Active Messages Implementations for the Meiko CS-2

Klaus E. Schauser and Chris J. Scheiman

Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106
{*schauser,chriss*}@*cs.ucsb.edu*

Abstract

Active messages provide a low latency communication architecture which on modern parallel machines achieves more than an order of magnitude performance improvement over more traditional communication libraries. It is used by library and compiler writers to obtain the utmost performance and has been used to implement the novel parallel language Split-C. This paper discusses the experience we gained while implementing active messages on the Meiko CS-2, and discusses implementations for similar architectures. The CS-2 is an interesting experimental platform, as it resembles a cluster of Sparc workstations, each equipped with a dedicated communication co-processor.

During our work we have identified two mismatches between the requirements of active message and the Meiko CS-2 architecture. First, architectures which only support efficient remote write operations (or DMA transfers as in the case of the CS-2) make it difficult to transfer both data and control as required by active messages. Traditional network interfaces avoid this problem because they have a single point of entry which essentially acts as a queue. To efficiently support active messages on modern network communication co-processors, hardware primitives are required which support this queue behavior. We overcame this problem by producing specialized code which runs on the communications co-processor and supports the active messages protocol. We also identify hardware primitives which are required to efficiently support active messages. The second mismatch is that active messages do not provide a non-blocking form of send, which is required to achieve the highest possible bandwidth while allowing the overlap of communication and computation when a communications co-processor is present. We propose to extend the current active message definition to include a non-blocking form of send. Our implementation of active messages results in a one-way latency of $12.3\mu s$ and achieves up to 39 MB/s for bulk transfers. Both numbers are close to optimal for the current Meiko hardware and are competitive with performance of active messages on other hardware platforms.

1 Introduction

Active messages provide an efficient communication architecture which exposes the raw capabilities of the underlying hardware [vECGS92]. Although usable by application programmers, it is mainly targeted at compiler and library writers who want to obtain the utmost in performance. Active messages associate with each message a handler function which is executed on the receiving processor upon arrival of the message. Message handlers are intended to be short functions which execute quickly; in particular, message handlers are not allowed to suspend. In essence, active messages can be viewed as a very restricted form of RPC. To eliminate complicated buffer management and simplify deadlock considerations, active messages divide handlers into two classes: requests and replies. A request can send a message to an arbitrary processor, and when it arrives the specified request handler is invoked. Request handlers may answer by sending a single reply message, while reply handlers are prohibited from additional communication. Under the active message model, messages travel from user space (the send instruction) directly to user space (the message handler), avoiding any form of buffer management and synchronization usually encountered in traditional send & receive. As a result, active messages achieve an order of magnitude performance improvement over more traditional communication mechanisms.

Although they are quite primitive, active messages have become an important communication layer because of their efficiency. The small overhead and low latency facilitates building more complicated communication layers [TM94] and makes it a desirable target for high-level language compilers [CGSvE93, CDG+93]. Over the past several years active messages have been implemented on many different hardware platforms, including the CM-5 and Ncube/2 [vECGS92], Paragon, as well as clusters of workstations connected by FDDI [Mar94] and ATM [vEABB94]. In most, if not all, of these cases, active messages are a natural solution for obtaining low latency communication. In most of these systems the main processor is involved in both the sending and receiving of messages. In such a setting it is natural to attach an arbitrary computation in the form of a message handler to the code which extracts a message out of the network interface. But what if the sending and receiving of messages are handled by a communications co-processor?

This paper reports on our experience with implementing active messages on the Meiko CS-2, a parallel architecture with a communications co-processor. The CS-2, the most recent MPP built by Meiko, consists of Sparc based nodes connected via a fat tree communication network [HM93]. The CS-2 is an interesting architecture as it closely resembles a cluster of workstations, which is widely accepted as the future of parallel processing. Each node is running a full Solaris 2.3 operating system and is equipped with a special communications co-processor, the Elan processor, which enables direct user level communication in the form of DMA transfers between processors running the same parallel program.

Surprisingly, it is not straightforward to implement active messages efficiently given only remote write operations when more than two processors are involved. The difficulty is to efficiently inform the main processor about the arrival of the message, without sending several additional messages or having the main processor search a large number of input buffers.[1] Traditional network interfaces avoid this problem because they have a single point of entry which essentially acts as a queue, where the main processor can pick up messages. Special hardware support enables the CS-2 architecture to autonomously process incoming messages, without the intervention of the main processor, and deposit each message in a memory location specified by the sender. While this simplifies the transfer of data it makes it harder to combine the transfer of data with the transfer of control as required by active messages. A similar situation also exists on the Cray T3D which supports extremely low latency remote read and write operations, but lacks the support for transfer of control between two processors.

Another difficulty occurs with the bulk transfer of data. (Though a typical active message consists of only a few bytes of data, the protocol allows for special bulk transfer operations, to maximize bandwidth.) The problem is that the active message protocol does not provide a non-blocking form of send which, in the presence of a communications co-processor, allows the overlap of communication and computation, while achieving the highest possible bandwidth. With only a blocking send, the main processor is forced to wait while the data is being transferred.

Despite this mismatch between the requirements of active messages and the basic Meiko CS-2 hardware, it is possible to obtain an efficient implementation of active messages for the Meiko CS-2. Besides the DMA engine, the Elan co-processor contains other functional units, including a freely programmable thread processor with powerful network transactions which allow us to solve the problem. Our implementation of active messages results in a one-way latency of $12.3\mu s$ (including the execution of the message handler), which is just $3\mu s$ above the best obtainable $9.5\mu s$ using direct DMA transfers. For bulk transfers we obtain up to 39 MB/s, compared to Meiko's peak bandwidth of 44 MB/s. Both the latency and bandwidth are close to optimal for the current Meiko hardware and are competitive with performance of active messages on other hardware platforms.

