Software Support for Virtual Memory-Mapped Communication

Cezary Dubnicki, Liviu Iftode, Edward W. Felten, Kai Li
Department of Computer Science
Princeton University

Abstract

*Virtual memory-mapped communication (*VMMC*) is a communication model providing direct data transfer between the sender's and receiver's virtual address spaces. This model eliminates operating system involvement in communication, provides full protection, supports user-level buffer management and zero-copy protocols, and minimizes software communication overhead. This paper describes system software support for the model including its API, operating system support, and software architecture, for two network interfaces designed in the* SHRIMP *project. Our implementations and experiments show that the* VMMC *model can indeed expose the available hardware performance to user programs. On two Pentium PCs with our prototype network interface hardware over a network, we have achieved user-to-user latency of 4.8 μsec and sustained bandwidth of 23 MB/s, which is close to the peak hardware bandwidth. Software communication overhead is only a few user-level instructions.*

1 Introduction

The trend in building scalable systems has been toward high-level integration using high-volume commodity components. While it is commonplace to build supercomputers using off-the-shelf microprocessor chips instead of custom-designed processing elements, the industry is now building large-scale systems using commodity board-level integration. For example, Convex uses HP workstation boards to construct their shared-memory multiprocessor, and IBM uses their high-end workstation boards to construct the SP2 multicomputer. The availability of new interconnection technologies has motivated several research projects to investigate how to use entire commodity workstation or PC systems and software to build high-performance scalable systems. This approach reduces system cost and tracks closely the exponential increase in all aspects of the technology base.

A challenge in building scalable systems using networks of commodity components is to achieve communication performance competitive with or better than custom-designed systems. The performance gap between the current networking technologies for local area networks (LANs) and those for custom-designed multicomputers is large. Message-passing latency for LANs, for example, is in the order of hundreds of microseconds whereas the latency for multicomputers is in tens. To bridge this gap, one needs to develop a very "thin" communication mechanism that allows applications to achieve communication latency and bandwidth very close to those available in hardware.

In order to satisfy the requirements of a variety of applications, the communication mechanism should support a wide range of communication facilities, including client-server protocols and multicomputer message-passing interfaces such as NX, PVM, and MPI. To achieve this goal, the communication mechanism should:

- support protected communication in multi-user environment,
- be easy to customize and integrate with commodity hardware and software, and
- provide user-to-user latency and bandwidth close to the underlying hardware and minimize CPU involvement in communication.

Traditional communication mechanisms for LANs and even for multicomputers do not meet these criteria. The total user-to-user message transfer latency is often one or even two orders of magnitude higher than the possible minimum time imposed by the underlying hardware. For example, on the Intel Paragon multicomputer, sending and receiving a message requires at least 30 μsec, of which less than 1 μsec is due to time on the wire [7].

Virtual memory-mapped communication (*VMMC*) is a communication model that allows user-to-user data transfers with latency and bandwidth close to those supported by the underlying hardware. This model requires very little host processor involvement in data transfer. The critical features of *VMMC* are:

To appear in *10th International Parallel Processing Symposium,* April, 1996.

- elimination of operating system involvement in communication,
- support for user-level buffer management and zero-copy protocols,
- no CPU involvement in data receiving,
- minimal CPU overhead when sending data, and
- support for protected communication in a multi-user environment.

The implementation of the *VMMC* model requires close cooperation between software and hardware. This paper describes the software support for *VMMC* and presents its hardware requirements. We have implemented *VMMC* for two network interface designs in the *SHRIMP* project [1, 2, 4]. We achieved user-to-user latency of 4.8 μsec and sustained bandwidth of 23 MB/s, which is close to the peak hardware bandwidth. The software communication overhead is only a few user-level instructions. In addition, the *VMMC* API is simple and its implementation requires minimal hardware and operating system support.

Section 2 discusses the reasons why traditional communication mechanisms deliver poor performance. Section 3 describes the model of virtual memory-mapped communication. Section 4 describes the application programming interface. In section 5 we discuss *VMMC*'s requirements for hardware and operating system support and give two examples of *VMMC* implementations. The performance of these implementations is discussed in Section 6. Section 7 contains discussion of related work. We present conclusions in section 8.

2 Problems with traditional communication models

Designing a new model requires understanding why the traditional send/receive communication model performs so poorly. Traditional systems provide protection by requiring applications to access the network through system calls, which significantly increases latency. The challenge for new communication mechanisms is to eliminate the system calls without sacrificing protection. On the sending side, additional sources of latency in the traditional model include verification in software of data access rights; checking for availability of receive buffers and preparation of packet descriptors. An interrupt is needed on the receiving side, resulting in even higher latency. Data copying on both ends reduces performance further. In our approach we would like to remove these overheads from the common case of data transfer.

