An Evaluation of the DEC Memory Channel
Case Studies in Reflective Memory and Cooperative Scheduling

Andrew Geweke and Frederick Wong
University of California, Berkeley
{geweke,fredwong}@cs.berkeley.edu

21 May 1999

Abstract

With the rise of clusters as a vehicle for very-high-performance computing, an increasing emphasis is being placed upon the communication interface between each processor and the underlying network. And while many studies have explored the design of both traditional send/receive network interfaces and shared-memory designs, a third alternative — reflective memory — offers a significantly different design space and provides its own, separate benefits and challenges.

In this paper, we investigate Memory Channel, a high-speed reflective-memory network, by implementing MPI above Memory Channel. Our results show that Memory Channel is a powerful substrate for message-passing programs, but that significant obstacles remain to building truly large-scale message-passing software above this network. Finally, we investigate time-sharing of a Memory Channel-based cluster and demonstrate the need for event support at the hardware layer.

1 Introduction

With the advent of fast communication interfaces for commodity workstations [1], powerful massively parallel processors (MPPs) can be quickly and cheaply constructed from commodity hardware. Such networks as Myrinet [4], Gigabit Ethernet, and others adapt easily to the standard PCI peripheral bus and provide low-latency, very-high-bandwidth access to the entire cluster from a single processor, creating a system-area network (SAN). This combination has proved to be extremely powerful, allowing clusters to supplant traditional supercomputers in many areas.

The semantics presented by these network interfaces vary significantly. Most popular commercial network interfaces, to date, provide one of two models to the processor: *shared-memory* or *send-receive*. By imitating a shared-memory multiprocessor (SMP), shared-memory network interfaces provide a well-known programming model with implicit communication to software. Send-receive network interfaces, on the other hand, provide a well-known programming model with *explicit* communication: in many ways, they imitate much-higher-performance versions of typical Ethernet and/or Token Ring LANs.

A third sort of communication interface, however, provides a separate but equally interesting design point. *Reflective-memory*[16,17] interfaces are a sort of hybrid between explicit send-receive interfaces and implicit shared-memory models. Like a shared-memory interface, communication is performed using the processor's native load and store instructions; like a send-receive interface, however, there is no single global view of "one large memory".

We have constructed a traditional message-passing library, MPI[13], on top of Compaq's Memory Channnel[15] reflective-memory network interface. Doing so provides a wealth of information about the usability and performance of reflective-memory interfaces under message-passing workloads.

This paper is organized as follows. Section 2 describes the underlying Memory Channel interface and its characteristics. Section 3 covers our multi-protocol implementation of MPI over shared memory and Memory Channel, and the micro-benchmarks performance. We then compare this data to the performance of MPI over true shared memory and the raw Memory Channel interface. Section 4 explores the implementation issues under a time-sharing workload. Section 5 then considers some of the factors affecting performance under shared and non-shared workloads, and some of the challenges poised by constructing a message-passing interface over a reflective-memory substrate. Finally, section 6 draws some conclusions and suggests possible improvements to hardware and software designs for large-scale reflective-memory designs.

2 Background

The first half of this section describes reflective memory in general and the Memory Channel hardware and software used; the second half then continues on to describe the MPICH software used to implement MPI.

2.1 Reflective Memory and Memory Channel

Reflective memory is a powerful communication abstraction, but one that also differs significantly from most currently widely-used models.

Reflective memory provides a process a write-only memory "window" onto another process's address space. Once the "window" has been set up, all processor writes into that "window" go not to memory, but instead directly into the address space of the destination process. This propagation of writes into another process is the basis of reflective-memory communication.

Reflective memory differs from shared memory in two important ways: first, the "window" is *write-only*, and attempts to read from it produce unpredictable results or a processor trap; second, the "window" is *one-way*: the second process must set up an entirely separate "window" of its own back into the first process if it wishes to return communication.

These distinctions are quite fundamental to reflective memory and are what makes the design space of reflective-memory communication interesting.

