Data Transfers between Processes in an SMP System: Performance Study and Application to MPI

Darius Buntinas Guillaume Mercier William Gropp
Mathematics and Computer Science Division
Argonne National Laboratory
email: {buntinas, mercierg, gropp}@mcs.anl.gov

Abstract— **This paper focuses on the transfer of large data in SMP systems. Achieving good performance for intranode communication is critical for developing an efficient communication system, especially in the context of SMP clusters. We evaluate the performance of five transfer mechanisms: shared-memory buffers, message queues, the Ptrace system call, kernel module-based copy, and a high-speed network. We evaluate each mechanism based on latency, bandwidth, its impact on application cache usage, and its suitability to support MPI two-sided and one-sided messages.**

I. Motivation and Scope

Designing a communication system tailored for a particular architecture requires understanding the achievable performance levels of the underlying hardware and software. Such understanding is key to a more efficient design and better performance for interprocess communication. Interprocess communication usually falls into two main categories: communication between processes within an SMP node, and communication between processes on different nodes. Considerable research has been carried out in the latter case where communication is involved over various high-performance networks. Communicating over shared memory is a field of study that regained popularity with the growing market of SMP clusters.

In this paper, we focus on the shared-memory case and analyze five methods of transferring data between processes on an SMP. We compare their performance based on the usual metrics of latency and throughput. We also consider three other important factors that have been generally overlooked in the past: scalability; the effects of the data transfer operation on processor caches, specifically application data located in the cache; and the setup time required to use the mechanism. We focus on mechanisms available on Intel Xeon-based SMP nodes; however, we believe that similar mechanisms can be used on other architectures with similar results.

The structure of this paper is as follows. In Section II, we describe the data transfer mechanisms that we considered. In Section III we present our performance evaluation of the mechanisms with regard to the different metrics chosen. In Section IV we discuss the suitability of the different mechanisms to support large MPI two-sided messages and one-sided messages. In Section V we conclude this paper and discuss future work.

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-ENG-38.

II. Transfer Techniques Considered

In this paper, we analyze mechanisms for transferring data between processes on an SMP. We consider only mechanisms appropriate for implementing MPI. Such mechanisms cannot have restrictions on where the source or destination buffers can be located in a process's address space. For example, a particularly efficient transfer method implemented at the kernel could manipulate page tables and transfer pages from the page table of the source process to that of the destination process, resulting in a true zero-copy transfer. Such a mechanism, while efficient, would require the source and destination buffers to have at least the same alignment, if not page alignment, and would therefore be beyond the scope of this paper.

We analyze five data transfer mechanisms. These are (1) shared buffers, (2) message queues, (3) the Ptrace system call, (4) kernel-module based copy, and (5) a network interface controller. Below we describe these mechanisms in more detail.

A. Copying through Shared Buffers

The most obvious technique is to have the processes copy through a buffer located in a shared-memory region. First, the processes allocate a shared-memory region between themselves. The mechanism is then straightforward: the sending process copies the data from the source buffer into the shared buffer, and the receiving process copies the data from the shared buffer into its final location in the destination buffer. Synchronization is needed to ensure that one process doesn't read the buffer before the other process has finished writing, and vice versa. We used a flag associated with the shared buffer to indicate whether it was full or empty.

This approach, however, has some limitations in the case of large data. If a single buffer is allocated to contain the entire data to be transferred, transferring large data has a negative impact on available memory. Also, the receiving process must wait until the whole message has been copied into the shared buffer before it can start to copy the data out. These drawbacks can be overcome by using a pair of such smaller buffers and switching between them. While one process is copying out of Buffer 0, the other process is copying into Buffer 1; then they switch. This double-buffered approach can reduce the latency because the receiving process doesn't have to wait for the

sending process to finish copying whole message before it can start copying. Moreover, such an approach can improve the throughput because two processors are transferring the data in parallel. The performance of this method depends on the size of the buffers. If the buffers are too small, the throughput will suffer because the processes have to synchronize more often and memory copy functions are not as efficient for moving small data as they are for moving large data. If the buffers are too large, then one does not get the benefit of double-buffering when transferring small- and medium-sized data. The optimal size of the buffers can be determined empirically.

In order to avoid the cost of setting up the shared-memory region each time large data has to be transferred, such shared buffers can be preallocated between each pair of processes. This approach requires $O(P^2)$ sets of buffers, for P processors. This may be acceptable for a small SMP node, but it does not scale for large SMPs. For large SMPs a method for creating and destroying shared buffers dynamically can be used.

In the rest of this paper we refer to this mechanism as the *shared buffer* mechanism.

