Design and Implementation of Virtual Memory-Mapped Communication on Myrinet

Cezary Dubnicki, Angelos Bilas, Kai Li
Princeton University
Princeton, New Jersey 08540
{dubnicki,bilas,li}@cs.princeton.edu

James Philbin
NEC Research Institute, Inc.
Princeton, New Jersey 08540
philbin@research.nj.nec.com

Abstract

This paper describes the design and implementation of the virtual memory-mapped communication model (VMMC) on a Myrinet network of PCI-based PCs. VMMC has been designed and implemented for the SHRIMP multicomputer where it delivers user-to-user latency and bandwidth close to the limits imposed by the underlying hardware. The goal of this work is to provide an implementation of VMMC on a commercially available hardware platform; to determine whether the benefits of VMMC can be realized on the new hardware; and to investigate network interface design trade-offs by comparing SHRIMP with Myrinet and its respective VMMC implementation.

Our Myrinet implementation of VMMC achieves 9.8 microseconds one-way latency and provides 108.4 MBytes per second user-to-user bandwidth. Compared to SHRIMP, the Myrinet implementation of VMMC incurs relatively higher overhead and demands more network interface resources (LANai processor, on-board SRAM) but requires less operating system support.

1. Introduction

Low cost and high performance are the potential advantages that motivate building a high-performance server out of a network of commodity computer systems. Such servers can cost substantially less than custom-designed ones because they can leverage high-volume commodity hardware and software. They can also deliver high performance because commodity hardware and software track technologies well.

A key enabling technology for this approach is a high-performance communication mechanism that supports protected, user-level message passing. In order for the communication mechanism to be non-invasive with respect to the commodity hardware and commodity operating system software, it must require no modifications to these components. This implies that the communication mechanism should support protection among multiple processes and should be able to deal with communication among virtual address spaces without using special scheduling policies. In order to achieve low latency and high bandwidth, the communication mechanism should also reduce message-passing software overheads to a minimum. This implies support for user-level message passing.

The virtual memory-mapped communication (VMMC) mechanism provides good support for protected, user-level message passing [7, 8]. Its main idea is to allow data to be transmitted directly from a source virtual memory to a destination virtual memory. For messages that pass data without passing control, the VMMC approach can completely eliminate software overheads associated with message reception.

The model has been designed and implemented for the SHRIMP multicomputer [3, 4, 8]. Since the the SHRIMP network interface supports VMMC, the implementation requires either no software overhead or only a few user-level instructions to transfer data between the separate virtual address spaces of two machines on a network. The VMMC model eliminates operating system involvement in communication while at the same time providing full protection. It supports user-level buffer management and zero-copy protocols. Most importantly it minimizes software overhead. As a result, available user-to-user bandwidth and latency are close to the limits imposed by the underlying hardware.

The goals of this study are threefold: First, we wanted to provide an implementation of VMMC on a commercially available hardware platform such as Myrinet,[1] and to describe its implementation and performance. Second, we wanted to investigate the extent to which the benefits of VMMC could be realized on such an implementation. For example, we wanted to know if user-level bandwidth and latency could be kept as close to the hardware limits as they are on the SHRIMP system. Third, we wanted to investigate

[1] Myrinet is a product of Myricom, Inc. in Arcadia, CA

network interface design tradeoffs by comparing SHRIMP with Myrinet and the resulting VMMC implementation. We were interested in how particular hardware features would impact the performance or protection between users and what it would cost in terms of design, implementation, and required software support.

This paper describes the design and implementation of VMMC for the Myrinet [5]. Our implementation achieves 9.8 microseconds one-way latency and provides 108.4 MB/s user-to-user bandwidth. When compared with the SHRIMP implementation, the Myrinet implementation of VMMC incurs relatively higher overhead and demands more network interface resources (LANai processor, on-board SRAM) but requires less operating system support.

2. Overview of VMMC

Virtual memory-mapped communication (VMMC) is a communication model providing direct data transfer between the sender's and receiver's virtual address spaces. This section provides a high-level overview of VMMC as implemented on Myrinet hardware. For an in-depth introduction to VMMC see [7].

The VMMC mechanism achieves protection by requiring that data transfer 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 that 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 the transferred data does not overwrite any memory locations outside the destination receive buffer.

Imported receive buffers are mapped into a `destination proxy space` which is a logically separate special address space in each sender process. It can be implemented as a subset of the sender's virtual address space or as a separate space (this is the case with the Myrinet implementation of VMMC, see section 4). Destination proxy space 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. A valid destination proxy space address can be translated by VMMC into a destination machine, process, and memory address.

