Low Latency Message-Passing for Reflective Memory Networks

Matt Jacunski, Vijay Moorthy, Peter P. Ware, Manoj Pillai,
D. K. Panda, and P. Sadayappan

Department of Computer and Information Science
The Ohio State University, Columbus, OH 43210
Phone: (614) 292-0053, Fax: (614) 292-2911
Email: {jacunski,moorthy,ware,pillai,panda,saday}@cis.ohio-state.edu

Contact Author: Matt Jacunski

Abstract: *In this paper we present an efficient design for message passing over a reflective memory network. First, we consider the attributes of reflective memory communication networks and the requirements to efficiently build message-passing functionality on these networks. We then introduce the Bill-Board Protocol, a lock-free protocol which provides low-latency send, receive, and multicast functionality to higher-level applications over reflective memory networks. The communication protocol and an implementation on SCRAMNet is described in detail. Lastly, the performance of this protocol is demonstrated.*

Keywords: message-passing, reflective memory networks, communication protocols, low-latency communication.

1 Introduction

Clusters of Workstations are becoming increasingly popular as a platform for parallel and distributed computing. For parallel computing, clusters offer a cost effective alternative to conventional parallel computers[1, 3, 13]. For distributed computing they offer good scalability and high availability. Since they use commodity processors and networks, the performance of clusters of workstations is often comparable to that of massively parallel processors (MPP's) but at a much lower cost.

Unlike conventional parallel processors which use customized interconnection networks for communication [6], workstation clusters use commodity networking technologies. In recent years many new high performance networking technologies have emerged such as ATM [2], Fast Ethernet [11] and FiberChannel [14]. These networks provide high bandwidth but have high latency due to network routing and high software overhead. For high performance in parallel processing, however, low latency is very critical. As a result considerable effort has been put into developing low latency networking technologies such as Myrinet [4] and messaging software such as Illinois Fast Messages [12] and U-Net [15].

Reflective memory networks provide a limited amount of non-coherent shared memory across a cluster of workstations. Each workstation is equipped with a network interface card (NIC) which possesses a memory bank. Memory on the NICs is not physically shared but each update to a location on any one of the NICs is transmitted to the other NICs on the network. This causes the memory at each NIC in each workstation to be a reflection of the memory of the other NICs. However, reflective memory networks provide non-coherent shared memory. That is, if multiple nodes make updates to the same shared location, different nodes may see the updates in a different order.

Examples of reflective memory networks include Encore's Reflective Memory Network (later DEC Memory Channel), SCRAMNet, and VMIC (VME Microsystem International Corporation) Fiber-Optic Reflective Memory product.

Reflective memory networks have so far been used almost exclusively for real time and embedded systems where communication follows a producer-consumer pattern. Non-coherent shared memory is ideally suited for such applications as the producer process writes a datum at a (well known) location on the NIC, and the consumer process, typically on a different node, can read the value. Latency is of primary concern as the volume of data is relatively small. An obvious observation is that message passing is inherently a producer-consumer relationship. We aim to use reflective memory to provide low latency messages for small message sizes and use the mirrored aspect for improved multicast messages.

Shared memory and message passing are the two programming models available for parallel programming. Applications which require shared memory tend to use physically shared memory for communication. However, in large systems physically shared memory is very difficult to build. Therefore, larger systems tend to use message passing even though message passing using conventional LAN networking technology is orders of magnitude slower. Distributed Shared Memory (DSM) systems provide an abstraction of shared memory built on top of message passing systems. For a DSM system to work, it has to provide cache coherence but scaling DSM systems while maintaining cache coherence is a difficult problem[8]. Reflective memory networks do not provide complete shared memory functionality or message passing functionality on their own, but such functionality can be built on top of these networks. In fact, reflective memory networks have advantages over conventional interconnects :

- Low latency

  Since communication is effected by writing to a location in memory, protocol overhead is done away with. Consequently, latency for sending messages is very low. This is particularly advantageous for shorter messages.