Memory Channel is an implementation of reflective memory; it consists of a switched system-area network, composed of a PCI network-interface adapter card, 32-bit-wide parallel copper cables, and (optionally) switches.

The raw hardware has an end-to-end latency claimed to be under five microseconds; indeed, we measured an end-to-end latency of approximately 2.7 microseconds for a four-byte message (the smallest possible). Bandwidth is claimed to be between 35 MByte/s and 70 MByte/s; our experiments produced a maximum bandwidth of approximately 62.5 MByte/s. The hardware has built-in error detection (though not correction); however, as the error rate is extremely low, we could not test the claimed rate of approximately three errors per year of continuous operation.

Our testbed consisted of a pair of Compaq AlphaServer 2100s, each of which contains two 533MHz DECchip Alpha 21164s, 256 MBytes of RAM, and a single Memory Channel PCI interface card. We connected the two machines directly (back-to-back) with a single Memory Channel cable.

The Memory Channel software introduces the concept of "Memory Channel address space". The hardware does not maintain any addressable state itself; instead, this "address space" is an abstraction (in reality simply a layer of indirection) provided to simplify Memory Channel programming.

"Address space" is allocated and named using the `imc_asalloc()` function; this function either creates a new segment of "address space" of the specified length and tags it with the given name, or returns a handle to existing "address space" with the specified name if such space exists. Once a handle is obtained, the `imc_asattach()` function maps the "address space" into the real address space of the calling process. The process must specify at this time whether the address space is mapped for transmission or receipt of data.

Using this concept of "address space", two processes can communicate: if they both obtain handles to the same "Memory Channel address space", and one process maps the region for transmission, and the other for receipt, then writes are automatically propagated from one process to the other across the Memory Channel bus. Indeed, multiple readers and writers can exist for the *same* address space, and writes are propagated properly.

The Memory Channel software will also allow a "loopback" mode, where memory mapped for transmit is also mapped back into the same process's address space for receipt at the same address. This causes writes to the memory to be visible to the calling process; however, because the writes must still be propagated out to the Memory Channel hardware first, there is a delay equal to the hardware latency before writes are "visible".

Also provided are a set of spinlock functions for Memory Channel. However, an uncontested acquisition of a spinlock requires approximately 120 microseconds, or about 40 times the latency of a four-byte message. As such, we avoided use of locks in our software.

Memory Channel software is also provided to manage the memory administratively, provide traditional user/group/other read/write permissions on "address space", and detect errors. We did not require any of these services and do not describe them further here.

Finally, Memory Channel software provides for *coherent* allocation of "address space". Basically, when a region of "address space" is allocated and coherent allocation is requested, every node on the Memory Channel network immediately maps that area into a special daemon process's address space for reception, thus ensuring receipt of all writes to that region. When another process then requests that area, the memory is simply mapped directly into the requesting process's address space; this ensures that, upon allocation, the requesting process has the same view of the contents of that memory as all other nodes. This process provides a straightforward solution to an otherwise-difficult problem.

2.2 MPI and MPICH

MPICH [11] is a free implementation of the MPI standard that is efficient and flexible enough to be the basis for our work. The MPI standard is rich with functionality, and attempting to implement it all directly would have extended our work greatly. However, MPICH introduces the Abstract Device Interface (ADI), a set of six basic communication functions that, when implemented, allow MPICH to use a new communication device. MPICH ships with quite a number of devices, including one for standard shared memory.

We have implemented the ADI functions for a Memory Channel communications substrate, allowing all MPI calls to be used on a Memory Channel-based network. We have also modified the MPICH shared-memory ADI to allow multi-protocol MPI when using SMPs:

3 Message Passing using Shared and Reflective Memory

3.1 Implementation

Because of the unique requirements of reflective memory, our implementation of the MPICH ADI imitates neither a typical ADI for a send-receive network interface nor the standard MPICH shared-memory ADI.