As implemented on Myrinet, VMMC supports the deliberate update mode of data transfer. Deliberate update is an explicit request to transfer data from sender's local virtual memory to a previously imported receive buffer. The basic deliberate update operation is as follows:

$$\texttt{SendMsg}(\texttt{srcAddr}, \texttt{destAddr}, \texttt{nbytes})$$

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.

When a message arrives at its destination, its data 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 also 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.

3. Myrinet and Its Network Interface

Myrinet is a high-speed local-area network or system-area network for computer systems. A Myrinet network is composed of point-to-point links that connects hosts and switches. The network link can deliver 1.28 Gbits/sec bandwidth in each direction [1, 5].

The Myrinet network provides in-order network delivery with very low bit error rates (below 10^{-15}). On sending, the 8-bit CRC is computed by hardware and is appended to the packet. On a packet arrival, CRC hardware computes the CRC of the incoming packet and compares it with the received CRC. If they do not match, a CRC error is reported.

We designed and implemented a VMMC mechanism for the Myrinet PCI network interface. The PCI network interface is composed of a 32-bit control processor called LANai (version 4.1) with 256 KBytes of SRAM (Static Random Access Memory). The SRAM serves as the network buffer memory and also as the code and data memory of the LANai processor. There are three DMA engines on the network interface: two for data transfers between the network and SRAM and one for moving data between the SRAM and the host main memory over the PCI bus. The LANai processor is clocked at 33 MHz and executes a LANai control program (LCP) which supervises the operation of the DMA engines and implements a low-level communication protocol. The LANai processor cannot access the host memory directly; instead it must use the host-to-LANai DMA engine to read and write the host memory. The internal bus clock runs at twice the CPU clock letting the two DMA engines operate concurrently.

4. Design

4.1. Overview

Our implementation of VMMC on Myrinet consists of a number of software components cooperating with each other. Trusted system software includes:

- VMMC daemon (one per node),
- kernel-loadable VMMC device driver,
- network mapping LANai control program,
- VMMC LANai control program which actually implements VMMC.

The user-level software consists of the VMMC basic library. A user program must link with it in order to communicate using VMMC calls.

In brief overview, user programs submit export and import requests to a local VMMC daemon. Daemons communicate with each other over Ethernet to match export and import requests and establish export-import relation by setting up data structures in the LANai control program. After a successful import, the user process submits send requests directly to the LANai processor by writing to an entry in the process send queue which is allocated in the LANai SRAM. A request consists of a local virtual address specifying send buffer, the number of bytes to send and the destination proxy address which is used by the LANai to determine the destination node and physical address. The VMMC LANai control program picks up a send request, translates the virtual send buffer address into a physical address and then sends the data to the destination node. The Myrinet driver assists the LANai control program in the virtual to physical translation of send addresses, if such assistance is necessary, i.e. the translation entry is not present in the LANai SRAM software TLB. The Myrinet driver also implements notification delivery to user processes.

4.2. Assumptions

We assume that the Myrinet topology is static during program execution. If nodes are added, deleted, or system topology changes in any way the system software must be restarted in order to remap the network.

We do check for CRC errors, but do not recover from them. According to Myricom these errors should be very rare and incorporating CRC error recovery in our system would complicate its design and add more software overhead. So far our experience has validated this decision. We were able to transmit gigabytes of data without a single CRC error. In our experience, if these error do occur they are clustered and indicate a serious hardware malfunction rather than a sporadic transmission error.

4.3. Network Mapping

When the system boots, each VMMC daemon loads a special LANai control program derived from software provided by Myrinet, that automatically maps the network. This program assembles routing information from each node to all possible destinations within a given network by exchanging special mapping messages between nodes. We refer the interested reader to the Myricom documentation for more detail on the implementation of the network mapping LCP [1].

After each node has mapped the entire network, each VMMC daemon extracts the routing information, and then replaces the mapping LCP with an LCP that implements VMMC. When the VMMC LCP operates, no dynamic remapping of the network takes place and all the routing information resides in static tables created by the mapping LCP.

4.4. Establishing Export-Import Relation

Similarly to the SHRIMP implementation of VMMC, there are page tables maintained in each network interface. The incoming page table (one per interface) has one entry for each physical memory frame, which indicates whether a given frame can be written by an incoming message and whether a notification should be raised when a message arrives.