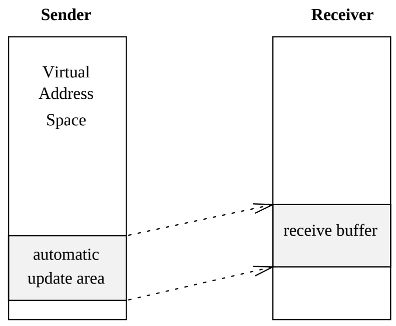

Figure 1: Example of automatic update mapping. The *sender* maps its local virtual address range to an imported buffer, for automatic update.

Existing communication models are difficult to customize because they are too high-level with too much semantics already built in. For example, the send/receive model includes complex buffer management functionality. If an application needs only fixed size buffers or can used preallocated buffers for communication, it cannot make use of this information to improve communication performance. For these reasons, our new communication mechanism should be low-level and minimal.

3 *VMMC* model

Essentially, virtual memory-mapped communication is a mechanism for protected data transfer from the sender's virtual address space to the receiver's virtual address space. Communication is protected because it may take place only after the receiver gives the sender permission to transfer data to a given area of the receiver's address space. The receiving process expresses this permission by `exporting` areas of its address space as receive buffers where it is willing to accept incoming data. A sending process must `import` remote buffers which it will use as destinations for transferred data. An exporter can restrict possible importers of a buffer; *VMMC* enforces the restrictions when an import is attempted. After a successful import, the sender can transfer data from its virtual memory into the imported receive buffer. *VMMC* makes sure that transferred data do not overwrite receiver's addresses outside the destination receive buffer.

VMMC supports two data transfer modes: `deliberate update` and `automatic update`. Deliberate update is an explicitly initiated transfer of a contiguous block of data from any (readable) virtual address in the caller's process to a previously imported receive buffer. *VMMC* guarantees in-order, reliable delivery of deliberate update messages.

Automatic update is performed implicitly by the system on each write to local memory. To use automatic update, a sender must first create an **automatic update mapping**. This designates a region of the sender's virtual memory as an automatic update area and "maps" it to a remote receive buffer (see Figure 1). Once the automatic update mapping exists, all writes by the sender to its automatic update area are automatically propagated to the remote receive buffer. Since writes are propagated automatically, there is no way to explicitly specify a destination for each write. The *VMMC* subsystem infers the destination from the local address written to. As a result, there may be only one automatic update destination associated with a given local address. *VMMC* guarantees in-order, reliable delivery of automatic update messages.

When a message sent by either transfer mode arrives at its destination, it is transferred directly into the memory of the receiving process, without interrupting the receiver's CPU. Thus there is no explicit receive operation in *VMMC*.

VMMC supports user-level buffer management because transfer is performed between user-level memory locations. Buffer management is divorced from the data movement mechanism and becomes the responsibility of the communicating parties. Zero copy protocols are possible because a direct application-to-application transfer of data can occur.

The CPU overhead to send data can be very small: one local write for automatic update, or a few user-level instructions for deliberate update. The model does not impose any CPU overhead to receive data, as there is no explicit receive operation. CPU involvement in receiving data can be as little as checking a flag; moreover program logic can be used to reason about what data has already arrived (since messages are delivered in order).

Our model as described in this section can be applied to communication between processes executing on one uniprocessor machine, on separate processors of a shared memory multiprocessor, or between processes executing on different nodes in a local area network. In the former two cases, *VMMC* is a special restricted case of shared memory communication with deliberate update added for bulk transfer. The LAN case is discussed in the next section.

4 Application programming interface

In this section we describe the *VMMC* API for distributed memory machines. We extend the model described in section 3 and introduce **destination proxy space** and **notifications**. Notifications are used to transfer control between processes and to notify receivers about external events. They are described in 4.4.

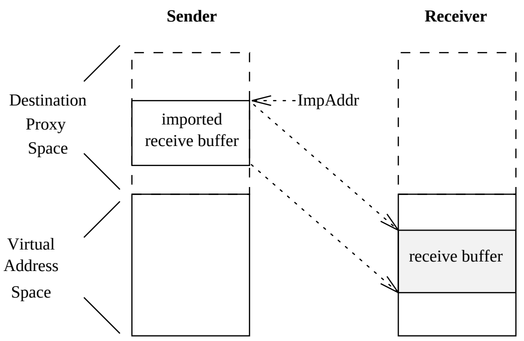

Figure 2: Mapping a receive buffer into a sender's destination space. The *receiver* exports the receive buffer which is imported by the *sender*.