A bove the Memory Channel ADI, we have implemented a multiprotocol layer. This layer is essentially a "switch" that routes incoming MPICH requests to the Memory Channel ADI if communication is to take place between two separate nodes, and to the shared-memory ADI (provided with MPICH) if communication is to take place within a node (between separate processors). Using this method, we can use the increased performance of shared memory when communicating within a physical node, and the Memory Channel interconnect when communicating across nodes.

Every Memory Channel node maintains two regions of Memory Channel address space for *every* other node: one for transmitting to the remote node, and one for receiving from that node. The transmit region on node *m* for transmissions to node *n* and the receive region on node *n* for transmissions from node *m* use the same name; thus, they are connected via the Memory Channel software and can communicate. Each region contains three segments:

- The *control block*, a three-word segment of memory that contains critical data for the region as a whole — specifically, the *descriptor-queue tail* and an area that stores the tokens used to handshake sender and receiver for flow control;
- The *descriptor queue*, an approximately eight-kilobyte (one-page) area of memory that stores descriptions of messages (or message fragments) to be sent and received;
- The *buffer*, a large (megabyte-sized) area of memory that stores the actual data of all messages sent.

The descriptor-queue tail is simply the index of the last free buffer in the descriptor queue. The waiting count and drain count will be described later. Further, each node maintains the following data in standard memory for each remote node:

- A local copy of the current descriptor-queue tail for the region that transmits to the remote node;
- The offset of the first unused byte in the buffer segment of the region that transmits to the remote node;
- An *unexpected queue* that stores descriptors and data of received messages skipped over due to tag/context mismatches;
- An *expected queue* that stores descriptors of receives requested by the process, but not yet fulfilled by incoming messages.

The process to send a message from node *s* to node *r* is thus the following:

- Node *s* stores the data in the message directly into the buffer on the remote node, updating its own offset of the first unused byte in that buffer;
- Node *s* uses its local copy of the descriptor-queue tail to store a descriptor into the next free descriptor in the queue;
- Node *s* increments the descriptor-queue tail on the remote node and updates its own local copy.

And the process for node *r* to receive a message from node *s* is the following:

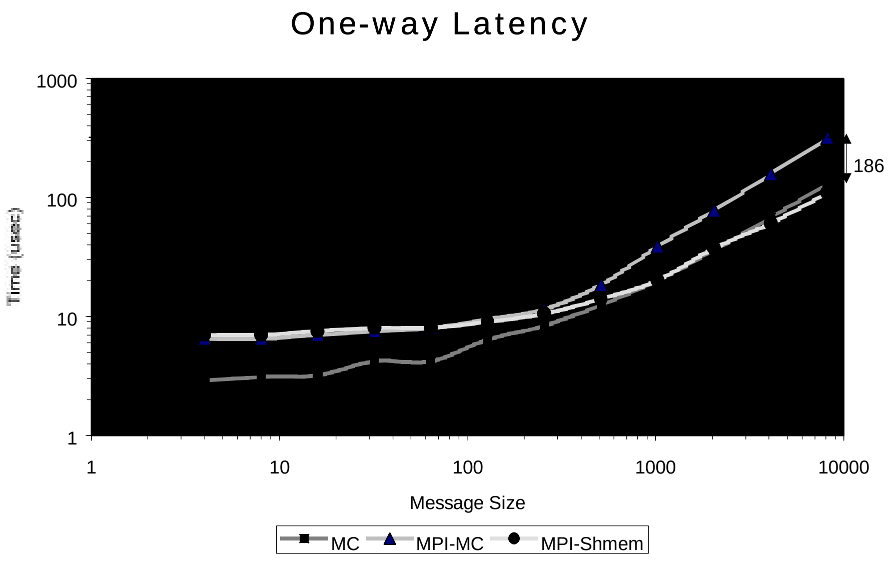
- Examine the first descriptor in the receive region from node *s*;
- If it matches the desired receipt parameters (in this case, tag and MPI context), copy the buffer into the user-data area specified in the receive call.
- Otherwise, examine the expected queue. If the descriptor in the receive descriptor queue matches an entry in the expected queue, the receive is completed by copying the buffer into the user-data area specified in the expected queue.
- If no match is found in the expected queue, copy the descriptor into the unexpected queue, and examine the next descriptor in the queue.
- If the end of the descriptor queue is reached without a match, *r* places the receive descriptor into the expected queue or spin-waits for a new message to arrive, depending