- Multicast Capability

  The very nature of reflective memory networks lends itself to multicast communication, since a write by one process becomes visible to many other nodes. It has been shown that the performance of other collective operations can be improved by improving multicast performance. Some conventional networks do provide some multicast ability. ATM provides multicast, but it comes with the overhead of establishing a multicast connection. Ethernet provides broadcast, but does not have the flexibility to specify a group of receivers. In reflective memory networks, it is possible to do multicast at a cost just a little higher than that for sending a message to one receiver.

This paper is organized as follows. Section 2 describes the motivation and design goals for developing low latency message passing for reflective memory networks. Section 3 describes the BBP architecture and our initial implementation on SCRAMNet, Section 4 the performance of BBP on the SCRAMNet network, and Section 5 presents our conclusions.

2 Motivation and design goals

The importance of low communication latencies and small-message bandwidth has been demonstrated in [15]. Additionally, for parallel computing, the latency of the multicast operation typically has an impact on application performance.

DEC's Universal Message Passing (UMP) library implementation on MEMORY CHANNEL [10] provides good message passing performance over reflective memory. Their implementation uses a combination of lock-free and lock-controlled buffers for point-to-point messages and only lock-controlled buffers for multicast functionality. The cost of acquiring and releasing an uncontested spin-lock takes approximately 130 and 120 microseconds, respectively [10]. This is a significant cost which would be eliminated if the reliance on spin-locks could be eliminated.

Here, we present a framework for implementing efficient message passing functionality on reflective memory networks. We describe the Billboard Protocol (BBP), and its implementation on SCRAMNet, a reflective memory network. There were several underlying design goals:

- Provide efficient user-level message passing primitives that provide typical send/receive functionality to application programs.

  Much of the overhead in messaging software is copying messages from user space to kernel space and in between layers of the messaging software. The most efficient way of sending messages is to eliminate this copying.

- Allow for multiple processes on a single machine.

  A single workstation may have multiple processes running which need to communicate through the network. The protocol should permit more than one process per node to share the same NIC.

- Provide feasibility for adding write protection.

  Messages are sent on the network by writing to network memory. When there are multiple participating processes, the protocol should implement write protection.

- Provide in-order delivery between any two processes.

  Messages from a process A to process B must be delivered in the same order in which they are sent.

- Minimize accesses to the NIC over the I/O bus.

  The processor in the workstation has to go through the I/O bus to read network memory. Since the network memory can receive updates over the network, and not just the host processor, it cannot be cached. Also I/O buses are slower than system buses so it takes the processor more time to access network memory than the main memory. The I/O bus is normally shared with other devices. Hence it can be a bottleneck, and accesses over the I/O bus must be minimized.

- Implementation of the protocol should not require any locks.

  In shared memory systems, there is substantial overhead involved in waiting for, acquiring and releasing locks. For low latency operation, the protocol should be lock-free.

- Design memory access and use protocol such that applications may benefit from the inherent broadcast/multicast capability of the network.

3 Design and implementation of Bill-Board Protocol (BBP)

In this section we present the design and implementation of the BBP. The general design goal of the BBP was to provide low-latency *send*, *receive*, and *multicast* primitives for reflective memory networks. Throughout the design process, the higher level requirements of MPICH [7] implementation and use of the library as a platform for TreadMarks [9], such as pairwise in-order delivery, were considered.

Although reflective memory hardware may provide a variety of features, the only functionality assumed by the BBP is the ability to map a portion of a process' virtual address space to the reflective memory area.

3.1 Reflective memory organization

The BBP divides the reflective memory into two distinct partitions: the message data partition and the control partition. The message data partition contains only the actual payload of a message while the control partition contains all other information pertaining to a message. Each of these partitions is further divided into NUM_PROC separate areas, one for each process as described below.

Each process is associated with two areas: a message data area and a control area. The message data area contains only the actual payload of a message while the control partition contains all other information pertaining to a message. A key aspect of the reflective memory organization for this protocol is that only a single process is permitted to write to any particular memory location, allowing lock-free operation.

The message data area is managed and written to by only the associated process. A portion of this area is allocated for each message sent (we'll refer to this as a message buffer). Other processes will read message data from this area when appropriate.

