of scheduling policies without sacrificing protection, and to overlap communication with computation. We address these challenges by designing a simple, low-cost, custom network interface, and using it to connect commodity computing engines with a commercially available routing backplane.

The first challenge is to minimize message passing overhead. The cost of communication on existing multicomputers with traditional network interfaces is due almost entirely to the amount of CPU time required to send and receive a message. Compared to these software overheads, hardware communication latencies are almost negligible. For example, on the Intel DELTA multicomputer, sending and receiving a message requires 67 μsec, of which less than 1 μsec is accounted for by hardware latency [18]. The main reason for such high overheads is that communication is usually provided 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. Our goal of low-overhead communication can be achieved by supporting communication directly at the user level.

The second challenge is to support multiprogramming, under a variety of scheduling policies, without compromising protection. Our user-level communication mechanism must be designed carefully to meet this challenge. Some existing network interface designs support user-level message passing, but they either prohibit multiprogramming, or constrain how processes are scheduled. Systems that prohibit multiprogramming use hardware to divide the machine into partitions, and then run a single parallel job in each partition. Systems that restrict multiprogramming allow several parallel jobs to share a partition, but guarantee protection only if certain scheduling policies are used. For example, the CM-5 hardware provides safe user-level communication, provided the operating system uses strict gang scheduling in each partition. Even under these constraints, the CM-5 network interface still requires significant user-level message handling overhead. The best case using the active message mechanism requires 3.3 μsec, or over 100 SPARC CPU cycles, for software message handling [29].

We require our mechanism to support general multiprogramming, which is needed on parallel systems for the same reasons as on uniprocessor systems: it allows useful processing to go on during I/O and paging, and it supports interactive jobs well. Restricting multiprogramming, as the CM-5 does with its gang-scheduling requirement, is probably not the right choice for a research machine. Having hardware that supports general multiprogramming gives us the ability to experiment with various scheduling policies, and allows us to support the best scheduling algorithm, whatever it turns out to be.

The third research challenge is to support the overlap of communication and computation effectively. This requires a communication mechanism with very low transfer initiation overhead in order to use the transfer time effectively. Some multiprocessors address this issue by using multithreading, but the commodity processors and operating systems targeted for use in a SHRIMP system do not support rapid context switching. Therefore, we provide two types of low-overhead transfer initiation mechanisms in the SHRIMP network interface: one with no overhead and one requiring a few user-level instructions. Neither of these mechanisms requires any operating system assistance to send or receive a message.

This paper describes the virtual memory-mapped network interface for the SHRIMP multicomputer being constructed at Princeton University. By taking a system level design approach that considers not only the hardware latency, but also the operating system and user-level overhead, we were able to move protection related work out of the critical message passing path and to provide applications or compilers with the ability to overlap communication with computation without using multiple threads of control. We have implemented several message passing primitives in an experimental environment, demonstrating that our approach can reduce the message passing overhead to a few instructions.

2 Main Ideas

There are three main ideas in our network interface design: providing applications with a virtual memory-mapped interface, separating protection from data movement, and delivering updates to mapped memory in an overlapped fashion. These ideas lead to a simple network interface with greatly reduced message passing overhead.

Our virtual memory network interface allows the virtual memory of a sending process to be mapped to the virtual memory of a receiving process in such a way that ordinary `store` instructions by the sending process cause data to be propagated to the virtual memory of the receiving process. This can be used to support a simple, flexible message passing mechanism, or a programming model with many of the properties of shared memory.

The separation of protection from data movement is accomplished by separating destination specification from data specification in message passing primitives. The traditional `send` primitives usually have the form:

```
send(destination, send-buf, nbytes ...)
```

where `destination` usually contains the destination node-id, process-id, and other information used to build a message header and check protection. The `send-buf` and `nbytes` specifications are used to tell the network interface where the data is.

Our approach is to use virtual memory-mapped segments for all message passing primitives. The traditional `send` primitive becomes two separate primitives:

```
map(send-buf, destination, receive-buf)
send(send-buf, nbytes ...)
```

where the `map` primitive is a kernel call that performs protection checking and stores memory mapping information on the network interface, and the `send` primitive initiates data transfer completely at user level. Once the mapping has been established, `send` can be used repeatedly to transfer data with minimal overhead. The mapping allows one-way data transfer using local virtual memory addresses. Communication can be made bidirectional by adding a complementary mapping in the reverse direction.

Separating protection from data movement allows the common case to be performed entirely at user-level. Setting up a mapping is necessarily slow, since it requires protection to be verified in the operating system kernel. Once a mapping has been set up, communication can proceed without any operating-system involvement. The common case, communication, is fast; the rare case, mapping, is slow but ensures protection.

The separation of destination specification and data specification fits multicomputer programs very well. Most multicomputer programs have a process on each node, and each process loops until the desired result is computed. Each iteration of the loop requires sending messages to, and receiving messages from other processes using the same buffers. For such processes, `map` calls can execute outside the loops and `send` can be performed completely at user level (Figure 1).

