Data Transfers Between Processes in a 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 the transfer of large data in SMP systems. Achieving good performance for intra-node communication is very important when one aims to develop an efficient communication system, especially in the context of SMP clusters. We present and evaluate the performance of five different transfer mechanisms: copying through shared-memory buffers, using message-queues, using the Ptrace system call, kernel module-based copy and using 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.

1 Motivation and Scope

Designing an efficient communication system tailored for a particular architecture requires its developers to comprehensively understand the achievable performance levels of the underlying hardware and software. This 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. A lot of 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 different methods of transferring data between processes on an SMP. We describe five data transfer mechanisms, and compare their performance based on the usual metrics of latency and throughput but we also consider other factors

*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.

such as scalability the effects of the data transfer operation on the application's data located in the cache, as well as the setup time required to use the mechanism. We think that these are important aspects that have been generally overlooked in the past. We narrowed our scope to consider 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 2, we will describe the various data transfer mechanisms that we took under consideration. In Section 3 we will present our performance evaluation of the mechanisms with regard with the different metrics chosen. In Section 4 will discuss the suitability of the different mechanisms to support large MPI two-sided messages, and MPI one-sided messages. In Section 5 will conclude this paper and discuss future work.

2 Transfer Techniques Considered

In this paper, we analyze five mechanisms for transferring data between processes on an SMP. These are 1) copying through shared buffers, 2) copying through message queues, 3) using the Ptrace system call, 4) using a kernel module to perform the copy and 5) using a network interface controller (NIC) to transfer the data. Below we describe these mechanisms in more detail.

2.1 Copying Through Shared Buffers

The first and 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 them. 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 is full or empty.

This approach, however, has some limitations in case of large data. If a single buffer is allocated to contain the entire data to be transferred, transferring large data would have a negative impact on available memory. Also, the receiving process would have to 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 switch 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 since the receiving process starts to get data before the whole message is copied, and can improve the throughput because two processors are transferring data at the same time. The performance of this method will depend on the size of the buffers. If the buffers are too small, the throughput will suffer because the memory copy functions are not as efficient for moving small data as they are for large data. If the buffers are too large, then you don't 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 would require $\mathrm{O}(P^2)$ sets of buffers, for P processors. This would be acceptable for a small SMP node, but it does not scale for large SMPs.

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

2.2 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 possesses a pair of queues that are shared with other 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 contains unused cells, called the *free queue*, whereas the other queue, called the *receive queue*, contains cells that hold the data that's being transferred. Fig. 1 depicts a simple send-receive sequence involving three processes where processes 0 and 2 send messages to process 1. A send transaction involves three steps:

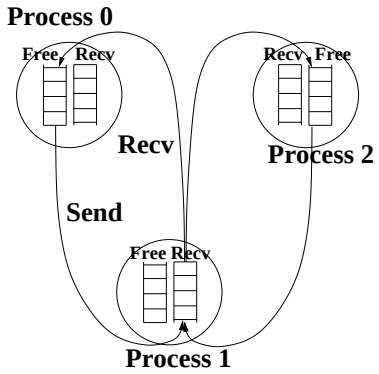


Figure 1: Sending and receiving with lock-free shared message queues

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

A receive transaction also involves three steps:

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

The queues are lock-free queues implemented using atomic operations such as *Compare-and-Swap* and *Swap*. The queue implementation is based on the lock-free MCS lock described in [8]. What makes this method scalable is that only one pair queues is needed per process, regardless of how many processes it may communicate with. This means 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, a process simply has to check whether its receive queue is not empty. It is to be noted that this mechanism can also be very useful to set-up multi-protocol systems. This lock-free queue system avoids the use of more costly locking mechanisms, such as semaphores or mutexes. It also enforces some *minimal* flow control since a process can only send data when free cells are available, meaning that other processes actually receive data. 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 induces extra handling since the dequeuing enqueuing operations will be performed for each chunk. This cost, however, is offset by the pipelining effect that comes from dividing up the data.