Also associated with each process is a portion of the control partition, a control area. A process' control area contains a message descriptor blocks, outgoing message (MSG) flag blocks, and incoming acknowledge (ACK) flags.

There's an array of NUM_BUF message descriptors which are written only by the associated process (NUM_BUF is the maximum number of available message buffers). Each message is associated with a message descriptor by buffer number. A message descriptor contains all information required by a destination process to receive the message. The current implementation's message descriptor includes fields for message length, message data location, message type, and a sequence number.

There are NUM_PROC arrays of MSG flags and NUM_PROC arrays of ACK flags, each associated with a single destination process. Each of these flag arrays contains NUM_BUF bits, one for each message buffer this process may send. The ACK flags are used to acknowledge a message and are written to by only one destination process.

Figure 1 illustrates the layout of reflective memory for the protocol.

3.2 Communication protocol

Use of reflective memory for message passing with the BBP involves allocation and initialization of a structure which provides storage of (a) logical address of this process, (b) pointers to message and control partitions and areas, and (c) communication state. After initialization, messages may be send and received using the functions provided by the API listed in Figure 2.

These are the communication components of sending a message or a multicast:

- Allocate a message buffer (which may require reclaiming previously used message data areas).
- Copy message data from process memory to message data area on reflective memory.
- Assemble and copy message descriptor to appropriate location in control area.
- Toggle state of appropriate MSG flags.

To send a message, a message buffer is allocated from the message data area. This allocation reserves a contiguous block of reflective memory and a logical buffer number. The message contents are then copied to the message data area at the allocated location. Message length, location, type, and a sequence number are written to the message descriptor corresponding to the allocated buffer number. Sequence numbers are generated independently by each process to provide an order to the stream of messages being sent by any particular process. Finally, the MSG flag corresponding to both the appropriate destination address and the allocated buffer number is toggled to indicate that a new message has been sent. The buffer number and message data block will not be reused until acknowledgement by the receiver is confirmed.

Multicast is handled in much the same way as a send. A buffer is allocated, message contents are copied, and the message descriptor is written. For multicast, however, multiple MSG flags are toggled, one for each of the specified destinations. Each of these MSG flags will be associated with the same message buffer but with different processes. For multicast, the buffer number and message data block will not be reused until acknowledgement is confirmed from all destinations.

There are two steps to receiving a message: checking for the presence of a new message and actually receiving the message. These two steps could be combined but are available separately to provide additional flexibility. These are the components of checking for the availability of new messages:

- Check for change of state in pertinent MSG flags.
- Of the MSG flags which have changed state, find the one with the earliest sequence number.