Destination proxy space is a logically separate special address space in each sender process; it is used for addressing imported receive buffers. This new address space is usually a subspace of the sender's virtual address space[1], but it is not backed by local memory and its addresses do not refer to code or local data. Instead, they are used only to specify destinations for data transfer. After sender imports a remote receive buffer, a representation of the imported buffer is mapped into the sender's destination proxy space. This representation is simply a range of addresses in destination proxy space. An address belonging to a given range can be translated by *VMMC* into a destination machine, process, and virtual address. Figure 2 shows the mapping of a receive buffer allocated on node *Receiver* into destination proxy space of node *Sender*.

4.1 Establishing import-export mappings

In *VMMC*, data transfer can occur only after receive buffer has been mapped into the sender's destination proxy space. The mapping is needed for several reasons. First, it enforces protection by allowing *VMMC* to verify that the sender has permission to transfer data into the receiver's buffer. Second, it allows receive buffers to be pre-allocated, so the sender can transfer data without having to check the availability of receive buffer space. Third, it allows *VMMC* to identify which memory areas are writable by incoming messages.

The sender and receiver cooperate to establish an import-export mapping. The receiving process exports a part of its address space with the following operation:

`Export(mapId, recvBuf, nbytes, permissions)`

where *recvBuf* is the virtual address of the first byte of the receive buffer to be exported, and *nbytes* is its size. *permissions* describes the set of processes that are allowed to import this receive buffer. *mapId* is an

[1]Depending on the implementation, destination space can be also separate from virtual space.

integer identifier, which is used by importer to establish import-export mapping with the following call:

`Import(mapId, exporter, &impAddr, &nbytes)`

where *exporter* identifies the exporting process. After a successful import, *impAddr* contains the starting address in the importer's destination proxy space of a local representation of imported receive buffer (see Figure 2) and *nbytes* contains the size of the imported buffer.

VMMC supports both blocking and nonblocking versions of the import operation. The blocking version waits until a matching buffer has been exported, while the nonblocking version returns a handle which can be used later to check whether the import operation has finished.

Establishing an import-export mapping requires a trusted third party (operating system kernel or daemon process) to verify protection, so it can be relatively expensive. However, it should occur infrequently, as import is required only once for a given receive buffer; afterward messages can be send cheaply.

4.2 Deliberate update

Deliberate update is an explicit request to transfer data from anywhere in virtual memory of a sender to a previously imported receive buffer. The basic deliberate update operation is as follows:

`SendMsg(srcAddr, destAddr, nbytes)`

This is a request to transfer *nbytes* of data from sender's virtual address *srcAddr* to a receive buffer designated by *destAddr* in the sender's destination proxy space. For example, if process *Sender* uses *ImpAddr* (see Figure 2) as a destination address, this request will transfer data to beginning of *Receiver*'s receive buffer. It is the responsibility of *VMMC* to ensure that data is not transferred outside the selected receive buffer.

If no error occurs, *SendMsg* returns after all data has been sent out to the network. *VMMC* also provides a non-blocking variant of *SendMsg*, which takes the same arguments, but initiates the data transfer and then immediately returns a handle identifying the request. The handle can be used later to inquire about the status of the request. The non-blocking send is designed to minimize the CPU overhead required to start a data transfer. *VMMC* does not notify the sender when the data arrives at the receiver or when the receiver consumes the data. Obtaining this information requires a higher-level protocol.

4.3 Automatic update

Deliberate update allows any local data to be sent to any previously imported receive buffer. Automatic update is introduced to obtain lower latency and to avoid explicit transfer initiation. (This can benefit applications like shared virtual memory).

Automatic update propagates writes to local memory to remote receive buffers. A sender process sends updates to a receiver simply by writing to the sender's local virtual memory. The latency of transfer initiation can be completely hidden from the sender. Since no special send operation is required, existing code using shared memory model can be often ported to *VMMC* system with little or no modification.

To allow automatic update, *VMMC* provides the `Map` operation. It maps an area in the virtual address space, for automatic update, to an already imported receive buffer, as in Figure 1. Separation *Map* and *Import* is useful since the former can be implemented as a local operation. *VMMC* also provides an `Unmap` operation which unmaps given range of local addresses mapped previously as an automatic update area. This operation can be also used to change automatic update destination (which requires unmapping the old destination and mapping the new one), or to avoid overwriting the remote receive buffer while performing local computations.

4.4 Notifications

VMMC provides `notifications` to support transfer of control and to notify receiving processes about external events. Attaching a notification to a message causes the invocation of a user-level handler function in the receiving process after the message has been delivered into the receiver's memory.

A variant of the deliberate update *SendMsg* operation is provided to send a message with a notification attached. When an automatic update mapping is created, a sender may specify that a notification should be attached to every automatic update message propagated by the mapping.

The receiving process can associate a separate notification handler with each exported buffer. If a message with notification arrives at a receive buffer that has no handler, the notification is ignored.