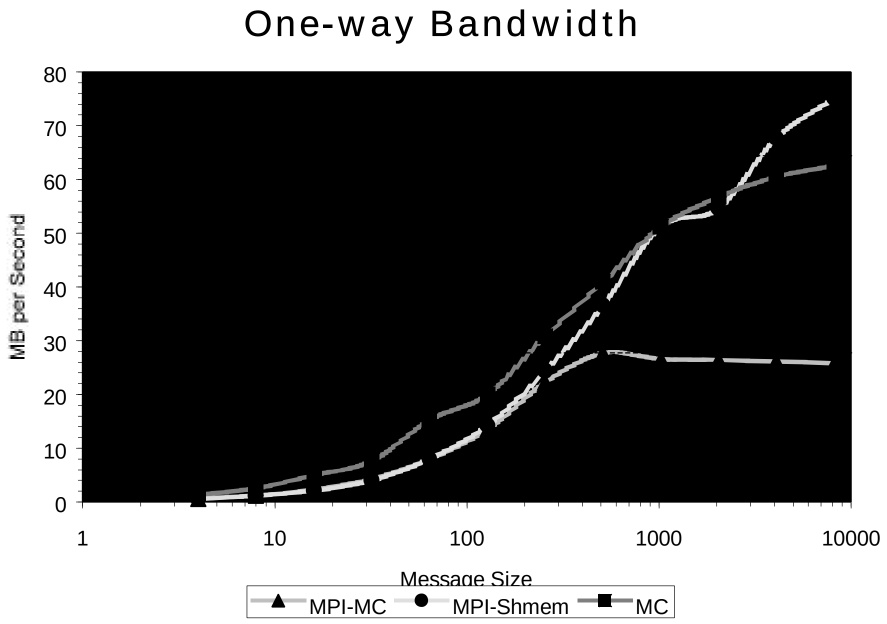
on whether the "post receive" or "complete receive" (used with blocking receives and the nonblocking completion calls) has been called.

There is one final case to consider: if the sender floods the receiver with messages, the descriptor queue (if the messages are small) or the buffer (if the messages are large) will become full. At this point, we drain the buffer as follows:

- The sender indicates to the receiver its desire to have the receiver drain the buffer by writing a special token to the receiver;
- The sender waits for the receiver to drain its buffer; however, while doing this, it also drains its own buffers to avoid deadlock;
- The receiver drains the buffer;
- The receiver acknowledges the buffer drain by sending a token to the sender (in the area normally used to send messages to the sender).

While more-sophisticated (and perhaps higher-performing) buffer-management schemes are certainly possible, we did not want to add the accompanying complexity to our implementation. For example, one can avoid additional memory copies by removing messages directly from the middle of the data buffer; however, this causes internal fragmentation in the buffer and necessitates a much more complex buffer-management scheme.





3.2 Microbenchmarks

Two microbenchmarks — one-way latency and streaming bandwidth — were run against each of raw Memory Channel, our implementation of MPI over Memory Channel, and MPI over true shared memory.

The one-way latency for each system is determined by causing a source node, *s*, to send a message of determined length to a receiving node *r*. Upon receipt of the entire message, *r* immediately issues an identical message in return to *s*. When the entire message has been received at *r*, we record the ending time; one-half of this total round-trip time is then determined to be the one-way latency.

The streaming bandwidth for each system is determined by simply calculating the effective bandwidth (the reciprocal of the latency) achieved in the round-trip-time test.

Memory Channel Performance. Raw Memory Channel hardware has performance very much in line with the claimed hardware specifications. We were able to achieve a one-way latency of approximately 2.9 microseconds consistently when using a four-byte packet. Latencies increased to a maximum of approximately 131 microseconds for an eight-kilobyte (8192-byte) packet.