There is also an outgoing page table, which is allocated separately for each process using the VMMC on a given node. An entry in this table corresponds to a proxy page of a receive buffer imported by this process. The format of this entry is a 32-bit integer which encodes the destination node index and physical page address. The size of the outgoing page table limits the total size of imported receive buffers. The current limit is 8 MBytes; however, the outgoing page table is only limited by the amount of available SRAM on the LANai card and the number of processes simultaneously using a given interface for VMMC communication.

To export a receive buffer, a user program contacts the local VMMC daemon which locks the receive buffer pages in main memory and sets up entries corresponding to the receive buffer pages in the incoming table to allow data reception. On an import request, the importing node daemon obtains the physical addresses of receive buffer pages from the daemon on the exporting node. Next, the importing node daemon sets up outgoing page table entries for the importing process that point to receive buffer pages on remote node.

When sending, the user process passes a proxy address that specifies the destination for the data. A proxy address consists of a proxy page number and an offset within the page. We ensure that a process can send only to valid destinations. Since the outgoing page table is local to the sending process, there is no way a process can use outgoing page table entries set up for others.

4.5. Sending and Receiving Data

Each process has a separate send queue allocated in LANai SRAM. To submit a send request, the user process writes the next entry in the send queue with the length of the data to be sent and a proxy address which specifies the destination, as described before.

There are two types of send requests: short and long. The short request sends a small amounts of data (currently up to 128 bytes) by copying it directly into the send queue in the LANai memory (no host-LANai DMA is used in this case). With the long send request (up to 8 MBytes), the user process passes only the virtual address of the send buffer. The existence of two send formats is transparent to user programs. The VMMC basic library decides which format to used for a particular *SendMsg* VMMC call.

A long message is sent in chunks. Each chunk consists of routing information, a header, and data. The routing information is in standard Myrinet format. The header includes the message length and two physical destination addresses. The two addresses are needed in order to perform two piece scatter in the case when the destination memory spans a page boundary. When no page boundary is crossed, the second physical address is set to zero. The sending node LANai prepares each header using the send proxy address passed by the user with the request. With this address, the LANai indexes the outgoing page table to determine the destination node and physical addresses. This information must have been set up on import, as described above, for the proxy address to be valid.

The LANai processor maintains in SRAM virtual-to-physical two-way set associative software TLB for each process using the VMMC on a given node. This TLB is rather large - it can keep translations for up to 8 MBytes of address space assuming 4 KByte pages. The LANai processor uses it on long sends to obtain the physical address of the next page to be sent. If the translation is missing, the LANai raises an interrupt and the VMMC driver provides the necessary translation to update the SRAM TLB. On one interrupt, translations for up to 32 pages are inserted into the SRAM TLB. Send pages are locked in memory by the VMMC driver when it provides the translations for the SRAM TLB.

After obtaining the physical address of the next source page, the LANai sends the next chunk of a long message. The chunk size is currently set to the page size (4K bytes), except for the first chunk which is equal to the amount of data needed to reach the first page boundary on the send side. The sending of each chunk of a long message requires two DMA transactions: host to LANai buffer and LANai buffer to network. These transactions are pipelined to improve bandwidth. Additionally, since the network bandwidth is higher than the PCI bus bandwidth, we prepare the header for the next chunk when network DMA of the previous chunk has finished, but while the host DMA of the current chunk may be still in progress.

When the last chunk of a long message is safely stored in the LANai buffer, the LANai reports (using LANai to host DMA) a one word completion status back to user space. This technique allows the user program to spin on a cache location while waiting for message completion, and it avoids consuming memory bus cycles.

On arrival of a message chunk, its data is scattered according to the two physical addresses included in the header. The LANai is able to compute scatter lengths using the total message length and the scatter addresses.

5. Performance

5.1. Implementation Environment

Our implementation and experimentation environment consists of four PCI PCs connected to a Myrinet switch (M2F-SW8) via Myrinet PCI network interfaces (M2F-PCI32). In addition, the PCs are also connected by an Ethernet. Each PC is a Dell Dimension P166 with a 166 Mhz Pentium CPU with 512 KByte L2 cache. The chipset on the motherboard is the Intel 430FX (Triton) chipset. Each PC has 64 MBytes of Extended Data Out (EDO) main memory and 2GB of EIDE disk storage.

Each node in the cluster runs a slightly modified version of the Linux OS version 2.0.0. The modifications are minimal and are needed to extend the device to kernel interface by exporting a few more symbols for loadable modules. Linux provided us with most of the functionality we needed, including calls to lock and unlock pages in physical memory. Loadable kernel modules proved to be a useful and powerful feature of Linux.