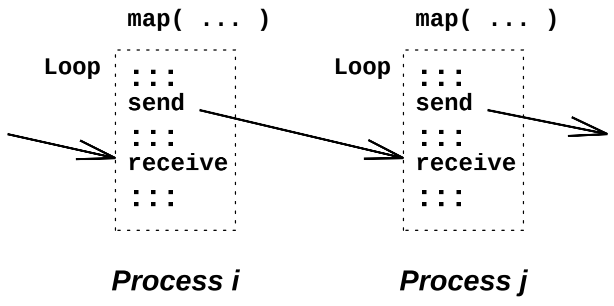


Figure 1: *A typical multicomputer program.*

The network interface supports two transfer strategies: *automatic update* and *deliberate update*. Under automatic update, a sending process can map the memory of its data structures as "send buffers" and there is no need to have an explicit `send` call. Any time the sending process writes to a mapped (outgoing) data structure, the newly written value is automatically propagated to the virtual memory of the destination process which is mapped to. The sending process continues its computation as the value is propagating. Automatic update requires the network interface to snoop the CPU's memory store operations, and requires the CPU to use a write-through caching strategy for its mapped memory. This method overlaps communication and computation at the user level without using multiple threads.

Under deliberate update, newly written values are not propagated immediately, but only after the sending process issues an explicit `send` command. Our network interface uses deliberate updates for user-level block data transfers to achieve high-bandwidth communication with no operating system overhead.

3 SHRIMP System

The SHRIMP system is a new multicomputer being designed at Princeton University. Each node in the SHRIMP multicomputer is an Intel Pentium Xpress PC system [13] and the interconnect is an Intel Paragon routing backplane. These state-of-the-art components allow SHRIMP to take advantage of the latest available technology at a fraction of the cost of current multicomputer systems.

The Xpress PC consists of a Pentium CPU [26, 14] with a second level cache connected to DRAM memory modules and I/O bus adapters (EISA [3] or PCI [23]) via the Xpress memory bus. Memory can be cached as write-through or write-back on a per-virtual-page basis, as specified in process page tables. The caches snoop DMA transactions and automatically invalidate corresponding cache lines, keeping consistent with *all* main memory updates. The PC baseboard has a memory extension connector which carries the majority of the Xpress bus signals, as well as a number of system expansion connectors (EISA or PCI).

The Intel Paragon routing backplane is a two-dimensional mesh of Intel iMRC routers [28], which are essentially faster and wider versions of the Caltech Mesh Routing Chip [7]. The backplane supports deadlock-free, oblivious wormhole routing [8] and preserves the order of messages from each sender to each receiver.

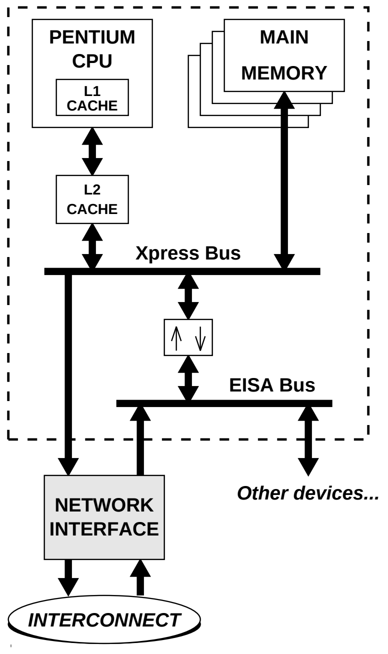


Figure 2: *A SHRIMP node with network interface*

The custom designed SHRIMP network interface is the key system component which connects each Xpress PC system to the processor port of an iMRC router on the routing backplane. The current generation of the network interface is connected to both the Xpress memory extension connector and an EISA expansion connector (Figure 2). 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 by way of the EISA expansion bus without involving the CPU. 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 normal, cacheable DRAM memory as send and receive buffers for message passing without any special hardware.

Note that future versions of the Xpress PC system will provide mastering capability for the Xpress memory bus, thereby simplifying the bus interface of the next generation SHRIMP network interface.

3.1 Physical Memory Mapping

The network interface implements virtual memory-mapping as described in Section 2 by using a *physical memory mapping* mechanism. Each page of local physical memory can be "mapped out" to a physical page of some other node in the system. We say that this other page is "mapped in", since it is the destination of data sent from the mapped-out page. The physical mapping information is retained in a Network Interface Page Table, described in Section 4.

To create a virtual memory-mapping from one node to another, the network interface requires a `map` system call to set up the appropriate physical mapping information in the page tables of both network interfaces, and configure the physical pages which are mapped out for write-through caching.

After the `map` call establishes the physical memory mapping, the network interface snoops all writes to the mapped memory directly off the Xpress bus, packetizes them, and sends the packets to the routing backplane (Figure 2). A packet consists of routing information, the absolute mesh coordinates of the intended receiver, destination memory address, data, and a CRC checksum to detect network errors.

When a packet is received by the network interface, the absolute mesh coordinates of the intended receiver and the CRC are used to verify that the packet was routed correctly and arrived intact. The network interface uses the destination address to transfer the data directly to the mapped-in physical memory without CPU assistance. This transfer is done through the EISA expansion bus on the current implementation of the network interface, although any path to the main memory could be used.