Bandwidth was measured on a very typical S-shaped increasing curve, starting out at negligible for four-byte messages and increasing rapidly to a maximum of approximately 62.5 MBytes/s at a packet size of eight kilobytes (8192 bytes). Of interest here is that the half-power point — the packet size for which bandwidth first exceeds one-half its maximum achievable value — is a relatively low 256 bytes; this is due to the low latencies incurred and is useful because it indicates that high communication efficiencies are achievable even with fine-grained communication.

MPICH-Shared Memory Performance. The performance of MPI over true shared memory is included with our benchmarks to provide a comparison point for Memory Channel and our MPI over Memory Channel. While not entirely comparable — reflective memory simply does not provide the same semantics as shared memory — this does provide a sense of the performance achievable with MPI without using any sort of traditional interconnect.

One-way latencies of MPI over shared memory achieve a minimum of approximately 6.5 microseconds for a four-byte packet. This latency is relatively high when contrasted with the raw Memory Channel performance, and is due to overheads imposed by MPI. Specifically, a minimum MPI header is typically twenty-eight bytes long; when sending a message of only four bytes, the header comprises 87.5% of the entire transmitted message size. Latencies approach 109 microseconds for an eight-kilobyte (8192-byte) message size.

MPI over shared memory achieves a maximum bandwidth of approximately 71.5 MBytes/s at a packet size of eight kilobytes. The half-power point here is somewhere between 512 and 1024 bytes; the higher overheads imposed by MPI mean that larger messages must be sent before high efficiency is achieved.

MPICH-Memory Channel Performance. Our implementation of MPI over Memory Channel achieves a minimum latency of about 5.5 microseconds, a surprising improvement over even the MPI-Shmem implementation provided with MPICH. At a message size of eight kilobytes, the latency has increased to about 317 microseconds — showing the overhead introduced by our implementation of MPI.

MPI-Memory Channel achieves a maximum bandwidth of about 27.7 MBytes/s with a 512-byte message size; the curve then interestingly dips a bit as the bandwidth decreases to approximately 25 MBytes/s at higher message sizes. The half-power point here is achieved with 128-byte messages, just as with the raw Memory Channel interface.

3.3 Summary

Memory Channel hardware and software itself performs extremely well. The low latencies of 2.9 microseconds for a single-word write (attributable entirely to hardware, as no software is involved) are faster than most, if not all, traditional send-receive interfaces. Bandwidth scales rapidly and well; the maximum achieved of approximately 63 MBytes/s is typical of a device limited by PCI bus bandwidth: although the raw data-transfer rate of the PCI bus is approximately 125 MBytes/s, the overhead imposed by bus arbitration and other devices decreases the achievable maximum transfer rate substantially.

MPICH's shared memory implementation provides us a guideline for the possible achievements of an MPI layer. Its latency of 6.5 microseconds is quite low; however, its latency is nevertheless higher than that of our MPI-Memory Channel implementation. We believe this to be due to the more sophisticated but more scalable algorithms used by MPICH's shared-memory device interface (as compared to our own Memory Channel interface). Bandwidth of the shared-memory implementation is clearly limited by MPI overheads; because at least two copies take place upon each message transmission (to and from the shared-memory region), bandwidth cannot achieve the maximum possible.

Our implementation of MPI over Memory Channel scales extremely well at the low end, achieving latencies approximately equal to (or lower than) those of MPI over shared memory. Again, we believe this is due to our simplistic buffering scheme, which reduces the amount of overhead in the sender. This scheme's scalability, however, is poor, as latencies increase substantially with higher message sizes and bandwidth peaks at about half that of raw Memory Channel.

4 Multiprogramming MPI Over Memory Channel

As clusters become an increasingly popular solution for high-performance computing needs, the need to share them increases dramatically. While traditionally such problems have been solved by allocating small pieces of a supercomputer to individual users, a time-shared approach — similar to that used on a uniprocessor machine with multiple users — offers many attractive benefits. Users get direct, immediate feedback for simple commands; users need not predict and reserve portions of the cluster weeks in advance (which nearly always leads to over-reservation by each user); and, if done efficiently, throughput is decreased only by a very small amount.