B. Copying through Message Queues

The scalability issues raised in the previous section can be avoided by organizing the shared buffers in a more sophisticated fashion. The design we propose is the following. Each process has a pair of queues located in a shared-memory region accessible by all processes. The elements of these queues, called *cells*, are fixed-size buffers. The number of cells in the queues is also fixed and independent of the total number of communicating processes. One of the queues, called the *free queue*, contains unused cells; the other queue, called the *receive queue*, contains cells, enqueued by other processes, that hold the data being transferred. Figure 1 depicts a simple send-receive sequence involving three processes, where Processes 0 and 2 send messages to Process 1.

A send transaction, as illustrated in Figure 1, involves three steps:

1s) The sending process dequeues a cell from its free queue.
2s) The process copies a cell's worth of data from the source buffer into the cell.
3s) The process then enqueues the cell on the receive queue of the receiving process.

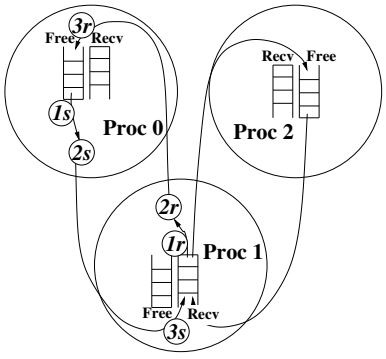


Fig. 1. Sending and receiving with lock-free shared message queues.

A receive transaction also involves three steps:

1r) The receiving process dequeues a cell from its receive queue.
2r) The process copies the data from the cell into the destination buffer.
3r) The process then enqueues the cell on the free queue of the sending process.

The queues are lock-free queues implemented by using atomic operations such as *Compare-and-Swap* and *Swap*, as described in [1]. What makes this method scalable is that only one pair of queues is needed per process, regardless of how many processes it may communicate with. Hence, the queues can be preallocated at initialization time on any size system. Furthermore, the fact that only one memory location has to be polled regardless of the number of processes makes checking for new messages also scalable. In order to receive a message from any process, a process simply has to check whether its receive queue is not empty. This lock-free queue system avoids the use of costly locking mechanisms using semaphores or mutexes. If the data to be transferred is larger than the size of a cell, it is divided into chunks, where each one is transferred in a separate cell. This procedure results in additional overhead because multiple queue operations will be performed to send a long message. The cost, however, is offset by the pipelining effect that comes from dividing up the data.

In the rest of this paper we refer to this mechanism as the *message queue* mechanism.

C. Copying with the Ptrace System Call

The shared buffers and message queue methods require the data to be copied twice: once from the source buffer into the shared buffer or cell, and once from the shared buffer or queue to the destination buffer. In order to eliminate one of those copies, a process would need to be able to directly access the other process's address space. One approach is to use the *ptrace* mechanism, which is designed to support debuggers. The operations supported by the ptrace system call depend on the architecture and operating system, but typically a mechanism is provided to allow the *controlling* process to *attach* to another process and access the memory of that process. On Linux 2.6, after attaching to another process, the controlling process opens the memory file in the `/proc` filesystem, of the other process and uses the `read()` system call to copy data out of the process's memory. Write access through the */proc* filesystem, however, is not supported in Linux 2.6. The steps to transfer data between processes using ptrace are listed below.

1) The destination process takes control of the source process by issuing a call to `ptrace` with the `PTRACE_ATTACH` parameter.
2) The destination process then opens the `/proc/`*pid*`/mem` file corresponding to the source process and reads the data using the `read` system call.
3) The destination process then releases the source process with another call to *ptrace*, with the `PTRACE_DETACH` parameter.

This technique has the advantages that it eliminates a copy and does not require any action from the remote process, making it a *one-sided* copy operation. However, this technique uses a system call that increases the latency of the transfer. Also, the ptrace system call stops the process that is being attached to. Hence, while a data transfer is being performed, the source process is frozen and cannot do any useful work.

In the rest of this paper we refer to this mechanism as the *Ptrace* mechanism.

D. Copying Using a Kernel Module (Kaput)

Features in recent Linux kernels allow a kernel module to map the pages of arbitrary user processes into kernel memory. Thus, a kernel module could directly copy data from one process's address space to another. Several implementations of this method exist. One is LiMIC [2], which is implemented on Linux 2.4. Another is Kaput [3], which is implemented on Linux 2.6. We evaluated Kaput because the source code was available to us, but we expect LiMIC to perform similarly.

In order for a process to access a memory region of another process using Kaput, that memory region must be *registered* with Kaput. By registering the memory region, the Kaput module stores the information about the process's pages that it will need when it maps those pages into kernel space later. The registration operation returns a *token* to the user application, which is used by the remote process to identify the registered memory region.

Once the memory has been registered, a process that has the token can perform a put or get between its local memory and the memory region associated with the token.

Using a kernel module in this way has the same advantages of the Ptrace mechanism, namely, eliminating a memory copy and being a one-sided operation. The kernel module method has the additional advantages of allowing data to be written as well as read and of not requiring that the remote process be frozen or otherwise interrupted during the transfer. This method does, however, still have the overhead of a system call, which the shared buffer and message queue methods avoid.

