Performance of User-Level Communication on Distributed-Memory Multiprocessors with an Optimistic Protocol

J. William Lee
Department of Computer Science and Engineering
University of Washington

Technical Report 93-12-06
December 1993

Performance of User-Level Communication on Distributed-Memory Multiprocessors with an Optimistic Protocol

J. William Lee
Department of Computer Science and Engineering
University of Washington

December 1993

Abstract

The scalability and performance of parallel applications on distributed-memory multiprocessors depends to a great extent on the performance of communication. Although the communication hardware on distributed-memory multiprocessors can transfer data quickly, applications often fail to achieve communication performance matching the hardware's capacity. Two key impediments to achieving high performance communication are flow control and protection boundaries.

This paper proposes and evaluates two techniques to reduce the cost of flow control and the cost of crossing protection boundaries. First, to optimize the common case, processors may send messages optimistically without ensuring that the destination processor of the message has enough buffer space to receive the message. Second, to avoid crossing protection boundaries through expensive traps and software interrupts, the kernel and the user-level address space may cooperate and communicate asynchronously through shared data structures. Implemented using these two techniques, our prototype system on an Intel iPSC/2 improves the performance of message passing primitives by a factor up to 2.5 and the performance of real applications by up to 30%.

1 Introduction

The scalability and performance of parallel applications on distributed-memory multiprocessors depends to a great extent on the performance of communication. Poor performance of communication not only limits the granularity of programs that can run successfully, but also degrades the performance of course-grain programs and limits their scalability.

Although most communication hardware can transfer data quickly, applications often fail to achieve communication performance matching the hardware's capacity. For example, on an Intel Delta multicomputer, the hardware can deliver a short message in less than one microsecond, but

This work was supported in part by the National Science Foundation (Grants No. CCR-8907666, CDA-9123308, and CCR-9200832), the Washington Technology Center, Digital Equipment Corporation, Boeing Computer Services, Intel Corporation, Hewlett-Packard Corporation, and Apple Computer.

the user-level application requires 67 microseconds. This *communication gap* [Felten 92a] is caused by the communication software, which typically performs flow control, manages message buffers, crosses protection boundaries, and ensures protection.

In this paper, we propose two new ideas to reduce this communication gap. The first idea attempts to decrease the cost of flow control on machines with a reliable network, such as a wormhole routing network. The second idea attempts to lower the cost of crossing protection boundaries on multicomputers, such as an Intel Paragon, that prohibit the user-level application from accessing the network interface directly. Briefly, the two ideas are as follows:

- *An optimistic flow control protocol.* In this protocol, each processor may send a message optimistically without ensuring that the destination processor has enough buffer space to receive the message. The processor retransmits messages failed due to buffer overflow using a protocol that takes advantage of the reliable network. This *optimistic* protocol is in contrast to *conservative* protocols, in which each processor sends messages only after it ensures that the receiver has enough buffer space [Pierce 88].

- *Crossing the protection boundaries in a communication system through a short cut — shared kernel/user data structures.* Through such shared data structures, a communication system may free message buffers managed in another protection domain, and the kernel may deliver messages to an application when the application promises to poll messages.

The motivation behind the optimistic protocol is to optimize the common case. When running parallel applications, each processor typically sends a limited number of messages at each step of the computation, and processors often consume incoming messages quickly. Thus processors are likely to have enough buffer space to receive messages.

The motivation behind the second idea is that traps, upcalls, and software interrupts are inherently expensive [Anderson et al. 91, Anderson et al. 92]. The idea of letting kernel and user communicate asynchronously through shared memory is not new: Marsh et al. used this technique to integrate kernel support with a user-level thread package on NUMA shared memory multiprocessors [Marsh et al. 91]. However, we believe we are the first to propose the extensive use of shared data structures in communication systems on distributed-memory multiprocessors. Our approach focuses on how a communication system may cross protection boundaries asynchronously instead of synchronously, and still maintain the same message passing semantics.

The goal of this work is two fold: (1) to propose and evaluate the two ideas described above, and (2) to build a fast communication facility suitable for implementing distributed applications, such as distributed shared memory systems.

To achieve this goal, we designed and implemented a communication facility called *Express Messages*, incorporating the two ideas described above. Express Messages currently runs on an Intel iPSC/2. It integrates message passing with a user-level thread package. The thread package executes message handlers in the context of user-level threads. It also polls incoming messages from the kernel transparently to the application programmer: messages are polled either when the user-level thread management system has no ready threads to run or when it is processing previous incoming messages.

The latency and throughput achieved by Express Messages are significantly better than that achieved by the native iPSC/2 message passing library: For small and medium messages, Express Messages reduces the latency by up to a factor of 2.5 and increases the throughput by up to a factor of 2.8.

Our experiments also show that Express Messages can effectively cut the communication time in real applications. Through a distributed shared memory system, we run two shared-memory programs on top of both the iPSC/2 message passing library and Express Messages. Express Messages reduces the application run time by from 0% to 30%. We found that, for these applications, messages sent optimistically rarely fail, and the applications often poll a large percent of the incoming messages through the user-level thread package.

The rest of the paper is organized as follows. The next section further motivates our approach by describing the problem that confronts high performance communication. Section 3 details the two ideas we propose. Section 4 describes the design and implementation of Express Messages. Section 5 evaluates the performance of Express Messages and the effectiveness of the two ideas. Section 6 describes some related work. Section 7 concludes.

