Early Experience with Message-Passing on the SHRIMP Multicomputer

Edward W. Felten, Richard D. Alpert, Angelos Bilas, Matthias A. Blumrich, Douglas W. Clark, Stefanos N. Damianakis, Cezary Dubnicki, Liviu Iftode, and Kai Li

Department of Computer Science
Princeton University
Princeton, NJ 08544 USA

Abstract

The SHRIMP multicomputer provides virtual memory-mapped communication (VMMC), which supports protected, user-level message passing, allows user programs to perform their own buffer management, and separates data transfers from control transfers so that a data transfer can be done without the intervention of the receiving node CPU. An important question is whether such a mechanism can indeed deliver all of the available hardware performance to applications which use conventional message-passing libraries.

This paper reports our early experience with message-passing on a small, working SHRIMP multicomputer. We have implemented several user-level communication libraries on top of the VMMC mechanism, including the NX message-passing interface, Sun RPC, stream sockets, and specialized RPC. The first three are fully compatible with existing systems. Our experience shows that the VMMC mechanism supports these message-passing interfaces well. When zero-copy protocols are allowed by the semantics of the interface, VMMC can effectively deliver to applications almost all of the raw hardware's communication performance.

1 Introduction

The trend in building scalable systems has been moving from chip-level integration toward board-level and system-level integration during the last few years. Using entire computer systems as basic building components to construct scalable multicomputers has two main advantages. The first is low cost because it can leverage the volume production of commodity computer systems. The second is performance because commodity systems track the rapid development of hardware and software technologies well. A key question regarding this approach is whether a system built this way can deliver communication performance competitive with or better than custom-designed parallel computers based on low-level integration.

This paper describes how the hardware and software of the SHRIMP multicomputer system interact to provide high-performance communication and reports our early experience with several message-passing implementations on the first operational 4-node SHRIMP system.

The SHRIMP multicomputer is a network of commodity systems. Each node is a Pentium PC running the Linux operating system. The network is a multicomputer routing network [40] connected to the PC nodes via custom-designed network interfaces. The SHRIMP network interface closely cooperates with a thin layer of software to form a communication mechanism called virtual memory-mapped communication. This mechanism supports various message-passing packages and applications effectively, and delivers excellent performance.

Several features of the virtual memory-mapped communication mechanism are important to high-level communication software. First of all, protected, user-level message passing is supported. Secondly, it allows communication buffer management to be customized for individual libraries and applications at user level. Thirdly, it allows user programs to implement zero-copy data transfer protocols. Finally, it can separate data transfers from control transfers so that a data transfer can be done without the intervention of the receiving node CPU; the implementation of control transfer such as the active message mechanism [18] is an option. Our early experience with the SHRIMP multicomputer indicates that the mechanism is easy to use and that these features are very important for applications to achieve low-latency message passing and to obtain communication bandwidth and latency approaching the values provided by the raw hardware.

The SHRIMP multicomputer communication mechanism is a compromise solution between software and hardware designers. Since a high-performance communication mechanism must support various high-level message passing mechanisms well, it requires close cooperation among network interface hardware, operating systems, message passing primitives, and applications. We believe that an early hardware and software "co-design" effort has been crucial to achieving a simple and efficient design.

The following sections describe virtual memory mapped communication and its implementation on the prototype SHRIMP system, followed by implementation descriptions and experimental results of the NX message passing library, the Sun RPC library, the socket communication library, and an RPC library specialized for SHRIMP. We will show that in each case, we can obtain communication performance close to that provided by the hardware. Finally, we will discuss the lessons learned and make concluding remarks.

2 Virtual Memory-Mapped Communication

Virtual memory-mapped communication (VMMC) was developed out of the need for a basic multicomputer communication mechanism with extremely low latency and high bandwidth. This is achieved by allowing applications to transfer data directly between two virtual memory address spaces over the network. The basic mechanism is designed to efficiently support applications and common communication models such as message passing, shared memory, RPC, and client-server.

The VMMC mechanism consists of several calls to support user-level buffer management, various data transfer strategies, and transfer of control.

2.1 Import-Export Mappings

In the VMMC model, an *import-export mapping* must be established before communication begins. A receiving process can *export* a region of its address space as a receive buffer together with a set of permissions to define access rights for the buffer. In order to send data to an exported receive buffer, a user process must *import* the buffer with the right permissions.

After successful imports, a sender can transfer data from its virtual memory into imported receive buffers at user-level without further protection checking or protection domain crossings. Communication under this import-export mapping mechanism is protected in two ways. First, a trusted third party such as the operating system kernel or a trusted process implements import and export operations. Second, the hardware virtual memory management unit (MMU) on an importing node makes sure that transferred data cannot overwrite memory outside a receive buffer.

The *unexport* and *unimport* primitives can be used to destroy existing import-export mappings. Before completing, these calls wait for all currently pending messages using the mapping to be delivered.

2.2 Transfer Strategies

The VMMC model defines two user-level transfer strategies: *deliberate update* and *automatic update*. Deliberate update is an explicit transfer of data from a sender's memory to a receiver's memory.

In order to use automatic update, a sender *binds* a portion of its address space to an imported receive buffer, creating an *automatic update binding* between the local and remote memory. All writes performed to the local memory are automatically performed to the remote memory as well, eliminating the need for an explicit send operation.

An important distinction between these two transfer strategies is that under automatic update, local memory is "bound" to a receive buffer at the time a mapping is created, while under deliberate update the binding does not occur until an explicit send command is issued. Automatic update is optimized for low latency, and deliberate update is designed for flexible import-export mappings and for reducing network traffic.

Automatic update is implemented by having the SHRIMP network interface hardware snoop all writes on the memory bus. If the write is to an address that has an automatic update binding, the hardware builds a packet containing the destination address and the written value, and sends it to the destination node. The hardware can combine writes to consecutive locations into a single packet.

Deliberate update is implemented by having a user-level program execute a sequence of two accesses to addresses which are decoded by the SHRIMP network interface board on the node's expansion bus (the EISA bus). These accesses specify the source address, destination address, and size of a transfer. The ordinary virtual memory protection mechanisms (MMU and page tables) are used to maintain protection [10].

VMMC guarantees the in-order, reliable delivery of all data transfers, provided that the ordinary, blocking version of the deliberate-update send operation is used. The ordering guarantees are a bit more complicated when the non-blocking deliberate-update send operation is used, but we omit a detailed discussion of this point because none of the programs we will describe use this non-blocking operation.

The VMMC model does not include any buffer management since data is transferred directly between user-level address spaces. This gives applications the freedom to utilize as little buffering and copying as needed. The model directly supports zero-copy protocols when both the send and receive buffers are known at the time of a transfer initiation.

The VMMC model assumes that receive buffer addresses are specified by the sender, and received data is transferred directly to memory. Hence, there is no explicit receive operation. CPU involvement in receiving data can be as little as checking a flag, although a hardware notification mechanism is also supported.

