DESIGN AND IMPLEMENTATION OF NX MESSAGE PASSING USING SHRIMP VIRTUAL MEMORY MAPPED COMMUNICATION

Richard Alpert, Cezary Dubnicki, Edward W. Felten, and Kai Li
Department of Computer Science
Princeton University
Princeton, New Jersey 08544-2009
{alpert|dubnicki|felten|li}@cs.princeton.edu

Abstract — *This paper describes the design, implementation and performance of the NX message-passing interface on the Shrimp multicomputer. Unlike traditional methods, our implementation, exploiting Shrimp's virtual memory-mapped communication facility, performs buffer management at user level without using a special message-passing processor, and requires no CPU intervention upon message arrival in the common cases. For a four-byte message, our implementation achieves a user-to-user latency of 12 microseconds, about factor of four smaller than that on the Intel Paragon. For large messages, our implementation quickly approaches the bandwidth limit imposed by the Shrimp hardware.*

1 INTRODUCTION

NX has been a common message-passing interface on multicomputers for about a decade [16]. Its implementations, however, require substantial software overhead. For example, one-way user-to-user latency for a four-byte message on the Intel Paragon is about 50 microseconds, of which only about a microsecond is in hardware [17]. Over 98% of the overhead is in software to achieve protected communication, buffer management, translation between virtual and physical addresses, DMA setup, message receiving, dispatching and delivery. It is clear that reducing the software overhead is the key to achieving low-latency message passing.

This paper describes the design and implementation of the NX message-passing interface on the Shrimp multicomputer. Unlike traditional NX implementations, ours operates entirely at user level. It takes advantage of Shrimp's virtual memory-mapped communication facility [4, 6] which provides a basic mechanism to allow a process to transfer data directly from its virtual memory into the memory of a remote process with full protection.

Although implementing a message-passing library entirely at user level can eliminate the overhead of checking for protection and translation between virtual memory and physical memory addresses, it requires careful design to achieve good performance. The main design challenge is to avoid interrupting the CPU on the receiving node. In traditional implementations of message-passing, the receiving processor decides where in memory to place each packet as it arrives. To avoid interrupting the receiving process upon packet arrival, the receive buffer must be reserved before the message is sent and the sending process must decide where the data is to be placed.

We have implemented the NX message-passing library and measured its performance on our Shrimp multicomputer prototype. For a four-byte message, our implementation achieves a user-to-user latency of 12 microseconds, a significant improvement over the the Intel Paragon implementation which relies on a special message-passing processor. For large messages, our implementation quickly approaches about 23 Mbytes/second, which is close to the bandwidth limit imposed by the Shrimp hardware.

The remainder of this paper is organized as follows: Section 2 describes the NX message passing interface, section 3 discusses the Shrimp multicomputer and its virtual memory mapped communication model. Section 4 presents our design, and section 5 analyzes some issues raised by the design. Experimental evaluation of our implementation is described in section 6. Section 7 gives related work, and section 8 concludes.

2 NX MESSAGE PASSING

The NX message passing interface, supported on the Intel iPSC/860 and Intel Paragon, is a tagged or "typed" message passing system. The tag, a sender-side, application-supplied integer, allows a receiver to exercise some control over the order in which it receives messages. The core of the NX interface provides calls to send and receive messages.

Synchronous Communication The most frequently used NX communication calls include `csend` and `crecv`. `csend` is a synchronous send operation in that it completes only when the buffer from which data are sent can be reused; completion of `csend` does not imply that data have left the sending node or have reached the receiver. The parameters of `csend` include destination node, process identifier, a tag (or "type"), a buffer address, and a size. The parameters of `crecv` are a chosen message tag, a buffer address, and a size.

Asynchronous Communication A receiver may "post" an asynchronous receive by making the `irecv` call. Data structures at the receiver are modified so the arriving message can be copied directly into user-level address space, then the receiving process resumes useful work. Asynchronous receive calls return an identifier that can be used to determine the state of the pending receive, checking whether it has completed (`msgdone`) or waiting for it to complete (`msgwait`).

Similarly, using `isend`, a process may send asynchronously. When an `isend` call completes, there is no guarantee that the buffer from which the data are being sent can be used again without unpredictable damage to the message. As with `irecv`, `isend` returns an ID which can be used to determine the state of the send buffer.

Interrupting Communication NX also supports communication calls that invoke an application-supplied handler when the operation completes. The "handler-send" function, `hsend`, initiates a send operation, then executes the specified application-supplied handler after the user-space buffer contents have been copied or dispatched. As with `csend`, semantics guarantee only that the user-space send buffer can be reused without compromising the integrity of the message, and not that the message has been delivered.