2 Sources of the Communication Gap

In this section, we motivate our approach by describing the sources of software overhead in conventional communication systems on distributed-memory multiprocessors. We can attribute the software overhead to two main reasons: protocol processing and protection boundaries [Felten 92a].

2.1 Protocol Processing Overhead

Protocol processing includes two tasks: buffer management and flow control. The challenge in buffer management is to reduce overhead and avoid copying messages. To reduce message copying, a communication system may map message buffers into both the kernel and the user-level address space [Schroeder & Burrows 90, Brustoloni & Bershad 93].

The challenge of flow control is to deliver messages between user-level processes reliably with a small cost. The cost of flow-control typically includes extra computation, as well as extra protocol messages. For example, the Intel NX/2 operating system, which runs on many of the Intel multicomputers with reliable networks, uses a *conservative* flow control scheme [Pierce 88]: each node sends a message only after it ensures that the destination node has enough buffer space to receive the message. Specifically, NX/2 partitions messages into short and long messages. Each node reserves some short message slots for each other node. A node may send a short message directly, as long as it has a short message slot to the destination node. However, each node has to send a long message using a three-way protocol: The sender first sends a short message to reserve the required buffer space; after receiving a grant message from the receiver, the sender then transfers the long message itself. In NX/2, the maximum size of short messages is 100 bytes.

It is inherently difficult for a conservative flow control scheme to avoid sending protocol messages in order to deliver medium or long messages reliably. The reasons are as follows: First, in general,

each processor does not know how much buffer space is left on another processor. In order for a processor to make sure that there is enough buffer space on another processor, it has to reserve that much buffer space either by sending a protocol message or by reserving the required buffer space in advance. Second, each processor in general does not know the size and destination of the messages it is going to send, because applications may decide these parameters at runtime. So in order to reserve space for future messages, a processor must in general reserve buffer space on all processors. This makes reserved buffer space a scarce resource, increasingly so for machines with a large number of processors. If processors use reserved buffer space to send medium or long messages, this scarce resource would be exhausted quickly, and the communication system will tend to send protocol messages frequently in order to reserve more buffer space.

2.2 Protection Boundaries

The problem of protection boundaries is two fold. First, because the kernel can not trust the user, it has to verify the parameters in system calls. Second, and more importantly, it can be costly to cross protection boundaries. Conventional communication software crosses protection boundaries through traps, upcalls, or software interrupts. These operations may save registers and flush cache and TLBs; they are inherently expensive [Anderson et al. 91, Anderson et al. 92, Felten 92a].

The protection boundaries problem described above exists only on machines that prohibit applications from accessing the network interface directly. By mapping the network interface into the user level, Thinking Machines' CM-5 eliminates the need to cross protection boundaries in communication systems [TMC 91, Felten 92a]. However, such a hardware solution has its cost: CM-5 has to support "all fall down" mode; it has to provide two separate networks, one for the operating system kernel and one for the user-level application; and it has to pay extra overhead when users time-slice partitions.

3 Two Techniques to Reduce the Communication Gap

We propose to reduce the communication gap in two ways: (1) reducing the cost of flow control through an optimistic protocol, and (1) lowering the cost of crossing protection boundaries through shared kernel/user data structures.

3.1 The Optimistic Flow-Control Protocol

The basic idea of the optimistic flow-control protocol is to send short and medium messages optimistically and use a conservative protocol to retransmit failed messages. The protocol divides messages into short, medium, and long messages. The size of short messages is subject to the same constraint as in the NX conservative protocol. The size of medium messages may be as large as a page, for example.

The basic scheme is as follows:

- Under certain condition as described below, each processor sends short and medium messages

optimistically, without ensuring that the destination processor has enough buffer space to receive the message. (Below, we call these messages *optimistic messages*.)

- In all other situations, each processor sends messages using the NX conservative protocol: it sends short messages directly, and it sends medium and long messages using the three-way protocol.

- If a receiver runs out of buffer space to receive an optimistic message, the receiver replies the sender with a negative acknowledgment. The sender then retransmits the failed message using a conservative protocol.

- Each processor frees an optimistic message only after the processor knows for sure that the message has been *accepted* by its receiver. A processor accepts a message when it commits resources to buffer the message.

There are two challenges in designing the optimistic protocol: (1) How can each processor know what optimistic messages have been accepted by their destination processors? Can this be done without requiring processors to reply to every optimistic message with a positive acknowledgment? (2) How does the system maintain the order of messages? These two issues are discussed below.

3.1.1 Freeing Optimistic Messages

We let messages piggyback information about what optimistic messages have been accepted, rather than requiring processors to acknowledge every optimistic message. This is done through sequence numbers. For each of the other remote processor d processor s communicates with, processor s maintains three sequence numbers: *out(s, d)* is the sequence number for the next optimistic message from processor s to processor d; *in(s, d)* is the sequence number of a past optimistic message from processor d to processor s: processor s has accepted all optimistic messages from processor d up to this message; *safe(s, d)* is the processor s's best guess of processor d's *in(d, s)*: all optimistic messages from processor s to processor d with sequence number less than *safe(s, d)* should have been accepted [1]. Whenever processor s sends a message to processor d, it attaches *in(s, d)*, which is used by processor d to update its *safe(d, s)*.