Like UNIX signals, notifications can be blocked and unblocked; they can be discarded or accepted (separately for each buffer); and a process may be suspended waiting for notifications. Unlike signals, notifications are queued when blocked.

4.5 Destroying import-export mappings

Either the receiver or the sender may destroy an import-export mapping. A sender calls `Unimport` to indicate that it no longer plans to transfer data to a given receive buffer. After the unimport completes, the range of sender's destination proxy space associated with the unimported buffer becomes free and can be reused for another imported buffer.

A receiver calls `Unexport` to destroy a given receive buffer. After this call completes, all import mappings to this buffer are gone and any messages sent using these mappings have been delivered. Since *Unexport* may require communication with multiple senders to flush and destroy mappings, *VMMC* API supports a non-blocking version of *Unexport*.

5 Implementation

In this section we discuss what the *VMMC* implementation needs from the hardware and the operating system. The hardware description is mostly functional and is presented to establish context for the discussion of operating system requirements. We will describe the software architecture of two *VMMC* implementations we have done as part of the *SHRIMP* project.

VMMC is implemented by cooperating software and hardware. Most of *VMMC*'s functionality can and should be provided by software, with hardware implementing only performance-critical features. The exact division of labor between hardware and software varies between implementations.

Data transfer, including direct placement of data into the receiver's memory, should clearly be provided by hardware, since it is the most common operation. Establishing and destroying import-export mappings can be relegated to software, since it is much less common than data transfer.

5.1 Hardware requirements

To realize the potential performance of *VMMC* some hardware support is necessary. We discuss it in stages, starting with hardware that is easy to implement (or already existing on commodity systems), proceeding to hardware that is useful but progressively more difficult to implement.

The necessary hardware features are as follows:

- *cache-coherent DMA to and from the network:* The network interface must be able to transfer data directly to and from cacheable memory. This allows the source and destination buffers to be in cacheable memory. Without cache-coherent DMA, incoming messages would have to be written into special non-cacheable memory, requiring an extra user-level copy, or would require taking an interrupt on data arrival. Many popular architectures, including PCs, support cache-coherent DMA.

- *ability to cheaply determine destination physical address for incoming data:* In order to transfer incoming data directly into memory without interrupting the CPU, the network interface should be able to figure out where to put the data. One simple way to do this is to pin receive buffers in memory and communicate the physical addresses of buffer frames to importers. In this approach, each message carries a destination physical address in its header. Thus, virtual memory mapping is implemented in software on top of hardware that supports physical memory mapping.

- *reliable, ordered network message delivery:* Delivering messages out of order directly into user space or losing packets is bound to cause a disaster unless user code runs an expensive higher-level protocol. Modern networks like SCI and Myrinet deliver messages in the order they were sent. SCI provides reliable message delivery, while Myrinet has extremely low error rates, so data loss can be treated as a catastrophic event.

Useful, but more difficult to implement hardware features include:

- *support for automatic update:* Automatic duplication of local writes to remote memory can be implemented by snooping the memory bus, or by allocating part of main memory on a standard I/O bus network interface.

- *support for user-level, protected initiation of deliberate update:* In the absence of automatic update, this feature is important for low latency. Even with automatic update, it is still useful for limiting the CPU overhead required to initiate a large transfer [2].

- *support for receive buffers larger than the size of DRAM:* This would require a TLB on the network interface and some support for handling misses when the destination page is not in main memory.

5.2 Operating system requirements

We assume that most of the *VMMC* functionality is provided by a user-level *VMMC* daemon (server) process; the server implements creation and destruction of import-export and automatic update mappings. We introduce the daemon to minimize the kernel modifications required.

The operating system requirements are a direct consequence of the support provided by the hardware. Here are the operating system features we deem necessary for *VMMC* implementation:

- *ability to DMA between any location in physical memory and the network:* We want to use cache coherent DMA to transfer data directly between user-level structures and network, so the operating system should allow any memory to be used as a source or destination of a DMA transfer.

- *DMA-aware pager:* The operating system must take care to avoid paging out a page that is being used as a source for a currently-executing outgoing DMA transfer. Such a page should be considered temporarily locked until the transfer is finished. In practice this means that OS pager should be able to quickly determine whether a given page is scheduled to be used, or is being used now, as a source for DMA transfer. The network interface knows the physical addresses of such pages; it must allow some way for the operating system to check whether a specific page is a source of a pending transfer.

- *support for* VMMC-*specific memory locking:* On the receiving side, the network interface hardware usually cannot translate virtual to physical addresses, so receive buffer pages must be locked into memory. Once a page of a receive buffer is locked in memory, it must remain locked until all of its import-export mappings are dead, even if the process owning it dies. As long as an import-export mapping exists, a message may arrive at the page, so the page cannot be reused.