Like `hsend`, the `hrecv` asynchronous receive function takes, as one of its parameters, a user-defined function. This function is executed after incoming data have been copied into their final user-level destination, whenever that may occur.

3 SHRIMP VIRTUAL MEMORY MAPPED COMMUNICATION

<u>Shrimp Hardware</u> The Shrimp system was designed with three goals in mind: (1) low cost, achieved by using commodity hardware and software, (2) low-latency communication, achieved by supporting communication directly at the user level, and (3) protection between parallel jobs, achieved by a combination of hardware and software techniques.

Each node of the Shrimp system is a 60 MHz Pentium-based Xpress PC system [13]. The Xpress PC consists of a Pentium CPU with a second-level cache connected to DRAM memory modules and an EISA [3] I/O bus adapter via the Xpress memory bus. The caches snoop DMA transactions and automatically invalidate corresponding cache lines, keeping consistent with all main memory updates, including incoming DMA.

The interconnect is an Intel Paragon routing backplane, a two-dimensional mesh of Intel iMRC routers [19], essentially faster and wider versions of the Caltech Mesh Routing Chip [8]. The backplane supports deadlock-free, oblivious wormhole routing [9] and preserves the order of messages from each sender to each receiver.

<u>Virtual Memory Mapped Communication</u>
The Shrimp hardware and software together support the *virtual memory mapped communication* (VMMC) model [10]. In VMMC, a process wanting to receive messages *exports* a region of its virtual memory. A remote process can then *import* the region[a], setting up an *export-import mapping*. Setting up such a mapping requires system calls by both processes.

Once an export-import mapping exists, the sending process may transfer data directly into the exported memory of the receiving process, without kernel involvement. When the data arrives at the destination process, the Shrimp hardware deposits it directly into memory, without any need to interrupt the destination processor. Although interrupts are not necessary in most cases, the programmer can choose to cause an interrupt on the arrival of a particular packet.

Virtual memory-mapped communication supports two data transfer modes, `deliberate update` and `automatic update`. In deliberate update, the sending process explicitly requests the transfer of data from the sender's virtual address space to a previously imported receive buffer. The Shrimp network interface has a user-level DMA mechanism [5] which allows a user program to initiate a deliberate update data transfer

[a] An exporter can restrict possible importers, verifying such restrictions when an import is attempted.

in two user-level instructions.

Automatic update transfers data without explicit processor involvement on either the sender-side or the receiver-side. To allow automatic update, a sender must first create an `automatic update binding`. This operation designates a region of its virtual memory as an automatic update area and binds it to an imported remote receive buffer. Once the binding has been established, any write to the source memory region by the sender is propagated to the receive memory region automatically.

A deliberate update can be made from any region in the sender's user-level virtual address space to any imported buffer. The location of the data source, its size, and the destination buffer all are designated at the time the data transfer is initiated. In contrast, the sender must designate an automatic update region in its user-level virtual address space and bind it to an imported buffer in advance of any automatic update data transfer.

A receiver, when it exports a buffer, can associate a user-level handler with that buffer. A sender, if it sends data to that region "with notification," will cause that user-level handler to be executed by the receiver after the data have been written into the receiver's address space.

Virtual memory-mapped communication supports protected, user-level message passing. The mappings or bindings can be created only by system calls which verify protection constraints. After the mappings or bindings are created, regular virtual memory hardware will ensure protection among multiple processes. Once a mapping or binding is created, it can be used to send and receive messages without violating the protections established by the kernel calls.

An important feature of virtual memory-mapped communication is that it permits and encourages user-level buffer management. This feature allows libraries and user programs to customize their own buffer management to implement zero-copy protocols to achieve very low-latency message passing and to exploit the raw bandwidth available in hardware. The challenge in implementing a message-passing interface such as NX at user level is to exploit fully the virtual memory-mapped communication mechanism, while maintaining protection constraints and NX semantics.

More complete descriptions of Shrimp can be found elsewhere [4, 5, 6].

4 IMPLEMENTING NX MESSAGE PASSING ON VMMC

Shrimp provides us the opportunity to design an NX implementation that avoids interrupts in the common case. This offers improved performance, but leads to several new design challenges.

While the advantages of a sender's writing directly into a receiver's address space are clear, the challenges this raises are less obvious. Not being immediately involved in each communication action, a receiver won't know when a message has arrived. Uncoordinated senders cannot share message buffers on a receiver, requiring distinct message areas for each sender.

A receiver expecting a message and not knowing the identity of the sender (or identities of a set of possible senders) must search a larger space to determine whether that message has arrived.