2.3 Notifications

The *notification* mechanism is used to transfer control to a receiving process, or to notify the receiving process about external events. It consists of a message transfer followed by an invocation of a user-specified, user-level handler function. The receiving process can associate a separate handler function with each exported buffer, and notifications only take effect when a handler has been specified.

Notifications are similar to UNIX signals in that they can be blocked and unblocked, they can be accepted or discarded (on a per-buffer basis), and a process can be suspended until a particular notification arrives. Unlike signals, however, notifications are queued when blocked. Our current implementation of notifications uses signals, but we expect to reimplement notifications in a way similar to active messages [18], with performance much better than signals in the common case.

3 SHRIMP Prototype

We have built and experimented with the SHRIMP multicomputer prototype in order to demonstrate the performance of hardware-supported virtual memory-mapped communication. Figure 1 shows the structure of our system.

3.1 Hardware Components

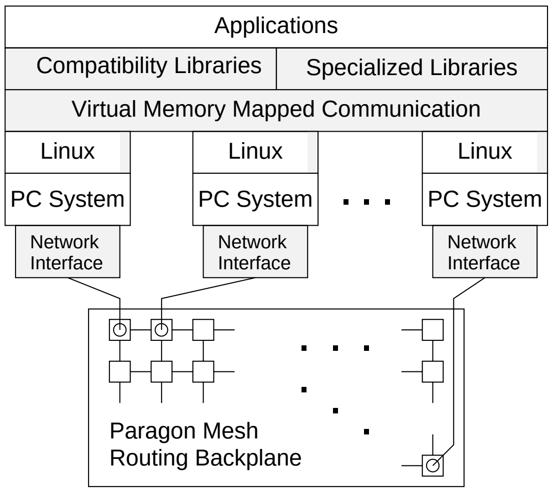


Figure 1: *The prototype SHRIMP system. Shaded components have been implemented by our team.*

The prototype system consists of four interconnected nodes. Each node is a DEC 560ST PC containing an Intel Pentium Xpress motherboard [25]. The motherboard has a 60 Mhz Pentium CPU [36] with an external 256 Kbyte second-level cache, and 40 Mbytes of main DRAM memory. The Xpress memory bus has a maximum burst write bandwidth of 73 Mbytes/sec, and includes a memory expansion connector which carries the majority of the bus signals. Peripherals are connected to the system through the EISA expansion bus [3], which has main memory mastering capability and a maximum burst bandwidth of 33 Mbytes/sec.

Main memory data can be cached by the CPU 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, thereby keeping consistent with *all* main memory updates, including those from EISA bus masters.

The network connecting the nodes is an Intel routing backplane consisting of a two-dimensional mesh of Intel Mesh Routing Chips (iMRCs) [40], and is the same network used in the Paragon multicomputer [24]. The iMRC is essentially a wider, faster version of the Caltech Mesh Routing Chip [14]. The backplane supports deadlock-free, oblivious wormhole routing [15], and preserves the order of messages from each sender to each receiver.

In addition to the fast backplane interconnect, the PC nodes are connected by a commodity Ethernet, which is used for diagnostics, booting, and exchange of low-priority messages.

The custom network interface is the key system component which connects each PC node to the routing backplane and implements the hardware support for VMMC.

3.2 SHRIMP Network Interface Datapath

The SHRIMP network interface hardware (Figure 2) is designed to support fast virtual memory mapped communication by providing the necessary support for eliminating the operating system from the critical path of communication. The hardware consists of two printed circuit boards, since it connects to both the Xpress memory bus (through the memory expansion connector) and the EISA expansion bus. The Xpress card is extremely simple; the EISA card contains almost all of the logic.

The network interface has two principal datapaths: outgoing and incoming. Outgoing data comes either from the memory bus Snoop Logic (automatic update) or from the Deliberate Update Engine, and incoming data comes from the Interconnect. The Network Interface Chip (NIC) is an Intel component which is designed to interface an iMRC router on the backplane to a bi-directional, 64-bit processor bus. Therefore, the Arbiter is needed to share the NIC's processor port between outgoing and incoming transfer, with incoming given absolute priority.

Consider first the outgoing automatic update datapath. After the address and data of an automatic update pass through the multiplexer, the page number portion from the address is used to directly index into the *Outgoing Page Table* (OPT) which maintains bindings to remote destination pages. If an automatic update binding exists, then the Packetizing hardware uses the destination page pointer from the OPT together with the offset portion of the automatic update address to form a packet header in the Outgoing FIFO. The data is appended to this header to form a packet.

If the indexed page has been configured for combining in the OPT entry, then the packet is not immediately sent. Rather, it is buffered in the Outgoing FIFO and if the next automatic update address is consecutive, the new data is simply appended to the existing packet. If not, a new packet is started. Pages can also be configured to use a hardware timer: a timeout causes a packet to be automatically sent if no subsequent automatic update has occurred.

The outgoing deliberate update datapath is identical to the automatic update datapath, except for the source of the address and data. The Deliberate Update Engine interprets the two-access transfer initiation sequence described in Section 2.2, and performs DMA through the EISA bus to read the source data from main memory. The data is passed through the multiplexer along with an address to select the destination in the OPT, derived from the transfer initiation sequence.

The incoming datapath is driven by the Incoming DMA Engine which transfers packet data from the NIC to main memory over the EISA bus, using the destination base address contained in the packet header. Before performing the transfer, however, the destination address is used to index directly into the *Incoming Page Table* (IPT) to determine whether the specified page is enabled to receive data. The IPT has an entry for every page of memory, and each entry contains a flag which specifies whether the network interface can transfer data to the corresponding page or not. If data is received for a page which is not enabled, then the network interface will freeze the receive datapath and generate an interrupt to the node CPU.

VMMC notifications (Section 2) are supported through a sender-specified interrupt flag in the packet header, and a receiver-specified interrupt flag in each IPT entry. An interrupt is generated to the destination CPU *after* a received packet has been transferred to memory if both the sender-specified *and* receiver-specified flags have been set. The sender-specified flag is set when either an automatic update OPT entry is configured for destination interrupt, or a deliberate update is initiated with a flag requesting a

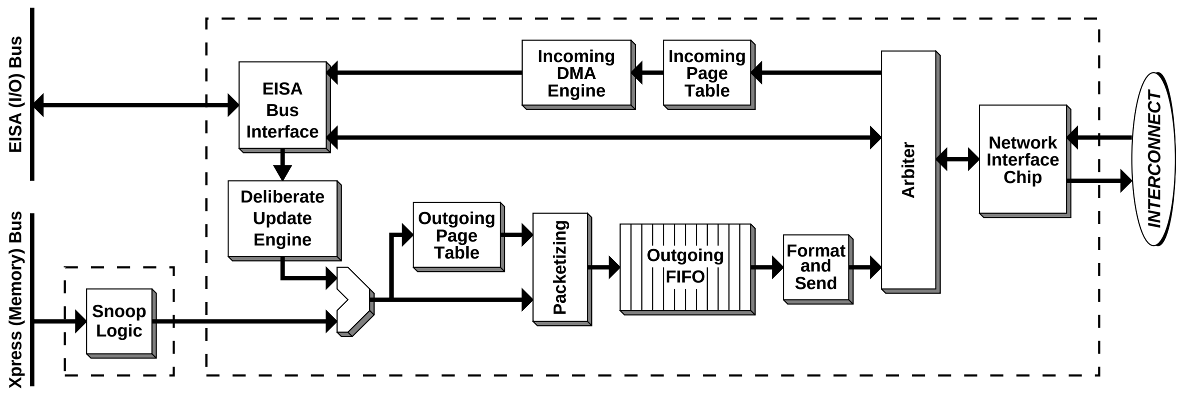


Figure 2: *Datapath of the SHRIMP network interface*

destination interrupt.