The new kernel-level code we needed is implemented in a loadable device driver including a function which translates virtual to physical addresses and code that invokes notifications using signals. We note here that we could have completely avoided any modification of the OS if we had provided a special variant of *malloc* which would allocate send and receive buffers in driver-preallocated memory mapped into user space. This solution, however, has the serious drawback of not supporting direct send and receive from user static data structures (which are not dynamically allocated).

5.2. Hardware Limits

The Myrinet network provides network bandwidth of about 160 MBytes per second. However, the total user-to-user bandwidth is limited by the PCI-to-SRAM DMA bandwidth.

Figure 1 presents the bandwidth achieved by the host-to-LANai DMA engine for various block sizes. The maximum bandwidth of the PCI bus is close to the 128 MBytes/sec which is achieved for 64 KBytes transfer units. However, without hardware support for scatter-gather any communication library which supports virtual memory and direct sending from user-level data structures cannot use transfer units larger than the size of a page (4 KBytes). This is because consecutive pages in virtual memory are usually not consecutive in the physical address space. As a result, user-to-user bandwidth is limited by host-to-LANai bandwidth achievable with a 4 KBytes transfer unit. This limitation is 110 MBytes per second.

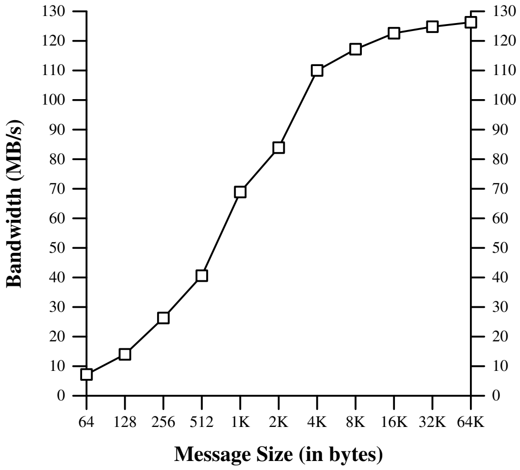

Figure 1. Bandwidth of DMA between the Host and the LANai.

The estimate of minimal latency imposed by the hardware includes the time to post a send request, the time needed for sending a one-word message between two LANais and the time to DMA data into the receiver host memory. We measured the costs of memory-mapped I/O over the PCI bus. For read, this cost is 0.422 μsec, for write it is 0.121 μsec. Posting a send request costs at least 0.5 μsec, if only writes are used (assuming reads are removed from critical path by preallocating items on send queue). Picking up a send request, preparing a network packet, initializing the DMA network transfer and receiving data on the receiving LANai cost about 2.5 μsec. Another 2 μsec are needed on the receive side to arbitrate for the I/O bus, initiate the receive host DMA and put data in the host memory. Adding all these overheads together, the minimum latency is about 5 μsec.

5.3. Micro-benchmarks

The basic operation in the VMMC is *SendMsg*. In an effort to capture the different cases that arise in message passing programs we analyze different forms and aspects of this basic operation. When a virtual-to-physical translation is needed on a send, we make sure that it is present in the LANai software TLB. This eliminates the cost of interrupting the host to ask its kernel for the missing translations. We feel this assumption is reasonable, as the LANai TLB can keep mappings for 8 MBytes of user space.

The first dimension we explore in our experiments specifies whether the send is synchronous or asynchronous. A synchronous send returns only after the data is transferred to the network interface and the send buffer can be safely reused. An asynchronous send returns immediately. If the sender needs to reuse the send buffer, he first has to verify that it is safe to do so. The second dimension specifies nature of the traffic in the network. We investigate three possible cases:

- one-way traffic: one node sends data to a second, idle node.
- bidirectional traffic - each node sends at the same time to each other,
- alternating traffic: nodes send to each other in alternate way.

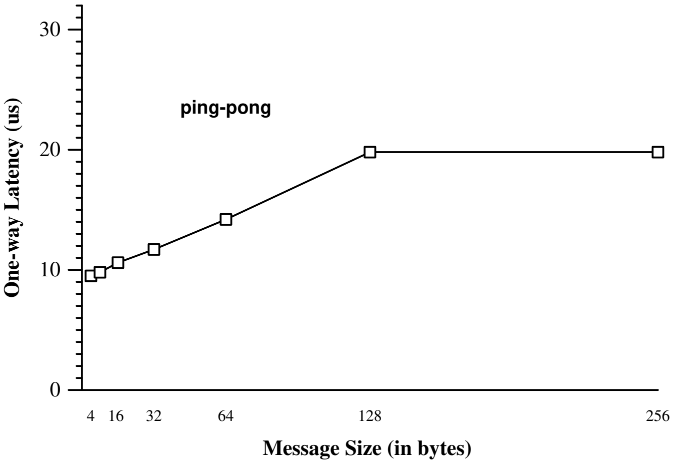

Figure 2. VMMC latency for short messages.