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

2.3 Copying with the Ptrace System Call

In order to avoid the additional copy associated with using shared-buffers or message-queues, a process would need to be able to directly access another process's address space. One mechanism to do this is to use the *ptrace* mechanism, which is designed to support the implementation of debuggers. The features supported by the ptrace system call depend on the architecture and operating system, but typically some mechanism is provided to allow the *controlling* process to access the memory of the process it has attached to. On Linux 2.6, the controlling process can open the memory file in the `/proc` filesystem, of the process it has attached to and use the `read()` system call to copy data out of the process's memory. However, write access through the */proc* filesystem is not supported. The steps to perform a copy 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 avoids an intermediate 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 which increases the latency of the transfer. Also, the ptrace system call stops the process that is being attached to. This means that 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 will refer to this mechanism as the *Ptrace* mechanism.

2.4 Copying Using a Kernel Module (Kaput)

Features in recent Linux kernels allow a kernel module to map into kernel memory the pages of user processes other than the current process. This means that a kernel module could directly copy data from one process's address space to another. Several implementations of this method currently exist. One is LiMIC [7] which is implemented on Linux 2.4. Another is called Kaput [5] 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 advantages of eliminating an intermediate memory copy and being a one-sided operation, just as the ptrace mechanism does. However the kernel module method has the additional advantages of allowing data to be written as well as read, and not requiring that the remote process be frozen or otherwise interrupted during the transfer. This method does still have the overhead of a system call which the shared-buffer and message-queue methods do avoid.

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

2.5 Copying Using the NIC

The last solution we study is to perform the copy 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 the 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 will replace whatever was in the processor's cache before the operation. This 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 the other methods. Similarly, bandwidth may also suffer because the I/O bus and network are typically slower than the system bus.

However, these drawbacks may be overcome by the fact that there is no host involvement in the operation once it has been initiated. The data transfer operations 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 will refer to this mechanism as the *NIC-copy* mechanism.

3 Performance Comparisons

In this section, we compare the performance of the data transfer mechanisms we described above. We benchmarked several factors: the usual metrics of latency and throughput, the effects on the L2 cache and the cost for setting up and tearing down the system. 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 latency, bandwidth, overhead to setup and tear-down the transfer, and the effects of the transfer operation on the L2 cache.

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 [3] "PCI64C" NIC connected to a 32-port switch using the GM [11] 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 [4] software library that offers a convenient interface to gather the results. In this paper, we consider one Megabyte (1 MB) as 1024×1024 bytes.

3.1 A Common Benchmarking Infrastructure

In order 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()`: initialize the transfer method
- `finalize()`: finalize 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()` are the functions that perform the data transfer operation. 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()`. 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 1 summarizes how these functions are implemented in the benchmark program.

Table 1: 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 `kaput_get()`	
NIC-Copy	`gm_put()` or `gm_get()`	

For all of the tests, we took an average of 1000 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, then all subsequent iterations are accessing the buffers from the cache. This 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 only reuse a buffer after we have shifted more than eight times the L2 cache size.

3.2 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 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. In order to find a better implementation, we evaluated libc `memcpy()` and two other memory copy implementations: one implemented using the IA32 string copy assembly instruction, and one implemented using MMX memory copy instructions. We used the MMX copy implementation from the MP_Lite [1] source code.

Figure 2 shows the results of our evaluation. We see that for smaller messages, up to about 2 KB, the implementation using the assembly string copy instruction (labeled asm copy) performs better