Unlike many supercomputers, however, clusters often schedule each processor entirely independently of the others (as each processor is running its own commodity, uniprocessor OS). With many common message-passing applications, this results in extremely poor performance unless some sort of global co-scheduling is implemented [2].

Implicit co-scheduling [3] uses the already-available data of message arrivals to synchronize scheduling of a job across a set of nodes "automatically", providing a simple, efficient, fast global co-scheduling algorithm without sending periodic "clock signals" across the interconnect. This method can provide enormous increases in efficiency under many circumstances [3].

Unfortunately, implicit co-scheduling relies upon the ability of the network interface to provide feedback about messages and data sent or received. This is provided, in the form of interrupts or handlers, on most traditional send-receive network interfaces; however, with a reflective-memory model, all writes are equal as far as the network interface is concerned, and information about messaging events is not directly available: such information is contained only in the user-level MPI library — which does not run unless the program is scheduled!

We considered carefully three separate potential mechanisms for implementing some sort of coscheduling under MPI for Memory Channel. Each seemed to show promise in the beginning, but eventually proved itself either unworkable or simply extremely difficult to implement — and with promise of little performance benefit. Following are the three methods we considered, and the conclusions we can draw from this experience.

4.1 Co-scheduling Mechanisms

Lowering Priority. One option we considered was to simply reduce the priority programmatically of a parallel process that was to begin busy-waiting for an incoming message. Theoretically, all parallel processes that received messages would raise their priorities, thus becoming active on all nodes, exchange messages, ensuring they remained active, and all tend to reduce priority at once as they all block for messages.

We discarded this method, however, as if a sequential job is introduced into the mix, it will tend to either always run or never run, depending on whether its priority is higher than or lower than (respectively) that of the parallel program.

Priority-Modification Daemon. Another considered option was to create a "daemon" process that would watch for inbound messages, and raise/lower the priorities of the affected parallel programs as in the first mechanism above. However, the daemon suffers from the same priority problems with respect to sequential jobs, and so this option was also discarded.

Semaphore Daemon. The basic thrust of this mechanism is to create a daemon process that watches for inbound messages for all MPI processes; when an inbound message is received, it unlocks (and then immediately re-locks) a semaphore that it maintains lock on constantly. When an MPI process needs to wait for an inbound message, it attempts to lock this semaphore (and then immediately unlock it), thus blocking in the kernel until the daemon unlocks the semaphore.

This method shows the most promise of any, but was eventually scrapped due to two major issues.

First, we consider the case where one parallel program is blocked for an inbound message, but another program (parallel or sequential) is continuing processing. If the daemon process's priority is lower than that of the other job, it will effectively never be scheduled, preventing it from noticing incoming messages and thus waking up the sleeping parallel program. If its priority is higher than that of the other job, the daemon process will run constantly, soaking up CPU cycles and preventing the other processes from running.

Second, this method requires deep integration between the daemon and the MPI runtime libraries. Our experience attempting to implement this method, which consumed a great deal of time and energy without producing a truly bug-free implementation, suggests that it may simply be approaching the limit of the complexity we can manage in this unfamiliar arena of reflective memory and a relatively complex MPI device implementation.

4.2 Sumamry

We can only conclude that attempting to introduce co-scheduling behavior into a Memory Channel-based system is inefficient at best and unmanageable at worst. Fundamentally, this is due to the fact that the communication assist provides no feedback with which it can signal the system processor(s). Without such a feedback-based mechanism, the computer's main CPU must be used to poll the network for inbound messages; this polling distorts — and, typically, destroys — the very scheduling mechanism that it is intended to implement.

By contrast, the familiar Active Messages network interface over Myrinet uses the network interface's processor to deliver events directly to the operating system, permitting this sort of scheduling.