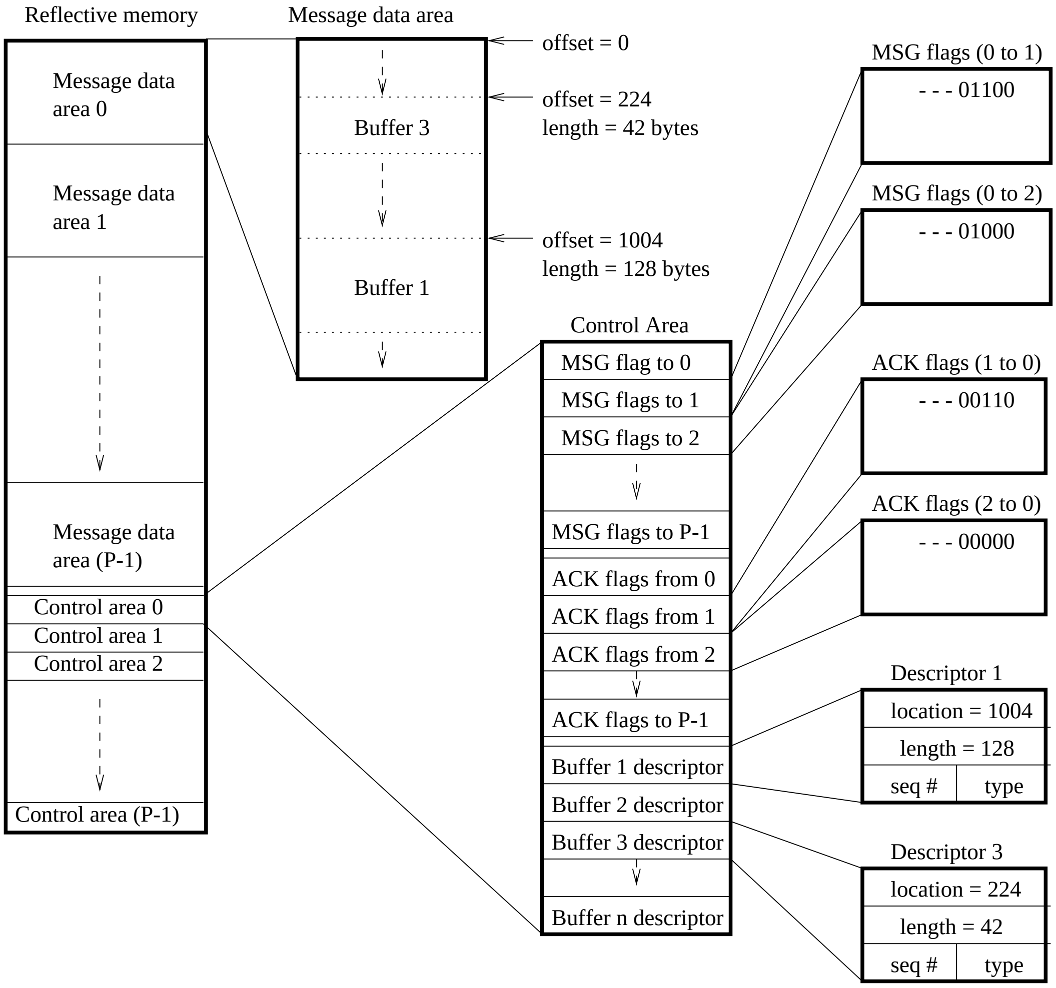

Figure 1: Reflective memory layout for the Bill-Board Protocol for P processes and n buffers. Process 0 has allocated 2 message buffers of length 128 and 42 bytes each. Both of these buffers are associated with outstanding (unacknowledged) messages. Noting that a message is unacknowledged when the MSG flag and ACK flag are in different states, buffer 1 is unacknowledged by process 1 and buffer 3 is unacknowledged by both processes 1 and 2.

These are the components to then receive a message:

- Copy the message data from reflective memory to the specified location.
- Toggle the state of the appropriate ACK flag to acknowledge receipt.

To determine if a new message is available to be received from a specified source, the MSG flag array affiliated with this process is read from the source's control area. All MSG flags which have changed state indicate a message from *source* destined for this process. Based on the sequence numbers read from the message descriptors associated with these toggled flags, the message sent earliest by *source* is selected as the next to be received. To actually receive this message, the specified number of bytes is copied from the *source* message data area to the specified receive buffer (the offset within the *source* message data area is found in the message descriptor). Finally, the appropriate ACK flag is toggled to indicate to the source that this process has received the message.

When either all buffers are allocated or message data memory is exhausted, previously used message buffers must be reclaimed. A message buffer may be reclaimed when all destinations have acknowledged receipt of the message. If the message buffer was allocated to send to a single destination, only a single ACK flag must be checked. However, if the message buffer was allocated for a multicast, state of a set of ACK flags must be verified to insure that all destinations have copied the message to a local receive buffer before reclaiming.

`bbp_init( int num_procs, int rank )`

> Allocates and initializes memory for the BBP state structure for process with logical address ($rank$) of (num_procs). This function returns a pointer to a bbp struct which is passed with all other function calls.

`bbp_Send( struct bbp *this_proc, char *buf, int len, int dest, short msg_type )`

> Sends len bytes located at buf to process with logical address $dest$ based on current bbp state.

`bbp_Mcast( struct bbp *this_proc, char *buf, int len, long *dests, short msg_type )`

> Sends len bytes located at buf to processes indicated by $dests$ based on current bbp state. $dests$ is an array of NUM_PROC bits indicating which logical addresses should be sent.