- *support for direct page modifications from the network:* The initiation of outgoing DMA to the network is synchronous with respect to operating system activities, while incoming transfer from the network is asynchronous. Receive buffer pages can be changed by incoming packets quietly, behind the back of the operating system. This has two consequences: optimizations like copy-on-write should not be allowed on locked pages, and, when deciding if such pages are dirty, the operating system should take into account the possiblity of modifications from the network. One simple solution is to assume that all locked pages are dirty; another is to use a per-page dirty bit implemented in the network interface hardware.

Additional operating system support is required when more features are provided by the hardware. For example, if the network interface supports user-level deliberate update, then special virtual memory mappings are needed [2]. In this case, the *VMMC* daemon must be able to manipulate the client's virtual address space.

If automatic update is supported by snooping, writes to automatic update areas should appear on the bus snooped by the network interface. Operating system support for selecting per-page caching strategy is useful in this case; we can cache pages mapped for automatic update in write-through mode and all other pages in write-back mode.

5.3 Software architecture

We describe our software architecture in the context of the *SHRIMP* project. Two network interfaces have been designed for the *SHRIMP* multicomputer; *SHRIMP* is constructed from EISA Pentium PCs and an Intel Paragon routing network. The first design explores how to minimally modify the traditional DMA-based network interface design, while implementing virtual memory mapping in software. It requires a system call to initiate outgoing data transfer, but received messages are delivered directly to memory by hardware, reducing the receive software overhead to only a few instructions in the common case. The second design implements virtual memory mapping completely in hardware. This approach provides fully protected, user-level message passing, and it allows user programs to initiate an outgoing block data transfer with two user-level instructions. Based on these network interfaces we built two implementations of *VMMC*.

SHRIMP-I [4], the first network interface designed by the *SHRIMP* team, uses the traditional DMA approach with added support for including destination physical address in the message header. On export, receive buffer pages are pinned in physical memory, so virtual memory mapping across nodes is implemented using physical memory mapping. For each process, the system software maintains a destination table, providing translation from destination space addresses to remote physical memory addresses. Deliberate update is initiated with a system call, and automatic update is not supported. On message arrival, the network interface hardware uses the physical address included in the message header to transfer message data directly into memory.

SHRIMP-II, like *SHRIMP-I*, includes destination physical address in the message header, allowing direct transfer into the destination memory. Also as in *SHRIMP-I*, receive buffers are pinned and physical memory mapping across nodes is used. *SHRIMP-II* extends the *SHRIMP-I* interface by implementing automatic update and user-level protected initiation of deliberate update. Destination space is now a part of the sender's virtual address space, and the destination table is maintained on the network interface. Automatic update writes are snooped off the memory bus. A user process can initiate a deliberate update transfer with just two memory-mapped I/O instructions [2]; these instructions access locations on the network interface board across I/O bus. Virtual memory mappings are used to verify permissions and provide protection between users on initiation of deliberate update.

Functionally, the *VMMC* software architectures on the two *SHRIMP* systems are similar. Each consists of

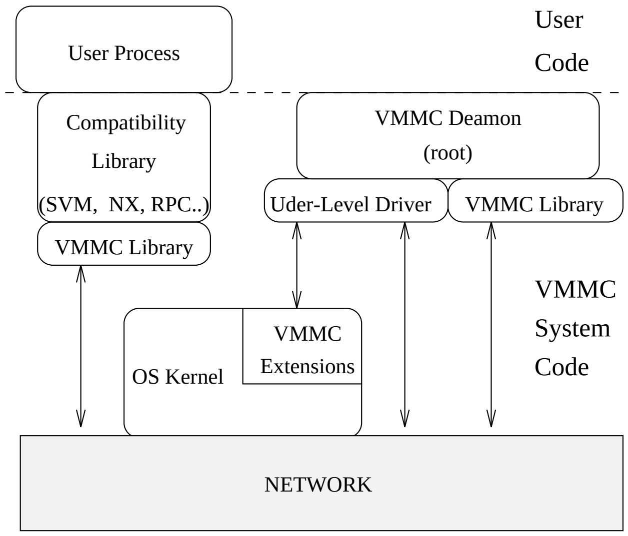

Figure 3: System software implementing *VMMC* in SHRIMP-II.

five parts:

- a *VMMC* daemon running on each node,
- a user-level network interface driver,
- *VMMC*-specific kernel extensions,
- the *VMMC* basic library, and
- compatibility libraries.

Figure 3 shows the relationship among these components in the case of *SHRIMP-II*. The one feature specific to *SHRIMP-II* is that the *VMMC* library has direct access to the network. In *SHRIMP-I* access to the network is through a system call.