Figure 3 shows how two processes coexist on a SHRIMP system. One process is using the gray mapping to send data from Node A to Node B. The other process is using the black mapping to do the same. The network interface stores information to maintain the mapping between the physical memories, and the virtual memory systems of the nodes maintain the mappings between virtual and physical

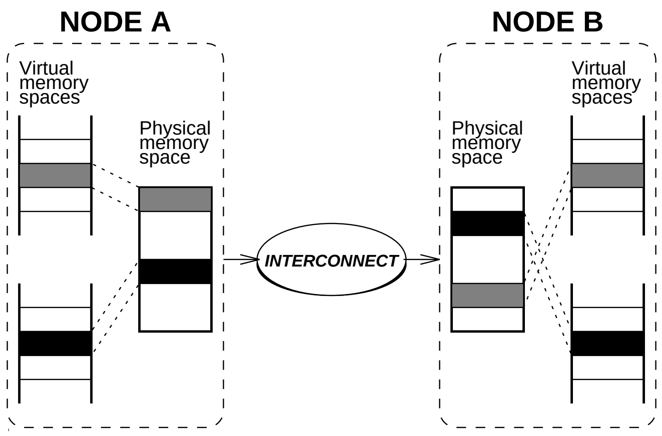


Figure 3: *Virtual memory mapping*

memory. Since the physical memory used by the two processes is distinct, a context switch between them does not require any action on the part of the network interface.

3.2 Mapping Alignment

An ideal virtual memory-mapped network interface would allow application programs to map arbitrarily sized segments of memory without any restriction, but it is difficult to implement such a scheme inexpensively. The SHRIMP network interface approximates this ideal by allowing a physical page to be split between two separate mappings at some configurable offset. As long as applications insure that the granularity of their mapped data structures exceeds the size of a page, this scheme can accommodate all mappings, including those which are not page-aligned.

For the rest of this paper we will discuss physical memory mappings on a page basis in order to avoid confusion. It should be understood, however, that any page can be split between two mappings.

4 The Network Interface

Figure 4 shows the datapath of the SHRIMP virtual memory-mapped network interface.

The key component of the network interface 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 bits to control how data is sent and received.

To illustrate how the network interface works, let us consider how the components of the datapath

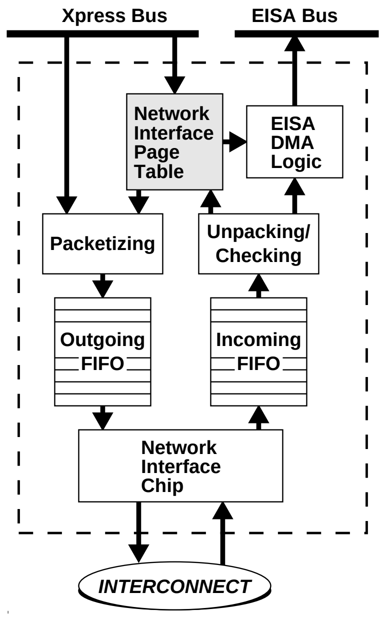


Figure 4: *SHRIMP Network Interface Data Path*

cooperate to propagate a written value from one node to another, assuming that a mapping has already been set up. This example does not explain all the features of the network interface; it is meant merely to illustrate one simple case.

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 indexes into the NIPT using the page number to reveal that the page is mapped out. Using the destination and physical mapping information from the NIPT entry along with the original offset from the write address, the network interface constructs a packet header. 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. The destination address from the packet is used by the EISA DMA logic to transfer the data directly to main memory.

The system flow control mechanism is very simple. If the Incoming FIFO becomes full (exceeding some programmable threshold), the NIC will cease to accept more packets from the network, possibly causing Outgoing FIFOs on other network interfaces to cease draining. If the Outgoing FIFO becomes full (exceeding another programmable threshold), the CPU is interrupted and waits until the FIFO drains. A full Incoming FIFO will eventually drain to memory, allowing the NIC to accept network packets. Since the routing network is deadlock-free, all packets will eventually be delivered to their destinations, allowing stalled Outgoing FIFOs to drain. Since the CPU does not write to any mapped pages while it is waiting, the Outgoing FIFO cannot overflow.

4.1 Supporting Automatic Update

Recall that "automatic update" means that an update to a mapped-out page initiates data transfer immediately (often called "eager" or "anticipatory" sharing). There are two implementations of automatic updates: single-write and blocked-write. The two implementations offer the same semantics, but differ in performance. While single write is optimized for low overhead, blocked write is optimized for efficient network bandwidth usage.

When the network interface snoops a memory write off the memory bus, it always looks up the referenced page in the NIPT. If the referenced address is mapped out for single-write automatic-update, the network interface packetizes the data of the write and sends the packet to the network immediately, as described in the beginning of Section 4. If the address is mapped for blocked-write automatic-update, the network interface buffers the data into the Outgoing FIFO without sending it out immediately. Subsequent writes are merged into the same packet if they are consecutive, occur within the same page, and occur within a programmable time limit from one another. Otherwise, the packet is terminated and sent.

The automatic-update method used in the network interface is an effective way to overlap communication with computation. The CPU issues regular `store` instructions to initiate data transfer, and suffers only the local write-through cache latency. The data propagates to the destination memory while the CPU goes on with its computation.