The structure of the remainder of the paper is as follows. Section 2 presents in more detail our

[1]Note that this problem is not specific to active messages. It also shows up when implementing other communication primitives which require either the intervention of the main processor or buffer management, e.g., tagged messages.

hardware platform, the Meiko CS-2 architecture, focusing in particular on the distinguishing characteristics of the Elan communications co-processor. Section 3 discusses the issues which arise when implementing active messages on an architecture with a communication co-processor and special support for remote read and writes, while Section 4 presents the solutions and implementations we developed for the CS-2. Section 5 evaluates the performance of our active messages implementations. Section 6 identifies issues in implementing high bandwidth bulk transfers for active messages. Section 7 discusses related work. Finally, Section 8 summarizes our experiences and concludes.

2 The Meiko CS-2 Architecture

The CS-2 is the most recent machine built by Meiko. It consists of Sparc based nodes connected via a fat tree communication network [HM93]. Running a slightly enhanced version of the Solaris 2.3 operating system on every node, it closely resembles a cluster of workstations connected by a fast network. For maximum floating point performance each node can optionally be equipped with vector units. Our machine is a 64 node CS-2 without vector units. Each node contains a 40 MHz SuperSparc processor with 1 MB external cache and 32 MB of main memory. The SuperSparc is three-way superscalar processor, and thus achieves a peak rate of 120 MIPS. The largest CS-2 built so far is a 256 processor machine with vector units installed at Lawrence Livermore National Lab. Each node in a CS-2 contains a special communications co-processor, the Elan processor, which connects the node to the fat tree.

The communication co-processor enables direct user-level communication between processors running the same parallel program. For every message, the co-processor handles the translations and checks required to ensure protection. When sending a message, the co-processor translates the virtual destination number into a physical processor and route. It also automatically prepends the destination process PID. On the receiving side the processor checks the validity of the PID and, when writing a message into memory, translates the virtual address into a physical address. This ensures protection, because the TLBs, context, virtual process, and route tables are all under the control of the operating system. These checks, which are done by the Elan hardware, achieve protected user-level communication without the intervention of the operating system for every message.

The Elan co-processor contains six functional units, two of which receive messages (the inputers), three of which can send messages (the DMA engine, thread processor, and reply processor), and one which forms the memory mapped interface to the main processor (the command processor). The Elan co-processor connects directly to the MBus, and implements the MBus Level 2 protocol to ensure consistency with the Sparc caches. The code for all functional units, with the exception of the thread processor, resides in the on-chip ROM. Although logically separate, the functional units, which Meiko refers to as processors, are currently implemented on top of the same microcoded engine, and execution is switched among these.

The *command processor* forms the interface to the main processor. It is memory mapped and the main processor issues commands by writing into certain locations and accesses status information by reading from them. These commands allow the main processor to schedule operations on the DMA engine or thread processor. The *inputers* process the incoming messages and execute the network transactions specified by the packet. They can write a cache line or word into a virtual address, perform an atomic add, test the status of some memory location, or enqueue a read request on the reply processor or DMA engine. The *reply processor* services read requests and sends the responses back to the requesting processor. The *DMA engine* performs DMA transfers of arbitrary size from a virtual address on the source node to a virtual address on the destination node. DMA transfers can even cross page boundaries since the Elan co-processor contains its own TLB. Long transfers are automatically split into smaller units, and the DMA engine is timesliced to allow other functional units to execute. The *thread processor* implements a Sparc-like instruction set enhanced with special instructions for opening and closing connections, sending network transactions, and transferring control to other functional units.[2] The user either writes code for the thread processor directly in assembly language or generates it through a modified version of gcc, the "thread compiler". Due to a micro-code implementation, the thread processor achieves a peak performance of only 3 MIPS. The current gate array implementation of the Elan has only a small 8 word instruction cache for the thread processor and no data cache, and all memory accesses go to the slow external DRAM.

The main processor and the Elan can synchronize their actions through the use of *events*. The main processor and the thread processor can set, clear, and wait on events. The DMA engine and input processors can also set events. The most common use of an event is as a flag to indicate the completion of a DMA transfer.

As reported in [HM93], the highest obtainable performance at the user level with DMAs is a $9\mu s$ one-way latency for small messages, and a peak sustained bandwidth of 44 MB/s for long messages. As our measurements indicate, the one-way latency in our own machine is $9.5\mu s$ because our machine additionally has a second level cache. Of this time, $2.4\mu s$ is spent on the sending processor, $5.5\mu s$ on the DMA engine (most of which is to ensure protection), only $0.5\mu s$ are required for the network latency, and another $1\mu s$ are spent on the inputer.

3 Issues in Implementing Active Messages

Active messages associate with each message a handler function which, upon arrival, is executed on the receiving processor. The purpose of these handlers is to integrate the message into the on-going computation. There are two forms of active messages: requests and replies. Handlers for requests may send a reply, while handlers for replies are refrained from any further communication.

There are several important issues that every implementation of active messages must deal with. Of these issues we will discuss protection, reliability, notification, and synchronization. The solutions to these issues are dependent on the hardware platform.