In the rest of this paper we refer to this mechanism as the *Kaput* mechanism.

E. Copying Using a NIC

The last solution we study is to perform the copy by using a network interface controller (NIC). Most modern user-level network libraries support remote direct memory access (RDMA) operations, such as put or get, which allow one process to transfer data to and from another process's memory. By using a NIC to transfer the data, once the operation has been initiated, the host processor is not involved in the transfer. This method is also one-sided. One large benefit of using the NIC is that the processor's cache is not affected. Normally, when the host processor performs a copy operation, the copy operation replaces whatever was in the processor's cache before the operation. This approach can severely increase the cache misses that the application sees. By using the NIC to transfer the data, rather than the processor, the processor's caches remain intact.

Latency and bandwidth performance can be a drawback to using this method. Because the data is going out over the I/O bus to the NIC and back again (and, depending on the specific network, even possibly going out over the network and back again), the latency of copying data using the NIC may be considerably higher than that of the other methods. Similarly, bandwidth may also suffer because the I/O bus and network are typically slower than the system bus.

These drawbacks may be overcome, however, by the fact that there is no host involvement once the operation has been initiated. The data transfer operation can be scheduled so it can be overlapped with other useful computation, thereby hiding the latency of the operation.

In the rest of this paper we refer to this mechanism as the *NIC-copy* mechanism.

III. Performance Comparisons

In this section, we compare the performance of the data transfer mechanisms described above. We benchmarked several factors: latency and throughput, the cost for setting up and tearing down the system, and the effects on the L2 cache. We feel that these are key characteristics to be taken into account when developing a high-performance communication system. We start by describing our benchmark infrastructure, evaluating different memory copy mechanisms, and determining the optimal size of shared buffers and message queue elements. Then we evaluate each mechanism based on the cited factors.

Our testbed consists of a dual-SMP 2 GHz Xeon node with 4 GB of memory. The Xeon processors have a 512 KB 8-way associative L2 cache with 64-byte cache lines. The OS is Linux 2.6.10. For the NIC-copy mechanism, we used a Myrinet 2000 [4] "PCI64C" NIC connected to a 32-port switch using the GM [5] message-passing system, version 2.0.21. The NIC is installed in a 64-bit 66 MHz PCI slot. In order to measure L2 cache misses, we used the PAPI [6] software library that offers a convenient interface to gather the results. In this paper, we consider one megabyte as 1024×1024 bytes.

A. A Common Benchmarking Infrastructure

To ensure fairness and accuracy in our evaluation of the transfer mechanisms, we developed a common benchmarking infrastructure that allows us to integrate and test the mechanisms in a modular and easy way. The infrastructure is based on a flexible interface so we can implement and evaluate the different transfer mechanisms with one generic test program. This test program has the interface shown below. A module was written for each transfer mechanism that implements each of these functions.

- `init()`: initializes the transfer method
- `finalize()`: finalizes the transfer method
- `register_mem()`: informs the transfer mechanism module about the memory that will be used for the transfer
- `deregister_mem()`: informs the transfer mechanism module that the memory will no longer be used for transfers
- `copy_local()`: performs the local portion of the memory transfer operation on the local process

– `copy_remote()`: performs the remote portion of the memory transfer operation on the remote process. For one-sided transfer mechanisms, such as NIC-transfer, this is an empty function.

The functions `copy_local()` and `copy_remote()` perform the data transfer. For the two-sided transfer methods, shared buffer and message queue, the remote process calls `copy_remote()`, which copies the data from the source buffer into either the shared buffer or queue. The local process calls `copy_local()`, which copies the data from the shared buffer or queue into the destination buffer. For the one-sided transfer methods Ptrace, Kaput, and NIC-copy, the transfer operation is performed only by the local process in `copy_local()`, while the `copy_remote()` function is an empty function. A fast shared-memory barrier is used to synchronize the processes before each iteration to ensure that one process doesn't start the next operation before the other process is finished with the current one. Table I summarizes how these functions are implemented in the benchmark program.

TABLE I
IMPLEMENTATIONS OF `copy_local()` AND `copy_remote()` FOR THE VARIOUS TRANSFER MECHANISMS

	`copy_local()`	`copy_remote()`
Shared buffer	copy out of shared buf	copy into shared buf
Message queue	copy out of queue	copy into queue
Ptrace	attach and `read()`	
Kaput	`kaput_put()` or `_get()`	
NIC-copy	`gm_put()` or `_get()`	

For all the tests, we took an average of 1,000 iterations. One common problem with performing memory transfer tests repeatedly is that the source and destination buffers are loaded into the cache on the first iteration and then all subsequent iterations access the buffers from the cache. This approach skews the results by making the performance seem higher than it should be. To reduce this effect, for each iteration we shift the source and destination buffers by a cache line and reuse a buffer only after we have shifted more than eight times the L2 cache size.