The automatic-update page type can be used to share memory between processes and support a programming model based on PRAM consistency [17]. That is, processes retain a local copy of a shared address "space" and maintain consistency between their local copy and all the other copies by duplicating local updates to remote copies. Although there is no global consistency mechanism preventing the individual copies from becoming inconsistent with one another, protocols can be used to maintain consistency within applications. In order to share memory, two processes on different nodes simply create complementary mappings. Writes to local shared memory are sent to the mapped memory of the remote node. Because there is a unique path from a source node to a destination node and the hardware guarantees that all messages from the same sender are delivered in the same order, software consistency schemes can be applied.

4.2 VM Mapped Commands

In order to allow an application to control some operations of the network interface without involving the kernel, we provide a mechanism called *Virtual Memory Mapped Commands*. The network interface command memory is located in the node's physical address space, but does not address any actual RAM. References to command memory simply transmit information to or from the network interface. For example, command memory might be used to switch some page from single-write to block-write mode, or it might request an interrupt the next time data arrives for some page.

The current network interface supports one command memory space the same size as the actual physical memory, and assigns a unique command page to each page of physical memory. Since both address spaces are linear and of equal size, the assignment is simply determined by the distance between them in the physical address space. 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.

4.3 Supporting Deliberate Update

In order to achieve high bandwidth data transfer, memory can be mapped out for deliberate update. Data written to a deliberate-update page is not automatically transferred to the destination node, but the transfer takes place only when the user-level application issues an explicit send command. The command is issued through a command page corresponding to the page from which the transfer is to occur. This method allows user programs to control the point at which data is transferred from a mapped-out data structure to its destination.

The protocol for the command to transfer a block of deliberate-update type data is nontrivial for two reasons. First, the network interface has only one DMA engine to perform deliberate updates, and it serves only one request at a time. The DMA engine is not always available, so attempts to initiate a transfer might fail. This requires the network interface to return a success/failure code to each process attempting to start a transfer, and it requires processes to retry operations that fail. Second, the network interface has no control over process scheduling, so a context switch could happen at any time. This requires processes to use a single, atomic instruction to start a transfer and determine whether it succeeded.

The current network interface supports deliberate-update transfer initiation using the compare-and-exchange (CMPXCHG) instruction [14], which generates a read cycle followed by a write cycle if the value returned by the read matches the accumulator. To transfer n words of data starting from a mapped-out base address, the application loads a source register with n, and issues a CMPXCHG instruction whose destination address is the *command page* address associated (same offset) with the mapped-out base address. If the DMA engine is free, the network interface reacts to the read cycle by returning zero, which causes the CMPXCHG to generate the write cycle and start the transfer. So, the application initiates a deliberate-update transfer by clearing the accumulator, setting up the source register with n, and repeatedly performing a locked (atomic) CMPXCHG instruction to the command page address until successful (zero returned).

When the DMA engine is busy, the network interface reacts to a read cycle by returning the number of words remaining to be transferred, and a binary flag to indicate whether the read address matches the current base address of the DMA engine. Therefore, a single read cycle allows an application to determine whether a transfer it initiated is complete, or the number of words remaining to be transferred if not. This feature can be used to implement backoff strategies to optimize the use of the memory bus for the DMA transfer.

Deliberate-update data transfer is accomplished with minimal additional hardware support because the network interface simply enables the Xpress memory bus snooping mechanism while the DMA engine reads the data from main memory. The outgoing datapath then captures the data in a manner equivalent to automatic-update writes (Section 4.1), and packetizes it for network transmission.

Because protection and mapping are on a page basis, each deliberate-update command can transfer at most one page of data. Larger transfers, or transfers that span a page boundary, must be broken up into several smaller transfers. The command sequence to send a large piece of data crossing page boundaries can easily be embedded in a macro or a run-time library routine.

4.4 Consistency of Mappings

Since entries in the network interface page table (NIPT) refer to remote *physical* addresses, the operating system must take some care when paging, to ensure consistency between its local virtual-to-physical page mapping, and the virtual-to-physical mappings implied by remote NIPTs. This is essentially the same as the TLB consistency problem in shared-memory multiprocessors.

Note that there is no consistency problem for pages that have only outgoing communication mappings. These pages are not referenced in remote NIPTs, so they can safely be replaced, provided that the outgoing mapping information is stored in the page table.

The simplest consistency policy is simply to pin into physical memory all pages that have incoming communication mappings, eliminating the consistency problem since a page referred to by a remote NIPT entry can never move. This solution is satisfactory if there are not too many communication mappings.

We can design a more sophisticated solution by exploiting the similarity to the TLB consistency problem, and borrowing the standard solution [25]. Space does not permit a detailed discussion of this policy. Briefly, before replacing a communication-mapped page, a node's kernel must cause all remote NIPT entries referring to that physical page to be invalidated. This is done by sending messages to the remote kernels, which invalidate their NIPT entries and then respond with an acknowledgement. When all acknowledgements are received, the page can be replaced. NIPT entries are "invalidated" by marking the source virtual pages as read-only. If the application later tries to initiate a transfer by writing such a page, a page-fault will occur and the kernel can try to re-establish the invalid mapping.

5 Performance

This section discusses the performance of the SHRIMP system. We evaluate two aspects of performance: hardware performance, and the software overhead required by various message passing operations.