3.3 Software Components

Figure 1 shows the existing software components on our prototype system.

There are two types of communication library: the compatibility libraries and the specialized libraries. The compatibility libraries currently consist of NX/2 message passing, Sun RPC, and stream sockets, all fully compatible with existing systems. Existing applications using the standard interfaces can run with our compatibility libraries without any modification. There is currently one specialized library, which implements remote procedure call, taking full advantage of the SHRIMP VMMC mechanism. Both the compatibility libraries and the specialized library run entirely at user level and are built on top of the VMMC layer.

The VMMC component is a thin layer library that implements the VMMC API, provides direct access to the network for data transfers between user processes, and handles communication with the SHRIMP daemon. SHRIMP daemons are trusted servers (one per node) which cooperate to establish (and destroy) import-export mappings between user processes. The daemons use memory-mapped I/O to directly manipulate the network interface hardware. They also call SHRIMP-specific operating system calls to manage receive buffer memory and to influence node physical memory management in order to ensure consistent virtual memory communication mappings across nodes.

3.4 Peak Performance of SHRIMP

To measure user-to-user bandwidth delivered by the SHRIMP basic library, we performed experiments using four different transfer strategies. In each of them, we had two processes on two different nodes repeatedly "ping-pong" a series of equally-sized messages back and forth, and measured the roundtrip latency and bandwidth. Figure 3 shows the resulting one-way latency and bandwidth for message sizes between 4 bytes and 10 Kbytes, using both automatic update (AU) and deliberate update (DU).

The AU-1copy case involves a message copy on the sender, but none on the receiver, whereas the AU-2copy case involves copying on both sides. The DU-0copy case avoids copying altogether and transfers the message directly from sender to receiver, while the DU-1copy case copies at the receiver. Since automatic update transfer always requires one memory copy by the sender (instead of an explicit send operation), the total number of copies in the automatic update cases is greater by one than in the corresponding deliberate update cases.

Because the performance of small message transfers is dominated by latency, the left-hand graph in Figure 3 shows our latency results for small messages only. The automatic update latency for a one-word, user-to-user data transfer is 4.75 μsec with both sender's and receiver's memory cached write-through, and 3.7 μsec with caching disabled (not shown in the graph). The deliberate update latency for a one-word, user-to-user data transfer is 7.6 μsec.

The right-hand graph in Figure 3 shows user-to-user bandwidth as a function of transfer size for all four cases. For smaller messages, automatic update outperformed deliberate update because automatic update has a low start-up cost. For larger messages, however, deliberate update delivered bandwidth slightly higher than automatic update because automatic update is limited by the bandwidth of the "extra" memory copy operation. In the DU-0copy case, the maximum bandwidth was almost 23 MB/sec, limited only by the aggregate DMA bandwidth of the shared EISA and Xpress buses.

4 User-Level Compatibility Libraries

The key software challenge in building SHRIMP was whether we could deliver the raw performance capabilities of the hardware to application programs. Our goal was to build compatibility libraries that support standard communication interfaces, without compromising performance.

We built user-level compatibility libraries to support three common communication interfaces: the Intel NX multicomputer message-passing system [35], Sun RPC [26], and stream sockets [29]. All three of these libraries are *fully compatible* with existing systems, so that existing application code can be used without modification. Full compatibility means that the entire interface, including all of the inconvenient corner cases, has been implemented and tested. Full discussions of the three libraries are found elsewhere [1, 6, 16]; here we give only brief descriptions of how the libraries operate in the common cases.

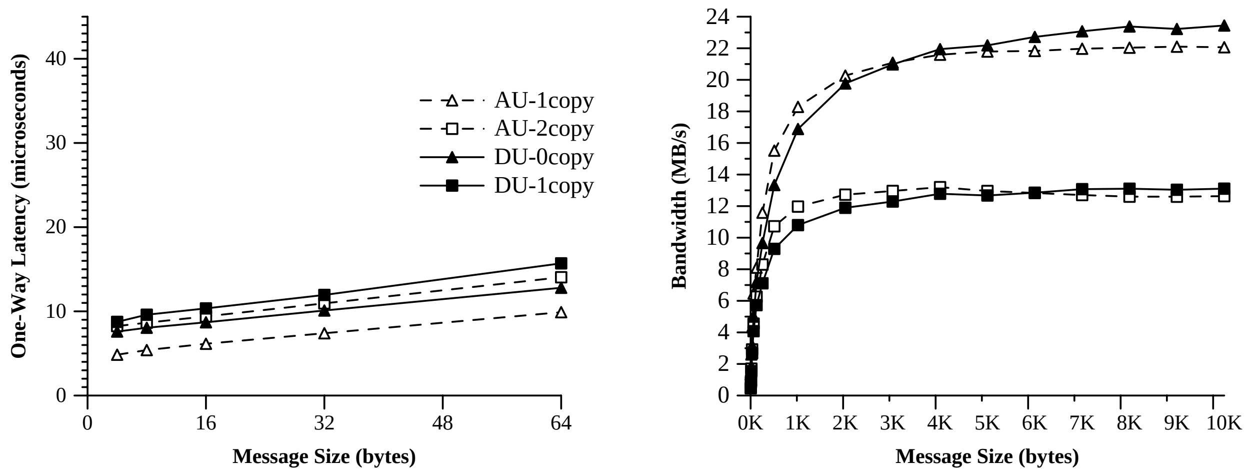


Figure 3: *Latency and bandwidth delivered by the SHRIMP VMMC layer.*

Although the libraries must meet different requirements, their designs do have some common elements. We now describe these common elements, before discussing the particular features of each library.

Connections All three libraries use some kind of point-to-point connection as a basic building block. A connection between two processes consists of a set of buffers, each exported by one process and imported by the other; there is also a fixed protocol for using the buffers to transfer data and synchronize between the two processes. In NX, a connection is set up between each pair of processes at initialization time; in RPC, a connection is established at binding time; connections are already part of the sockets model. The details of how buffers are mapped to create a connection differ, because the libraries have differing requirements.

Synchronization and Buffer Management A connection generally consists of some buffers for transmitting the contents of user messages, and some buffers for transmitting control information. To pass a message, the sender first chooses a remote data buffer into which to place the message contents. After transferring the message contents, the sender transmits some control information to tell the receiver that the data is in place, and possibly to provide descriptive information such as the message size. Since SHRIMP guarantees in-order delivery of packets, the control information arrives at the receiver after the data.