Synchronous Communication We begin by describing how the synchronous communication calls `csend` and `crecv` are implemented. Once this is done, we will explain how the synchronous mechanisms are generalized to handle asynchronous and interrupting communication.

We use two different protocols, depending on the size of the data transfer. Small messages use a simple one-copy protocol, while large messages use a more complicated zero-copy protocol. In our baseline system, messages carrying payloads of size 1012 bytes or smaller use the small-message protocol (resulting in 1024-byte packets).

Small Messages Our small-message protocol uses three data structures.

- *Buffers* are regions in the receiver's memory where messages are temporarily stored. Each buffer is dedicated for the use of a single sender; there are generally several buffers per sender[b].
- A *freelist* is a list of buffers that a particular receiver has authorized a particular sender to fill. There is a separate freelist for each sender/receiver pair.
- A *receive queue* is a linked list of messages that have arrived at a node and have been seen by the

[b] This design uses more memory than alternatives which dynamically allocate buffers to senders, but we believe the memory requirements are still reasonable. See section 5 for more discussion of this point.

NX library, but have not yet been received by the application program.

Before describing these data structures in more detail, we first give a rough description of how the protocol works.

The `csend` operation first allocates a receiver-side buffer into which to place the message, by examining the appropriate freelist. It then transmits the message body, along with a small header, into that buffer, and returns.

The `crecv` operation first scans the receive queue, looking for a message with the right tag. If it finds one, it copies the message from its buffer into the application's memory, puts the buffer on the appropriate freelist, and returns. Otherwise, it starts polling in a loop, scanning its buffers for newly arrived messages. If an arriving message has the wrong tag, the message is put into the receive queue. When an arriving message with the right tag is found, the message data is copied into the application's memory, the message buffer is put on the appropriate freelist, and `crecv` returns immediately.

Note that some time may elapse between the point when a message arrives in a receiver's buffer and the point when the message is put into the receive queue. Messages are moved to the receive queue lazily, only when the processor is spinning in `crecv` and has nothing better to do.

Buffers When our NX library is initialized, each processor allocates a block of buffers for each other processor to send to. Each buffer is a fixed-size region of memory for message data, followed by a small "header." Each block of buffers is `export`ed. Each processor then `import`s the blocks of buffers that other processors have allocated for it. Once this has occurred, Shrimp allows the sender to transfer data directly into any of the receiver-side buffers allocated for it.

Note that the "header" is actually at the end of the buffer slot. When a message is sent, the message data is put at the end of the data area, so that the message body and the header lie in contiguous addresses; this allows the body and header to be sent in a single data transfer. Since the header is after the data in memory, it is delivered to memory after the data, so the receiver may infer from the header's arrival that the data have arrived. The start address of the data is determined by examining the *size* field of the header.

Buffer Allocation using Freelists Before a sender may use a buffer, the receiver must first *grant* the buffer to the sender, signifying that the buffer is empty and is available to be used. The sender fills the buffers in the same order in which the receiver granted them. The receiver keeps track of the sequence of buffers it has granted each sender, so the receiver always knows which buffer the sender will use next.

A receiver uses a *freelist* to grant buffers to a particular sender. Logically, a freelist is just a producer-consumer queue which allows `put` operations by the receiver and `get` operations by the sender. To grant a buffer, the receiver `put`s the buffer ID into the queue; to get the next granted buffer, the sender does a queue `get` operation.

Each queue is implemented as a simple array. The queue's memory is `export`ed by the sender and `import`ed by the receiver at initialization time. This allows the receiver to write data into the queue, and the sender to read data from it. Successive `put` operations write their values to successive locations in the array, in round-robin fashion. Successive `get` operations read their values in a similar round-robin fashion. Empty slots are filled in with a special invalid value, and the array is sufficiently large that the queue never becomes full. (The number of entries in a queue can never exceed the number of buffers the receiver has allocated for the sender.) This implementation of distributed queues is very efficient: a `put` operation requires one word to be written into remote memory across the network, and a `get` operation can be satisfied with a single local memory read. In practice, `put`s are bundled and transferred by deliberate update.

Allocating More Buffers Recall that once a message has been put into a buffer by the sender, the message continues to occupy the buffer until the message is delivered to the application process. In the unlikely event that all buffers are full when a sender wants to send a message, something must be done to cause more buffers to be allocated.

When a sender finds a freelist empty, it transmits a message with notification (interprocessor software interrupt) to a small, special buffer just for this purpose at the receiver, asking for buffers to be freed. The receiver responds by allocating more buffer memory, copying some messages into the new buffers, and granting the vacated buffers back to the sender. We expect this operation to be very rare.