Ping-pong *(synchronous send, alternate traffic)***:** This is the traditional ping-pong benchmark. Figure 2 gives one-way latency for small messages. One word latency is 9.8

μsec. Messages of up to 32 words are copied to the SRAM send queue using memory-mapped I/O and then the LANai copies message data into the network buffer. For longer messages, host-to-LANai DMA copies data from a host send buffer to a network buffer.

Figure 3 shows the bandwidth for varying message sizes. The highest bandwidth provided by VMMC is equal to 108.4 MBytes per second. As explained in section 5.2, the available bandwidth is limited to 110 MBytes per second, with our implementation delivering 98% of this limit. Such good performance results from 1) a tight sending loop, 2) pipelining the host send DMA with the net send DMA and 3) precomputing the headers as explained in section 4.5. However, this loop also needs to be responsive to unexpected, external events, such as the arrival of incoming data packets. In this event, we abandon the tight sending loop and return to the main loop of the VMMC LANai control program.

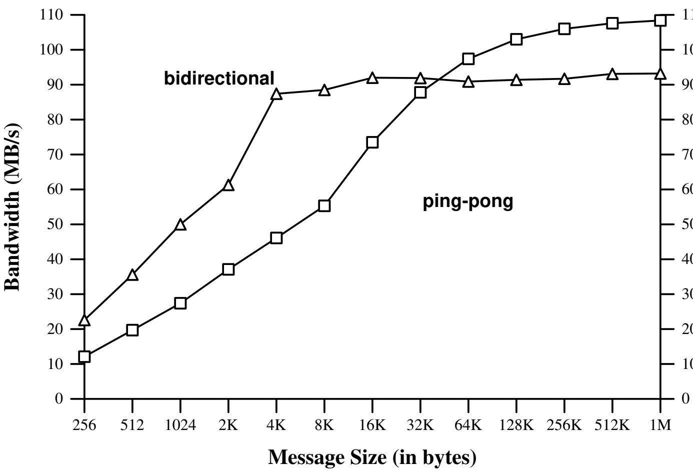

Figure 3. VMMC bandwidth for different message sizes.

Bidirectional traffic: *(synchronous send, bidirectional traffic)***:** This benchmark tries to capture the behavior of VMMC in the presence of simultaneous bidirectional traffic. Both sides start sending at the same time, they wait until they have received the message from the peer process and then go on to the next iteration. Figure 3 presents the bandwidth in this case. Note that this is the total bandwidth of both senders. The maximum available bandwidth, equal to 91 MBytes/sec, is lower than in the previous case. With bidirectional traffic we cannot use the tight sending loop because packets leave and arrive simultaneously. Instead we have to go through the main loop of our software state machine which slightly increases the software overhead, and reduces the bandwidth.

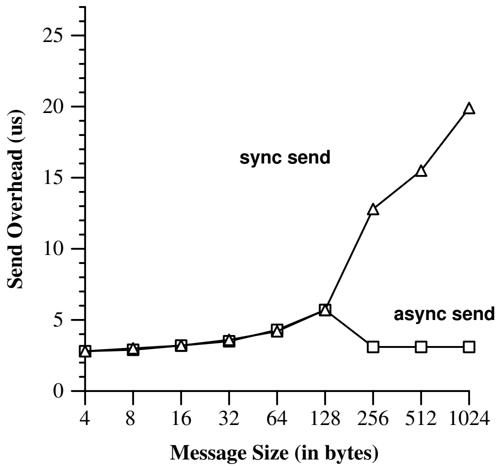

Figure 4. Overhead of the synchronous and asynchronous send operations.

Synchronous Send Overhead *(synchronous send, one-way traffic)***:** Another important aspect of the synchronous send operation is the cost of copying the data to the network interface. In this case we measure the time it takes to transfer the data to the network interface, at which point it is safe to reuse the send buffer. Figure 4 shows that synchronous send overhead is about 3 μsec and grows slowly up to 128 bytes, but for longer messages there is a significant jump in overhead caused by the change of protocols and the use of host-to-LANai DMA on the send side. Synchronous send overhead, not latency is the motivation why the threshold at which send protocol changes from short send to long send is not lower than 128 bytes. Setting this threshold to 64 would dramatically increase synchronous send overhead for messages between 64 and 128 bytes long, although latency would not change much, as shown on Figure 2. On the other hand, we cannot set this threshold higher than 128 bytes because of limited size of LANai SRAM.