In general, each processor frees optimistic messages lazily. When a processor detects that its free buffer space is low, it frees optimistic messages according to the *safe* sequence numbers maintained by the processor. Because messages usually go both ways during parallel computation, this scheme is likely to free most of the optimistic messages.

It is possible, however, that messages may go only one way, as depicted in Figure 1. In the figure, application messages only flow from processor A to processor B. Our solution takes advantage of the fact that the underlying network delivers messages reliably and maintains the order of messages. When processor A runs out of buffer space to send messages, it sends processor B a short inquiry, to which processor B must reply. When processor A receives the reply, it knows for sure that it has

[1] Sequence number overflow is handled in the same way as in conventional communication systems.

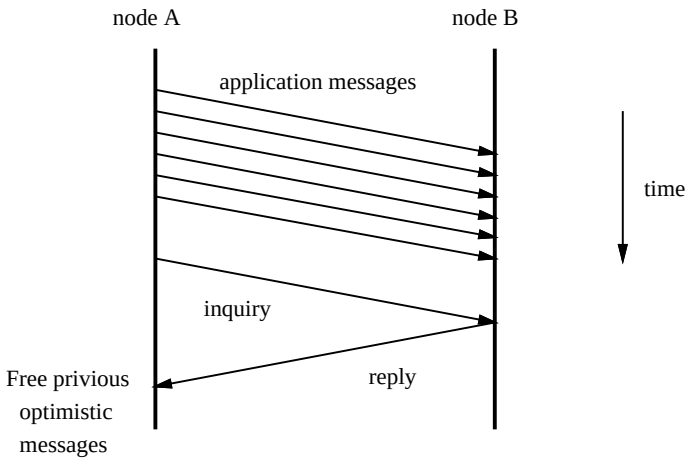

Figure 1: Freeing optimistic messages in the "worst case."

received all the negative acknowledgments for optimistic messages sent before the inquiry; those messages should either have been accepted or have been retransmitted.

3.1.2 Maintaining the Order of Messages

Each processor maintains the order of incoming messages using a FIFO queue. Whenever a processor hears about an incoming application message the first time, it appends an element at the end of the FIFO queue to remember its order. When the processor receives a retry message, it finds the order of the message by searching through the FIFO queue.

The use of the FIFO queue poses a problem: the FIFO queue of a processor may overflow if the processor can not keep up with the speed of incoming optimistic messages. We solve this problem by giving each processor a limited amount of optimism: Each processor can only send short and medium messages optimistically when the difference between the corresponding *out* and *safe* sequence numbers is under a threshold. This scheme bounds the number of elements in any FIFO queue representing failed messages. Because a processor can always reject incoming optimistic messages when the FIFO queue fills up, the processor can always make sure that the FIFO queue never overflows.

In essence, each processor reserves a certain number of elements in each other's FIFO queue for failed optimistic messages. This is similar to reserving short message slots. However, because the size of each element in the FIFO queue is small, each processor can afford to let other processors be reasonably optimistic.

3.2 Crossing the Protection Boundaries through Kernel/User Shared Data Structures

Crossing the protection boundaries through traps, upcalls, or software interrupts is expensive, but there is a short cut: the kernel and the address space may communicate asynchronously through data structures mapped into both address spaces [Marsh et al. 91]. The difficulty, however, is to

maintain the same message passing semantics when the kernel and the address space communicate asynchronously instead of synchronously.

To illustrate, consider a parallel program running on top of an RPC system implemented at the user level. Each processor may multiplex both user-level application threads that perform real computation and user-level server threads that processes incoming RPC calls. When the kernel receives an incoming RPC call, it may notify a user-level server thread either synchronously through an upcall or asynchronously through shared memory. But if the kernel notifies the server through shared memory, the address space may ignore the incoming call for a long time because application threads may enter a long computation phase. Due to this delay of processing incoming messages, the processor that issued the call may idle for a long time waiting for the reply. This is especially a problem in distributed shared memory systems [Chase et al. 89, Bershad et al. 93], in which a request may hop across several processors through a chain of "best guesses" before it fetches some remote data.

Our approach to the problem is to let the kernel and the address space form a contract: If an address space promises to poll messages, the kernel must place incoming messages at certain place in the user's address space. If the address space does not promise to poll messages, the kernel must crawl out to the user level synchronously through upcalls or software interrupts. In this way, the kernel and the address space cooperate to guarantee that incoming messages are never ignored.

4 The Design and Implementation of Express Messages

Express Messages is a thread and communication system implemented using the two techniques described above. In Express Messages, the kernel implements the optimistic flow-control protocol, and a user-level thread package contracts with the kernel: the user-level thread package polls messages either when it has no ready threads to run or when it is processing previous incoming messages. This approach makes polling transparent to application programmers.

4.1 The Message Passing Interface

Express Messages provides user-level threads, thread synchronization objects, and message passing primitives. The message passing primitives include routines to allocate message buffers and a single *send* primitive. The *send* primitive takes three parameters: a unique identifier for the remote destination process, a pointer to a message, and a pointer to a message handler. A thread sending a message continues as soon as *send* returns: the message buffer is freed automatically later when the message has been transferred reliably. When the message arrives at the destination process, a user-level server thread processes the message automatically by executing the message handler with message body as argument.