`bbp_MsgAvail( struct bbp *this_proc, int source )`

> Returns a valid logical address from which a message is available or (-1) if no message is available. If a valid $source$ is supplied, only that source is checked. If (-1) is supplied, multiple sources may be checked in a fair manner.

`bbp_Recv( struct bbp *this_proc, char *buf, int max_len, int source, short msg_type )`

> Copies the next message (up to max_len bytes) sent by $source$ into buffer beginning at buf.

Figure 2: Bill-Board Protocol API.

3.3 Implementation

We have initially implemented the BBP on Systran's SCRAMNet reflective memory product [] with 2MB of memory on each card.

3.3.1 Initialization

Proper operation relies on each process initializing BBP communication with a unique logical address and the same total number of processes. A fixed size control area of 4096 bytes is allocated to each process. The remaining memory is divided into equally sized message data areas, one for each process. Defining NUM_PROCS as the maximum number of processes, NUM_BUFS as the maximum number of logical message buffers for each process, and $DESC\_LEN$ as the length of a message descriptor in bytes, the memory bytes required for each control area, $M_{control}$, is

$$M_{control} = 2\left(\frac{NUM\_PROCS \times NUM\_BUFS}{8}\right) + (NUM\_BUFS \times DESC\_LEN)$$

When initializing for a 64 process system with a maximum of 64 buffers and a message descriptor length of 8 bytes, 1536 bytes of control area are required and the size of the message data area allocated to each process is 28,672 bytes. Pointers to each message data area as well as any flag arrays of interest are calculated and stored in the BBP structure. There is additional state information used to reduce the required traffic over the I/O bus which is also initialized. This state information includes space to buffer portions of reflective memory, message descriptor buffers, structures used to maintain the allocation of message buffers, and space to hold a list of new messages which have been detected but not yet received. Use of this state information to improve efficiency is discussed in detail below.

All outgoing flags are set to 0 at initialization.

3.3.2 Flags

When a message is sent, one or more MSG flags are toggled. If the flag bit is 0, it is set to 1 and vice versa. At the receiver, instead of taking action when a flag that is set, action is taken when a flag changes state. To detect changes in state, a copy of a flag array's last state is stored as the basis for comparison. When changes in state are detected, action is taken for each flag with state change and the new array is stored for future use.

Because all flags are initially in the same state, 0, a message is unacknowledged while corresponding MSG and ACK flags are in different states.

3.3.3 PCI memory access optimizations

Because of the high cost of accessing reflective memory over the PCI bus compared to cached main memory, our implementation attempts to reduce PCI accesses where possible. Four implementation details which reduce either the frequency or length of reflective memory accesses are discussed below.

- Keep cached copy of outgoing MSG and ACK flags. This avoids a read each time a new flag is written. Because only a single process writes any particular array of flags, that process may keep a cached copy of the state of these flags and use the cached copy to perform change of state logic instead of repeatedly reading the flag arrays over the I/O bus.
- Create a message list in sequential order when new message(s) detected. To ensure in-order delivery, the descriptors of all new messages must be accessed to read sequence numbers. To avoid repeated reads of the same descriptors over the I/O bus, all new descriptors are read and inserted into a list ordered by sequence number. If 5 new messages are detected, 5 descriptors are read and inserted into the list. The next 4 calls to check message availability from this source return immediately without accessing the I/O bus.
- Shorten the length of the message descriptor. This is especially significant for keeping short-message latencies low. We have shortened the length of the message data location by encoding the offset as a number of multi-byte blocks rather than in bytes.
- Reduce the number of logical message buffers. The potential for this optimization will vary, depending on application requirements. To provide in-order delivery, all MSG flags must be checked so the earliest message sent may be received next. Therefore, the time required to read these flags increases as the number of buffers increases. Reducing the number of buffers per process reduces the time required to read the flags but also reduces the number of concurrent outstanding messages possible.