Large Messages Larger messages are transferred using a different protocol. A sender first sends a "scout" message to the receiver. The scout, carrying the message tag, message size, and a sender-supplied key, is sent like any other small message.

Once the scout packet has been sent, the sender waits for the receiver to reply. Since the sender has nothing to do in the meantime, it begins copying the large message into a local (sender-side) buffer. Although the sender is copying, we still consider this a zero-copy protocol; the copying is not on the critical path. If the receiver replies to the scout while the sender is copying, the copy operation is abandoned and the message is sent directly from the original send area. If, on the other hand, the copy completes before the receiver has replied, then the sender-side application program is resumed, and the message is eventually sent from the copy area. Since the semantics of NX prohibits returning from a synchronous send (`csend`) call before the send buffer can be reused, this safe copy permits the sender to continue computing even if the receiver isn't ready to receive the data.

When the receive call finds the scout message during the normal execution of a `crecv` call, the receiver, if it has not done so already, `export`s the message's final destination memory. The receiver then transfers (with notification) that buffer's ID into a special location in the sender's address space.

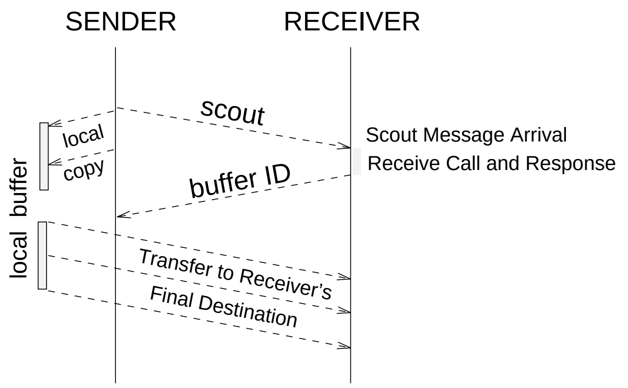

Figure 1: *Large Message Transfer — The sender may or may not complete the local copy prior to sending (see text).*

Upon receipt of the buffer ID, a user-level library handler is executed on the sender side. (See Figure 1.) This causes the sender to abandon its copy operation, if it hasn't finished yet, and to send the message data directly into the receiver-side application memory.

Asynchronous Communication Because a receiving Shrimp node's processor is not involved in data transfer, implementing the asynchronous receive call `irecv` efficiently presented a particular challenge.

irecv By posting an NX asynchronous receive (`irecv`), an application provides enough information to enable an incoming message to be written directly to its final destination upon arrival. Since NX semantics do not allow the receiver-side library to know in advance which processor a message will come from next, the receiver cannot ask the sender to take special action when a particular message is sent.

In our implementation of `irecv`, the parameters of the `irecv` call are stored in a data structure at the receiver. A user-level library handler is designated when the message buffer area is exported during initialization. If an unsatisfied `irecv` is pending, the receiver enables notifications for the small-message buffer area. When a message arrives, the handler compares the type of the arriving message with the types of unsatisfied `irecv` calls. If there is a match, the data are copied into their final user-level destination, and a flag, read by `msgwait` and `msgdone`, is set to indicate completion of the receive.

isend The NX asynchronous send, `isend`, initiates a send, returning immediately. Like `irecv`, `isend` returns a message ID that can be used by `msgwait` and `msgdone` to determine whether the original send buffer can be reused. On Shrimp, `isend` simply calls `csend` and returns a flag that `msgwait` and `msgdone` always interpret as *true*.

Interrupting Communication The two interrupting NX communication functions, `hsend` and `hrecv`, permit a user to define and specify handlers to be executed upon completion of the communication (see Section 2).

On Shrimp, `hsend` is the less interesting of the two, simply sending `csend` and calling the handler upon return from the send.

Because a Shrimp node is not actively involved in receiving a message, implementing `hrecv` poses a more interesting challenge. When the library exports incoming message areas, a user-level library notification handler is designated, exactly as is done for `irecv`. All sends to that area are "with notification". When an `hrecv` is posted, the receive area first is examined to determine whether a message carrying the requested tag already has arrived. If it has, the incoming data are placed in the user's address space and the handler is invoked. If not, notifications on the receiver's message area buffer are enabled, the parameters of the `hrecv` call (message type, final destination buffer address, final destination buffer size, and the handler) are stored in a list of unsatisfied `hrecv` handlers. When a small message is received, the library notification handler compares the message tag with the tags of messages for which an `hrecv` has been posted. If there is a match, the library handler calls a normal `crecv` and, upon its completion, executes the handler specified in the `hrecv` call. If there is no match, the notification handler simply returns.