Express Messages supports an execution model in which the same program image is loaded onto all the participating processors. Each process multiplexes user-level threads on a single processor. This execution model was chosen mainly for two reasons. The first reason is simplicity. Applications that run on distributed-memory multiprocessors, including message passing programs

and distributed shared memory systems, often load the same program image onto all participating processors. These applications do not require a general naming and binding facility such as those found in RPC systems. The second reason is performance. User-level threads provide a way to manage parallelism with inexpensive primitives [Anderson et al. 92]. User-level threads also provide a way to hide communication latency by allowing the communication of one thread to overlap with the computation of other threads [Felten 92b].

Many characteristics of Express Messages are motivated by our desire to support the implementation of efficient distributed applications, especially distributed shared memory systems:

- Express Messages invokes message handlers automatically, and message handlers may block. This feature helps to implement distributed applications because, in a distributed application, incoming messages may be unanticipated and message handlers may have to share synchronization objects with application threads.
- The send primitive is one-way asynchronous. Compared to the request-reply model used in RPC systems, this approach exposes more power [Lampson 83, Walker et al. 90, Ananda et al. 91].
- The system maintains the order of messages. Maintaining the order of messages makes it easier to reason about correctness in a distributed application.
- In each process, there is a single server thread processing incoming messages one after another. Because each processor only has a limited amount of physical memory, a robust system has to bound the number of server threads that may be created in a process during computation. We choose a single server thread for simplicity and performance.

4.2 Implementation

As mentioned earlier, we implemented Express Messages on an Intel iPSC/2, which runs the native Intel NX/2 operating system and incorporates a reliable hypercube network [Pierce 88]. The DMA controller of the network can perform scattered-gather. The NX/2 operating system provides virtual memory, but it does not provide paging.

We modified the NX/2 kernel and implemented a user-level thread package. Where the original NX/2 anticipates a short incoming message, the modified kernel anticipates a message of size up to a page. The modified kernel still handles its native iPSC/2 messages in the same way as it used to, making it possible for existing iPSC/2 message passing programs to run as before. The new kernel can also perform the optimistic flow-control protocol for the new type of messages, and it can upcall into the user level. The user-level thread package schedules and synchronizes user-level threads, allocates message buffers, processes upcalls, polls incoming messages, and implements a thin layer of flow control at the user level.

4.2.1 Buffer management

Message buffers are mapped into both the kernel and the user-level address space. The output message buffer is managed mainly by the thread package, and the input message buffer is managed

mainly by the kernel. Both the thread package and the kernel allocates short, and medium and long messages differently: short messages are allocated through a free list, and medium and long messages are allocated at the granularity of a page.

In most cases, Express Messages does not copy messages while delivering messages from the output buffer of one process into the input buffer of another process. The kernel always sends messages directly from their original buffer space. The kernel receives different messages in different ways: an optimistic message is received into a page and mapped into its destination process; a short message sent via the conservative protocol is received into a page and then copied into a short message slot; a medium or long message sent via the conservative protocol heads directly into the pre-allocated buffer. This scheme minimizes copying in the kernel while retaining protection.

Both the input and output message buffers are freed lazily through kernel/user shared data structures. For example, after the kernel has transferred a short message, the kernel inserts the message into a list shared by the kernel and the address space. When the address space runs out of buffer space to send short messages, the address space may examine the list and claim all messages returned by the kernel.

4.2.2 Flow Control

The kernel implements the optimistic flow-control protocol described in Section 3.1. The actual implementation is process-based rather than processor-based: In Express Messages, each process maintains information about flow control and the kernel acts on behalf of the process to carry out the flow-control responsibility. Because each application using Express Messages starts one process on each participating processor, there is no basic difference between the process-based and processor-based approach.

Because messages are allocated and eventually consumed at the user level, there is also a thin layer of flow control in the user level thread package. On the send side, when an address space runs out of output buffer space, the address space blocks any thread that attempts to allocate buffers and traps into the kernel, informing the kernel to return all the messages already accepted. Those blocked threads may be unblocked later by the user-level scheduler when the scheduler discovers that the kernel has returned more buffers. On the receive side, when the kernel requests the address space to return buffers synchronously, the address space traps into the kernel every time a message handler returns. The kernel may request the user level to return buffers synchronously when it runs of of buffer space or short message slots.

4.2.3 Polling

The user-level thread package polls incoming messages from the kernel through the FIFO queue used in the optimistic protocol. The FIFO queue is a circular queue mapped into both the kernel and the address space. The address space maintains a pointer to the front of the queue, while the kernel maintains another pointer that indicates up to which message the address space may consume. The address space polls messages simply by comparing the two pointers and grabbing

messages from the queue. Security is not compromised: both the FIFO queue and the pointer maintained by the kernel are mapped read-only at the user level.

4.2.4 Complying with the Contract

In Express Messages, the address space polls messages either when it has no ready thread to run, or when the single server finishes processing an incoming message. The address space promises to poll messages either when the user-level scheduler is running the idle polling loop or when the server is handling an incoming message. The information about the promise reaches the kernel via a bit in a kernel/user shared data structure. When the address space does not promise to poll messages, the kernel upcalls into the user level to deliver incoming messages. The address space responds to an upcall by starting the server thread, which runs at a user-level priority higher than application user-level threads.