3.1 Protection

To provide protection, we must ensure that a message cannot be sent to a destination process which is not part of the currently running parallel program. This is either ensured by involving the operating system on every message (Ncube/2 [vE93], SparcStation cluster [vEABB94]), using gang scheduling (CM-5 [vE93]), or combining gang scheduling with some hardware support (HP cluster [Mar94]). Protection on the Meiko CS-2 is trivial since it is done completely by the Elan processor in hardware.[3]

3.2 Reliability

To provide reliability, we must ensure that each message is delivered to the destination once and only once. This is fairly easy in the context of dedicated parallel supercomputers as usually the network guarantees reliable delivery once the message is accepted. On the other hand, reliability is still a major research issue for clusters of workstations which usually are connected by an unreliable network which may drop packets. Thus, the source must buffer messages for potential retransmission. The situation on the Meiko CS-2 is

[2] As we inadvertently found out, one way to "crash" the machine is to execute an infinite loop on the thread processor. Since the thread processor continues running forever, no other functional units are scheduled and messages, not even systems messages, get through, in effect hanging the node.

[3] While this makes writing communication systems code easier, is also the main inhibitor to achieving a lower latency than the $9.5\mu s$ we currently observe. Every communication operation is required to do the protection, which on the sending side involves 3 table lookups to main memory requiring about $3\mu s$. In certain situations (e.g., a dedicated machine with gang scheduling) not all of this protection is required and some or all of these table lookups could be omitted, thereby substantially decreasing the cost to send a message.

somewhere in between. The CS-2 network may drop packets due to congestion. But each message is acknowledged, i.e., returns either an ACK if it was correctly received or a NACK if it was dropped or some other transmission (CRC) error occurred. In the case of a NACK we just retransmit the message.

3.3 Notification and synchronization

For proper notification, we must ensure that the main processor is notified upon the arrival of a message so that it can invoke the messages handler. For proper synchronization, we must ensure that that the message handlers for incoming messages do not interfere in unexpected ways with the running computation. The issue of notification is one of the most difficult in implementing active messages. There are two ways to notify the main processor: one involves interrupts, and the other polling. Since interrupting the main processor when a message arrives is far too costly on most architectures, the common solution is to have the main processor check (poll) at regular intervals whether a new message has arrived. This has the added benefit that any required synchronization between ongoing computation and the incoming messages is easily handled: the processor just does not poll in critical sections.

The CS-2 with its co-processor is quite different from all of the other architectures on which active messages have previously been implemented, in that the main processor is not involved in the actual sending and receiving of messages. Rather, the co-processor transfers messages from the memory of the source processor directly to the memory of the destination processor. As it turns out, the fact that the main processor is not involved in the reception of messages has a big impact. On the other architectures, the main processor regularly polls the network interface for the arrival of a message. Since the main processor itself is involved in extracting a message from the network interface, it is easy to immediately invoke the corresponding message handler.

On the CS-2, however, the message directly goes into memory without intervention of the main processor. This in itself is not bad, as we can statically allocate the buffer space necessary. The real difficulty arises because the destination address of the message is solely determined by the sender. On a machine with P processors, if every processor owns a destination buffer on every other processor, then a poll for the arrival of a message involves checking P locations for the presence of a message. Obviously, this cost — which grows with the number of processors — is far too expensive if many processors are involved. On the other hand, checking whether a new message has arrived is cheap if all processors send to the same location. The difficulty now is that a new incoming message may overwrite some other message which has not yet been processed. Thus, this approach opens the can of worms associated with unreliable message delivery, that we thought was already adequately addressed.

Traditional network interfaces avoid this problem because they have a single point of entry which essentially acts as a queue. Checking for the arrival of a new message, and extracting it from the network, can be done in constant time. It is this queue that we somehow have to emulate with the hardware capabilities of the Meiko. We have developed several implementations of active messages which use the hardware capabilities of the CS-2 to a varying degree. The first solution just uses the remote write capabilities provided by the DMA processor. The second solution uses the programmable Elan thread processor on the sending side and makes use of special network transactions which allow to check the state of a remote location. Finally, the last solution additionally uses the thread processor on the receiving side to pick up messages and enqueue them for the main processor.

We now discuss the details behind these three solutions.

4 Solutions

This section describes the three solutions we have implemented and discusses their relative advantages and disadvantages. Performance measurements are presented in the next section. For now we limit the discussion to fixed size active messages. In the case of the Meiko CS-2 a natural choice is 32 bytes, the size of a cache line. Later, in Section 6, we discuss how the extend this for bulk transfers.

4.1 Active messages given only remote writes

We first discuss how to implement active messages using only remote write capabilities. We divide our discussion into two parts. In the first, we intend to lead the reader through the many design decisions involved in the implementation of active messages given only remote writes. In the second part, we present one solution in more detail. The main reason why it is difficult to implement active messages using only remote writes is that it is the sender who specifies the destination address of a message.

4.1.1 Alternatives and tradeoffs

Active messages have to do two things: they have to transfer the data to the destination processor and they have to transfer control to the message handler. As mentioned above, transfer of control (the notification problem) is best solved by polling, as interrupts are far too expensive. Thus on a receiving processor we need a buffer to store the incoming message and a flag to indicate that a message has arrived. (By combining the flag and the data, they can be transferred together in a single message, thus incurring the sending overhead only once.)

Assume that every processor owns an incoming message buffer on every other processor.[4] The poll on the receiving processor needs to check all of the flags (buffers). If any is set, a message is present and the corresponding message handler is invoked. The sending processor needs to know when a message has been processed so that it can reuse the receiver's incoming message buffer. Thus, every message requires a round trip acknowledgment.[5]

On a system with P processors the poll needs to check P locations. Thus, this scheme only works well for a small number of processors. But there are alternatives. One solution would be to have all processors send to the same incoming location. Now the poll only needs to check a single location, ensuring minimal latency. Unfortunately, messages can get overwritten before they have been processed by the main processor. Therefore, some error recovery mechanism has to be employed to ensure that lost messages are resent. This alternative is not attractive in the case of the Meiko CS-2, where reliable delivery is already present.