3.4 Realization of design goals

Our implementation of the BBP meets the stated design goals:

- We have provided efficient user-level message passing primitives including send, receive, and multicast.

- Multiple processes on a single machine may access the same NIC. In fact, there is no distinction made between processes on the same node and those on other nodes. The only identifier used is the logical process address assigned at initialization time.
- Addition of write protection is feasible with minor modification to the control area layout. Currently the control area consists of outgoing MSG flags, incoming ACK flags, and outgoing descriptors. If this layout was adjusted to include outgoing ACK flags (instead of incoming), the associated process would be the only writer to both the message data area and the control area.
- In-order delivery between any two processes is accomplished through the use of sequence numbers and the protocol requirement that all newly detected messages be considered when selecting the next to be received.
- The implementation details described above seek to minimize the costliest component of communication over the reflective memory systems being considered, data copy over the I/O bus.
- The implementation of the protocol does not require locks for either point-to-point operations or for multicast.
- The protocol makes use of the inherent multicast capability of reflective memory networks by allowing a single copy of message data to be read by multiple receivers.

4 Performance of BBP on SCRAMNet

This section describes SCRAMNet, a reflective memory network, and the performance of the BBP implementation on SCRAMNet.

4.1 Description and characterization of SCRAMNet network

SCRAMNet [5] is one type of reflective memory network. It provides a limited amount of non-coherent shared memory across the cluster. The network consists of a set of workstations with NICs containing memory banks. Writes to any one memory bank are replicated on all the other NICs on the network. Though physically distributed, the memory is shared; writes by any node are visible to all other nodes. However, SCRAMNet shared memory is not coherent; different nodes may not see updates in the same order.

Physically, SCRAMNet is arranged as a ring of up to 256 nodes. Two neighboring nodes can be separated by a maximum of 300m. The maximum separation can be increased up to 3500m using 1300nm laser. The ring is normally made of fiber-optic cable which allows such distances but for shorter distances, coaxial cabling can also be used. A fiber optic bypass switch ensures continuity in the ring when not all nodes are powered on. For systems larger than 256 nodes, a hierarchy of rings is used to accommodate more nodes. SCRAMNet is extremely fast. The data transfer latency between two neighboring nodes is 250-800 nanoseconds, depending on the mode of transmission being used and packet size. SCRAMNet normally uses fixed 4-byte packets for transmission. It provides a maximum throughput of 6.5 MBytes/sec. Variable length packets with lengths varying from 4 bytes to 1 KByte can also be used. In this mode, the maximum throughput is 16.7 MBytes/sec. However data latencies are much higher when variable length packets are used.

SCRAMNet uses Register Insertion for Media Access Control on the ring. At each node, there is a switch which multiplexes incoming messages from the incoming link on the network and the NIC to the outgoing link. If a message arrives on the network, while the node is in the process of injecting a message into the network, then the network message is queued. A message is generated whenever a host writes to a location in SCRAMNet memory. Multiple writes are put in a FIFO queue on the NIC and sent out one by one. Each message traverses the entire ring once and then is removed from the network by the source node.

Communication is done on SCRAMNet by writing and reading to SCRAMNet memory. On average it takes 0.54 μs to write one word to SCRAMNet memory and 1.02 μs to read one word from SCRAMNet memory.

4.2 Performance experiments and results

We have implemented the BBP for SCRAMNet on a cluster of Pentium II SMP boxes running *Linux* version 2.0.30. We have measured the amount of time spent in the various tasks involved in communicating over the network using the BBP. Figures 3, 4 and 5 show these results. Results have been shown for two message lengths 4 bytes, and 128 bytes.

Figure 3 shows the cost of sending a message using the BBP. The bulk of the time is spent in copying the message from user space to SCRAMNet memory. This component of the cost grows linearly with the message length. By avoiding unnecessary copying, the cost of sending has been kept down by keeping this component to a minimum. The software overhead of sending a message is the time spent in buffer allocation, writing the descriptor and notifying the destination process(es) of the message by toggling MSG flags. We show results for both 1 destination and 8 destinations. For the 128-byte case, there is only a 3.25 microsecond increase in the time required to complete the send to 8 destinations compared with the time required for a single destination. The dominating component, message copy, is independent of the number of destinations.