This contract not only makes it possible for the kernel to cross into the user-level quickly, but also simplifies the implementation. Without the contract, an upcall may occur while the address space is still processing previous upcalls. With the contract, the address space may safely assume that upcalls will not occur until the server thread revokes its promise of polling messages.

Our implementation chooses to use a single upcall stack to avoid managing upcall stacks and passing them between the kernel and the address space. This scheme, however, poses a minor problem: If the kernel upcalls after the server has revoked its promise but before the server has completely switched to another thread, the kernel may overwrite the upcall stack which is still in use. We solve this problem by detecting such a situation. If before the upcall, the kernel discovers that the stack pointer of the interrupted process is still within the upcall stack, the kernel does not push the whole process state onto the upcall stack; instead, it simply upcalls and inform the upcall handler of such event.

4.2.5 Critical Sections

Critical sections pose problems. User-level thread packages on uniprocessors typically use critical sections to guarantee atomicity. However, an upcall may occur when an application thread is inside a critical section, and the message handler triggered by the upcall may also need to enter the same critical section.

We use a *lazy* strategy to solve this problem. When the server detects that a critical section it is going to enter is occupied, it resumes the interrupted process, which must be currently in the critical section. When the resumed process exits the same critical section, it hands off the processor back to the server, at which point the server enters the critical section and continues.

Our lazy scheme is in contrast to the *eager* recovery schemes used in Scheduler Activations, which always give the interrupted thread a chance to exit critical sections at the time of the upcall [Anderson et al. 92, Barton-Davis et al. 93]. Such an eager scheme is necessary on shared-memory multiprocessors, because if a thread is preempted in a critical section, it may impede the progress of other threads on other processors. This problem, however, does not exist on uniprocessors. Our

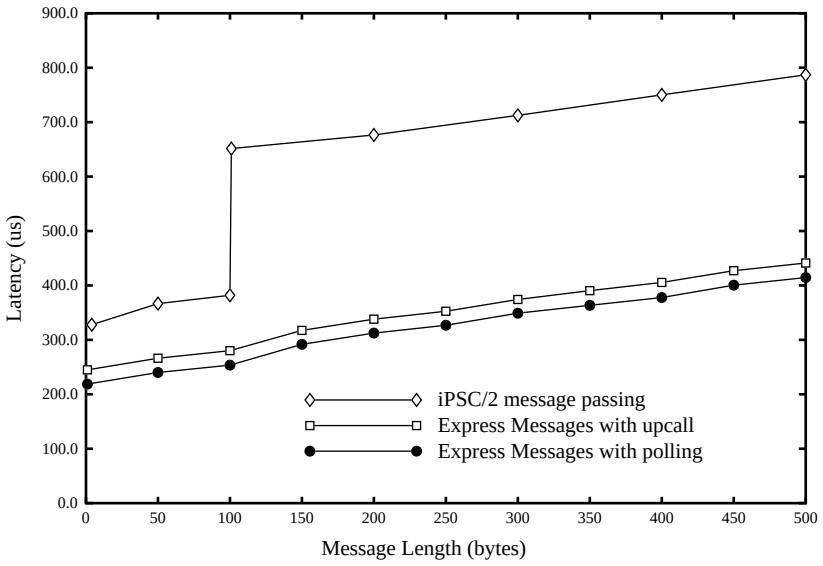

Figure 2: The best-case latency of the iPSC/2 message passing library and Express Messages.

lazy scheme simplifies the implementation and optimizes the case in which the server does not block.

5 Performance

In this section, we describe the performance of Express Messages measured on the Intel iPSC/2. We focus on the effect of the two techniques described in Section 3 on the performance of message passing primitives, as well as real applications.

5.1 The Performance of Message Passing Primitives

Figure 2 compares the latency achieved by Express Messages and the native iPSC/2 message passing library. The latency is measured by a simple ping-pong program, in which two processes on two nodes send messages back and forth. Each measured run of the program represents the best case in each system: in the case of iPSC/2 messages, messages are not copied, and short message slots are piggybacked both ways. In the case of Express Messages, all messages less than about a page are sent optimistically, and optimistic messages never fail.

There are two lines in the figure representing the latency of Express Messages. One of them, Express Messages with polling, is measured when both processes in the ping-pong program have no ready thread to run. The other, Express Messages with upcall, is measured when both processes run an application thread that spins in a loop while a message is bounced between the two processes. The difference in latency is about 27 *us*. This relative large difference is due to the cost of an upcall: the upcall saves all the registers and the state of the floating point unit on the upcall stack.

As shown in the figure, the latency achieved by Express Messages is significantly lower than that achieved by the iPSC/2 message passing library. More importantly, the latency of iPSC/2 messages