5 Observations

5.1 Benchmarks and Quantitative Data

Memory Channel, as a *hardware* solution, is a very-high-performance interconnect. Measured latencies and bandwidths are competitive with the field of cluster interconnects as a whole and tend to be better than those interconnects based on traditional send-receive models.

Further, our demonstration of a MPICH device interface for Memory Channel indicates that, while difficult, it is entirely possible to construct a usable messaging interface over Memory Channel. While performance with large message sizes tended to be quite poor by comparison with the raw Memory Channel hardware interface, these figures could be increased substantially with additional implementation engineering and careful optimization. However, it is worth remarking that such optimization comes with an increasing difficulty in engineering due to the reflective-memory model.

5.2 Implementation Difficulties and Qualitative Observations

A great deal of experience has been gained from the implementation of MPI over a Memory Channel substrate. Here, we attempt to identify the weaknesses and advantages of the reflective-memory model in general, and Memory Channel specifically as we have experienced them.

Advantages of the reflective-memory model and Memory Channel are:

- *Extremely low latency*. Because no explicit send operation need be initiated, we achieve an interesting point where small messages are cheap — whereas usually they are quite expensive. This may permit much finer-grained programming and allows many more detailed interactions in our MPI implementation.

- *Simple semantics*. While the model of write-only memory is difficult to use in practice, it is conceptually simple — indeed, it is arguably simpler than a model such as Active Messages [9] (which nonetheless may prove to be a superior general-purpose cluster interconnection protocol).

Weaknesses of the reflective-memory model and Memory Channel include:

- *Very slow locks*. Because the provided Memory Channel spinlocks are extremely slow — approximately 40 times slower than a simple write — any use of them in common operations will have an enormous impact on performance. A natural design point for reflective memory would ordinarily be locked data structures, similar to those built in standard shared memory; however, due to the lack of locks, these structures cannot be used.

- *Memory consumption*. Because we cannot use locks, we are forced to use individual, point-to-point data structures. Unfortunately, this does not scale well, especially under Memory Channel: each node must allocate a region of Memory Channel memory for *every other node*; given a reasonable buffer size of 1 MByte, this limits scalability. Memory Channel has a limit on total memory used of 512 MBytes; at 1 Mbyte per point-to-point connection, this produces a limit of a 22-node network. Also, as all receive buffer areas must of necessity be pinned into physical RAM on destination nodes, this also limits the amount of buffer space available to each node. Both these factors conspire to put serious doubt upon Memory Channel's scalability.

- *Pinned memory*. Memory Channel also requires that memory mapped for receipt of data must be pinned on the host; this limits the amount of memory that can be mapped within a node, without regard to the total amount mapped in the cluster. For example, in our simple two-node cluster, Memory Channel refused to allocate a total amount of memory greater than 20 MBytes total (the sum of both nodes' allocations). This also can provide serious hindrance to message-passing and other traditional parallel-programming techniques that might otherwise be used with Memory Channel.

- *Lack of events*. Perhaps the most serious problem associated with the use of Memory Channel in a shared environment is its total lack of support for events. Because it is impossible to act upon inbound message events without destroying the efficiency of the main operating system's task of running multiple parallel or sequential tasks, we cannot hope to effectively share a Memory Channel-based cluster among multiple parallel tasks. We believe this will become an increasingly important problem in the future.

6 Conclusions and Future Work

Memory Channel is a high-performing implementation of reflective memory that can be successfully used as a cluster interconnect for message-passing systems. The low latencies provided by Memory Channel can be leveraged for more complex, and thus more robust or highly-performing, communications protocols; further, the low latencies promote fine-grained parallel programming (as they do not penalize programs for sending short messages).