A simple way to combine reliable transmission with fast polling is to introduce additional flags which indicate the presence of a message. Each processor sends a message to a separate buffer and, in addition, sets a common flag. When this common flag is not set we know that no message has arrived. Only in the case where the common flag is set do we need to scan the other flags. Instead of just representing the flag by a single bit, we could put the sender's processor number into the location. This makes it easier to identify the sender, but because of possible overwrites we still need to scan all other locations. To avoid scanning all locations, we could use a tree structure, reducing the cost of a poll to $\log(P)$. Of course, then $\log(P)$ remote writes are required, leading to a much higher communication cost.

[4] Actually, we need a separate input buffer for requests and replies. When requests and replies do not share resources, it is easy to ensure that no communication deadlock can occur. In the case where a send inside a request handler fails, all that need to be done is to process incoming replies [vECGS92]. To simplify the description, we will limit our discussion throughout this paper to a single incoming buffer. The reader should keep in mind that in reality we always have two incoming buffers.

[5] The scheme described so far only uses one incoming buffer for every other processor. Thus, only a single message can be outstanding for any pair of processors. It is easy to extend this using k message deep input buffers [Mar94]. Now it is possible to overlap several communication operations to the same processor and it is possible to combine the acknowledgment of multiple messages. Unfortunately, as we have found out, using many buffers leads to a larger working set and therefore to a higher cache miss rate. Thus, there is a tradeoff between how many communication operations can be overlapped and the efficiency of each.

As this discussion shows, there are many alternatives in creating active messages on a machine with only remote writes. Our first implementation is a basic scheme, involving 1 output buffer and P input buffers, where each input buffer acts as its own flag.

4.1.2 Implementation

A detailed walk-through of an active message transaction is presented below. This extremely simple implementation forms the basis for our comparison with other methods in Section 5.

Implementation 1 (Active messages using remote write)

- *The sender stores the message in OUT-BUFFER. The buffer is needed since the co-processor (DMA processor) transfers the data from memory.*
- *The sender then sets up a DMA descriptor to send OUT-BUFFER to IN-BUFFER[s] on the destination processor, where s is the processor number of the sender.*
- *The sender issues a DMA request, and the DMA co-processor sends the message to IN-BUFFER[s].*
- *The receiver, when polling, checks each of the P input buffers.*
- *When it finds that IN-BUFFER[s] is non-empty, it executes the corresponding message handler.*
- *The receiver clears IN-BUFFER[s].*
- *If the message handler does not send a reply, poll makes sure to send one. This way, the sending processor knows it is free to send again.*

There can only be a single message outstanding from one processor to any other processor. While this is perhaps the simplest possible implementation, any other implementation would still involve either $O(P)$ input buffers, or an error recovery scheme.

As the above discussion shows, remote writes alone are not powerful enough to implement active messages efficiently, the reason being that it is the sender who decides the destination address of a message. What we really want is to enqueue a message on the destination site. This requires more powerful functionality than plain reads or writes. It requires some sort of atomic read/write operation. We examine such an implementation in the next section.

4.2 Active messages given remote test

To efficiently implement active messages, we would like a single point of entry for messages. This could be accomplished with the use of a queue. To implement a queue where many processes can enqueue data we need some sort of atomic read/write operation, such as test-and-set, to prevent write conflicts. Given test-and-set, our scheme for sending a message could be: Lock the remote queue with test-and-set, get the destination counter, increment the counter by message size, write the counter back to the receiver, put the message in the queue, then release the lock. Unfortunately, such an implementation would require many round trip transmissions.

We can avoid some of this overhead by implementing a queue of depth 1. By using a single buffer, essentially a queue of depth 1, our scheme for sending a message is now: Lock the buffer with test-and-set and move the message into the buffer if empty. The receiver is now responsible for releasing the lock. This scheme requires no roundtrip overhead for managing the counter.

To implement this scheme on the Meiko, we use special network transactions which test a remote location and, depending on the outcome, transmit a message or not. The sequence of special thread instructions used is: open a connection, test a remote location, do a remote write, and close the connection. On the Meiko, the write will not be done unless the test succeeds. To insure reliable delivery, it is important that the testing and writing of the buffer be a single atomic transaction with respect to other processors. Even though the Meiko CS-2 has more than one input channel, it guarantees that a complete network transaction involving a remote test is atomic.

Unlike the previous implementation involving only remote writes, only one buffer is required by the receiver, and no roundtrip communication is required. We still use an output buffer that is not re-used until the previous message has been delivered.

A detailed walk-through of an active message transaction is:

Implementation 2 (Active messages using remote test-and-set)

- *The sender waits until any previous message in OUT-BUFFER has left.*
- *The sender stores the message in OUT-BUFFER. After filling OUT-BUFFER, the sender continues on with other code.*
- *The thread processor continually scans OUT-BUFFER, and sees that a message is ready.*
- *Using remote test-and-set, it checks the remote IN-BUFFER and if it is empty, writes OUT-BUFFER into it.*
- *If the IN-BUFFER is not empty, it performs an exponential backoff and repeats the step above.*
- *Once it succeeds in transmitting the active message, the thread processor clears the OUT-BUFFER, and begins scanning the OUT-BUFFER again, waiting for a new message.*
- *On the receiver, a poll checks IN-BUFFER. If something is present, it copies the IN-BUFFER into registers, clears the IN-BUFFER so that a new message can arrive, and then executes the active message function.*

This implementation provides a single point of entry for active messages, which makes the poll efficient. The drawback is that only a single message can arrive at each destination since we have only a queue of depth one.