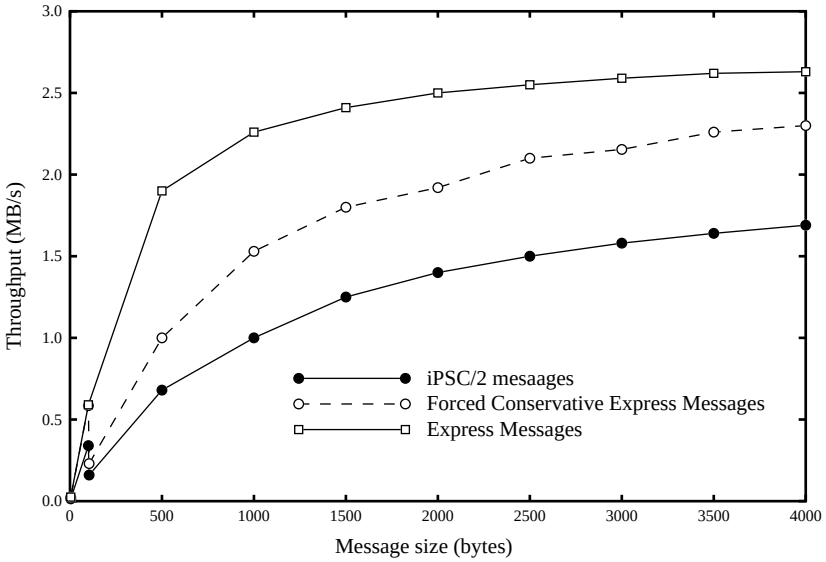

Figure 3: The throughput of the iPSC/2 message passing library and Express Messages. The curve "Forced conservative Express Messages" is measured when optimistic messages are manually disabled in Express Messages.

increases dramatically at 100 bytes, but the latency of Express Messages increases gradually as the message size increases, until the message size reaches about the 4k page size (which is not shown in the figure). This difference is the inherent advantage of the optimistic protocol.

Figure 3 compares the throughput achieved by Express Messages and the iPSC/2 message passing library. The throughput is measured by an application, in which one process sends thousands of messages to another. In the Express Messages program, the receiver has no ready thread: the user-level scheduler consumes messages through polling.

Again, Express Messages significantly outperforms the iPSC/2 message passing library. The difference in throughput is mainly due to two reasons. First, the iPSC/2 message passing system may copy a message if the message arrives at its destination before the corresponding receive is posted. Express Messages, on other hand, never copies messages when it passes messages larger than 100 bytes. This difference contributes mainly to the gap between the two curves "iPSC/2 Messages" and "Forced conservative Express Messages." The latter is measured when optimistic messages are manually disabled in Express Messages, in which case Express Messages is forced to use the NX conservative protocol to send messages. Second, the iPSC/2 message passing system sends a lot more protocol messages. The number of protocol messages sent by the iPSC/2 message passing system is always twice the number of application messages, as long as the application message is bigger than 100 bytes. On the other hand, the number of protocol messages sent by Express Messages is typically a small fraction of the number of application messages. For example, for messages of 500 bytes, Express Messages sends 88% of the messages optimistically. None of the optimistic message fails because the receiver receives these messages quickly. The number of protocol messages sent by Express Messages is only 24% of the number of application messages. This inherent advantage of the optimistic protocol contributes to the difference between the two

Application	Red/Black SOR	LocusRoute
Performance improvement	7.1%	33%

Table 1: Performance improvement of running on top of Express Messages over running on top of the iPSC/2 message passing library.

curves "Express Messages" and "Forced conservative Express Messages."

5.2 Application Performance

In order to measure the effect of Express Messages on real applications, we ported a distributed shared memory (DSM) system, the Concord DSM system, onto Express Messages. Concord attempts to provide a practical DSM system for real applications through a careful division of responsibilities among the programmer, the compiler, and the runtime system [Lee 93]. Currently, the same machine-independent code of Concord has been implemented on top of three message passing platforms: Unix sockets on a network of workstations, the iPSC/2 message passing library on an iPSC/2, and Express Messages on the iPSC/2.

In this section, we evaluate the effectiveness of Express Messages through two shared memory programs that run on top of Concord. The first application is Red/Black SOR. Red/Black SOR solves the two dimensional Laplace equation using the successive over-relaxation method. The second application is LocusRoute from the SPLASH benchmark [Singh et al. 92]. LocusRoute is a commercial quality VLSI standard cell router. The original LocusRoute code consists of 6400 lines of C code.

We first compare the performance of the two programs running on top of Express Messages and the iPSC/2 message passing library. Table 1 shows the speedup of the two programs running on top of Express Messages over running on top of the iPSC/2 message passing library. In the measured run, the Red/Black SOR program used 32 processors to perform Red/Black over-relaxation on 512 $\times$ 256 data points; the LocusRoute program used 16 processors to route 1266 wires.

As shown in the table, Express Messages speeds up these two programs by 7.1% to 33%. The speedup of two programs are very different because the two applications spend different amount time in communication. The Red/Black SOR program spends a small percent of time in communication, while LocusRoute spends a large percent of time in communication.

5.2.1 The Effectiveness of the Optimistic Protocol

To assess the effectiveness of the optimistic protocol, we gathered statistics when the Red/Black SOR program and the LocusRoute program run on top of Express Messages. These statistics are shown in table 2.

As shown in the table, when these two applications run, processors send almost all the messages optimistically and few optimistic messages fail. This is not surprising: For example, in the Red/Black SOR program, threads exchange boundary data points with their neighbors. As long

Application	Red/Black SOR	LocusRoute
Percent of messages sent optimistically	100%	99.9%
Percent of optimistic messages failed	0%	0%

Table 2: Some statistics gathered when the two programs run on top of Express Messages.

Application	Red/Black SOR	LocusRoute
Performance improvement	2.5%	11%

Table 3: The performance benefit of using the optimistic protocol over the conservation protocol.

as the amount of buffer space is more than the space occupied by boundary points of a few data blocks, optimistic messages should never fail during the inner loop of the computation.