Asynchronous Send Overhead: *(asynchronous send, one-way traffic)***:** In many applications, it is useful to be able to post a send request and then proceed with computation. In this case the send buffer cannot be reused, unless the application verifies that it is safe to do so by using another call to the network interface to check the status of the posted request. Figure 4 shows that the asynchronous overhead of a long send is slightly lower than for short sends. This is because the long send request is of fixed size and does not require data copying, whereas on a short send the host has to copy data with memory-mapped I/O to the LANai buffer. We note, however, that asynchronous and synchronous send overheads for short sends are equal, since in both cases the host executes the same code while the synchronous overhead of a long send is much higher than its asynchronous overhead.

5.4. vRPC

In order to evaluate the performance of our implementation with a standard interface we run *vRPC*([2]). This is an RPC library that implements the *SunRPC* standard and uses VMMC as its low-level interface to the network.

Our strategy in implementing *vRPC* was to change as little as possible, and to remain fully compatible with the existing *SunRPC* implementations. In order to meet these goals, we changed only the runtime library; we made minor changes to the stub generator[2] and we did not modify the kernel. The server in *vRPC* can handle clients using either the old (UDP- and TCP-based) or the new (VMMC-based) protocols. Two main techniques are used to speed up the library. First, we re-implemented the network layer directly on top of the new network interface. 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 collapse certain layers into a new single thin layer that provides the same functionality. The low cost of VMMC on fast interfaces like SHRIMP and Myrinet allows a simpler design. Moreover, user to user communication in VMMC exposes the full communication stack to the programmer and allows for customized protocols. A detailed description of *vRPC* can be found in [2].

Although *vRPC* was tuned for the SHRIMP hardware, on the Myrinet network interface it achieves a round trip latency of 66 microseconds (compared to 33 microseconds on SHRIMP). The maximum bandwidth for SHRIMP is 8-12 MBytes/sec, but for Myrinet we achieved 32-33 MBytes/sec. The reduction in bandwidth compared to peak VMMC bandwidth happens because *vRPC* performs one copy on every message receive (for a total of two copies in a round-trip call). We measured the *bcopy* bandwidth in the library code to be in the range of 20–50 MBytes/sec depending on the size of the data copied. The one copy on the receive side is necessary, if compatibility with *SunRPC* is to be maintained. We note here that when compatibility restriction is removed it is possible to implement an RPC interface which has bandwidth close to this delivered by VMMC (see [2] for details).

6. Network Interface Design Tradeoffs

We are interested in investigating how Myrinet and SHRIMP network interface designs impact the performance of VMMC implementation. In our comparison we need to factor in important hardware differences: first of all, SHRIMP interfaces to the EISA bus, whereas Myrinet is a PCI interface. Second, the SHRIMP interface also has a memory bus snooping card, but Myrinet cannot access the memory bus directly[3]. To avoid comparing apples with oranges we take into account only deliberate update and measure the performance of both VMMC implementations against the limits imposed by each system's hardware.

The overall designs of both implementations are similar in many ways. Both systems implement export-import link in the same way (in fact the same daemon code is used in both cases). Also both systems include a destination physical address in the message header. The incoming page table is used in the same way in both implementations. The outgoing page table also exists in both designs - in SHRIMP there is one such table per network interface, in Myrinet we have one per sending process.

The main difference is that the SHRIMP network interface supports initiation of deliberate update directly in hardware. The destination space is a part of the sender's virtual address space and virtual memory mappings are used to verify permissions, to translate virtual to physical addresses of send buffers, and to provide protection between users on initiation of deliberate update. A user process can initiate a deliberate update transfer with just two memory-mapped I/O instructions [4]; these instructions access locations on the network interface board across I/O bus.

Picking up a send request in Myrinet requires scanning send queues of all possible senders, whereas in SHRIMP it is done immediately by the network interface state machine responding to a wide range of memory-mapped addresses. Processing the send request of a message which does not cross page boundaries is also much faster in SHRIMP than in Myrinet, even if we consider deliberate update only. It takes about 2-3 microseconds in SHRIMP to verify permissions, access the outgoing page table, built a packet and start sending data. In Myrinet, even when one sender is active, it takes at least twice as long, because virtual-to-physical translation and header preparation is done by the LANai in software.