We define communication *latency* to be the time between a write operation by the sending CPU, and the arrival of the written data in the destination memory. The SHRIMP hardware has a communication latency of less than 2 μsec, and a peak communication bandwidth of 33 Megabytes/second. These figures are comparable with several modern multicomputers, and demonstrate that we do not lose significant hardware performance by using commodity parts rather than building the machine from scratch.

Table 5 summarizes the software overheads for various message passing primitives presented in this section. Overhead is measured in the number of CPU instructions required to carry out each operation. The expressions in parentheses divide each overhead into instructions executed by the source, and instructions executed by the destination.

Message Passing Primitive	Software Overhead (instructions)
single buffering	9 (4+5)
single buffering + copy	21 (4+17)
double buffering (case 1)	2 (1+1)
double buffering (case 2)	8 (3+5)
double buffering (case 3)	10 (5+5)
deliberate-update transfer	15 (15+0)
csend and crecv	151 (73+78)

Table 1: *Software overhead of message passing primitives*

A discussion of the experimental procedure, and descriptions of the primitives, are given below.

With the exception of csend and crecv, software overhead is very small, requiring no more than 21 instructions for each primitive. The more complex semantics of csend and crecv result in larger software overhead, but these operations are still relatively cheap, with overheads of under 80 instructions each. Overall, we conclude that the virtual memory mapped network interface supports efficient implementation of a range of message passing primitives.

5.1 Hardware Performance

We evaluate the performance of the SHRIMP hardware by measuring the latency and bandwidth of transfers. Of course, the performance of individual processors when computing locally is not determined by us.

Latency

We evaluate latency for the automatic-update mechanism, using the single-write mechanism. This is the strategy recommended for applications requiring very low latency. Each write by the originating processor becomes a network packet immediately, allowing the hardware latency to be completely overlapped with computation.

In the absence of bus and network contention, the propagation latency on a 16-node system with the current EISA-based prototype network interface is estimated to be slightly less than 2 μsec. This latency includes the time to get the data off the Xpress bus, to build a packet, to transfer the packet through the Outgoing FIFO and the NIC, to route it through the backplane, to pass it through the destination NIC and Incoming FIFO, and finally to transfer it to the destination address over the EISA bus. Our next implementation of SHRIMP will bypass the EISA bus and drive the Xpress memory bus directly, thus reducing the latency to less than 1 μsec.

Peak Bandwidth

We evaluate bandwidth for the deliberate-update mechanism, since it is the strategy recommended for applications that require the highest bandwidth. Deliberate-update communication takes place as a single burst, driven by DMA engines on both the sending and receiving nodes.

The EISA bus on the receiver's side is the bottleneck that limits bandwidth. The peak bandwidth of the EISA bus in burst mode is 33 Mbytes/second [3]. All other parts of the datapath have at least twice this bandwidth. Our next implementation of SHRIMP will bypass the EISA bus, thus achieving peak bandwidth of about 70 Mbytes/second.

5.2 Software Overhead

We measured the software overhead required by typical message passing primitives. Because SHRIMP offers user-level communication, applications are free to use customized message passing operations rather than a single, generic mechanism. As a result, the software overhead cannot be characterized by a single number; we must measure the overhead induced by a variety of message passing primitives. Generally, primitives with richer, more useful, semantics require higher overhead than do simpler primitives.

We measured software overhead as the total number of instructions required by the sending and receiving processors to implement a particular primitive.

Experimental Environment

Because we do not yet have SHRIMP hardware, we measured software overheads on an experimental implementation environment. This environment can be viewed as a restricted version of SHRIMP – application code that works on the implementation environment will run without change on a real SHRIMP system. Hence, our instruction counts are accurate.

The implementation environment consists of two i486-based Xpress PCs, connected via a pair of Pipelined RAM (PRAM) network interfaces [17]. Each network interface contains 32 Kbytes of dual-ported SRAM which is mapped to the SRAM of the other in a manner similar to a complementary SHRIMP single-write, automatic-update mapping. The PCs run a modified OSF-1/MK AD operating system.

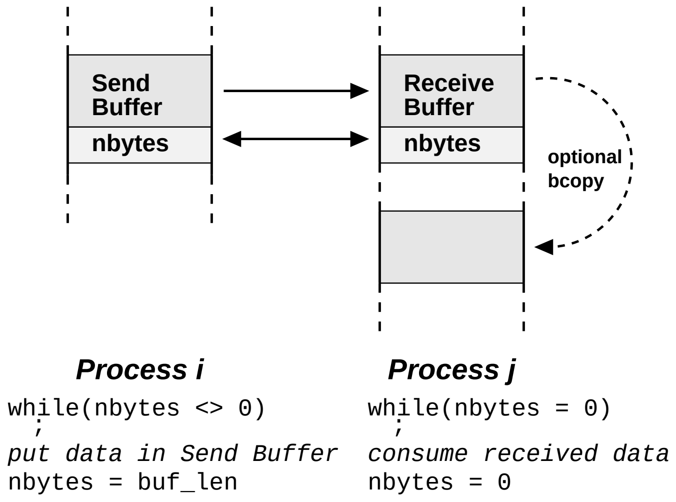
Because the PRAM interface does not support deliberate-update transfers, we could not run the deliberate-update code in the experimental environment. However, we are confident that this code is correct, since it is very small (less than 50 lines).