How does the optimistic protocol affect performance? In order to answer this question, we manually disabled optimistic messages in Express Messages and measured the speedup of using the optimistic protocol in Express Messages over using solely the conservative protocol in Express Messages. The result is shown in table 3.

The benefit of the optimistic protocol ranges from 2% to 11%. These values are significantly lower than those in table 1. We believe that this is mainly because the iPSC/2 message passing primitives may be copying messages.

5.2.2 The Effectiveness of Crossing Protection Boundaries through the Short Cut in Shared Memory

Figure 2 shows delivering incoming messages through shared data structures save 27 microseconds from the latency of a one way message. This advantage, however, will only benefit applications if applications receive messages through polling. Table 4 shows the percent of messages polled when these two applications run. As shown in the table, majority of the messages are polled.

The fact that the thread package polls for a majority of the messages suggests that the thread package might as well poll all messages. To assess the effect of this alternative choice, we disabled upcalls in Express Messages and measured the performance degradation due to polling-only. The results are shown in table 5.

Table 5 shows that there can be a huge win to let the kernel and the address space form a

Application	Red/Black SOR	LocusRoute
Percent of messages polled	94%	83%

Table 4: The percent of messages polled when the two programs run on top of Express Messages.

Application	Red/Black SOR	LocusRoute
performance degradation	0%	39%

Table 5: Performance degradation of polling-only compared to having a contract.

contract. Polling-only degrades performance of LocusRoute by 39%. This is because LocusRoute uses a work heap model and it is very dynamic: incoming messages may often find that a thread management system is not idle. The performance degradation for the Red/Black SOR program is zero. This is because, in Red/Block SOR, threads execute in lock-step using barriers: they are likely to be either all computing or all communicating.

6 Related Work

There is a large body of literature related to improving the performance of communication systems. Below we compare our work with four other systems that either have similar goals or use similar techniques.

Protocol compilation [Felten 93] is a technique in which a compiler generates tailored protocols to reduce the cost of protocol processing. The key observation of this approach is that the compiler, unlike processors, may have global knowledge of how a parallel program runs, and this knowledge may be used to reduce the cost of protocol processing. For example, the compiler may ensure that processors have enough buffer space to receive messages. Felten's prototype protocol compiler successfully speeds several message passing programs by 0-20%. This approach, however, works only for data parallel programs. There are still many open research problems related with this pioneering work: For example, can compilers recognize communication pattern automatically? Can runtime compilation work successfully for programs in which the size and destination of messages can only be determined at runtime? Unlike Felten's approach, our approach — the optimistic protocol — requires no help from either the compiler or the programmer, and it can reduce protocol overhead for any application in which processors are likely to have enough buffer space to receive messages. Our approach also reduces the copying of messages, though we require the programmer to allocate message buffers from dedicated memory regions.

Active Messages [von Eicken et al. 92] is a communication system that attempts to reduce the cost of protocol processing using a minimum protocol approach. Because Active Messages invokes message handlers while interrupts are masked off, it does not need a flow-control scheme, and it requires only two message buffers. This simplicity reduces message latency dramatically. However, Active Messages is a low-level communication system. Using Active Messages as a high-level communication system confronts three difficulties. First, it is difficult to ensure security on multicomputers that do not use a separate network for operating systems. If a user-level message handler refuses to return to the kernel either because of an error or a malicious act from a programmer, the message handler may block messages destined to other processes as well as the operating system. Second, it is difficult to structure applications that use Active Messages. A message handler doing

useful work may have to share synchronization objects, such as locks, with application threads. When a message arrives at a processor, an application thread on the processor may hold some of the locks required by the message handler, preventing the message handler from running without blocking. Third, it is difficult to avoid deadlock. Because there is only a single send message buffer, a message handler may find that the send message buffer is still in use when the message handler attempts to send a message. To avoid deadlock, the message handler must buffer the message somewhere else, and the application must send the message later. Unlike Active Messages, Express Messages is a high-level communication system. In Express Messages, message handlers execute while interrupts are enabled; message handlers can block; and the flow control in Express Messages frees the programmer from manually buffering and re-sending messages in order to avoid deadlock.

Optimistic implementation of bulk data transfer protocols [Carter & Zwaenepoel 89] is a technique to speed up transferring bulk data across a packet-based network. Using this technique, a bulk data transfer protocol optimistically predicts that, during receipt of bulk data, the next packet that comes from the network is the next packet of the bulk data transfer. By being optimistic, the protocol may avoid an extra copy of packets. Although our approach shares similar philosophy in protocol design, our techniques and our targeted network environment are very different. Our optimistic protocol is designed to speed up transferring medium messages across reliable networks, and the protocol attempts to avoid sending protocol messages by making optimistic assumptions.

URPC [Bershad et al. 91] is an RPC system for shared-memory multiprocessors. URPC lets two address spaces on the same machine communicate through a short cut — shared memory regions pairwise mapped into both address spaces. In URPC, each address space polls incoming RPC calls. The effectiveness of URPC for normal RPC calls relies on its optimistic assumption: (1) the client has other work to do, and (2) the server is polling, or will soon poll, incoming messages. Unfortunately, as argued in section 2.1 and demonstrated in section 5.2.2, the optimistic assumption does not always hold for parallel computing on distributed-memory multiprocessors. Our approach of polling differs from URPC mainly in that, in our approach, the kernel and the address space forms a contract to make sure that messages are always delivered quickly.