Faster send initiation in SHRIMP requires more operating system modifications - special proxy mappings must be maintained by the OS, whereas in Myrinet we only need to be able to translate virtual to physical addresses for pinned pages. Additionally, since the two instructions that initiate a send DMA in SHRIMP are not atomic, the state machine needs to be invalidated on context switch to avoid mixing partial requests issued by two users. In Myrinet, no such invalidation is necessary, as each sender has its own send

[2] We only added support for the new protocol, so that code for it is automatically generated.

[3] As a result SHRIMP supports besides deliberate update another mode of transfer, called automatic update which snoops writes directly from the memory bus and sends it to a destination node

queue.

When a send spans multiple pages, its overhead in Myrinet is not that high, as only one request can be posted for very long sends. However, in SHRIMP we need to issue two memory-mapped instructions for each page. The Myrinet approach requires many more resources on the network interfaces, including SRAM needed to keep the software TLB for each sender.

On receive side, actions of both systems are similar. Both network interfaces DMA message data into pinned receive buffers using addresses taken from the message header.

For a one word message, deliberate update latency in SHRIMP takes about 7 microseconds as compared to 10 microseconds on Myrinet, in spite of the fact that the EISA bus is much slower than the PCI bus. For longer messages, the Myrinet implementation approach of posting the send virtual address once and translating it on the network interface incurs lower host CPU overhead on send.

Bandwidth-wise both implementations of VMMC do a good job, with SHRIMP being slightly better as it delivers user-to-user bandwidth equal to achievable hardware limit (23 MB/s), whereas Myrinet implementation provides bandwidth (108.4 MB/s) equal to about 98% of available bandwidth. The main reason for lost bandwidth is the overhead of running a software state machine (i.e. LANai LCP) compared to a hardware state machine as implemented in SHRIMP.

In conclusion, SHRIMP provides (not surprisingly) a better platform for VMMC implementation at the cost of specialized hardware and more OS modifications. However, the Myrinet platform allows a VMMC implementation that also has impressive performance close to the hardware limits. The cost here is more resources used by the network interface, including the LANai processor and on-board SRAM.

7. Related work

The Myrinet switching fabric has been used by several different research projects to produce high performance message passing interfaces. These include the Myrinet API [1] from Myricom, Active Messages [12] (AM) from the Univ. of California at Berkeley, Fast Messages [10] (FM) from Illinois University and PM [11] from the Real World Computing Partnership.

The Myrinet API supports multi-channel communication, message checksums, dynamic network configuration and scatter/gather operations; however, it does not support flow control or reliable message delivery. On our hardware platform the Myrinet API has a latency of 63 microseconds for a 4 byte packet and a peak ping-pong bandwidth of 30 MBytes per second for an 8 KByte messages.

In Active Messages [6] each communication is formed by a request/reply pair. Request messages include the address of a handler function at the destination node and a fixed size payload that is passed as an argument to the handler. Notification is done using either waiting for response, polling or interrupts. The current implementation of active messages does not support channels or threads. Active Messages does not yet run on our hardware.

The Fast Messages (FM) 2.0 [9] is a user-level communication interface which does not provide protection, i.e. only one user process per node is assumed at any given time. FM design favors low latency, rather than high bandwidth communication. FM 2.0 supports a streaming interface that allows efficient gather/scatter operations. It also supports reliable delivery, polling as a notification mechanism, and thread safe execution. FM 2.0 is similar to AM in that each message is associated with a handler that will process the message on the receive side. On our hardware FM has a latency of 10.7 microseconds for an 8 byte packet and a peak ping-pong bandwidth of 30 MBytes per second for an 8 KByte messages. The low latency is achieved by using a small buffer size (128 bytes) and programmed I/O on the sending side. Using programmed I/O avoids the need for pinning pages on the sender side. On the receiver side, DMA is used to move the message data from the LANai to the receive buffers, which are located in pinned memory. The handlers then copy the data from the receive buffers to the user's data structures. In contrast, VMMC avoids copying on the receiver side by allowing the user to access data directly in the exported (pinned) memory.

PM, like AM and FM, is a user space messaging protocol. It requires gang schedulling to provide protection. PM supports multiple channels, the Modified ACK/NACK flow control, and notification by either polling or interrupts. In PM's model the user first allocates special send buffer space, then copies data into the buffer, and finally, sends the buffer contents to the destination node. The receiver provides a buffer into which the contents of the message are received.