B. Determining the Optimal Memory Copy Routine

In the shared buffer and message queue transfer mechanisms, the data is copied into and out of the shared-memory buffer or queue by using a memory copy operation. The most common implementation of this operation is to use the libc `memcpy()` function. However, this may not be the most efficient method. To find a better implementation, we evaluated libc `memcpy()` and two other memory copy implementations: one implemented by using the IA32 string copy assembly instruction, and one implemented by using MMX nontemporal move instructions. We used the MMX copy implementation from the MP_Lite [7] source code.

Figure 2 shows the results of our evaluation. For smaller messages, up to about 2 KB, the implementation using the assembly string copy instruction (labeled asm copy) performs better than the other two. Beyond 2 KB, the MMX copy implementation performs much better than the others. For our evaluation of the shared buffer and message queue data transfer mechanisms, we used asm copy to copy data up to 2 KB and MMX copy for larger data.

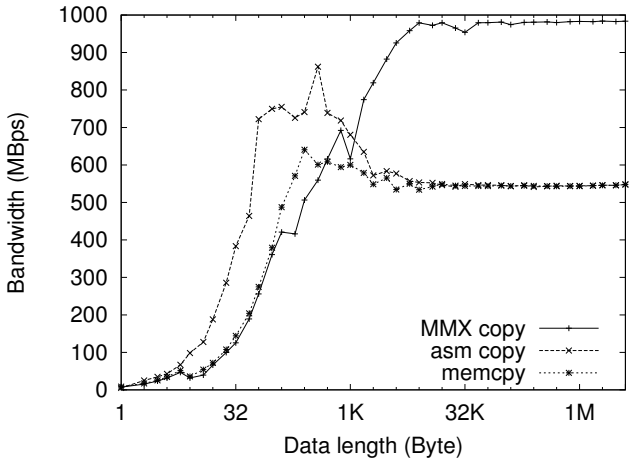


Fig. 2. Bandwidth of three memory copy implementations.

C. Determining the Optimal Shared Buffer and Queue Element Size

To determine the optimal size of the buffer for the shared buffer mechanism and the size of queue elements for the message queue mechanism, we measured the bandwidth of the mechanisms for transferring 4 MB of data while varying the buffer and queue element size. Figure 3 shows that, for the shared buffer mechanism, using an 8 KB buffer gives the highest throughput. That is, at 8 KB, the pipeline between the two processors copying in and out is most efficient. For the message queue mechanism, however, the throughput increases with the queue element size, even up to 1 MB. This result is most likely due to the overhead of the queuing operation: there are fewer queuing operations per transfer operation as

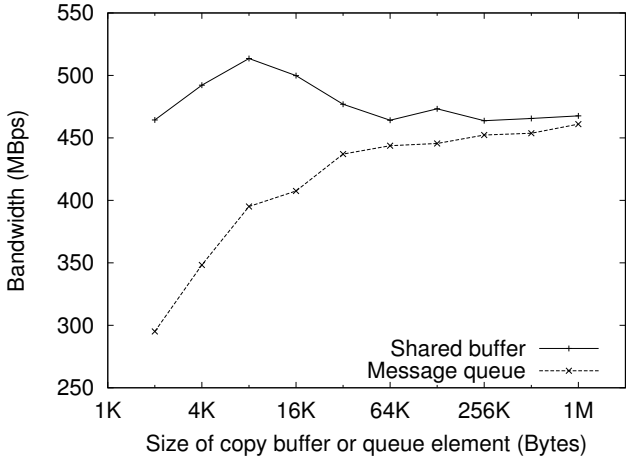


Fig. 3. Effects of copy buffer size and queue element size on the performance of the shared buffer and message queue mechanisms.

the elements get larger, so larger queue elements give better performance.

For the remaining evaluations, we used 8 KB buffers for the shared buffer mechanism and 32 KB queue elements for the message queue mechanism. We chose 32 KB queue elements rather than larger ones because, given the memory needed to implement a queue with a reasonable number of elements, it would be unrealistic to implement a queue with 1 MB or larger elements. Also, 32 KB is around the "knee" of the curve, and so the benefit of using larger queue elements decreases with larger sizes. The optimal values will vary depending on the specific hardware and software being used.

D. Latency and Bandwidth

Using the benchmarking infrastructure described above, we evaluated each of the memory transfer mechanisms. Table II and Figure 4 show the results of these tests for both latency and bandwidth, respectively.

TABLE II
ONE-BYTE LATENCIES FOR THE DATA TRANSFER MECHANISMS

	Latency (μs)
Shared buffer	1.5
Message queue	3.3
Kaput put	2.1
Kaput get	2.1
NIC-copy put	11.6
NIC-copy get	14.3
Ptrace	20.2

We can see that three mechanisms offer low latencies: shared buffer, message queue, and Kaput. Shared buffer is the most efficient, but Kaput also performs well, despite the fact that system calls are necessary to transfer the data. NIC-copy performance is an order of magnitude higher than the previous solutions, but this is a characteristic of the specific hardware we used. The mechanism featuring the highest latency is Ptrace. The penalty comes from the system-call overhead and from the fact that the target process (from which the data is read) needs to be stopped before the transfer can be performed.