7 Conclusion

The basic ideas behind our approach are relatively simple: Processors may send medium messages optimistically because other processors may consume messages quickly, making enough room for these messages. Software communication system may cross protection boundaries through kernel/user shared data structures because the kernel and the address space may cooperate, making it possible to replace synchronous operations with asynchronous operations.

Implementing the two ideas turned out to be non-trivial. The optimistic protocol is complicated because it must retransmit failed messages and maintain the order of messages without paying a big cost in the common case. The cooperation between the kernel and the address space can also be complicated because the kernel may synchronously crawl out to the user level either when the address space is in a critical section or when the address space is changing its promise.

Despite its complexity, our prototype communication system achieved performance significantly

better than that achieved by the iPSC/2 message passing library. The latency and throughput of message passing primitives are improved by a factor up to 2.5, and the performance of two real applications is improved by 0-30%. Our measurements show that, in running parallel applications, messages sent optimistically rarely fail, and the communication system can often cross the protection boundaries through the short cut — kernel/user shared data structures.

Acknowledgments

The author is grateful to Ed Lazowska, Ed Felten, Dylan McNamee, Chandu Thekkath, and Raj Vaswani for useful conversations and helpful comments during this project. Ed Lazowska supervised this project, and provided valuable opinions at various stage of the project and valuable comments on various drafts of this paper. Chandu Thekkath taught the author a few techniques often used in conventional communication systems. Raj Vaswani and Dylan McNamee shared with the author their experience in implementing Scheduler Activations. And Ed Felten pointed out a problem concerning deadlock.

References

[Ananda et al. 91] A. L. Ananda, B. H. Tay, and E. K. Koh. ASTRA - An Asynchronous Remote Procedure Call Facility. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 172–179, May 1991.

[Anderson et al. 91] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. The Interaction of Architecture and Operating System Design. In *Proceedings of the 4th International Conference on Architecture Support for Programming Languages and Operating Systems*, pages 108–120, 1991.

[Anderson et al. 92] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[Barton-Davis et al. 93] Paul Barton-Davis, Dylan McNamee, Raj Vaswani, and Edward D. Lazowska. Adding Scheduler Activations to Mach 3.0. In *The USENIX Mach III Symposium Proceedings*, Santa Fe, New Mexico, April 1993.

[Bershad et al. 91] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. User-Level Interprocess Communication for Shared Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(2), May 1991.

[Bershad et al. 93] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. The Midway Distributed Shared Memory System. In *Proceedings of the 38th IEEE Computer Society International Conference (COMPCON 93)*, pages 528–537, February 1993.

[Brustoloni & Bershad 93] José C. Brustoloni and Brian N. Bershad. Simple Protocol Processing for High-Bandwidth Low-Latency Networking. Technical Report CMU-CS-93-132, School of Computer Science, Carnegie Mellon University, March 1993.

[Carter & Zwaenepoel 89] John B. Carter and Willy Zwaenepoel. Optimistic Implementation of Bulk Data Transfer Protocols. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 61–69, May 1989.

[Chase et al. 89] Jeffrey S. Chase, Franz Amador, Edward D. Lazowska, Henry M. Levy, and Richard J. Littlefield. The Amber System: Parallel Programming on a Network of Multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 147–158, December 1989.

[Felten 92a] Edward W. Felten. The Case for Application-Specific Communication Protocols. In *Proceedings of Intel Supercomputer Systems Division Technology Focus Conference*, pages 171–181, 1992.

[Felten 93] Edward W. Felten. *Protocol Compilation: High-Performance Communication for Parallel Programs.* PhD thesis, Department of Computer Science & Engineering, University of Washington, Technical Report 93-09-09, 1993.

[Felten 92b] Edward W. Felten and Dylan McNamee. Improving the Performance of Message-Passing Applications by Multithreading. In *Proceedings of the Scalable High Performance Computing Conference (SHPCC-92)*, Williamsburg, VA, 1992.

[Lampson 83] Butler W. Lampson. Hints for Computer System Design. In *Proceedings of the 9th ACM Symposium on Operating System Principles*, pages 33–48, October 1983.

[Lee 93] J. William Lee. Concord: Re-Thinking the Division of Labor in a Distributed Shared Memory System. Dept. of Computer Science & Engineering, University of Washington. *Submitted for Publication*, November 1993.

[Marsh et al. 91] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. First-Class User-Level Threads. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 110–121, October 1991.

[Pierce 88] Paul Pierce. The NX/2 Operating System. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, vol. 1*, pages 384–390, January 1988.

[Schroeder & Burrows 90] Michael D. Schroeder and Michael Burrows. Performance of Firefly RPC. *ACM Transactions on Computer Systems*, 8(1):1–17, February 1990.

[Singh et al. 92] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[TMC 91] Thinking Machines Corporation. *CM-5 Technical Summary*, 1991.

[von Eicken et al. 92] Thorsten von Eicken, David E. Culler, Seth Copen Coldstein, and Klaus Erik Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 256–266, May 1992.

[Walker et al. 90] Edward F. Walker, Richard Floyd, and Paul Neves. Asynchronous Remote Operation Execution in Distributed Systems. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 172–179, May 1990.