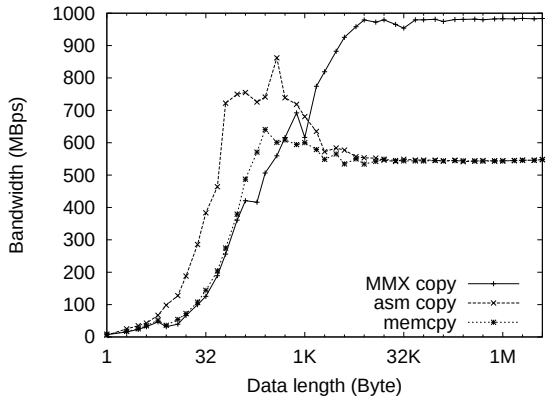


Figure 2: Bandwidth of three memory copy implementations

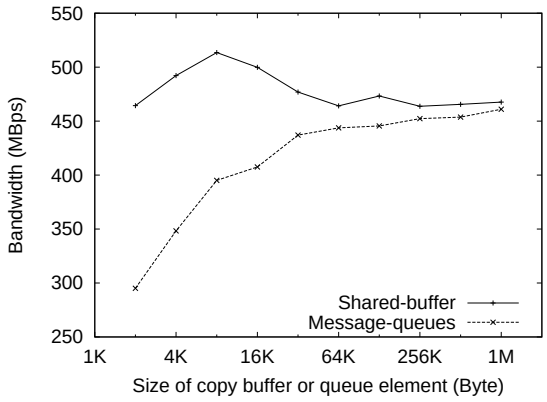


Figure 3: Effects of copy buffer size and queue element size on the performance of the shared-buffer and message-queue mechanisms.

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.

3.3 Determining the Optimal Shared Buffer and Queue Element Size

In order 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. In Fig. 3 we see that for the shared-buffer mechanism using an 8 KB buffer gives the highest throughput. This indicates that 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 is most likely because of the overhead of the queuing operation. There are fewer queuing operations to transfer the data as 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, due to 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.

3.4 Latency and Bandwidth

Using the benchmarking infrastructure described above, we evaluated each of the memory transfer mechanisms. Table 2 and Fig. 4 show the results of these tests for both latency and bandwidth.

Table 2: 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 being the most efficient. We notice that Kaput performs very well given the fact that it relies on system calls. NIC Copy performance is one magnitude higher than the previous solutions but this result is conditioned by the hardware employed. 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) is frozen while the transfer is performed.

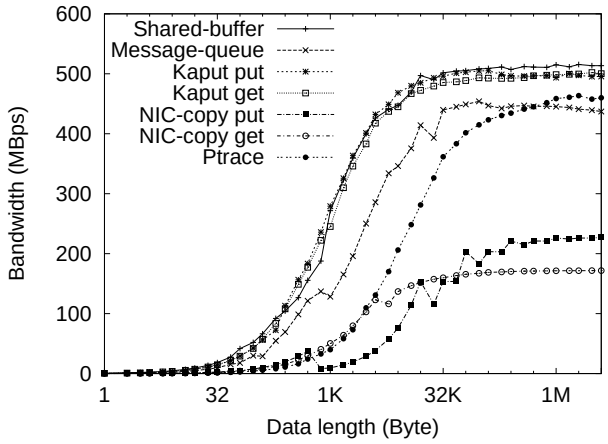


Figure 4: Comparing the performance of the data transfer mechanisms.

As for bandwidth, 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.

3.5 Transfer Setup and Tear-Down 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 can be setup once when the window is created, then many one-sided operations can be performed in that window. Table 3 shows the setup and tear-down overhead for each mechanism. The overhead for the shared-buffer and the message- queue mechanisms are the same: a temporary file is created and is truncated to the desired size by one process, it is then mapped into each of the process's address space, the file is closed, and unlinked. The tear-down consists of simply unmapping the mapped file. For the Kaput mechanism the setup consists of registering the target buffer. The tear-down 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 tear-down overhead.