To receive a message, a process looks for incoming control information that denotes the arrival of a suitable message. Once this is found, the receiver can consume the message data. The receiver then sends some control information back to the sender, to signal that data buffer space can be re-used. As above, the exact details differ depending on the requirements of each library.

Reducing Copying In each library, a naive protocol would copy data twice: from user memory to a staging buffer on the sending side, and from the connection's data buffer to user memory on the receiving side.

The send-side copy can generally be removed, by transmitting data directly from the user's memory. Since the source address for deliberate update can be anywhere in memory, this is a simple improvement to make. The result is a one-copy protocol.

One complexity does arise in one-copy protocols. The SHRIMP hardware requires that the source and destination addresses for deliberate updates be word-aligned. If the user's source buffer is not word-aligned, then deliberate update will send some data that is not part of the message. Solving this problem adds complexity to the library, and hence makes it slightly slower; but this is worthwhile to avoid a copy, except for very small messages.

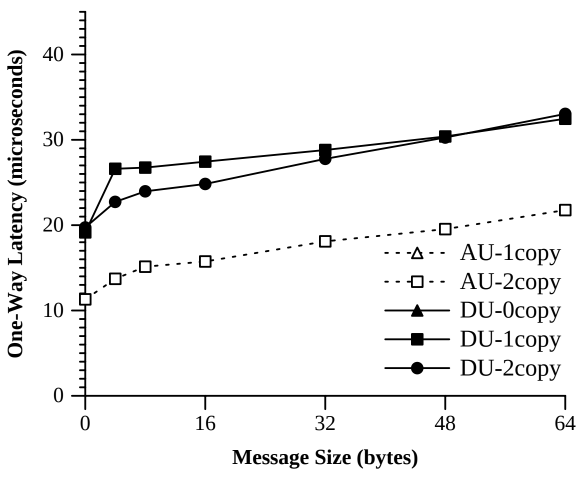
It is also possible in some cases to use a zero-copy protocol. This requires transferring data directly from the sender's user buffer to the receiver's user buffer; the sender-side library must somehow learn which address the user passed to the receive call, so an extra control packet must usually be sent. SHRIMP does not allow zero-copy protocols when the sender's and receiver's buffers are not word-aligned.

Deliberate vs. Automatic Update Each library must choose when to use the deliberate update strategy and when to use automatic update. Automatic update is clearly faster for small transfers, so all three libraries use automatic update to transfer control information.

For transferring message data, the choice is less clear-cut. Deliberate update has slightly higher asymptotic bandwidth, but automatic update is convenient to use and does not restrict buffer alignment.

When we use automatic update, the sender transmits the message data to the receiver by simply copying the data from the user memory to the send-side communication buffer. We count this as a data copy. Thus, every automatic update protocol does at least one copy. The benefit of this "extra" copy is that no explicit send operation is required.

Experiments Below, we give the results of performance experiments with each library. In all cases, we ran a program that does a large number of round-trip "ping-pong" communications between two processes. The message *latency* is half of time required for a round-trip communication; the *bandwidth* is the total number of the user's bytes sent, divided by the total running time. For each library, we show two



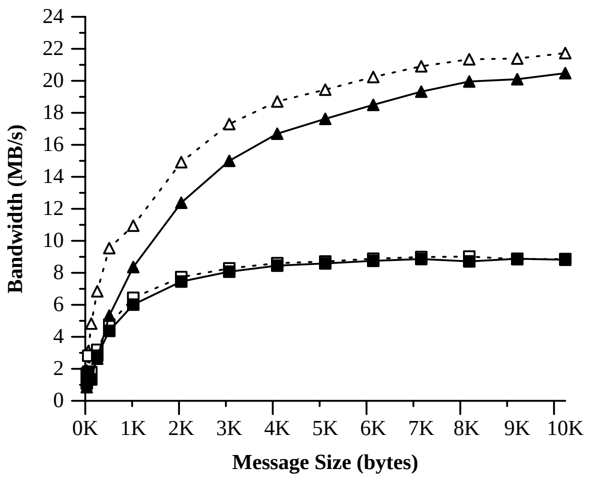


Figure 4: *NX latency and bandwidth*

graphs. One graph shows the latency for small message sizes, and the other shows bandwidth for large message sizes.

Each graph shows several different variants of a library. The variants are classified according to whether they use automatic update or deliberate update, and according to whether they copy the message contents zero, one, or two times per one-way transfer. We omit the combinations that don't make sense for a particular library.

4.1 NX Message Passing

Our compatibility library for the Intel NX multicomputer message-passing interface uses different protocols, depending on message size. Small messages use a simple one-copy protocol, while large messages use a more complex zero-copy protocol.

One-Copy Protocol At initialization time, a connection is set up between each pair of processes. In addition to buffers for passing control information, each connection contains a buffer for passing data in each direction. These data buffers are divided into fixed-size packet buffers.

To send a message using the one-copy protocol, the sender writes the data and a small descriptor into a packet buffer on the receiver.

The receiver, when a receive is called, examines the *size* field of the descriptor to determine whether a message has arrived. After the receiver consumes the message, it resets the size field to a special value and uses the control buffer to return a send credit to the sender. Since the receiver may consume messages out of order, the credit identifies a specific packet buffer which has become available.

Small messages require at least one copy from the receive buffer into the user's memory. The sender may choose to send the data along with the header directly via automatic update as it marshals the header and data in its send buffer, saving the cost of an additional deliberate update. Alternatively, the data can be sent by deliberate update directly from user space into the receiver's buffer, saving the cost of a local copy.

Zero-Copy Protocol Large messages use a zero-copy protocol. The sender sends a "scout" packet, a special message descriptor, to the receiver (using the one-copy protocol); then the sender immediately begins copying the data into a local (sender-side) buffer[1]. The receive call, upon finding the scout message, sends back a reply, giving the sender the buffer ID of the region of address space into which the sender is to place the data. If it hasn't done so already, the sender imports that buffer. Data are transferred, and the sender sets a flag in the control buffer to notify the receiver of the data's arrival.

If the sender has not finished copying the data into its local buffer by the time the receiver replies, the sender transmits the data from the sender's user memory directly into the receiver's user memory. If the sender finishes copying the message before the receiver's reply arrives, the sending program can continue, since a safe version of the message data is available on the sender side.

Performance Figure 4 shows the performance of our NX library as a function of message size. The left-hand graph illustrates the tradeoff between a local copy and an extra send. Values on the lower deliberate-update curve, labeled "2copy," are observed when the sender copies data into the header marshaling area, then sends header and data with a single deliberate update. Values on the higher curve, labeled "1copy," result when a sender dispatches data and the message header with two separate deliberate updates. As the size of the message increases, the cost of copying begins to exceed the cost of the extra send.

For small messages with automatic update, we incur a latency cost of just over 6 μs above the hardware limit. This time is spent in buffer management, including cost to the receiver of the return of a credit to the sender. For large messages, performance asymptotically approaches the raw hardware limit, as the small message round-trip communication cost incurred by the protocol is amortized

[1]Despite this data copying, we consider this a zero-copy protocol, because the copy is not on the critical path. The sender copies only when it has nothing better to do; as soon as the receiver replies, the sender immediately stops copying. The only purpose of the copying is to make a safe copy of the message data, so the sender-side application program can be resumed.