The *VMMC* daemon runs on each node of the *SHRIMP* machine with superuser permission. The purpose of the daemon is to assist user processes in establishing and destroying import-export mappings and automatic update mappings (if automatic update is supported by hardware). Client processes communicate with their local daemon, never with remote daemons. The daemons communicate with each other, exchanging information about clients, exported buffers, and outstanding import requests, as well as global configuration information. Export requests are maintained by the local daemon in a hash table. Import requests are propagated to the daemon of the exporter where they are matched to the appropriate export, or queued in the daemon hash table if the requested export has not been performed yet.

The network interface driver, linked into the daemon's address space, allows daemons to manipulate the protected hardware state, for example to establish an import-export mapping.

Kernel [2] extensions are used by a daemon to lock receive buffer memory and to create destination space mappings for both *SHRIMP* systems. Additionally, in *SHRIMP-I* deliberate update send is implemented as a system call. In *SHRIMP-II*, there are two system calls to manipulate destination space (which is a part of client's virtual address space) and one more system call to control page caching mode. This latter call is used to specify write-through caching for automatic update pages.

Programs that use *VMMC* are linked with the *VMMC* library which provides system-specific implementation of the *VMMC* operations. The library implements the interface described in section 4. Most of the API calls are implemented as IPC to the local daemon in both *SHRIMP* systems. The exception is deliberate update initiation, which in *SHRIMP-I* is a system call, and in *SHRIMP-II* is a small user-level procedure. The *VMMC* library is intended to provide a basic, low-level interface for libraries implementing higher-level specialized interfaces.

To exploit the performance benefits of *VMMC* without extensive rewriting of applications, we provide compatibility libraries. By using the *VMMC* library, which hides the details of memory-mapped communication implementation, compatibility libraries are system-independent. Each compatibility library implements one programming model interface like RPC, stream sockets, NX/2 [7] message passing, or shared virtual memory. Compared to traditional implementations of these models, we avoid receive system calls in both *SHRIMP* systems, and send system calls in *SHRIMP-II*.

Depending on the model, some additional savings can be realized by using *VMMC*. For example, in our RPC implementation [1], calls to the stream and network layers are replaced with calls to the *VMMC* library; as a result, a copy between the user buffer and the stream buffer can be avoided. Compared to the standard TCP-based implementation, null RPC on *SHRIMP-II* executes only one fourth as many instructions and has four fewer system calls per RPC. The *VMMC* implementation of NX/2 [1] achieves a user-to-user four-byte message latency of 12 μseconds, which is about factor of four smaller than that on the Intel Paragon. Shared virtual memory on *SHRIMP-II* [5] uses automatic update to merge fine-grain updates to shared pages and to avoid page faults if pages are shared by only two nodes. As a result, this implementation achieves substantial speedups compared to start-of-the art SVM implementations, even if the same network is

[2] We use the LINUX operating system.

used in both cases [5].

6 Performance

To verify how our two *VMMC* implementations compare with the traditional send/receive model, we performed a number of experiments. All performance numbers are preliminary, as our hardware is untuned and the *SHRIMP* system is still in the phase of debugging and integration.

Our system consists of 60 MHz Pentium PCs with Xpress memory bus and EISA I/O bus, connected by a high-performance Intel Paragon backplane. The hardware local-to-remote memory bandwidth is limited by the EISA bus bandwidth. The theoretical peak EISA bandwidth is 33 MB/sec, but this cannot be achieved in practice because some cycles are lost because of memory refresh and mandatory periodic rearbitrations for the bus. The hardware local-to-remote memory latency is about 4 μsec.

In *SHRIMP-II*, we measured user-to-user latency using a ping-pong test. Automatic update latency was about 4.8 μsec; deliberate update latency was 7.8 μsec. This one-way time includes transfer initiation, data transfer across the network, and DMA into an already exported receive buffer, plus overhead caused by the test code. Deliberate update latency is longer than automatic update latency because of two memory-mapped I/O references over the slow EISA bus, additional DMA on the sending side, and checking for page alingment. Both latency numbers are close to the hardware minimum.

User-to-user bandwidth in both *SHRIMP* systems is about 23 MB/s, which is close to the raw bandwidth we were able to achieve over the EISA buses in our PCs. It is also 70% of 33 MB/sec, which is the maximum theoretical bandwidth of the EISA bus. The discrepancy between achieved and theoretical bandwidths is not caused by the *VMMC* protocol, but by practical limitations of the EISA bus. Automatic update bandwidth is 20 MB/sec when writes are to consecutive locations (because they can be combined by network interface).

VMMC provides good support for overlapping communication and computation, because it is cheap to initiate a message. For automatic update, the CPU overhead to send is single local write. For deliberate update on *SHRIMP-II*, initiating a transfer requires two references to memory-mapped I/O space, plus some instructions to check for page boundaries within the message. The total cost is about 2.8 μsec. In *SHRIMP-I*, this overhead is higher — about 6.7 μsec for 1 word transfer — because a send system call is required.