Figure 4 shows the bandwidth comparison. We see that the NIC-copy mechanism, whether using get or put, has the lowest throughput. This is because of the low bandwidth of the PCI bus compared with the system bus. The Ptrace mechanism performs better than the NIC-copy mechanism for transfers larger than about 4 KB. Message queues perform better than the previous two, up to about 512 KB; then Ptrace performs slightly better. The best performance is seen by the Kaput and shared buffer mechanisms. The shared buffer mechanism performs slightly better than the kernel copy mechanisms for data larger than about 32 KB.

E. Transfer Setup and Teardown Overhead

Each of these mechanisms requires some setup before the transfer can take place. The results shown above do not include this overhead. Because this overhead can be significant, it would be desirable to perform several transfers per setup. For example, in MPI one-sided operations, the transfer mechanism

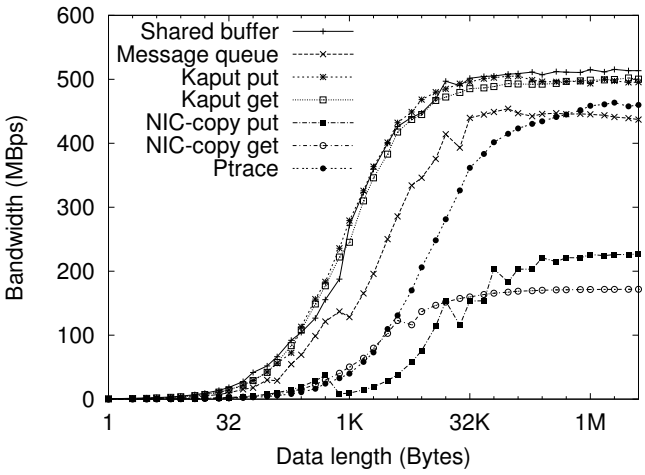


Fig. 4. Comparing the bandwidth of the data transfer mechanisms.

can be set up once when the window is created; then, many one-sided operations can be performed in that window.

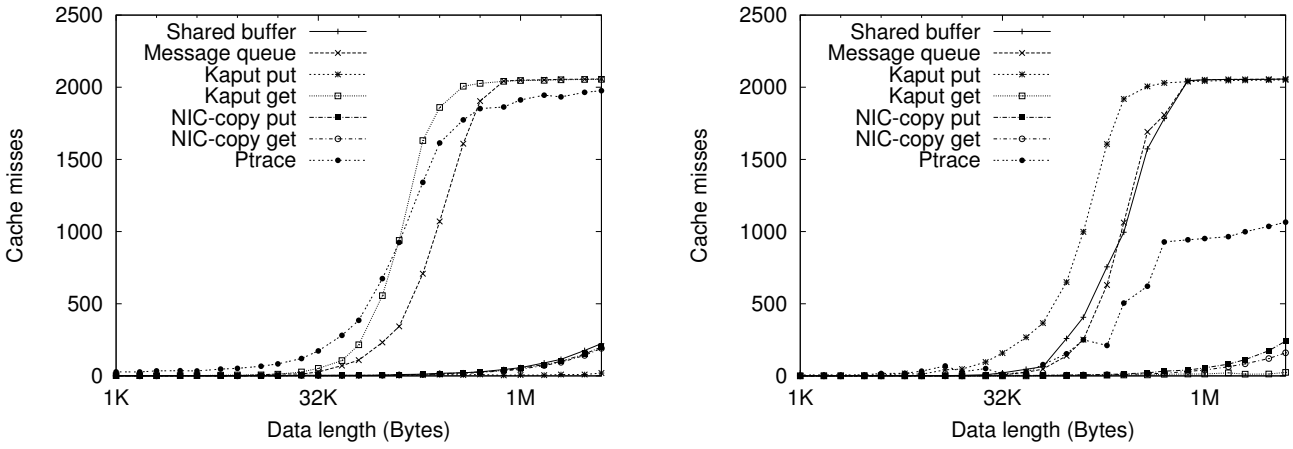
Table III shows the setup and teardown overhead for each mechanism and indicates whether the setup performed is specific to a memory location. When the setup is not specific to a memory region, the setup can be performed once, and data located anywhere in a process's address space can be transferred. If the setup is specific to a memory region, only data located in that region can be transferred. For example, when the shared buffer mechanism is set up, it is not specific to any memory region, while the setup for the NIC-copy and Kaput mechanisms do depend on the memory region because they require the specific memory region to be registered before data can be transferred into or out of it.