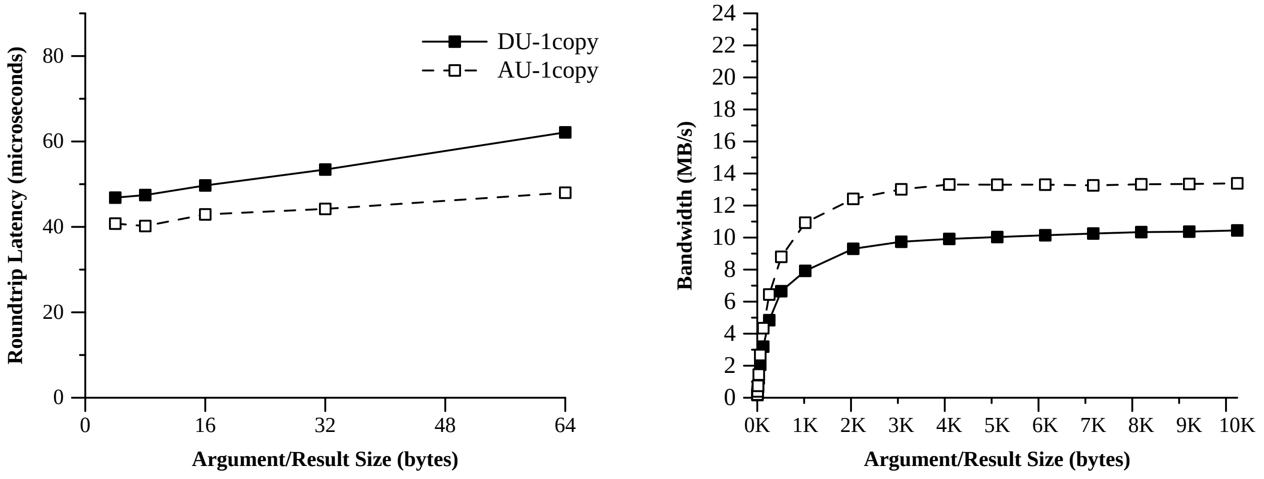


Figure 5: *VRPC latency and bandwidth*

over more data. A small "bump" is seen where the protocol changes. For in-depth analysis see [1].

4.2 Remote Procedure Call

VRPC is a fast implementation of remote procedure call (RPC) for SHRIMP, fully compatible with the SunRPC standard. In implementing VRPC we changed only the SunRPC runtime library; the stub generator and the operating system kernel are unchanged. SunRPC is implemented in layers, as shown on the left side of Figure 6. RPCLIB is the set of calls visible to the user, XDR implements architecture-independent data representation, and the stream layer decouples XDR from the expensive network layer.

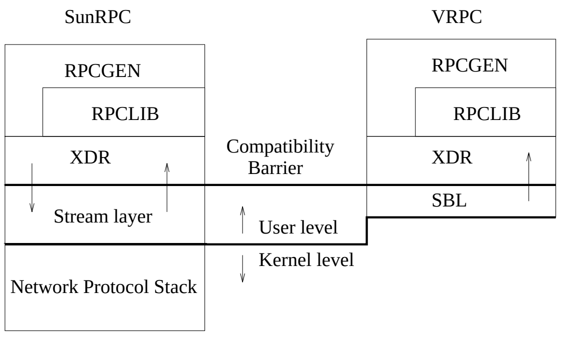


Figure 6: *The structure of two SunRPC systems: on the left, the existing implementation; on the right, our implementation. The arrows indicate the direction and number of copies at user level. The existing implementation also does copying in the kernel.*

We used two main techniques to speed up the library. First, we reimplemented the network layer using virtual memory mapped communication. Because VMMC is simple and allows direct user-to-user communication, our implementation of the network layer is much faster than the standard one. Our second optimization was to fold the simplified stream layer directly into the XDR layer, thus reducing the number of layers by one. The resulting system is shown on the right side of Figure 6.

Data Structures The communication between the client and the server takes place over a pair of mappings which implement a bidirectional stream between the sender and the receiver. Essentially, we implement a cyclic shared queue in each direction. The control information in each buffer consists of 2 reserved words. The first word is a flag and the second the total length (in bytes) of the data that has been written into the buffer from the last and previous transfers. The sender (respectively, receiver) remembers the next position to write (read) data to (from) the buffer. The XDR layer sends the data directly to the receiver, so there is no copying on the sending side.

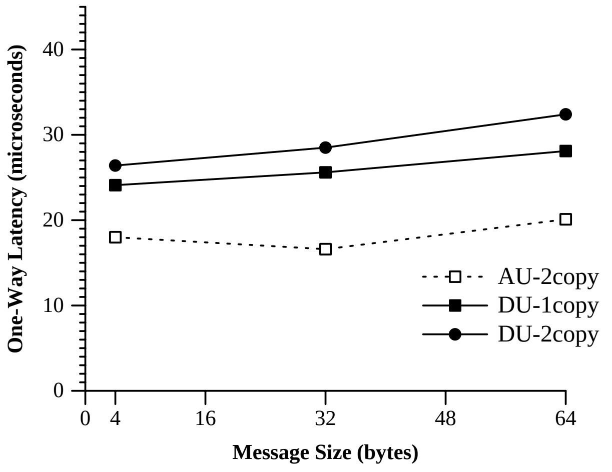
Performance Figure 5 shows the performance of a null VRPC measured by varying the size of a single argument and a single result, starting with a 4-byte argument and a 4-byte result. Our experiments show that even without changing the stub generator or the kernel, RPC can be made several times faster than it is on conventional networks. The resulting system has a round-trip time of about 29 μsec for a null RPC with no arguments and results. Of these about 21-22 μsecs are spent in transferring the headers and waiting for the server. About 7 μsecs are spent in preparing the header and making the call, 1-2 μsecs in returning from the call and the remaining 5-6 μsecs in processing the header. More details are found in [6].

Further optimizations It is also possible in principle to eliminate the receiver-side copy. This violates the initial constraint that the stub generator remain unchanged. However, in the current implementation of SunRPC, only slight modifications are required to the stub generator; alternatively, the user could modify the stub code by hand.

A consequence of avoiding all copies is that the server must consume the data before the client can send more data. This is not a problem since the server must finish processing the current call before going to the next in any case.