Single Buffering

We used the experimental environment to measure the performance of single-buffered send and receive operations. These primitives use a single memory buffer, mapped by the sender and receiver, to communicate data.

A simple way to implement single buffering is to use an automatic-update virtual memory mapping. Figure 5 shows an example implementation using this idea. A send-buffer in the memory of the sending process is mapped to a receive-buffer in the memory of the receiving process using an automatic-update mapping. The processes use a single flag, mapped for bidirectional automatic update, to synchronize their access to the buffer and to transmit the message size from sender to receiver.

To send a message, the sending process waits until the `nbytes` flag is set to zero, signifying that the buffer is empty. The sender puts the message data into the send buffer, then sets the `nbytes` flag equal to the message size. Because the send-buffer is mapped to the remote receive-buffer, the SHRIMP hardware automatically propagates the data to the receive-buffer. To receive a message, the destination process waits until `nbytes` is nonzero. After consuming the message data, it sets `nbytes` to zero to indicate that the buffer is available.



```
while(nbytes <> 0)      while(nbytes = 0)
  ;                       ;
put data in Send Buffer consume received data
nbytes = buf_len        nbytes = 0
```

Figure 5: *Single-buffered transfer*

Depending on the circumstances, the receiver may choose to copy the message out of the receive buffer. Although this uses some CPU time, it allows the sender to start the next transfer sooner. Since the decision to copy depends on the situation, we measured single-buffering overhead both with and without copying. Without copying, a single-buffered message requires 9 instructions: 4 for the sender and 5 for the receiver. Copying adds an additional overhead of 12 instructions on the receiving side, not including per-byte copying costs.

Double Buffering

A disadvantage of single buffering is that the sender cannot start transmitting data until the receiver has finished consuming the previous message. A technique known as *double buffering* can be used to overlap the consumption of one message with the transmission of the next.

Figure 6 shows the double-buffering method used in a typical multicomputer program. This method requires unrolling the loop of each process once, and using two buffers for each communication buffer of the original loop, one for odd iterations and another for even iterations.

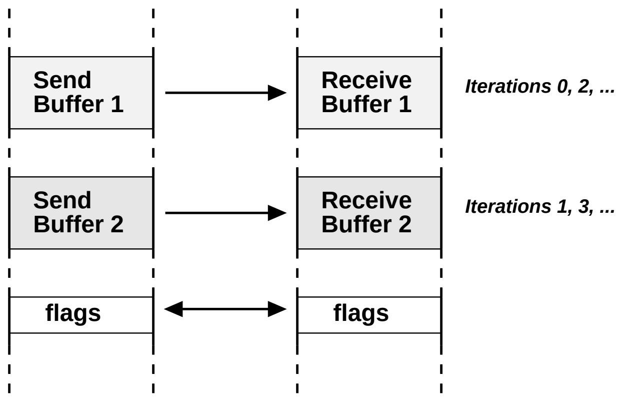


Figure 6: *Double buffered transfer*

The cost of double-buffering depends on the structure of the loop in the original program. There are three kinds of loops, two of which require synchronization between iterations. Since this synchronization is not necessary for message transport, we will not consider it part of the message-passing overhead.

In the first kind of loop, iteration $i+1$ uses data produced by iteration i. In this case, neither the sender nor the receiver must wait for buffers to empty or fill, since these conditions are ensured by the barrier synchronization. Therefore, the software overhead is 2 instructions to swap buffer pointers between odd and even iterations.

In the second kind of loop, the receive side uses data sent in the same iteration, so it needs to spin on a data-arrival flag. The sender does not need to wait for the send buffer contents to be consumed, because this is ensured by the barrier synchronization. In this case, the software overhead is 8 instructions: 3 for the sender and 5 for the receiver.

In the third kind of loop, synchronization is not required between iterations; all the necessary synchronization is provided by messages. This type of loop requires the receiving process to spin until data has arrived in the receive buffer. Additionally, before re-using a send buffer, the sender waits until the buffer's previous contents have been consumed. In this case, the software overhead is 10 instructions: 5 for the sender and 5 for the receiver.

Deliberate-Update Transfer

Pages mapped in deliberate-update mode do not transfer data until the originating process issues an explicit "send" operation. This operation is issued by accessing memory-mapped command pages from the user level, as described in Section 4.3.

We wrote a small macro that implements deliberate-update send. If the data to be transferred spans a page boundary, our implementation uses a series of single-page DMA transfers (Section 4.3). Software overhead is minimized between successive transfers because the preparation of the next page transfer command is overlapped with the outgoing DMA of the current transfer. In the simplest case, when only a single data transfer is required, this initiation requires 13 instructions. Checking whether a DMA operation has finished is much cheaper, requiring only 2 instructions.

NX/2 Primitives

Finally, we implemented versions of the standard send and receive operations offered by most message passing systems. We used the Intel NX/2 csend and crecv primitives, which buffer incoming messages in system-managed memory, and use message types to dispatch messages in a FIFO fashion [24]. In our implementation, we restrict the message type to a 16-bit integer, and assume that each message type represents only point-to-point communication (i.e. there is only one sender associated with each message type).