The cost of a notification is incurred whenever there is an unsatisfied `hrecv` or `irecv` pending. Notifications are normally disabled by the receiver, so the extra cost is incurred only by programs which use `hrecv` or `irecv`, and then only when such calls are outstanding. This highlights an advantage of a user-level implementation of a message-passing library: an application's performance need not incur the penalty imposed by some given functionality if that functionality is not required by the application.

5 ANALYSIS OF THE DESIGN

Several issues surfaced while designing the Shrimp NX library.

Multiple versus Single Message Areas To avoid requiring senders to coordinate their message transmission with one another, a given receiver-side buffer can be used by only one sender at a time. In our implementation, message buffers are assigned forever to the same sender, with a fixed number of buffers for each sender-receiver pair, for a global total of $O(p^2)$ message buffers. While simple, this wastes space, since some pairs of nodes communicate rarely or not at all.

Because the communicating processes trust one another, a more economical use of space could be achieved by having each receiver export a single large message area (still divided into individual message buffers as before). All potential senders would import that message area (giving a many-to-one "mapping" of that message area). The message buffers would be assigned as before, with different buffers assigned to different senders, and receivers would send buffer credits to senders as before. The only difference would be that a buffer would sometimes be taken from one sender and granted to another. Such a design would permit dynamic allocation of the message buffers, allocating buffers only where they are needed. The design of dynamic buffer allocation algorithms is an area for future research.

Copying In a traditional receiver-managed communication system, the buffer allocation code is aware of the state of each receive buffer. Without communication from the receiver, the sender in a sender-managed communications scheme cannot determine when a receive buffer can be reused. The separation of sender-managed message areas from receiver-managed message queues solves this problem, at the cost of an extra copy. By having the receiver determine explicitly grant buffers to senders, we can keep senders aware of the state of the buffers they are filling at the receiver.

In a zero-copy protocol, senders must deposit data directly in its final destination in the receiver's virtual address space. This requires a an exchange of packets between sender and receiver so the sender can determine where to put the message data and when it is safe to overwrite the receiver's memory. For small messages, it is cheaper to copy than to perform the extra packet exchange. For large messages, however, it pays to use a zero-copy protocol.

Our zero-copy protocol uses the scout message to inform the receiver that a large message is ready to be sent. (In NX, the receiver-side library does not know *a priori* who the sender is or how large the message is.) The scout message, traveling like any other message using the small message protocol, carries this information. The scout packet also serves as a "placeholder" to ensure that large messages and small messages do not pass each other but are delivered in the same order they were sent.

Once the sender and receiver are aware of each other's identities, the sender has the buffer ID of the final destination of the message, and knows that the receiver's buffer is ready, so data can be transferred directly.

Large Message Buffering The Shrimp NX library buffers large messages on the sender side. Buffering on the receiver side would require the receiving processor to suffer an interrupt on each large message arrival. More importantly, if the corresponding receive call had not yet been reached, the receiver would not be able to supply the sender with enough information to write the data directly to its final destination, completely eliminating any possibility of a zero-copy large message protocol. This places a large message copy operation on the critical path. Our large message zero-copy protocol removes copying from the critical path.

Buffer Availability In a receiver-managed system, the receiver immediately recognizes when its buffer supply is nearing exhaustion. New buffers can be allocated or existing ones emptied as needed. In sender-managed communication, a sender detects an inadequate supply of buffers at the receiver before the receiver does. Yet it is the receiver that must allocate new buffer space or consume messages to make room for incoming data. Our solution is to have the sender cause a user-level handler to be invoked at the receiver; the handler reclaims occupied buffers, freeing them for reuse. This is a rare event, so the cost of interrupting the receiver and invoking a user-level handler is amortized over many messages.

Polling vs. Blocking Our library, as described so far, always polls when it is waiting for a message. If the machine is shared with other jobs, it makes sense for a process to block when it has nothing useful to do. It would be easy to extend our library to do this. When a process had been polling for a while without any activity, it could enable notifications on its receive buffers and then suspend itself by calling a Shrimp procedure which blocks until a notification occurs. When a message arrived, the notification would cause the process to awaken immediately.

6 PERFORMANCE

The main goal of our performance experiments is simply to measure the user-to-user latency and bandwidth of the NX primitives for various message sizes. We measured two implementations, one using the automatic update mechanism and the other using the deliberate update mechanism. Since we use different protocols for large and small messages, we also conducted experiments with both protocols for each implementation. Such experiments allow us to determine the best message size at which to switch protocols for optimal message-passing performance.