4.3 Stream Sockets

The SHRIMP socket API is implemented as a user-level library, using the VMMC interface. It is fully compatible



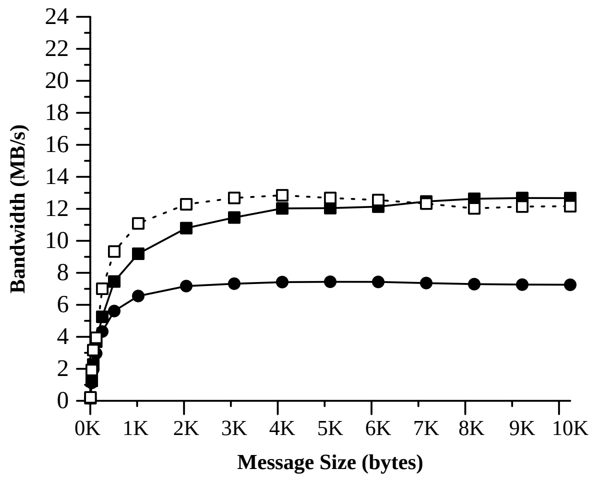


Figure 7: *Socket latency and bandwidth*

with the Unix stream sockets facility [29].

Connection Establishment During connection establishment, the implementation use a regular internet-domain socket, on the Ethernet, to exchange the data required to establish two VMMC mappings (one in each direction). The internet socket is held open, and is used to detect when the connection has been broken.

Data Structures Internally, the sockets library uses a straightforward implementation of circular buffers in order to manage incoming and outgoing data. For each socket descriptor, two structures are maintained that group data by who will have write access: incoming data (written by the remote process) and outgoing data (written by the local process). The incoming and outgoing structures are used to build the send and receive circular buffers.

We implemented three variations of the socket library, two using deliberate update and one using automatic update. The first protocol performs two copies, one on the receiver to move the data into the user memory and the other on the sender to eliminate the need to deal with data alignment. We can improve the performance by eliminating the send-side copy, leading to a one-copy protocol, although we must still use the two-copy protocol when dictated by alignment. The automatic-update protocol always does two copies, since the sender-side copy acts as the send operation.

It is not possible to build a zero-copy deliberate-update protocol or a one-copy automatic-update protocol without violating the protection requirements of the sockets model. Such a protocol would require a page of the receiver's user memory to be exported; the sender could clobber this page at will. This is not acceptable in the sockets model, since the receiver does not necessarily trust the sender.

Performance Figure 7 shows the performance of our sockets library. For large messages, performance is very close to the raw hardware one-copy limit. For small messages, we incur a latency of 13 μs above the hardware limit. This extra time is divided roughly equally between the sender and receiver performing procedure calls, checking for errors, and accessing the socket data structure.

We also measured the performance of our implementation using `ttcp` version 1.12, a public domain benchmark originally written at the Army Ballistics Research Lab. `ttcp` measures network performance using a one-way communication test where the sender continuously pumps data to the receiver. `ttcp` obtained a peak bandwidth of 8.6 MB/s using 7 kbyte messages. Our own one-way transfer microbenchmark obtained a bandwidth of 9.8 MB/s for 7 kbyte messages. Finally, `ttcp` measured a bandwidth of 1.3 MB/s, which is higher than Ethernet's peak bandwidth, at a message size of 70 bytes. For more analysis see [16].

5 Specialized Libraries

While our compatibility libraries have very good performance, their implementation was limited by the need to remain compatible with the existing standards. To explore the further performance gains that are possible, we implemented a non-compatible version of remote procedure call.

SHRIMP RPC is not compatible with any existing RPC system, but it is a real RPC system, with a stub generator that reads an interface definition file and generates code to marshal and unmarshal complex data types. The stub generator and runtime library were designed with SHRIMP in mind, so we believe they come close to the best possible RPC performance on the SHRIMP hardware.

Buffer Management The design of SHRIMP RPC is similar to Bershad's URPC [5]. Each RPC binding consists of one receive buffer on each side (client and server) with bidirectional import-export mappings between them. When a call occurs, the client-side stub marshals the arguments into its buffer, and then transmits them into the server's buffer. At the end of the arguments is a flag which tells the server that the arguments are in place. The buffers are laid out so that the flag is immediately after the data, and so that the flag is in the same place for all calls that use the same binding. A single data transfer can transmit both the data and the flag.

When the server sees the flag, it calls the procedure that the sender requested. At this point the arguments are still in the server's buffer. When the call is done, the server sends

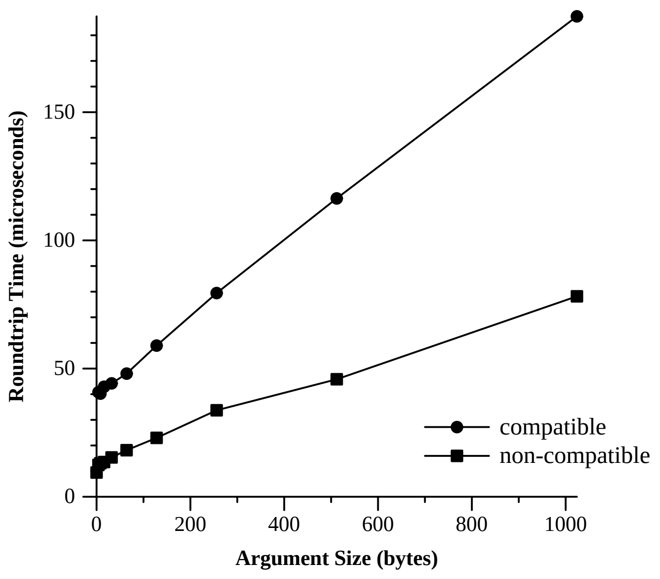


Figure 8: *Round-trip time for null RPC, with a single INOUT argument of varying size.*

return values and a flag back to the sender. (Again, the flag is immediately after the data, so that only one data transfer is required.)

Exploiting Automatic Update The structure of our SHRIMP RPC works particularly well with automatic update. In this case, the client's buffer and the server's buffer are connected by a bidirectional automatic update binding; whenever one process writes its buffer, the written data is propagated automatically to the other process's buffer. The data layout and the structure of the client stub cause the client to fill memory locations consecutively while marshaling the arguments, so that all of the arguments and the flag can be combined into a single packet by the client-side hardware.

On the server side, return values (`OUT` and `INOUT` parameters) need no explicit marshaling. These variables are passed to the server-side procedure by reference: that is, by passing a pointer into the server's communication buffer. The result is that when the procedure writes any of its `OUT` or `INOUT` parameters, the written values are silently propagated back to the client by the automatic update mechanism. This communication is overlapped with the server's computation, so in many cases it appears to have no cost at all. When the server finishes, it simply writes a flag to tell the client that it is done.

Performance Figure 8 compares the performance for a null call of two versions of RPC: the SunRPC-compatible VRPC, and the non-compatible SHRIMP RPC. We show the fastest version of each library, which is one-copy automatic update in both cases.

The difference in round-trip time is more than a factor of three for small argument sizes: 9.5 μsec for the non-compatible system, and 29 μsec for the SunRPC-compatible system. The difference arises because the SunRPC standard requires a nontrivial header to be sent for every RPC, while the non-compatible system sends just the data plus a one-word flag, all of which can be combined by the hardware into a single packet. For large transfers, the difference is roughly a factor of two. This occurs because the non-compatible system does not need to explicitly send the OUT arguments from the server back to the client; the OUT arguments are implicitly sent, in the background, via automatic update as the server writes them.