TABLE III
TRANSFER SETUP AND TEARDOWN OVERHEAD

	Setup (μs)	Teardown (μs)	Region specific
Shared buffer	10.9	5.1	No
Message queue	10.9	5.1	No
Kaput	2.8	1.4	Yes
NIC-copy	4.5	212.4	Yes
Ptrace	0.0	0.0	No

The overhead for the shared buffer and the message queue mechanisms are the same. A System V shared-memory segment is created and attached to. Then the segment is marked as destroyed, so that once the last process detaches from it, the segment is actually destroyed. Teardown consists of simply detaching from the region. For the Kaput mechanism the setup consists of registering the target buffer, and the teardown consists of deregistering the memory. Similarly, for the NIC-copy mechanism, the memory just needs to be registered and deregistered. For Ptrace there is no setup or teardown overhead.

We can see that the shared buffer and message queue mechanisms have high setup and teardown costs. However, these can be set up once when the communication library is initialized, since the setup is not specific to the source or



(a) Destination process

(b) Source process

Fig. 5. Effects of data transfer operations on application data in L2 cache.

destination buffers of a message. For large shared-memory systems, creating a queue or copy buffer between every pair of processes at initialization time may not be scalable. In this case they may have to be created dynamically. The setup for the Kaput and NIC-copy mechanisms is specific to the source and destination buffers and so must be performed each time a different source and destination buffer is used. The setup and teardown overheads for Kaput depend on the implementation of the underlying mechanism. While the Kaput overheads are quite low, an implementation that does not require per buffer registration and thus has no overhead is conceivable. The high overhead for the NIC-copy teardown shown in the table is a characteristic of the GM memory deregistration operation and may be smaller with other communication libraries.

F. L2 Cache Disturbance

Utilizing the cache effectively is central for achieving good performance for the application. We therefore want to examine the effects the data transfer mechanisms may have on the application's data stored in the cache. To do so, we allocated a buffer to represent the user data, and filled it. Next we performed a data transfer operation of some data outside of the user buffer, and checked how many L2 cache misses were encountered when reading the user buffer. In this test, our user buffer was 256 KB, half the size of the L2 cache on the machines we were using. Figure 5 shows the number of cache misses encountered on the destination and source nodes. Note that for the one-sided mechanisms, the source process is the initiator of the put operation, and the destination process is the initiator of the get operation.

We see in these graphs that for the NIC-copy mechanism, the impact on the cache is low on both the source and destination processes, whereas for the other mechanisms the impact on the cache is high on at least one of the processes. For the shared buffer mechanism the cache impact on the destination process is low, only around 225 cache misses, whereas at the source process, the impact is very high, over 2,050 cache misses. The reason is that the optimized memory copy function uses nontemporal move instructions, which bypass the cache, to write the data to the destination buffer. No nontemporal move instructions are available that read from memory. In the shared-memory mechanism, the source process reads the entire source buffer, thus causing cache lines to be allocated and the application's cache lines to be evicted. On the destination side, the process reads only from two 8 KB copy buffers and writes to the destination buffer. Because of the nontemporal move instructions, cache lines are not allocated when writing to the destination buffers, and so the application's cache lines are preserved.

The message queue mechanism sees a high cache impact on both the source and destination processes because the destination must read from many queue elements. Each time data is read from a new queue element, more cache lines are allocated, and more application cache lines are evicted. The effects of this could be reduced if the same queue elements were reused by the source after being freed by the destination, rather than using a new queue element each time.

For the Kaput mechanism, while there is a large impact on the cache at the initiating process, there is almost no impact on the cache at the target process. The reason is that the Kaput put or get operation is performed only on the processor of the initiator process, so only the cache on that processor is affected. The Ptrace mechanism has a high cache impact on the cache at the initiating process and a moderate impact (just over 1,060 cache misses) on the cache at the target process. It is unclear why the cache at the target process is affected. Table IV summarizes these results.

IV. Suitability for Use in MPI Implementations

In this section, we discuss the suitability of the mechanisms for supporting MPI operations [8], [9]. Specifically, we examine large MPI two-sided messages using a rendezvous protocol and MPI one-sided messages. Since this paper concentrates on transferring large data, we do not examine short MPI two-sided messages. In this analysis we assume that there exists a shared queue between the processes that is used for small messages.

TABLE IV
IMPACT ON APPLICATION CACHE BY THE DATA TRANSFER MECHANISMS AT THE SOURCE AND DESTINATION PROCESSES

	Source	Destination
Shared buffer	High	Low
Message queue	High	High
Kaput put	High	Low
Kaput get	Low	High
NIC-copy put	Low	Low
NIC-copy get	Low	Low
Ptrace	Medium	High

A. Large MPI Two-Sided Communication

Large MPI messages are typically transferred by using a rendezvous protocol, where the sender and receiver first exchange small messages to match the send and receive requests and then transfer the actual data of the message. This approach reduces the amount of data that has to be buffered at the receiver for messages that don't yet have a matching receive request.