We conducted our experiments on a four-node Shrimp multicomputer prototype. Each node of the Shrimp system is a 60 MHz Pentium-based Xpress PC system [18, 14, 13], with 32 MB DRAM and a 256 kB second-level cache. The nodes use an EISA [3] I/O bus. The operating system on each node is Linux with a thin layer of software to provide virtual memory-mapped communication [10].

Results To measure latency and bandwidth for each message size, we performed $10,000$ message-passing round trips, "bouncing" between two Shrimp nodes. We used the Pentium cycle counter to measure the elapsed time, and then divided by $20,000$ to derive the latency and bandwidth results. Figure 2 and Figure 3 show those results.

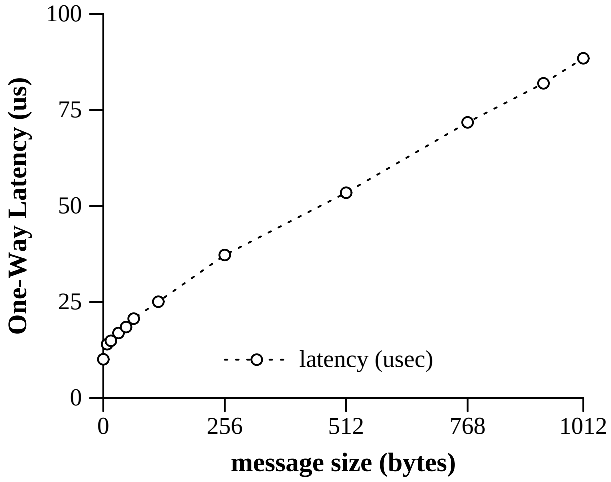

Figure 2: *Latency*

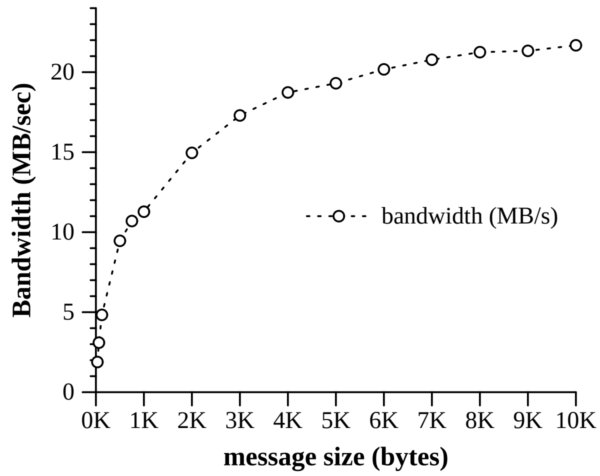

Figure 3: *Bandwidth*

As we can see, the one-way latency for a zero-byte message is 10 μsec. The minimum communication latency presently achievable on the raw Shrimp hardware and system software is 4.8 μsec [10]. (The main contribution to this latency is time needed for arbitration of the EISA bus). Thus, the software overhead of our NX implementation is about 5 μsec. This is an order-of-magnitude overhead reduction compared to the Intel Paragon implementation.

The bandwidth for 8KB messages approaches 23MB/sec, which is close to the limit of the Shrimp hardware (25MB/sec). 1KB messages can already achieve about 50% of the peak hardware bandwidth.

This result is again due to the low overhead implementation.

The bump in each curve at 1024 bytes reflects the switch between the small-message protocol and the large-message protocol. The small-message protocol spends an extra copy in order to avoid a packet round-trip, while the large-message protocol pays one packet round-trip to achieve a zero-copy protocol.

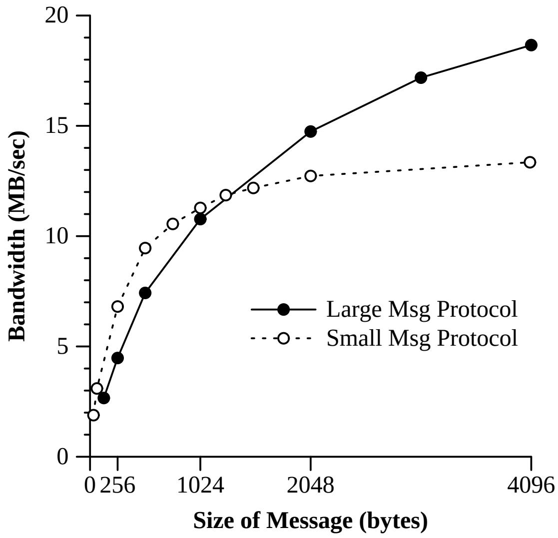

Figure 4: *Tradeoff between small-message protocol and large-message protocol*

Figure 4 justifies our decision to switch protocols at a message size of 1012 bytes, or a buffer size of 1024 bytes (including header). The figure shows that the small-message protocol is best for messages less than about 1300 bytes. We rounded down to 1024 bytes per buffer, so an even number of buffers would fit on a page.