6 Discussion

We learned many interesting things in the process of building, testing, and benchmarking our user-level communication libraries on SHRIMP. We now discuss some of these lessons.

User-level Buffer Management Leaving buffer management to the user-level libraries was the key to achieving good performance in all of the libraries. Superficially, the libraries use the same buffer management technique: transferring message contents into a large buffer region in linear fashion, and then sending control information to a separate area. If we look at the details, however, we see that there are crucial differences among the buffer management strategies. These differences are driven by the differing requirements of the libraries, and they lead to differences in performance.

For example, the sockets and VRPC interfaces require that the receiver consume messages in the order they were sent, while the NX interface does not. As a result, the sockets and VRPC interfaces use a circular buffer for each sender/receiver pair, while NX divides the buffer into fixed-size pieces that can be reused in any order.

There are several other examples in which different interface semantics led to different implementations:

- NX has a simple model of connection startup and shutdown, but these are more complicated in the sockets and RPC models. Thus, sockets and RPC must expend more effort on getting startup and shutdown behavior correct.
- NX divides data into messages; sockets uses a byte stream. Therefore, NX must provide separate header information for each user message transferred, while sockets can avoid using per-message headers.
- NX allows communication between a fixed set of processes only, while sockets and RPC allow communication between any pair of processes. At initialization time, NX sets up one set of buffers for each pair of processes; sockets and RPC set up separate buffers for each connection or binding when it is established.

If we had attempted to provide a single kernel-level (or architecture-level) mechanism to handle the buffer management requirements of all of the libraries, the result would have been very complicated and hence slow.

Separating Data Transfer from Control Transfer The SHRIMP architecture allows the user-level library to choose not only *how* buffer management is handled, but also *when* the buffer management code is executed. In other words, it separates data transfer from control transfer.

This is an important feature, since data transfer is much more common than control transfer in our libraries.

Transmitting a user message requires several data transfers (two for sockets and NX, and four for VRPC) but at most one control transfer. In fact, it is common in NX and sockets for a sender to send a burst of user messages, which the receiver processes all at once at the end of the burst. When this happens, there is less than one control transfer per message.

Interrupts Another key to high performance is reducing the number of interrupts. Typically, our libraries can avoid interrupts altogether. Since a sender can transmit several messages without any action from the receiver, the sender usually requires no action from the receiver until the receiver is ready to consume a message, so there is no reason to force the receiver to execute communication-library code before it is ready.

One exception to this rule occurs when a sender wants to transmit data but finds all of the buffers leading to the receiver are full. When this happens, the NX library generates an interrupt on the receiver to request more buffers. This is not necessary in the sockets library because sockets semantics do not guarantee extensive buffering. It is not necessary in RPC because it serves no purpose: on the call, the sender (client) is about to wait for the receiver (server) anyway, and on the return, the receiver (client) is known to be in the RPC library already.

Polling vs. Blocking All of our libraries are designed to prefer polling over blocking when they have to wait for data to arrive. We believe that polling is the right choice in the common case, although there are situations in which blocking is appropriate.

Our libraries are all written to switch between polling and blocking as appropriate. A notification (software interrupt) is sent along with each update to the control variables for each connection. When the process wants to start polling, it asks that notifications be ignored; the kernel then changes per-page hardware status bits so that the interrupts do not occur. When the process wants to block, it re-enables the notifications before blocking.

Benefits of Hardware/Software Co-design Our strategy in designing SHRIMP was to design the hardware and software components in tandem, to match hardware services to software requirements as closely as possible. Several features of the current design are consequences of this co-design.

For example, early versions of the SHRIMP hardware required that a single send-buffer be bound to each receive-buffer; the hardware could transfer data only between buffers that were bound together. Preliminary software designs found this restriction too limiting, so the architecture was changed to allow data transfer from arbitrary send buffers.

Another effect of co-design was the removal of a multicast feature from the hardware. Originally, we intended to have the hardware forward special multicast packets from node to node, to support one-to-many communication in hardware. This would have added significant complexity to the hardware, because of the danger of deadlock. The software designers found that the multicast feature was not as useful as we originally thought, and that software implementations of multicast would likely have acceptable performance. As a result, the multicast feature was removed from the hardware.

On the other hand, we failed to realize until too late how inconvenient the deliberate update word-alignment restriction would be for software. Removing this restriction would have complicated the hardware, and we are not yet sure whether this cost would be justified. Furthermore, automatic update, which has no such restriction, performs about as well as, and usually better than, deliberate update in our prototype.

7 Related Work

There is a large body of literature on interprocessor communication mechanisms. Most related work is in using remote memory reference models and implementing various message-passing libraries.

Spector originated a remote memory reference model [38]; it has recently been revived by several later research efforts as a model to program message-passing primitives for high performance [39, 18, 41]. All of these implementations require explicit CPU action or new instructions to initiate communication and require CPU intervention to receive messages.

The idea of automatic-update data delivery in our virtual memory mapped communication is derived from the Pipelined RAM network interface [31], which allows physical memory mapping only for a small amount of special memory. Several shared memory architecture projects use the page-based, automatic-update approach to support shared memory, including Memnet[17], Merlin [32] and its successor SESAME [43], the Plus system [9], and GalacticaNet [27]. These projects did not study the implementation of message-passing libraries.

Wilkes's "sender-based" communication in the Hamlyn system [42] supports user-level message passing, but requires application programs to build packet headers. They have not tried to implement message-passing libraries using the underlying communication mechanism.

Several projects have tried to lower overhead by bringing the network all the way into the processor and mapping the network interface FIFOs to special processor registers [11, 21, 13]. 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 [30, 19]. 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 the application must build packet headers, message passing overhead is still hundreds of CPU instructions.

Old multicomputers have traditional network interfaces and thus their implementations of the NX message-passing library manage communication buffers in the kernel [35, 34]. Current machines like the Intel Paragon and Meiko CS-2 attack software overhead by adding a separate processor on every node just for message passing [33, 24, 23, 22, 20]. This approach, however, does not eliminate the overhead of the software protocol on the message processor, which is still tens of microseconds in software overhead.

Distributed systems offer a wider range of communication abstractions, including remote procedure call [8, 37, 4], ordered multicast [7], and object-oriented models [2, 12, 28]. As above, the performance of these systems is limited by the hardware architecture.

Active Messages is one well-known system that attempts to reduce communication overhead on multicomputers [18]. Overhead is reduced to a few hundred instructions on stock hardware, but an interrupt is still required in most cases.

8 Conclusion

This paper reports our early experience with message passing on our prototype SHRIMP multicomputer system. The main positive experience is that the virtual memory-mapped communication (VMMC) model supported in the system works quite well.

- We could quickly implement NX, Sun RPC and stream sockets using the VMMC mechanism. One of the reasons is that VMMC does not impose restrictive buffering or synchronization semantics and another is that it is easy to program and debug message-passing libraries at user level.
- User-level buffer management allows us to conveniently implement a zero-copy data transfer protocol for NX's large messages, and a one-copy data transfer protocol for Sun RPC and stream sockets. Traditional systems often require two-copy protocols.
- Separating data transfer from control transfer allows us to transfer data without interrupting the receiver's CPU. This is one key to high performance.
- VMMC allows users to customize their specialized library to obtain performance very close to the hardware bandwidth limit. We have demonstrated this with a specialized RPC implementation. The software overhead of this implementation is under 1 μsec.