To implement remote test and set we made use of the Meiko co-processor on the sending side. It is also possible to use the co-processor on the receiving side. Actually, since the co-processor is fully programmable and can execute user code, we can even execute active message handlers on the co-processor without any intervention of the main processor. We discuss this in the next section.

4.3 Active messages given a co-processor

Since the Meiko CS-2 has a co-processor, it seems natural to use it to also receive incoming active messages and execute message handlers. In this way, the main processor is not interrupted from its current computation. Another advantage of this scheme is that the co-processor is likely to respond faster to an incoming message than the main processor because the co-processor is continuously checking for new messages (unless it is handling one). The main processor, on the other hand, only polls at sporadic intervals since it is also doing the regular computation. Message handlers which require special interaction with the on-going computation may still have to be executed on the main processor. Thus we can distinguish two forms of active messages:

those for the main processor and those for the co-processor. If a program does not contain any active messages for the main processor, then the main processor does not need to poll.

We have incorporated the co-processor message handling into the previous implementation of active messages based on remote test-and-set. Buffer management is the same as the previous approach; the only difference is that the thread code on the receiver now manages the receive buffer.

A walk-through of an active message transaction is:

Implementation 3 (Active messages using co-processor)

- *The sender transmits a message as before (It stores a message in OUT-BUFFER; the co-processor reads it and sends it to IN-BUFFER if free).*
- *On the receiver, the co-processor continuously checks IN-BUFFER. If a message is present, it has two choices for how to process it:*
- *If it is a message for the main processor, the co-processor enqueues it onto IN-QUEUE. Else, the co-processor executes the active message handler.*
- *The co-processor then clears the IN-BUFFER so that a new message can arrive.*
- *When the main processor polls, it checks IN-QUEUE. If messages are present, it executes the corresponding message handlers.*

The co-processor is now responsible both for sending and receiving of messages. When a message appears in the outgoing buffer, the co-processor picks it up and sends it to the destination processor. When a message appears in the incoming buffer the co-processor either executes the corresponding message handler itself or forwards the message to the main processor by depositing it into a queue in main memory. The main processor polls on this queue and picks up the message from there. Using the queue we have solved the original problem of ensuring that active messages have a single point of entry. In this case, it is the co-processor which collects messages from the network and inserts them onto a queue for the main processor.

This scheme has the advantage that messages are processed relatively fast and multiple messages may accumulate for the main processor thereby solving the problem encountered in the previous approach where only a single message can be destined to any processor. The co-processor continuously pulls new messages of the input buffer thus resulting in good communication performance, even when the main processor does not poll frequently. We should note a drawback of this scheme. The latency of active messages for the main processor has increased because the co-processor is involved twice, once on the sending and once receiving side. Previously it was only involved on the sending side and the the main processor directly picked up the message from the incoming buffer.[6]

Our implementation on the CS-2 is based on the Meiko event mechanism. This event mechanism automatically enables the execution of a thread on the communications co-processor when a message arrives. Thus the communications co-processor does not have to continually poll the input buffer. Another advantage of this approach is that we can use P input buffers without the overhead of constantly checking them. The thread code automatically executes whenever a buffer is loaded.

Writing active messages for the co-processor probably is too difficult for a casual applications programmer. On the other hand, this approach is very much suited for library and compiler writers which write a fixed number of message handlers and encapsulate them in a library or some language construct.

[6]As in the previous sections, there are various alternatives one can consider on this basic scheme. For example, if latency is a major concern, it is possible to implement a shortcut where the main processor and the co-processor both poll on the incoming buffer from the network, and whoever gets there first processes them appropriately.

For example, the co-processor can support communications protocols such as collective communication, distributed shared memory or cache coherence protocols.

Such is the case for the parallel language Split-C [CDG+93]. Split-C is a C-like language with a number of communication primitives; among them are the data transfer primitives get, put, read, write, and store. Split-C data transfers are kept track of using counters, and Split-C uses active messages to increment and decrement these counters. Essentially all of the active messages in Split-C do nothing more than access memory, add, and subtract. These operations are well suited for the thread co-processor.

5 Performance Measurements

We now discuss the performance measurements of the three active message implementations presented in Section 4, But first we examine the performance characteristics of the underlying architecture, the Meiko CS-2.

5.1 Meiko Primitives

The main processor is a 40 MHz SuperSparc with 3-way superscalar issue achieving a peak performance of 120 MIPS. In contrast, the Elan thread processor is a microcoded unit achieving only 3 MIPS. Thus the main processor is roughly 10-40 times faster than the thread co-processor.

main processor	40 MHz
	120 MIPS
thread co-processor	40 MHz
	3 MIPS
DMA transfer (4-32 bytes), 1-way latency	9.5 μs
main proc issues DMA	2.5 μs
co-processor processes DMA	5.5 μs
network latency	0.5 μs
inputer	1 μs
Thread code	
simulating DMA transfer, 1-way latency	12 μs
open	3 μs
remote test	0.6 μs
remote write block	1 μs
load/store	1-2 μs
instruction cache miss	1 μs

Table 1: *Cost of Meiko primitives.*

Table 1 presents the cost of various operations on the Meiko CS-2. For transferring 4–32 bytes of data, it takes $9.5\mu s$ from the time a processor starts a DMA until the data appears on the remote processor. This time is divided into four components: the time the main processor spends in launching the DMA ($2.5\mu s$), the time the co-processor spends in processing the remote write ($5.5\mu s$), the transmission time in the network ($0.5\mu s$), and time the remote co-processor needs to store the value in memory ($1\mu s$).