Automatic Update vs. Deliberate Update As mentioned above, one of the implementations uses the automatic update mechanism and the other uses the deliberate update mechanism. Figure 5 shows the latency results for both cases for small messages.

The deliberate update version uses the Shrimp user-level DMA mechanism to transfer the message body. The automatic update version simply copies the data, followed by the message header, into a special send-buffer memory region; the Shrimp hardware snoops the writes to the send-buffer memory and automatically propagates them to the receiver. The Shrimp hardware combines these writes into a single network packet, since the writes are to consecutive memory addresses.

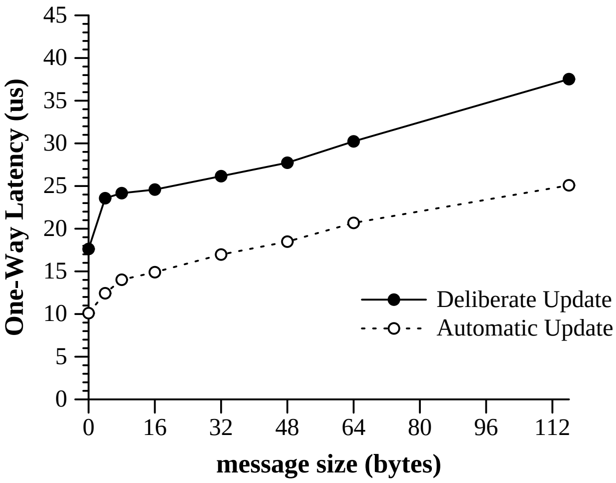

Figure 5: *Small Message Latency*

The deliberate update version is slower for two reasons. First, although the user-level DMA mechanism [5] requires only two user-level instructions to initiate a deliberate update DMA send, these instructions are a read and a write to the EISA I/O space, which require about half a microsecond each. Second, a separate deliberate update operation is required for each contiguous region that is sent. Since the message data and the message header are not contiguous, either two deliberate updates must be used, or some copying must be done to make the header and data contiguous.

Small messages are sent to a buffer area at the receiver which then copies them into their final destination, so there always is at least one copy in any small message protocol. Since the initiation of a message using the automatic update mechanism requires a local copy on the sender side, the automatic update protocol has two copies: one on the sender side and one on the receiver side. The deliberate update protocol, on the other hand, requires one copy at the receiver side for small messages.

Figure 6 shows the bandwidth results of the two implementations using the automatic update and deliberate update mechanisms.

Both curves approach their maximums quickly (at about 8KB message size). The automatic update mechanism has a slightly higher peak bandwidth. This occurs because the memory-copy (`bcopy`) operation to initiate an automatic update uses the memory bus, while an outgoing deliberate update DMA uses the EISA bus, which has a lower bandwidth than the memory bus. Thus the automatic update case can transmit the packet faster than the deliberate update

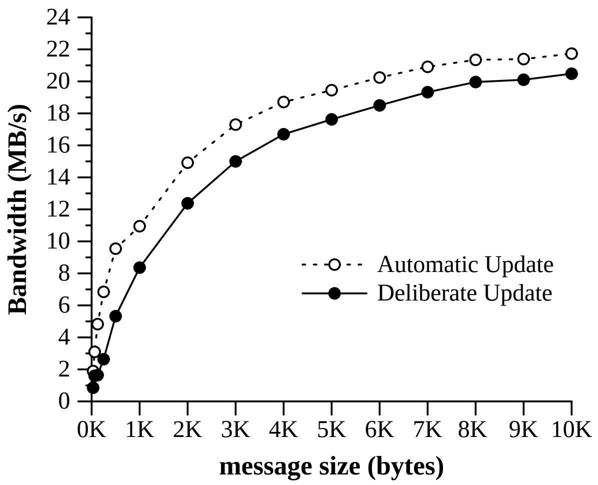

Figure 6: *Message Bandwidth*

case. In fact, automatic update performance is limited by the speed of the EISA bus on the receiving side.

The DMA bandwidth over the EISA bus is limited by its 8.25 MHz clock rate. The theoretical peak for the EISA bus is 33 MB/sec, or one four-word transfer per EISA clock cycle. However, it is impossible to transfer a word every cycle, because some cycles are stolen for memory refresh, and because the EISA specification requires that all DMA devices rearbitrate for the bus every 64 EISA cycles. The practical peak bandwidth of the EISA bus in our system is thus about 25 MB/sec. Many commodity PCs with an Xpress memory bus use faster I/O buses such as VESA or PCI buses. They deliver four times higher bandwidth than the EISA bus. If our Shrimp nodes used the PCI bus, we would see much higher peak bandwidths.