Table 3: Transfer setup and tear-down overhead

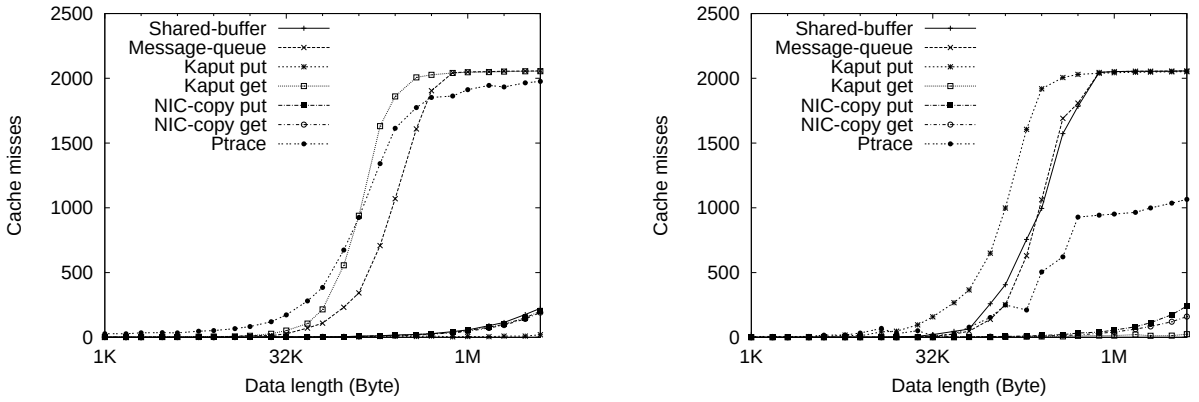
	Setup (μs)	Tear-down (μs)
Shared-buffer	96.3	23.2
Kaput	2.8	1.4
Message-queue	96.3	23.2
NIC-copy	4.5	212.4
Ptrace	0.0	0.0

We can see that the shared-buffer and message-queue mechanisms have high setup and tear-down costs. However, these can be set-up once when the communication library is initialized, since the setup is not specific to the source or destination buffers. For large shared memory systems creating a queue, or copy buffer between each pair of processes may not be scalable, so they may have to be created as needed. The setup for the Kaput and NIC copy mechanisms is specific to the source and destination buffers, and so must be performed for each different source or destination buffer. The high overhead for the NIC copy tear-down is a characteristic of the GM memory deregistration operation, and may be smaller with other communication libraries.

3.6 L2 Cache Disturbance

Utilizing the cache effectively is central for achieving good performance for the application. We therefore want to examine what effects the data transfer mechanisms may have on the application's data stored in the cache. In order to do this, we allocated a buffer to represent the user data, and filled it. Next we performed a data transfer operation of some data outside of this buffer, then checked

how many L2 cache misses were encountered when reading the 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.



(a) Destination process

(b) Source process

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

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 one of the processes. For the shared-buffer mechanism, the cache impact for 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. This is because on the source process the process is reading the entire source buffer which would cause cache lines to be allocated, and therefore the application's cache lines would be evicted. On the destination side, the process is only reading from two 8 KB copy buffers and writing to the destination buffer. 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 a new queue element is read from, 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 re-used by the source after being freed by the destination, rather than using a new queue element each time.

For the Kaput mechanism we see that 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 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. Table 4 summarizes these results.

Table 4: Summary of 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

4 Suitability for Use in MPI Implementations

In this section, we discuss of the suitability of the mechanisms for supporting MPI operations [9, 6]. 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 will not examine short MPI two-sided messages.

In this analysis we will assume that there already exists a shared queue between the processes which is used for small messages. We will assume that this is the default mechanism to transfer data and compare the other data transfer mechanisms to the message-queue mechanism.

4.1 Large MPI Two-Sided Communication

Large MPI messages are typically transferred using a rendezvous protocol, where the sender and receiver first match the send and receive requests and then transfer the data. This 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 rendezvous data. However, the advantage to using one of the one sided mechanisms, Kaput, NIC copy or Ptrace, is that there is less synchronization required between the processes, and there can be more overlap of computation and communication. If the MPI send or receive is non-blocking, once the send and receive requests have been matched, there is no need for the process initiating the transfer to synchronize with the other process. 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 leads to better CPU utilization. Among these methods the Kaput mechanism provides the highest throughput, but 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 thing that needs to be considered for the one-sided mechanisms is the setup and tear-down times. The Kaput and NIC copy mechanisms require that the source and destination buffers be reg-

istered before the data can be transferred. This is an additional 2.8 μs to 4.5 μs added to the transfer time. 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 avoid this overhead, the deregistration can be deferred until the amount of registered memory exceeds a threshold then all of the unused buffers can be deregistered at once.