We compared *SHRIMP* performance with an optimistic estimation of latency, bandwidth and software overhead in the traditional send/receive model (assuming the same network and interface as in our other experiments). We considered two kernel implementations of traditional send/receive communication: the *Kernel-1-Copy* protocol is similar to the NX implementation for Intel multicomputers; the *Kernel-2-Copies* protocol corresponds to a standard workstation implementation. Both models require send and receive system calls and an explicit receive operation that copies data from the system receive buffer into an application buffer. They differ in how the send operation is implemented: *Kernel-2-Copies* copies data from user space to kernel send buffer, whereas *Kernel-1-Copy* sends data directly from user space.

Model	Latency (μsec)	Bandwidth (MB/sec)
SHRIMP-I	10.7	23
SHRIMP-II deliberate update	7.8	23
SHRIMP-II automatic update	4.8	20
Kernel-1-Copy	13.7 + recv. interrupt	12
Kernel-2-Copies	15.7 + recv. interrupt	8

Table 1: User-to-user latency and bandwidth of traditional send/receive and *VMMC*. The same host and network is assumed for all models.

Table 1 shows a latency and bandwidth comparison. The latency of a one word transfer, with send and receive system calls, but without interrupt on receive, is more than 15 μsec, which is about three times our automatic update latency. In our experiments, user buffers are initially in the cache and kernel buffers are not. Cache-to-memory copy bandwidth is low, about 22 MB/sec, because of slow DRAM. As a result, the bandwidth of pipelined 4KB transfers with copying between user and kernel buffers on both send and receive sides is about 8 MB/s, about three times lower than our deliberate update bandwidth. With one copy, the bandwidth is 12 MB/sec, still half of the deliberate update bandwidth. With new high-speed I/O buses like PCI, this comparison will be even more favorable for *VMMC*, because deliberate update bandwidth is limited by I/O bus.

Table 2 shows the CPU overhead required to transfer a one word message. For *VMMC*, receive-side overhead is zero, as data is delivered directly into user memory.

Model	Sending Overhead (μsec)	Receiving Overhead (μsec)	Total Overhead (μsec)
SHRIMP-I	6.7	0	6.7
SHRIMP-II deliberate update	2.8	0	2.8
SHRIMP-II automatic update	0.01	0	0.01
Kernel-1-Copy	6.7	5 + recieve interrupt	11.7 + receive interrupt
Kernel-2-Copies	6.9	5 + recieve interrupt	11.9 + receive interrupt

Table 2: Communication overhead on CPU imposed by send/receive and *VMMC* models. The same host and network is assumed for all models.

SHRIMP-I send-side overhead is high because of the cost of a system call and verification of send rights in software. For both send/receive models the overhead is substantial because of send and receive system calls, verification of send rights, kernel buffer management, and DMA setup in software.

The latency of send/receive communication is about three times that of automatic update. The bandwidth of both *VMMC* implementations is three times higher than in the send/receive model where copies are performed by both sender and receiver; and two times higher if only one copy takes place. Total CPU overhead is four times lower in *SHRIMP-II* deliberate update than in send/receive communication.

7 Related work

Spector proposed a *remote reference/remote operation model* [9] in which a master process on a local processor performs remote reference and a slave process on another remote processor implements this reference by executing the remote operation. Compared to this model, *VMMC* is much simpler, as we envision hardware support only for data transfer. We also use virtual memory mappings across the network to provide protection and to eliminate software overhead on communication, whereas Spector's model did not discuss virtual memory or protection issues.

Thekkath, Levy and Lazowska [10] proposed a model similar to *VMMC*. Both models use protected memory-based communication. However, their model is not based on memory mappings across the network; instead their API uses memory regions and offsets. This difference leads to differences in implementation: they cannot use the existing virtual memory system to verify send rights, as we do in *SHRIMP-II*. Instead, their send operation requires trapping into the operating system kernel to verify send rights in system software, which leads to increased latency. Also, their model does not include support for automatic update, which is important for applications like shared virtual memory.

Hamlyn [13] is a new network interface under design at HP Laboratories. It provides some functionality useful for *VMMC*: user-level protected access to the network and direct transfer to receive-side user buffers. However, Hamlyn does not support automatic update and both send and receive areas must be pinned in physical memory.

Thekkath and Levy [11] investigated factors limiting performance of communication in high-speed networks. Their results indicate that modern distributed systems require both high throughput and low latency. As a result, support for both fast short messages and longer high bandwidth data transfers is needed.

Karamcheti and Chien studied software overhead on message passing [6]. They argue that the networking hardware should provide reliable, in-order message delivery (as *VMMC* assumes). They show that omitting these features, can more than double software overhead for message passing.