It is interesting to compare the time for a DMA write to the time it takes the thread to do the same operation. For the thread processor to simulate a DMA transfer, it must perform the following steps: wait for a signal from the main processor, open a connection to the destination processor, write a 32 byte block of memory to that processor, then close the connection. Using a flag in main memory as the start signal, this process takes $12\mu s$ versus $9.5\mu s$ using DMA write. Table 1 also shows other timings for thread instructions. Most thread assembly level instructions take about $0.3\mu s$ to execute, but special ones such as establishing a connection, take more. The most expensive operation is opening a connection to another processor, which takes $3\mu s$.

We compute the one-way latencies above by halving the round-trip latency involved in transferring a flag between two processors. This roundtrip operation is performed repeatedly so that overhead costs incurred by starting and stopping a timer can be ignored. The timer resides on the Elan co-processor chip and has a clock rate of 200 nano-seconds. In measuring the individual components, most of the timings are derived experimentally by executing the appropriate code fragment within a loop, while some of the timings, such as the remote write and network latency, come from Meiko documentation. Similar methodology is used to compute the remaining numbers in this section.

5.2 Implementation Results

We now discuss the performance of the three implementations of active messages presented in Section 4.

MAM-W (Meiko Active Messages using remote Write):
Based on DMA transfers, with 1 output buffer and P input buffers.

MAM-T (Meiko Active Messages using remote Test):
Based on the thread's remote test with 1 output buffer and 1 input buffer.

MAM-C (Meiko Active Messages using Co-processor):
Based on the same approach as above, but having the receiver's thread co-processor execute the message handler using events.

Table 2 summarizes the results, which are discussed in more detail below. The performance is presented in terms of the parameters of the LogP model [CKP$^+$93].

L : is the latency, or time for a message to travel from a source to a destination processor, disregarding overhead.

o_s : is the sending overhead. This is the time the main processor is involved in sending a message. Time spent by the co-processor in sending a message is added to the latency.

o_r : is the receiving overhead. This is the time the receiving processor spends in processing an incoming message, including the overhead to starting the active message handler. This number is 0 if the co-processor handles the message completely.

g : is the gap, or minimum interval needed between successive messages. In the optimal case, the main processor can send messages out as fast as its overhead and $g = o_s$. However, many implementations cause the gap to be larger. For instance, if the co-processor takes long to process a message or a message must block for available buffer space (on the sender or receiver), the gap may be larger than the send overhead.

one-way is the time it takes from when a processor starts sending a message until the destination processor receives it. This time should be equal to $o_s + L + o_r$.

We make a few comments on Table 2 before examining each implementation in detail. For active messages via DMA transfer, we note the following: The latency is the lowest of all methods: It is essentially the time it takes for the co-processor to perform a DMA plus the time it takes to travel across the network. The receiving overhead depends on the number of processors used. It is this lack of scalability which makes

Method	o_s	L	o_r	one-way	g
plain DMA	2.4	7.0	0.1	9.5	6
MAM-W (remote write), P = 8	2.8	7.0	6.4	16.2	32.4
MAM-W (remote write), P = 32	2.8	7.0	8.8	18.6	37.2
MAM-T (remote test)	1.1	10.5	0.7	12.3	14.2
MAM-C (co-processor)	3.4	28.5	0	31.9	31.9

Table 2: *Comparison of active message implementations (times in μs).*

this scheme unattractive. Because we require roundtrip messages, we see that the gap g is very large: the time for one roundtrip. A variation of this scheme which uses an array of buffers can reduce the gap to roughly the time it takes the DMA engine to process a request.

Active messages via the remote test give the best overhead results. The sending overhead is low, since the main processor must only buffer the data and tell the co-processor to do the rest. The receive overhead is low since only a single input buffer must be checked. However, the latency has increased. This is because the co-processor is handling the transmission of data. The co-processor also affects the gap: another message cannot be sent until the co-processor has finished sending the previous one. The $12.3\mu s$ one-way time compares favorably with the $9.5\mu s$ one-way time of a plain DMA.

The use of the co-processor to receive active messages increases the latency because the reception of a message has been moved to the co-processor. A hidden advantage of this last scheme is that messages can be serviced while the main processor is busy with other tasks.

More detailed timing analysis of the three schemes is found in Tables 3–5.

5.3 Active message implementation using remote writes (MAM-W)

In Table 3, we see the cost of each operation involved in an active message request when active messages are implemented using only remote writes.

Since there is only one send buffer, the sender must wait for it to be free. For the timing in Table 3, we assume it is. After filling the buffer, we setup the parts of the DMA descriptor which vary from message to message, such as the destination processor. Finally, the sender initiates the DMA transfer. The co-processor takes $5.5\mu s$ to transmit the data to the remote processor. The receiver, when polling, checks each input buffer. If a buffer is empty, the total time for that iteration of the loop is $0.1\mu s$; if the buffer contains a message, it takes $5.5\mu s$. The most expensive operation is the reply, because it uses the event mechanism to make sure the DMA processor transmitted the last reply. The total time for poll, assuming only 1 of the buffers contains a message, is $(5.6 + 0.1P)\mu s$. If messages are arriving from all processors — a common occurrence in many parallel algorithms — then we can assume that more than 1 buffer contains a message.

5.4 Active message implementation using remote tests (MAM-T)

In Table 4, we see the costs associated with an active message request involving a remote test performed by the sender's thread processor.