PM runs on hardware similar to ours where it has a latency of 7.2 microsecond for an 8 byte message and 118 MBytes per second peak pipelined bandwidth achieved with a transfer unit size of 8 KBytes. The ping-pong bandwith performance for PM is currently not available. PM can use transfer size bigger than a page size because it sends data only from special pre-allocated send buffers. As a result, a user must often copy data on sender side before transmitting it. The cost of this copy is not included in the peak bandwidth number above and it will reduce available user-to-user bandwidth. Even ignoring the cost of copying, when the transfer unit is limited to the page size (4 KBytes) both PM and VMMC have a pipelined bandwidth close to 110 MBytes per second (see Figure 1). PM achieves slightly lower latency than VMMC because it allows the current sender exclusive access to the network interface. This solution involves saving and restoring of channel state on a context switch, an expensive operation.

There are two important advantages that VMMC has over PM and FM messaging protocols. First, VMMC provides protection between senders on one node, as each sender has its own send queue. This design works well on both uniprocessor and SMP nodes. Other protocols assume only one sender process per node (FM) or require gang schedulling (PM). Second, peak performance of these messaging protocols relies on a receive request being posted before the message arrives at the receiver. If the message arrives *before* a receive is posted it must either be a) buffered or b) dropped. Either alternative significantly reduces bandwidth. With VMMC this performance restriction does not apply because the sender specifies the address at which to store the data.

8. Conclusions

This paper describes the design and implementation of virtual memory-mapped communication (VMMC) on a Myrinet network of PCI-based PCs. 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, zero-copy protocols, and minimizes the software overheads associated with communication.

VMMC has been designed and implemented for the SHRIMP multicomputer where it delivers user-to-user latency and bandwidth close to the limits imposed by the underlying hardware.

With this work we provide a first implementation of VMMC on a commercially available hardware platform. Similarly to the SHRIMP implementation, VMMC on Myrinet delivers user-to-user latency and bandwidth close to hardware limits. For short sends, software overhead is just a couple of microseconds. The maximum user-to-user bandwidth is 108.4 MBytes per second which is about 98% of available bandwidth. As with SHRIMP, we support multiple senders and implement protected, user-level send. Except for making a few static symbols external in Linux source, all system software is implemented at user-level or as a loadable device driver.

With this work we prove that VMMC is a fairly portable communication model, not tied to one particular hardware platform. By comparing Myrinet with SHRIMP we have found, not surprisingly, that the latter provides a better platform for a VMMC implementation at the cost of a specialized network interface and more OS modifications. However, the Myrinet implementation offers good performance and makes it possible to build a high-performance multicomputer out of commodity hardware.

Acknowledgments

We would like to thank Yuqun Chen and Stefanos Damianakis for their help in the development of the SHRIMP driver.

References

[1] The myrinet on-line documentation. http://www.myri.com:80/scs/documentation, 1996.

[2] A. Bilas and E. Felten. Fast rpc on the shrimp virtual memory mapped network interface. *Journal of Parallel and Distributed Computing*, 14:to appear, Feb. 1997.

[3] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. A virtual memory mapped network interface for the shrimp multicomputer. In *Proceedings of the 21st Annual Symposium on Computer Architecture*, pages 142–153, Apr. 1994.

[4] M. A. Blumrich, C. Dubnicki, E. 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*, Feb. 1996.

[5] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, Feb. 1995.

[6] D. Culler, L. Liu, R. P. Martin, and C. Yoshikawa. Logp performance assessment of fast network interfaces. *IEEE Micro*, 1996.

[7] C. Dubnicki, L. Iftode, E. Felten, and K. Li. Software support for virtual memory-mapped communication. In *Proceedings of the 1996 International Parallel Processing Symposium*, 1996.

[8] E. Felten, R. Alpert, A. Bilas, M. Blumrich, D. Clark, S. Damianakis, C. Dubnicki, 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.

[9] S. Pakin, M. Buchanan, M. Lauria, and A. Chien. The fast messages (fm) 2.0 streaming interface. Submitted to Usenix'97, 1996.

[10] S. Pakin, M. Lauria, and A. Chien. High performance messaging on workstations: Illinois fast messages (fm) for myrinet. In *Supercomputing '95*, 1995.

[11] H. Tezuka, A. Hori, and Y. Ishikawa. Pm: a high-performance communication library for multi-user parallel environments. Submitted to Usenix'97, 1996.

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