The *VMMC* model provides similar functionality to Active Messages [12]. In this asynchronous communication mechanism each message contains at its head the address of a user-level handler which is executed on message arrival. A *VMMC* message with notification is very similar to an active message. Unlike active messages, *VMMC* provides a way to transfer data without transferring control. Executing a handler can be expensive, since the processor must be interrupted. We believe that data transfer is the common case, so *VMMC* should have a performance advantage. *VMMC* also allows the sender more control over where in memory data is to be delivered, which makes it easier to implement zero-copy protocols.

Scalable, coherent shared memory is an attractive model for large scale parallel computing. The main obstacle to widespread use of shared-memory systems is high cost and high hardware complexity. The NUMA model of global address space partitioned across all nodes (implemented for example in the Cray T3D [8]) is also easy to use and provides a low latency data transfer mechanism. Compared to *VMMC*, its support requires more hardware and closer integration with host processor and memory. Moreover, reading remote memory incurs substantial overhead which needs to be addressed by migrating or replicating of data [3].

8 Conclusions

This paper describes software support for the virtual memory-mapped communication mechanism. This approach allows data transfer directly from the sender's to the receiver's address space, and eliminates operating system involvement in communication while providing full protection. The API is simple and yet supports user-level buffer management and zero-copy protocols.

The *VMMC* model supports two data transfer modes: deliberate update and automatic update. Neither case requires CPU involvement in data transfer. With deliberate update, bandwidth close to the hardware limit can be achieved as DMA moves data directly from user-level send buffer to the network and back to user-level receive buffer. With automatic update, the overhead on CPU of send initiation is one local write, as writes to local memory are automatically propagated to remote destination. In both cases, no receive interrupt is needed.

The *VMMC* model and its implementations are the result of hardware and software co-design. As a result, our model requires minimum hardware support, limited to the network interface. It concentrates on speeding up the most common case: data transfer between user address spaces. Most of the system software is implemented by a user-level daemon; only small changes to the operating system kernel are needed.

The model's API is small, low-level and flexible. It supports establishing and destroying mappings to remote buffers as well as operations to initiate deliberate update. Additionally, the API provides notifications to facilitate transfer of control. We have implemented several compatibility libraries using *VMMC* library, to support traditional interfaces like NX/2, RPC, sockets and shared virtual memory. Using compatibility libraries, the legacy code written to these interfaces can be ported to a new model without source code modifications and with substantial gains in performance.

Acknowledgements

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

References

[1] R. Alpert, A. Bilas, M. Blumrich, D.W. Clark, S. Damianakis, C. Dubnicki, E. Felten, L. Iftode, and K. Li. Early experience with message-passing on the shrimp multicomputer. In *Proceedings of the 23rd Annual Symposium on Computer Architecture*, 1996.

[2] M. A. Blumrich, C. Dubnicki, E.W. Felten, and K. Li. Protected, user-level dma for the shrimp network interface. In *Proceedings of the Second International Symposium on High Performance Computer Architecture*, February 1996.

[3] A.L. Cox and R.J. Fowler. The implementation of a coherent memory abstraction on a numa multiprocessor: Experiences with platinum. In *Proceedings of the Twelfth Symposium on Operating Systems Principles*, pages 32–44, December 1989.

[4] C. Dubnicki, K. Li, and M. Mesarina. Network interface support for user-level buffer management. In *Workshop on Parallel Computer Routing and Communication Workshop*. Springer-Verlag, April 1994.

[5] L. Iftode, C. Dubnicki, E.W. Felten, and K. Li. Improving release-consistent shared virtual memory using automatic update. In *Proceedings of the Second International Symposium on High Performance Computer Architecture*, February 1996.

[6] V. Karamcheti and A. Chien. Software overhead in messaging layers: Where does the time go? In *The 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 51–60, October 1994.

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

[8] Cray Research. Cray T3D system architecture overview. Technical Report HR-04033, Cray Research Inc, September 1993.

[9] A. Z. Spector. Performing remote operations efficiently on a local computer network. *Communications of the ACM*, 25(4):260–273, April 1982.

[10] Ch. A. Thekkath, H.M. Levy, and E.D. Lazowska. Separating data and control transfer in distributed operating systems. In *The 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, October 1994.

[11] Ch.A. Thekkath and H.M. Levy. Limits to low-latency communication on high-speed networks. *ACM Transactions on Computer Systems*, 11(2):179–203, May 1993.

[12] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of 19th ISCA*, pages 256–266, May 1992.

[13] J. Wilkes. Hamlyn – an interface for sender-based communications. Technical Report HPL-OSR-92-13, Hewlett-Packard Laboratories, November 1993.