Figure 4 shows the cost of receiving a message using the BBP. Again, we see that bulk of the cost is involved in copying the message from SCRAMNet memory to user memory and since this is done without any intermediate copying, the receive overhead has been minimized. The receive components other than message copy include the time required to (a) read incoming message flags, (b) read the message descriptor, and (c) acknowledge the message once copied.

Figure 5 shows a comparison of the components times separately and in the context of actually exchanging messages. This adds the latency encountered through the interface cards and across the network, call it NIC-to-NIC latency. It can be seen that the protocol costs dominate. As seen previously, these protocol costs are, in turn, dominated by the time required to copy to and from the NICs.

The above results describe the performance of the BBP on an unloaded network (no other messages on the network). Figure 6 shows the performance of BBP with increasing network load. The latencies measured are for point to point communication between two processes. Load is generated on the network by communication between other processes on other nodes. As can be seen from the figure, there is degradation of performance with increasing network load. No degradation of performance was observed when multiple processes sent concurrent messages of the same length. The size of the messages sent to generate additional network traffic which did degrade performance is indicated in the figure.

5 Conclusions

Reflective memory has two principle advantages: low latency and inherent multicast. We've taken advantage of these characteristics to develop the Bill-Board Protocol that provides point to point, in-order message delivery and an efficient multicast delivery. In particular, the multicast latency (on SCRAMNet) is comparable to that of a point to point message which takes 9.8 μ-seconds.

Our protocol also avoids the principle problem with reflective memory systems: the relative high cost of accessing NIC memory over the PCI bus compared to accessing system memory. By allowing only a single process to write to each region of NIC memory we avoid the high cost associated with a locking protocol built on top of NIC memory. We've minimized the number of writes to the reflective memory thereby insuring the best possible performance.

Providing an efficient, low latency multicast layer improves the performance of many collective operations. By offering better performance for collective operations the overall performance of parallel applications on clusters of workstations are improved as well.

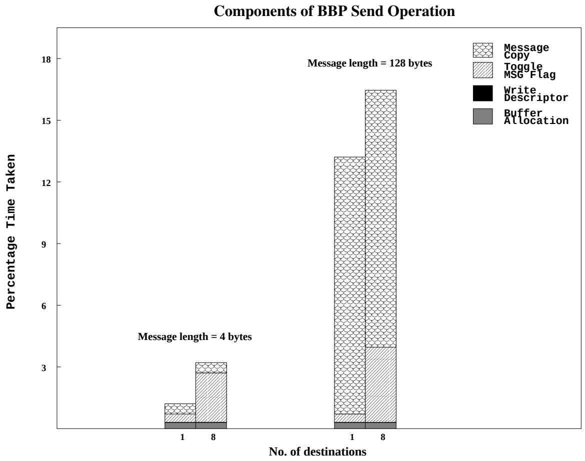

Figure 3: Components of the Send operation in the BBP for 4-byte and 128-byte messages sent to 1 destination using bbp_Send and 8 destinations using bbp_Mcast.

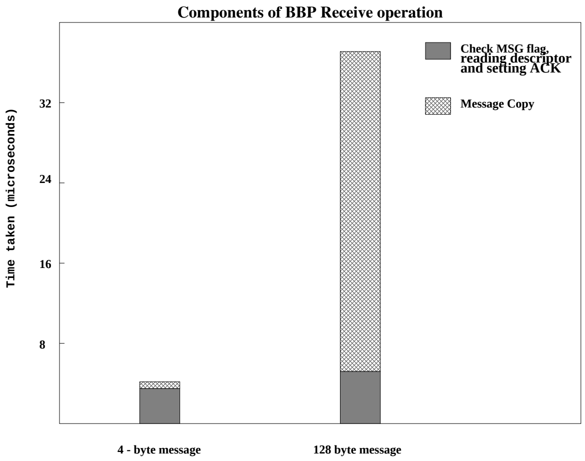

Figure 4: Components of the Receive operation in the BBP for 4-byte and 128-byte messages.