However, several obstacles remain in the path of Memory Channel's possible adoption as a widely-used interconnect. First, the reflective-memory semantics that it provides, while simple in the abstract, create undue headaches for the programmer — very few existing protocols or data structures are designed for "write-only" memory. As a case in point, while MPI has three different possible communications protocols — the "eager" protocol, which pushes data directly to the receiver upon a send; the "rendezvous" protocol, which matches sends and receives in the network; and the "get" protocol, which waits for a receive before moving data across the interconnect — we were only able to use the "eager" protocol, because we can only write, not read, a remote memory. Second, the limitation of total Memory Channel address space to 512 MBytes is a significant limitation for building large clusters, as is the requirement for pinned memory on each host (and its limitations). Finally, the lack of any sort of feedback from the communications interface to the main processor about messaging events is a significant hindrance to coscheduling, which is a virtual prerequisite for time-sharing of large parallel systems [2].

In the future, we would like to examine DEC's MPI implementation over Memory Channel; published results indicate an achieved bandwidth of over 60 MBytes/s.

7 References

[1] T. E. Anderson, D. E. Culler, D. A. Patterson, and the NOW Team. A Case for NOW (Networks of Workstations). *IEEE Micro*, February, 1995.

[2] Remzi H. Arpaci, Andrea C. Dusseau, Amin M. Vahdat, Lok T. Liu, Thomas E. Anderson, David A. Patterson. The Interaction of Parallel and Sequential Workloads on a Network of Workstations. In *Proceedings of ACM SIG-METRICS'95/PERFORMANCE'95 Joint International Conference on Measurement and Modeling of Computer Systems*, page 267-278, May 1995.

[3] Andrea C. Arpaci-Dusseau, David E. Culler, Alan Mainwaring. Scheduling with Implicit Information in Distributed Systems. *Sigmetrics'98 Conference on the Measurement and Modeling of Computer Systems.*

[4] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. K. Su. Myrinet - A Gigabet-per-Second Local-Area Network. *IEEE Micro*, 15(1):29-38, February 1995.

[5] M. Crovella, P. Das, C. Dubnicki, T. LeBlanc, and E. Markatos. Multiprogramming on Multirocessors. In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, pages 590-597, Dec. 1991.

[6] J. Dongarra, and T. Dunnigan. Message Passing Performance of Various Computers. *University of Tennessee Technical Report CS-95-299*, May 1995.

[7] Andrea C. Dusseau, Remzi H. Arpaci, and David E. Culler. Effective Distributed Scheduling of Parallel Workloads. In *Proceedings of 1996 ACM Sigmetrics International conference on Measurement and Modeling of Computer Systems*, 1996.

[8] Andrea C. Dusseau, Remzi H. Arpaci, David E. Culler. Re-examining Scheduling and Communication in Parallel Programs.*University of California, Berkeley, Computer Science Technical Report: UCB//CSD-95-881*, 1995.

[9] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser, ``Active Messages: a Mechanism for Integrated Communication and Computation", In *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992, Gold Coast, Qld., Australia, pp.256-266.

[10] D. G. Feitelson and L. Rudolph. Gang Scheduling Performance Benefits for Fine-Grained Synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306- 318, Dec. 1992.

[11] W. Gropp and E. Lusk and N. Doss and A. Skjellum. A high-performance, portable implementation of the (MPI) message passing interface standard. *Parallel Computing* 22(6):789-828, September 1996.

[12] A. Gupta, A. Tucker, and S. Urushibara. The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications. In *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 120-131, May 1991.

[13] Message Passing Interface Forum. The MPI Message Passing Interface Standard. *Technical Report, University of Tennessee*, Knoxville, April 1994.

[14] J. K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Third International Conference on Distributed Computing Systems*, pages 22-30, May 1982.

[15] R. Gillett, "MEMORY CHANNEL Network for PCI: An Optimized Cluster Interconnect," IEEE Micro (February 1996):12-18.

[16] M. Blumrich et al., "Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer," Proceedings of the Twenty-first Annual International Symposium on Computer Architecture (April 1994): 142-153.

[17] M. Blumrich et al., "Two Virtual Memory Mapped Network Interface Designs," Proceedings of the Hot Interconnects II Symposium, Palo Alto, Calif. (August, 1994): 134-142.