The steps involved on the sending side are similar to the steps in the previous implementation: The sender sets up a buffer and then notifies the co-processor, which does the actual transmission. Here however, the sender's main processor saves time since it doesn't have to setup a DMA descriptor. On the other hand the co-processor code is time consuming. In particular, just waiting for a signal from the main processor takes 3-4μs, because it must test a flag, and if no message is present, temporarily suspend the thread code

send code	wait for free send buffer fill send buffer setup dma launch dma Subtotal	$0.1\mu s$ $0.2\mu s$ $0.1\mu s$ $2.4\mu s$ **2.8** μs
sender's co-processor	dma does remote write	**5.5**μs
	transmission time	**1.5** μs
receive code	check incoming reply buffer for i=0..p-1 check receive buffer i if full, call message handler clear buffer if no reply required send dummy reply Subtotal	$0.1\mu s$ - $0.1\mu s$ $0.3\mu s$ $0.1\mu s$ $5\mu s$ **(5.6 + P $\times$ 0.1)** μs
Total		**(15.4 + P $\times$ 0.1)** μs

Table 3: *Costs involved in an active message request using MAM-W, an implementation involving only remote writes.*

so that other functional units of the co-processor can be scheduled. Loading the flag from main memory, testing, and branching takes approximately $2.5\mu s$, mostly because of the load. The thread break takes $1\mu s$.

When sending, the most expensive operation involves opening a connection to the remote processor, which costs $3\mu s$. After establishing a connection, it tests the remote buffer, then immediately tries to send the data and close the connection. The test instruction on the Meiko is such that if the test fails, no further transactions will be honored. Therefore, the data is not written unless the destination buffer is free. The close instruction returns an acknowledgment indicating whether the entire transmission was successful or not. If the transmission was unsuccessful we perform an exponential back-off, taking at least $3\mu s$, and try the send again. When successful, the co-processor marks the output buffer as free and returns to await another send request.

If we succeed in sending the message the on first attempt, the total time used by the thread code is $9\mu s$. Even though the open, test, and transmit times amount to less than $5\mu s$, the remaining thread code incurs an additional delay of $4\mu s$. The instructions after the remote write do not affect the latency, as they occur in parallel with the transmission of data. However, they affect the gap g, as shown in Table 2.

On the receiver's side, a poll on the main processor checks the incoming buffer. If a message is present, it executes the message handler, then clears the buffer so that a new message can arrive.

The total time to send a message is completely dominated by the thread's code.

5.5 Active message implementation using a co-processor (MAM-C)

Finally Table 5 shows the costs associated with an active message request when the thread handles incoming messages. The sending code is roughly the same as that shown in Table 4, except that we are now using events to trigger the sending of a message. Handling of messages by the co-processor is also triggered by events. The steps involved with the receiving code are similar to those shown in Table 4. The only difference is that the code, including the message reply, is executed on the co-processor. Note, that the

send code	wait for free send buffer	0.6μs
	fill send buffer	0.2μs
	set flag notifying thread	0.3μs
	Subtotal	**1.1** μs
send thread code	wait for launch flag	3.5μs
	(instruction cache miss)	1μs
	open connection	3μs
	remote test	0.5μs
	send message	1μs
	Subtotal (if succeed on 1st try)	**9** μs
	close connection	0.7μs
	if successful, return to wait	1μs
	if close not successful,	-
	pause	3+ μs
	return to open connection	1μs
	transmission time	**1.5** μs
receive code	check incoming request buffer	0.1μs
	if non-empty	-
	copy buffer into registers	0.2μs
	clear buffer	0.1μs
	call message handler	0.3μs
	Subtotal	**0.7** μs
Total		**12.3** μs

Table 4: *Costs involved in an active message request using MAM-T, an implementation involving remote test.*

execution takes much longer, because the co-processor is a lot slower than the main processor.

6 Bulk Transfers

While implementing the bulk transfer functionality of active messages we came across a second mismatch between the current definition of active messages and the CS-2 architecture. The objective of bulk transfers is to maximize bandwidth. This objective is clearly met by the Meiko's DMA transfer, which can transfer data in excess of 40 MB/s. Moreover, the DMA transfer is handled by the co-processor, potentially freeing the main processor for other tasks. Unfortunately, all forms of active message sends (including bulk transfers) are blocking, i.e., logically wait until the message has left the source processor (which does not imply that it already has been received). Thus, to adhere to this semantics we have only two implementation choices: (i) a send can copy the array into a temporary buffer and then the DMA transfers the temporary buffer or (ii) the main processor blocks until the DMA has finished transferring the array.

The first case is inefficient in that it effectively halves the bandwidth, since data is copied multiple times and bulk data transfer on an uncongested network is limited by the available bandwidth on the MBus. Therefore the second case is the only possible solution to obtain maximal bandwidth, but the main processor would idle during a potentially long operation, effectively wasting processor cycles. Note that this dilemma only occurs for bulk writes. For bulk reads, the sender cannot know the state of the requested data (without

send code	wait for free send buffer fill send buffer set event notifying thread Subtotal	$0.6\mu s$ $0.2\mu s$ $2.6\mu s$ **3.4** μs
send thread code	event triggers thread open,send (as in Table 4) set remote event Subtotal (if succeed on 1st try)	$5.5\mu s$ $5.5\mu s$ $2.5\mu s$ **13.5** μs
	transmission time	**1.5** μs
receive thread code	event triggers thread call message handler clear buffer Subtotal	$5.5\mu s$ $7\mu s$ $1\mu s$ **13.5** μs
Total request		**31.9** μs
send reply thread code	cache miss open,send (as in Table 4) set remote event Subtotal (if succeed on 1st try)	$1\mu s$ $5.5\mu s$ $2.5\mu s$ **9** μs
	transmission time	**1.5** μs
receive reply thread code	event triggers thread call message handler clear buffer Subtotal	$5.5\mu s$ $2.8\mu s$ $1\mu s$ **9.3** μs
Total reply		**19.8** μs
Roundtrip		**51.7** μs

Table 5: *Costs involved in an active message request using MAM-C, an implementation involving message handling on thread co-processor.*

the use of other primitives), so the data can be safely transferred since it is the applications programmer's responsibility to ensure that the remote processor does not change the data unexpectedly.