We expect SHRIMP to have two main advantages over existing systems when implementing these primitives. First, SHRIMP allows user-level communication, so buffer management can be moved to the user level, thus avoiding the overhead of copying data across the user/kernel boundary. Second, our memory-mapped interface semantics will allow data transport to be controlled using fewer instructions, and interrupting the CPUs less often.

The current implementation requires 73 instructions for csend and 78 instructions for crecv, which is about 1/4 of the overhead of the Intel implementation for the iPSC/2. We compare to the iPSC/2, since it uses i386 CPUs which have the same instruction set as the CPUs in both our implementation environment and the final SHRIMP system. The NX/2 implementation is more general, but introduces much higher overhead than our implementation. The NX/2 csend requires 222 instructions on the fast path to send a message, plus the cost of a system call and a DMA send interrupt. The NX/2 crecv overhead includes 261 instructions on the fast path to receive and dispatch a message, plus the cost of a system call and a DMA receive interrupt.

6 Related Work

The traditional method of designing network interfaces for multicomputers is based on DMA data transfer. Examples include the NCUBE [22], iPSC/2 and iPSC/860 [21]. The network interface designs of these machines are very similar. 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 it can dispatch the arrived message. The application makes a system call to receive the message. The network interface based on this approach is simple and can be used for commodity PCs or workstations. The main disadvantage is that message passing costs are usually thousands of CPU cycles, with the best implementation [29] still requiring over 100 CPU cycles.

One solution to the problem of software overhead is to add a separate processor on every node just for message passing [20, 12, 11]. Examples of this approach are the Intel Paragon and Meiko CS-2. 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 [5, 10, 6]. 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.

The Connection Machine CM-5 implements user-level communication through memory-mapped network interface FIFOs [16]. Protection is provided through the virtual memory system, which controls access to these FIFOs. However, there are a limited number of FIFOs so they must be shared within a partition (subset of nodes), restricting the degree of multiprogramming. Protection is provided between separate partitions, but not between processes within a partition. Since packet headers must be constructed by applications, the message passing overhead is still hundreds of CPU instructions.

Several shared memory architecture projects use the page-based, automatic-update approach to support shared memory. Examples include Memnet[9], Merlin [19] and its successor SESAME [30], the Plus system [4], and Galactica Net [15]. These systems do not provide a mechanism for high-bandwidth, low-overhead block data transfer.

Several parallel architectures use multiple threads [20, 27, 2, 1] to overlap communication with computation. These approaches require applications or compilers to create multiple threads on each node, and require the node CPU to switch thread contexts very fast.

The idea of automatic-update data delivery in our network interface is derived from the Pipelined RAM network interface [17]. The PRAM network interface allows physical memory mapping only for a small amount of memory on the network interface board.

7 Conclusions

The SHRIMP project uses commodity components, and a custom-designed network interface to build a new multicomputer. Communication in SHRIMP is based on remote virtual memory mapping. The SHRIMP interface offers advantages of both flexibility and performance over traditional multicomputer interfaces.

The communication abstraction offered by SHRIMP has many advantages over the interfaces offered by most other multicomputers. Separating destination specification from data movement allows the common case of communication to execute entirely at user-level, even in the presence of arbitrary multicomputer scheduling policies.

The memory-mapped communication model is more flexible than the traditional FIFO-based approach. FIFOs can easily be emulated using memory mappings, and memory mappings offer a wealth of additional possibilities, most notably automatic replication of data structures to remote processes. The tradeoff in the memory mapped communication model is its property of connection-oriented communication. Certain programs may need to use the same data buffers to communicate with more than one node in the system. The remote virtual memory-mapped communication model requires either using extra buffering and copying, or changing mappings.

As a consequence of user-level communication, SHRIMP does not impose any buffer management policy on applications. This flexibility allows applications, user-level libraries, or compilers to develop their own, customized buffering strategies.

One interesting feature of the virtual memory-mapped network interface is that it snoops the memory bus of the local node. Using a single-write automatic-update mapping, this feature allows the local CPU to see only the cache write latency and to overlap its subsequent instruction executions with network communication.

In this paper, we have presented limited experimental results on a few message-passing primitives. We are currently carrying out a more careful study to compare the remote virtual memory-mapped communication model with other message passing models.

As mentioned in the paper, we are in the process of prototyping our network interface for the Xpress Pentium PC system and the Paragon routing backplane. We expect to complete our first working hardware by the second quarter of 1994.

Acknowledgements

We would like to thank Otto J. Anshus, Douglas W. Clark, and Richard J. Lipton sincerely for their numerous contributions to the SHRIMP project and to this paper. This work was supported by National Science Foundation Grant CCR-9020893, ARPA and ONR under contracts N00014-91-J-4039, and Intel Supercomputer Systems Division. Matthias Blumrich and Richard Alpert were supported in part by ARPA Fellowships in High Performance Computing administered by the Institute for Advanced Computer Studies, University of Maryland.

References

[1] Anant Agarwal, David Chaiken, Kirk Johnson, David Kranz, John Kubiatowicz, Kiyoshi Kurihara, Beng-Hong Lim, Gino Maa, and Dan Nussbaum. The MIT Alewife machine: A large-scale distributed-memory multiprocessor. Technical Report MIT/LCS/TM-454, Massachusetts Institute of Technology, June 1991.