Good performance comes from close cooperation among network interface hardware, operating systems, message passing primitives, and applications. We feel that an early hardware and software "co-design" effort has been crucial in achieving a simple and efficient design.

The experiments reported in this paper are still limited. We have reported only the results of micro-benchmark experiments. We plan to study the performance of real applications in the near future. We also plan to expand the system to 16 nodes.

Finally, the performance numbers reported in this paper reflect the state of the system early in its lifetime, without a great deal of optimization of the hardware and software. As we tune the system and gain experience, we expect further performance improvements.

9 Acknowledgements

This project is sponsored in part by ARPA under contract N00014-95-1-1144, by NSF under grant MIP-9420653, by Digital Equipment Corporation and by Intel Corporation. Felten is supported by an NSF National Young Investigator Award. Alpert is supported by an ARPA Fellowship in High Performance Computing.

References

[1] Richard Alpert, Cezary Dubnicki, Edward W. Felten, and Kai Li. Design and Implementation of NX Message Passing Using SHRIMP Virtual Memory Mapped Communication. Technical Report TR-507-96, Dept. of Computer Science, Princeton University, January 1996.

[2] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. A Distributed Implementation of the Shared Data-Object Model. In *USENIX Workshop on Experiences with Building Distributed and Multiprocessor Systems*, pages 1–19, October 1989.

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

[4] B.N. Bershad, T.E. Anderson, E.D. Lazowska, and H.M. Levy. Lightweight Remote Procedure Call. *ACM Transactions on Computer Systems*, 8(1):37–55, May 1990.

[5] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. User-Level Interprocess Communication for Shared Memory Multiprocessors. *ACM Trans. Comput. Sys.*, 9(2):175–198, May 1991.

[6] Angelos Bilas and Edward W. Felten. Fast RPC on the SHRIMP Virtual Memory Mapped Network Interface. Technical Report TR-512-96, Dept. of Computer Science, Princeton University, February 1996.

[7] K.S. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[8] A.D. Birrell and B.J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[9] R. Bisiani and M. Ravishankar. PLUS: A Distributed Shared-Memory System. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 115–124, May 1990.

[10] 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*, pages 154–165, February 1996.

[11] S. Borkar, R. Cohn, G. Cox, T. Gross, H.T. Kung, M. Lam, M. Levine, B. Moore, W. Moore, C. Peterson, J. Susman, J. Sutton, J. Urbanski, and J. Webb. Supporting Systolic and Memory Communication in iWarp. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, June 1990.

[12] J.S. Chase, F.G. Amador, E.D. Lazowska, H.M. Levy, and R.J. Littlefield. The Amber System: Parallel Programming on a Network of Multiprocessors. In *Proceedings of the 12th Symposium on Operating Systems Principles*, pages 147–158, December 1989.

[13] William J. Dally. The J-Machine System. In P.H. Winston and S.A. Shellard, editors, *Artificial Intelligence at MIT: Expanding Frontiers*, pages 550–580. MIT Press, 1990.

[14] William J. Dally and Charles L. Seitz. The Torus Routing Chip. *Distributed Computing*, 1:187–196, 1986.

[15] 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.

[16] Stefanos Damianakis, Cezary Dubnicki, and Edward W. Felten. Stream Sockets on SHRIMP. Technical Report TR-513-96, Dept. of Computer Science, Princeton University, February 1996.

[17] 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.

[18] T. Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, pages 256–266, May 1992.

[19] FORE Systems. *TCA-100 TURBOchannel ATM Computer Interface, User's Manual*, 1992.

[20] John Heinlein, Kourosh Gharachorloo, Scott A. Dresser, and Anoop Gupta. Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor. In *Proceedings of 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 38–50, October 1994.

[21] Dana S. Henry and Christopher F. Joerg. A Tightly-Coupled Processor-Network Interface. In *Proceedings of 5th International Conference on Architectur al Support for Programming Languages and Operating Systems*, pages 111–122, October 1992.

[22] Mark Homewood and Moray McLaren. Meiko CS-2 Interconnect Elan – Elite Design. In *Proceedings of Hot Interconnects '93 Symposium*, August 1993.

[23] Jiun-Ming Hsu and Prithviraj Banerjee. A Message Passing Coprocessor for Distributed Memory Multicomputers. In *Proceedings of Supercompu ting '90*, pages 720–729, November 1990.

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

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

[26] Internet Network Working Group. *RPC: Remote Procedure Call Protocol Specification Version 2*, June 1988. Internet Request For Comments RFC 1057.

[27] 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.

[28] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[29] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD Unix Operating System*. Addison Wesley, 1989.

[30] C.E. Leiserson, Z.S. Abuhamdeh, D.C. Douglas, C.R. Feynman, M.N. Ganmukhi, J.V. Hill, D. Hillis, B.C. Kuszmaul, M.A. St. Pierre, D.S. Wells, M.C. Wong, S. Yang, and R. Zak. The Network Architecture of the Connection Machine CM-5. In *Proceedings of 4th ACM Symposium on Parallel Algorithms and Architectures*, pages 272–285, June 1992.

[31] Richard Lipton and Jonathan Sandberg. PRAM: A Scalable Shared Memory. Technical Report CS-TR-180-88, Princeton University, September 1988.

[32] Creve Maples. A High-Performance, Memory-Based Interconnection System For Multicomputer Environments. In *Proceedings of Supercompu ting '90*, pages 295–304, November 1990.

[33] 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.

[34] 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.

[35] Paul Pierce. The NX/2 Operating System. In *Proceedings of Third Conference on Hypercube Concurrent Computers and Applications*, pages 384–390, January 1988.

[36] Avtar Saini. An Overview of the Intel Pentium Processor. In *Proceedings of the IEEE COMPCON'93 Conference*, pages 60–62, February 1993.

[37] Michael D. Schroeder and Mike Burrows. Performance of Firefly RPC. *ACM Transactions on Computer Systems*, 8(1):1–17, 1990.

[38] Alfred Z. Spector. Performing Remote Operations Efficiently on a Local Computer Network. *Communications of the ACM*, 25(4):260–273, April 1982.

[39] Chandramohan A. Thekkath, Henry M. Levy, and Edward D. Lazowska. Efficient Support for multicomputing on ATM Networks. Technical Report 93-04-03, Department of Computer Science and Engineering, University of Washington, April 1993.

[40] Roger Traylor and Dave Dunning. Routing Chip Set for Intel Paragon Parallel Supercomputer. In *Proceedings of Hot Chips '92 Symposium*, August 1992.

[41] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of 15th ACM Symposium on Operating Systems Principles*, 1995.

[42] John Wilkes. Hamlyn – An Interface for Sender-Based Communications. Technical Report HPL-OSR-92-13, Hewlett-Packard Laboratories, November 1993.

[43] 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.