7 RELATED WORK

NX and similar libraries have traditionally been implemented in two ways. The first is to make communication a service of the operating system kernel, as on the iPSC/860 implementation of NX [16]. In this approach, arrival of a packet at a processor triggers and interrupt, which causes a kernel handler to run. The kernel handler examines the message header and then dispatches the message to memory. Buffer management is handled in the kernel, and the application program makes system calls to send and receive messages. The drawback of this method is that system calls and interrupts are expensive, and that kernel-level memory management can be complicated.

The second approach is to use a special processor to handle communication, as in the Paragon implementation of NX [17]. The communication processor runs specially written messaging code. It constantly polls the network interface to retrieve incoming packets without requiring interrupts. Application processes communicate with the communication processor by writing commands into a queue implemented in shared memory; no system calls are necessary. Although this approach eliminates the need for system calls and interrupts, it ties up a processor that could otherwise be used to run an application. In addition, buffer management is still done at kernel level.

Another way to improve NX message-passing performance is to take advantage of application-specific communication patterns to perform buffer management at compile time instead of run time to reduce the overhead of message passing [11]. This method works only for applications that have fixed communication patterns and requires a compiler to detect and generate application specific message-passing libraries.

MPI-FM [1] is a high-performance implementation of the MPI message passing interface, based on a port of the portable MPICH [2] layer to the minimal Illiois Fast Messages [15] library on the Myrinet [7] network. MPI-FM achieves a latency of 25 μsec and a peak bandwidth of 17.5 MB/s. The latency is lower than in many commercial MPP's, but is still about double the latency of our library.

8 CONCLUSIONS

We have described the design, implementation and performance of an NX message-passing interface library on the Shrimp multicomputer. For a four-byte message, our implementation achieves a user-to-user latency of 12 microseconds. The software overhead is only a few microseconds. For large messages, our implementation quickly approaches 23 MB/sec which is the bandwidth limit imposed by the Shrimp hardware. The results show that virtual memory-mapped communication can indeed support the NX message-passing interface well. A careful data structure design is the key to obtaining an efficient implementation.

We learned from our experiments that the deliberate update mechanism on the Shrimp multicomputer is not useful for NX to achieve high bandwidth. This is because the memory copying bandwidth is higher than the DMA bandwidth on the EISA I/O bus in our Shrimp nodes. Our experience suggests that the deliberate update mechanism should be implemented on a faster bus whose DMA bandwidth is close to the bandwidth of the memory system.

The software overhead of NX message-passing primitives in our implementation is an order-of-magnitute smaller than that of commercial systems such as CM-5, Intel Paragon and Meiko CS-2 [12]. We have not provided detailed comparision between our implementation and the traditional approaches. Doing so would require careful instrumentation to analyze the differences. We plan to conduct such studies in the near future.

References

[1] . MPI-FM version 1.0. `http://www-csag.cs.uiuc.edu/projects/comm`.

[2] . MPICH–A Portable Implementation of MPI. `http://www.mcs.anl.gov/home/lusk/mpich`.

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

[4] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputers. In *Proc. of the 21st Annual International Symposium on Computer Architecture*, pages 142–153, July 1994.

[5] Matthias Blumrich, Cezary Dubnick, Edward Felten, and Kai Li. Protected, user-level dma for the shrimp network interface. In *IEEE 2nd International Symposium on High-Performance Computer Architecture*, February 1996. To appear.

[6] Matthias Blumrich, Cezary Dubnicki, Edward Felten, Kai Li, and Malena Mesarina. Virtual-memory-mapped network interfaces. *IEEE MICRO*, 15(1):21–28, February 1995.

[7] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, February 1995.

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

[9] 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.

[10] Cezary Dubnicki, Liviu Iftode, Edward W. Felten, and Kai Li. Software support for virtual memory-mapped communication. to appear in 96' IPPS, May 1996.

[11] Edward W Felten. *Protocol Compilation: High-Performance Communication for Parallel Programs.* PhD thesis, University of Washington, August 1993.

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

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

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

[15] S. Pakin, M Lauria, and A. Chien. High performance messaging on workstations. In *Proceedings of Supercomputing '95*, 1995.

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

[17] Paul Pierce. The Paragon implementation of the NX message passing interface. In *Proceedings of SHPCC 94*, 1994.

[18] Avtar Saini. An overview of the Intel Pentium processor. In *Compcon Spring '93*, pages 60–62, February 1993.

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