[2] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *Proceedings of International Conference on Supercomputing*, pages 1–6, 1990.

[3] BCPR Services Inc. *EISA Specification, Version 3.12*, 1992.

[4] Roberto Bisiani and Mosur Ravishankar. Plus: A distributed shared-memory system. In *Proceedings of 17th International Symposium on Computer Architecture*, pages 115–124, May 1990.

[5] Shekhar Borkar, Robert Cohn, George Cox, Thomas Gross, H.T.Kung, Monica Lam, Margie Levine,

Brian Moore, Wire Moore, Craig Peterson, Jim Susman, Jim Sutton, John Urbanski, and Jon Webb. Supporting systolic and memory communication in iwarp. In *Proceedings of 17th International Symposium on Computer Architecture*, pages 70–81, May 1990.

[6] William J. Dally, Roy Davison, J. A. Stuart Fiske, Greg Fyler, John S. Keen, Richard A. Lethin, Michael Noakes, and Peter R. Nuth. The message-driven processor: A multicomputer processing node with efficient mechanisms. *IEEE Micro*, 12(2):23–39, April 1992.

[7] William J. Dally and Charles L. Seitz. The torus routing chip. *Distributed Computing*, 1:187–196, 1986.

[8] William J. Dally and Charles L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, C-36(5):547–553, May 1987.

[9] G. S. Delp, D. J. Farber, R. G. Minnich, J. M. Smith, and M. C. Tam. Memory as a network abstraction. *IEEE Network*, 5(4):34–41, July 1991.

[10] Dana S. Henry and Christopher F. Joerg. A tightly-coupled processor-network interface. In *Proceedings of 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 111–122, October 1992.

[11] Mark Homewood and Moray McLaren. Meiko CS-2 interconnect elan – elite design. In *Proceedings of Hot Interconnects '93 Symposium*, August 1993.

[12] Intel Corporation. *Paragon XP/S Product Overview*, 1991.

[13] Intel Corporation. *Express Platforms Technical Product Summary: System Overview*, April 1993.

[14] Intel Corporation. *Pentium Processor Data Book*, 1993.

[15] Andrew W. Wilson Jr. Richard P. LaRowe Jr. and Marc J. Teller. Hardware assist for distributed shared memory. In *Proceedings of 13th International Conference on Distributed Computing Systems*, pages 246–255, May 1993.

[16] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, Daniel Hillis, Bradley C. Kuszmaul, Margaret A. St. Pierre, David S. Wells, Monica C. Wong, Shaw-Wen Yang, and Robert Zak. The network architecture of the connection machine CM-5. In *Proceedings of 4th ACM Symposium on Parallel Algorithms and Architectures*, pages 272–285, June 1992.

[17] Richard J. Lipton and Jonathan S. Sandberg. PRAM: A scalable shared memory. Technical Report CS-TR-180-88, Princeton University, September 1988.

[18] Richard J. Littlefield. Characterizing and tuning communications performance for real applications. In *Proceedings of the First Intel DELTA Applications Workshop*, pages 179–190, February 1992. Proceedings also available as Caltech Technical Report CCSF-14-92.

[19] Creve Maples. A high-performance, memory-based interconnection system for multicomputer environments. In *Proceedings of Supercomputing '90*, pages 295–304, November 1990.

[20] R.S. Nikhil, G.M. Papadopoulos, and Arvind. *T: A multithreaded massively parallel architecture. In *Proceedings of 19th International Symposium on Computer Architecture*, pages 156–167, May 1992.

[21] Steven Nugent. The iPSC/2 direct-connect communication technology. In *Proceedings of 3rd Conference on Hypercube Concurrent Computers and Applications*, pages 51–60, January 1988.

[22] John Palmer. The NCUBE family of high-performance parallel computer systems. In *Proceedings of 3rd Conference on Hypercube Concurrent Computers and Applications*, pages 845–851, January 1988.

[23] PCI Special Interest Group. *PCI Local Bus Specification, Revision 2.0*, April 1993.

[24] Paul Pierce. The NX/2 operating system. In *Proceedings of 3rd Conference on Hypercube Concurrent Computers and Applications*, pages 384–390, January 1988.

[25] R.F. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architecture. In *Proceedings of 2nd International Conference on Architectural Support for Programming Lanugages and Operating Systems*, pages 31–41, October 1987.

[26] Avtar Saini. An overview of the intel pentium processor. In *Compcon Spring '93*, pages 60–62, February 1993.

[27] Burton J. Smith. A pipelined, shared resource MIMD computer. In *Proceedings of International Conference on Parallel Processing*, pages 6–8, 1978.

[28] Roger Traylor and Dave Dunning. Routing chip set for Intel Paragon parallel supercomputer. In *Proceedings of Hot Chips '92 Symposium*, August 1992.

[29] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of 19th International Symposium on Computer Architecture*, pages 256–266, May 1992.

[30] Larry D. Wittie, Gudjon Hermannsson, and Ai Li. Eager sharing for efficient massive parallelism. In *Proceedings of the 1992 International Conference on Parallel Processing*, pages 251–255, August 1992.