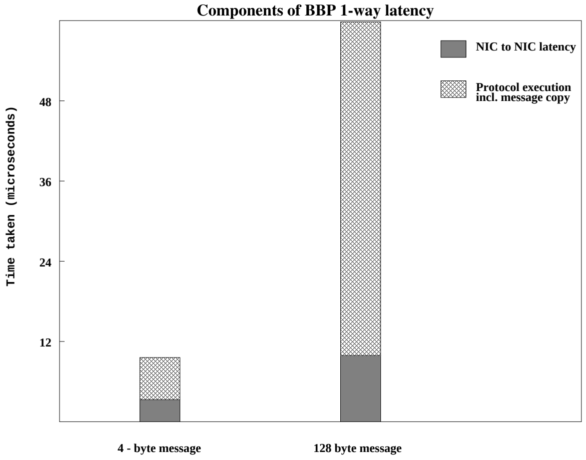

Figure 5: Total one-way message latency including BBP components and NIC-to-NIC latency for 4-byte and 128-byte messages.

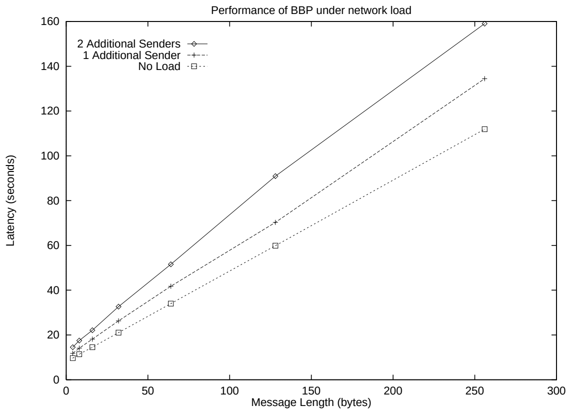

Figure 6: Performance of BBP for 4-byte point-to-point messages with additional network load. Additional network load created by adding 1 or 2 additional pairs of concurrent point-to-point messages with size=768 bytes.

References

[1] T. Anderson, D. Culler, and Dave Patterson. A Case for Networks of Workstations (NOW). *IEEE Micro*, pages 54–64, Feb 1995.

[2] ATM Forum. *ATM User-Network Interface Specification, Version 3.1*, September 1994.

[3] Donald J. Becker et al. Beowulf: A Parallel Workstation for Scientific Computation. In *International Conference on Parallel Processing*, 1995.

[4] N. J. Boden, D. Cohen, et al. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, pages 29–35, Feb 1995.

[5] T. Bohman. Shared-Memory Computing Architectures for Real-Time Simulation-Simplicity and Elegance. Technical report, Systran Corporation, 1994.

[6] J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks: An Engineering Approach*. The IEEE Computer Society Press, 1997.

[7] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI, Message Passing Interface Standard. Technical report, Argonne National Laboratory and Mississippi State University.

[8] C. Holt, J. P. Singh, and J. Hennessy. Application and Architectural Bottlenecks in Large Scale Distributed Shared Memory Machines. In *Proceedings of the International Symposium on Computer Architecture*, pages 134–145, May 1996.

[9] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the 1994 Winter Usenix Conference*, Jan. 1994.

[10] J. V. Lawton, J. J. Brosnan, M. P. Doyle, S. D. O Riordain, and T. G. Reddin. Building a High-Performance Message-passing System for MEMORY CHANNEL Clusters. *DIGITAL Technical Journal*, Oct 1996.

[11] Lee Melatti. Fast Ethernet: 100 Mbit/s Made Easy. *Data Communications*.

[12] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM). In *Proceedings of the Supercomputing*, 1995.

[13] G. F. Pfister. *In Search of Clusters*. Prentice Hall, 1995.

[14] M. W. Sachs and A. Varma. Fibre Channel. *IEEE Communications*, pages 40–49, Aug 1996.

[15] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-level Network Interface for Parallel and Distributed Computing. In *ACM Symposium on Operating Systems Principles*, 1995.