Any of the mechanisms could be used to transfer the message data in the rendezvous protocol. However, the advantage to using one of the one-sided mechanisms Kaput, NIC-copy, or Ptrace is that less synchronization is required between the processes, and there can be more overlap of computation and communication. If nonblocking MPI send and receive operations are implemented by using a one-sided transfer mechanism, then once the send and receive requests have been matched, one side initiates the transfer, and there is no need to synchronize with the other process except to notify it when the transfer has completed. Furthermore, the operation can be overlapped with computation on the remote side when using the kernel copy mechanism and on both sides when using the NIC-copy mechanisms. This approach leads to better CPU utilization. Among these methods the Kaput mechanism provides the highest throughput, but it does have a large impact on the cache at the initiating process. If cache impact is more of a concern, or if the transfer can be scheduled in such a way to hide the latency of the operation, the NIC-copy mechanism would be the best choice.

One factor that needs to be considered for the one-sided mechanisms is the setup and teardown times. The setup for the Kaput and NIC-copy mechanisms is memory region specific and so requires that the source and destination buffers be registered before the data can be transferred. This process adds to the transfer time 2.8 μs for Kaput and 4.5 μs for NIC-copy. Once the transfer has completed, the buffers can be deregistered. This is only 1.4 μs for Kaput but is 212.4 μs for the NIC-copy mechanism. In order to reduce the effects of this overhead, deregistration can be deferred until the amount of registered memory exceeds a threshold, at which time all of the unused buffers can be deregistered at once. In addition, a registration cache can be used along with delayed deregistration. Before registering a page, the process checks the registration cache to see whether the page has already been registered.

The shared buffer and message queue mechanisms can also be used even with nonblocking MPI sends and receives because each process will eventually have to wait for the operation to complete. At that time both processes are available to perform the transfer operation. If the SMP node has a small number of processors, a copy buffer can be set up at initialization time between each pair of processes. The shared buffer can then be used any time large data needs to be transferred. In this case, this mechanism would give the best throughput and the least cache impact of all of the methods except for NIC-copy.

B. MPI One-Sided Communication

In contrast to MPI two-sided messages where the sender specifies the source buffer and the receiver specifies the destination buffer, in an MPI one-sided operation one process specifies both the source and destination buffers. Before performing one-sided operations, each target process defines a *local window* describing the memory region on which one-sided operations can be performed. One-sided operations are initiated by a process during its *access epoch*. Operations initiated during an epoch are not guaranteed to complete, either on the initiator or at the target, until after the epoch. Specifically, the results of a put operation are not necessarily visible at the target process, and the results of a get operation are not necessarily visible at the initiating process, until after the epoch. MPI provides two modes, active and passive, that define how access epochs are started and completed.

In active mode, both the initiating process and target process must call specific MPI functions to start and complete an epoch. Note that the MPI standard does not require that any one-sided operations complete until after the epoch ends. Hence, the implementation can delay the transfer of the data until the target process calls the function to end the epoch. At that time both sides can be actively involved in the transfer of the data.

In passive mode, however, only the initiating process needs to make calls to start and complete an epoch. The target process is not required to make any MPI function calls in order to start or end an epoch. Hence, the implementation cannot depend on the target process to be involved in the transfer. However, the MPI standard does allow an implementation to require that the memory used for passive more one-sided operations be allocated using a special allocation function `MPI_Alloc_Mem()`.

Because in active mode the implementation can count on both the initiator and target processes to be involved in the actual transfer of the data, the situation is essentially the same as the two-sided rendezvous case, except that the buffers need be registered only once when the window for the one-sided operations is opened, and deregistered only when the window is closed. This approach allows the cost of the registration and deregistration to be amortized over many one-sided operations.

In passive mode we cannot expect the target node to participate in the transfer unless a separate thread or interrupt context is used at the target process. If a separate thread or interrupt is used, both the initiator and target processes can

TABLE V
SUMMARY OF CHARACTERISTICS OF EACH DATA TRANSFER MECHANISM

	Latency (μs)	Throughput (MBps)	Setup (μs)	Teardown (μs)	Cache Impact (source, dest.)	MPI Two-Sided Can Overlap Computation	MPI Two-Sided Per Msg. Overhead	MPI One-Sided Active Mode	MPI One-Sided Passive Mode
Shared buffer	1.5	513.5	10.9	5.1	(High, Low)	No	No	Yes	with thread
Message queue	3.3	437.1	10.9	5.1	(High, High)	No	No	Yes	with thread
Kaput put	2.1	495.5	2.8	1.4	(High, Low)	Yes	Yes	Yes	Yes
Kaput get	2.1	500.3	2.8	1.4	(Low, High)	Yes	Yes	Yes	Yes
NIC-copy put	11.6	227.1	4.5	212.4	(Low, Low)	Yes	Yes	Yes	Yes
NIC-copy get	14.3	171.7	4.5	212.4	(Low, Low)	Yes	Yes	Yes	Yes
Ptrace	20.2	460.0	0.0	0.0	(Medium, High)	No	No	Yes	with thread

participate in the transfer so the situation is similar to the active mode case. However, using interrupts to initiate data transfers incurs high latency for an OS context switch, and a separate thread polling for incoming one-sided messages wastes CPU time, making these not attractive options for implementing MPI one-sided operations.