The shared-buffer and message-queue mechanisms can also be used even with non-blocking MPI sends and receives because each process will eventually have to wait for the operations to complete. At that time both processes are available to perform the transfer operation. If there are a small number of processors on the SMP node, 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 transfered. In this case, this mechanism would give the best throughput of all of the methods, and the least cache impact of all of the methods except for NIC copy.

4.2 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 message, one process specifies both the source and destination buffers. There are two modes in which MPI one-sided operations can be used, active, and passive.

In active mode, the target process first calls an MPI function to allow other processes to perform one-sided operations on its memory, then it calls another MPI function that ensures that the operations have completed. The MPI standard does not require that any one-sided operations performed after the first call actually complete until after the second function is called. This means that the implementation can delay the transfer of the data until the second function is called, and that the both sides can be actively involved in the transfer of the data.

In passive mode, however, the target node does not have to call any functions in order for one-sided operations initiated by remote processes to complete. This means that the implementation cannot depend on the target process to be involved in the transfer. The implementation, can however, 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, this is essentially the same as the two-sided rendezvous case, except that the buffers need only be registered once when the window for the one sided operations is opened, and deregistered only when the window is closed. This allows the cost of the registration and deregistration to be amortized over many one-sided operations.

In passive mode the 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 thread is not used, only the one-sided transfer methods, Ptrace, Kaput and NIC copy, can be used to transfer the data. Ptrace and NIC-copy have high small-message latencies, over 20 μs and 10 μs respectively. This makes them not ideal for transferring small messages. The Kaput mechanism, which has a small message latency of around 2 μs, would be preferable in this case.

Table 5: Summary of characteristics of each data transfer mechanism

	Latency (μs)	Throughput (MBps)	Setup (μs)	Tear-down (μs)	Cache impact (source, dest.)
Shared-buffer	1.5	513.5	96.3	23.2	(High, Low)
Message-queue	3.3	437.1	96.3	23.2	(High, High)
Kaput put	2.1	495.5	2.8	1.4	(High, Low)
Kaput get	2.1	500.3	2.8	1.4	(Low, High)
NIC-copy put	11.6	227.1	4.5	212.4	(Low, Low)
NIC-copy get	14.3	171.7	4.5	212.4	(Low, Low)
Ptrace	20.2	460.0	0.0	0.0	(Medium, High)

The most efficient method for passive mode, if it is available, is to make the memory allocated by `MPI_Alloc_Mem()` sharable. Then when one-sided operations are to be used, this memory at the target process can be 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, 96.3 μs to allocate the memory and 23.3 μs to free it, but if the number of allocations and deallocations is low compared to the number of one-sided operations, the allocation and deallocation costs can be amortized.

5 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 5 summarizes the results of our evaluation.

We note that not all mechanisms may be available in all environments. For instance, normal users 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. There are faster NICs and communication subsystems, such as Myricom's MX [2], that can provide up to 495 MBps bandwidth and latency down to 2.6 μs [10]. 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 Sparcs or G5s, as well as to large shared memory machines.

References

[1] MPLite. http://www.scl.ameslab.gov/Projects/MP_Lite/.

[2] MX. http://www.myri.com/scs/download-mx.html.

[3] N. J. Boden, D. Cohen, et al. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, pages 29–35, Feb 1995.

[4] 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.

[5] Phil Carns. Kaput. A kernel module for copying data between process., 2004.

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

[7] Hyun-Wook Jin, Sayantan Sur, Lei Chai, and Dhabaleswar K. Panda. "LiMIC: Support for High-Performance MPI Intra-Node Communication on Linux Cluster". In *2005 International Conference on Parallel Processing (ICPP'05)*, pages 184–191, 2005.

[8] J. Mellor-Crummey and M. Scott. "Algorithms for scalable synchronization on shared-memory multiprocessors". *ACM Transactions on Computer Systems*, vol 9(1):pp. 21–65, February 1991.

[9] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, Mar 1994.

[10] Myricom. Myrinet performance measurements. http://www.myri.com/myrinet/performance/.

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