If a thread or interrupt context is not used, only the one-sided transfer methods Kaput and NIC-copy can be used to transfer the data. Without a separate thread or interrupt context, these mechanisms are the only way to perform MPI one-sided operations, so the mechanisms must be used to transfer small messages as well as large messages. NIC-copy has high small-message latency, over 10 μs, making it not ideal for small message transfers. The Kaput mechanism, which has a small message latency of around 2 μs, would be preferable in this case. Note that because Ptrace can transfer data only from the target to the initiating process, it cannot be used to support active mode without a separate thread or interrupt context.

An efficient method for implementing passive mode, which may not be appropriate for all applications, is to make the memory allocated by `MPI_Alloc_Mem()` sharable, by creating a shared-memory region or mapping a local file. Then when one-sided operations are to be used, this memory created at the target process can be attached to or mapped into the initiator's address space. The initiator of the one-sided operations can then directly access the target process's memory using loads, stores, or optimized memory copy functions. The overhead of making the memory sharable is relatively high, 10.9 μs to allocate the memory and 5.1 μs to free it, and this overhead must be incurred for every window allocated. Furthermore, some applications may use `MPI_Alloc_Mem()` as a general memory allocation function, not just to allocate memory to be used for windows. It would be undesirable to incur this overhead each time memory is allocated. But if an application uses `MPI_Alloc_Mem()` only for memory used for windows, and the number of allocations and deallocations is low compared to the number of one-sided operations, the overhead can be amortized.

V. DISCUSSION AND FUTURE WORK

In this paper, we have described five mechanisms for transferring data between processes in an SMP machine and evaluated them based on bandwidth, latency, setup costs, and their impact on the application's cache. Table V summarizes the results of our evaluation.

We note that not all mechanisms may be available in all environments. For instance, users typically would not be able to load a kernel module for the Kaput mechanism, and the machine may not have a high-performance user-level network. The Ptrace mechanism should be available on IA32 machines with a recent version of Linux; however, it's not clear whether this feature of the ptrace system call will be supported in the future. The shared buffer and message queue mechanisms should be available on any machine, and these mechanisms give relatively good performance.

The NIC-copy mechanism we analyzed in this paper was performed with a NIC that is a few years old. Faster NICs and communication subsystems, such as Myricom's MX [10], can provide up to 495 MBps bandwidth and latency down to 2.6 μs [11]. Such networks would make the NIC-copy mechanism perform just as well as the shared buffer and Kaput mechanisms with the added benefits of a one-sided mechanism and low cache impact.

Future work in this area would be to expand the study to other workstation architectures, such as X86_64, Sparcs, or G5s, as well as to large shared-memory machines.

REFERENCES

[1] D. Buntinas, G. Mercier, and W. Gropp, "The design and evaluation of Nemesis, a scalable low-latency message-passing communication subsystem," in *Proceedings of International Symposium on Cluster Computing and the Grid 2006 (CCGRID '06)*, May 2006.

[2] H.-W. Jin, S. Sur, L. Chai, and D. K. Panda, "LiMIC: Support for high-performance MPI intra-node communication on Linux cluster," in *2005 International Conference on Parallel Processing (ICPP'05)*, 2005, pp. 184–191.

[3] P. Carns, "Kaput," July 2004, a kernel module for copying data between process.

[4] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. Seizovic, and W. Su, "Myrinet - A gigabit per second local area network," in *IEEE Micro*, February 1995, pp. 29–36.

[5] Myricom, "Myricom GM Myrinet software and documentation," http://www.myri.com/scs/GM/doc/gm_toc.html, 2000.

[6] S. Browne, C. Deane, G. Ho, and P. Mucci, "PAPI: A portable interface to hardware performance counters," in *Proceedings of Department of Defense HPCMP Users Group Conference, Monterey, California*, 1999.

[7] D. Turner, "MP_Lite: A lightweight message-passing library," http://www.scl.ameslab.gov/Projects/MP_Lite/.

[8] *MPI: A Message-Passing Interface Standard*, Message Passing Interface Forum, March 1994.

[9] Message Passing Interface Forum, "MPI-2: Extensions to the message-passing interface," http://www.mpi-forum.org/docs/mpi-20.ps, July 1997.

[10] "MX: Myrinet Express," http://www.myri.com/scs/download-mx.html.

[11] "Myrinet performance measurements," http://www.myri.com/myrinet/performance/.