A blocking send is a natural choice when the main processor is involved in inserting the message into the network interface. In the case of the CS-2, however, the best solution would be to have the main processor initialize the DMA, have the DMA processor do the transfer in the background, and at some later stage check for completion. The programmer could then continue on with other work, as long as the data being transferred is not touched until the send is complete.

To alleviate this mismatch future versions of active messages, particularly the Generic Active Messages definition [CKL+94], include non-blocking sends for bulk transfers. Of course this also requires introducing a new synchronization primitive into active messages which allows to test for the completion of a bulk transfer. Notice that the proposed extensions to active messages are still quite different from traditional non-blocking send & receive, as bulk transfers under active messages do not issue a corresponding receive.

The same issue also applies to Split-C bulk puts, which are blocking. The programmer can immediately modify an array after it has been sent with a put. The irony is that Split-C derives its name from split-phase. One starts an operation involving remote transfer, continues with other work, then checks if the transfer is complete at its convenience. However, the current version of bulk put is not entirely split-phase. We propose exposing three different variants of bulk put to the programmer: a blocking version which waits until the put has been completed on the destination processor, the current version which only waits until the put has left the sending processor, and finally a fully non-blocking version.

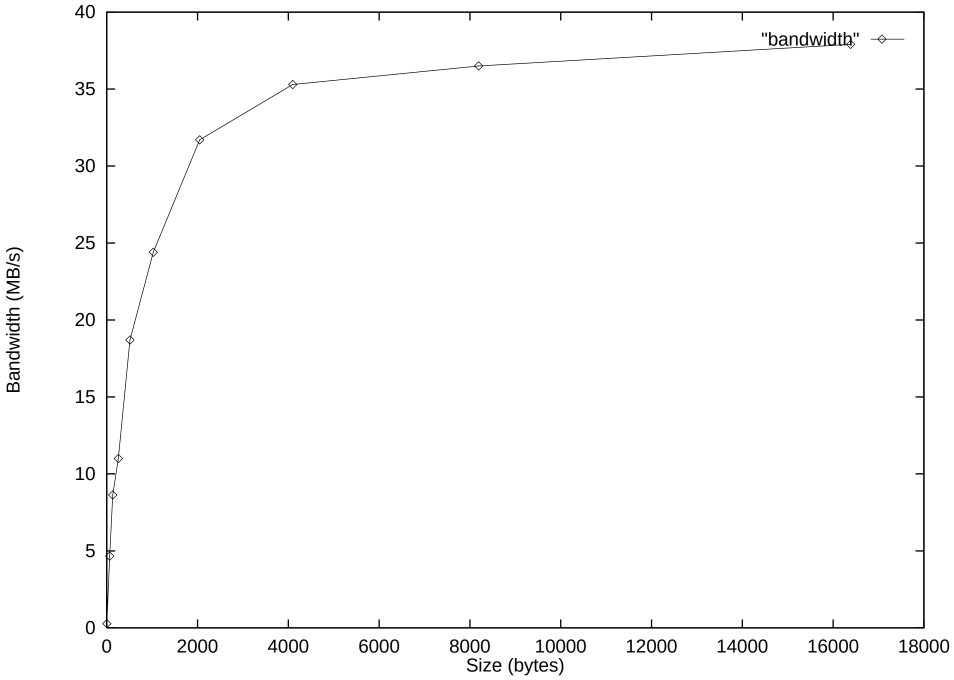

Figure 1: *Bulk transfer bandwidth.*

In our current implementation of bulk put, the main processor has to wait until the DMA engine has sent the message. This implementation does not allow the overlap of communication and computation but achieves the highest bandwidth. From Figure 1, we see that bulk puts, via the active message implementation, achieve a bandwidth of up to 39 MB/second.

7 Related Work

Active messages originated out of the Berkeley TAM group which was looking for efficient communication primitives required for fine-grained parallel languages [CGSvE93]. The first active message implementation was developed for the Ncube/2 and CM-5 and resulted in more than an order of magnitude performance improvement over more traditional communication architectures [vECGS92]. As a consequence, active messages were embraced by applications and systems programmers and were integrated into the CM-5 message library [TM94]. Since then active messages have been implemented on several other hardware platforms, including the Paragon, a cluster of HP workstations connected by FDDI [Mar94], and a cluster of SparcStations connected by ATM [vEABB94]. The last two implementations were especially challenging, since they had to solve the problem of unreliable message delivery.

As previously mentioned, active messages are essentially a very limited form of RPC. Thus one would expect the large body of work on RPC to be directly related to active messages. This is not the case for early research on RPC, which mainly focuses on solving problems in the case of distributed systems and relatively slow communication media (resulting in performance numbers in the millisecond range). Like active messages, RPCs require the transfer of both data and control. Unlike active messages, the control transfer is much more expensive because no limitations are placed on what an RPC may contain. As pointed out by [TLL93], the control transfer of RPC is often unnecessary. These researchers have shown that advances in operating systems research and more modern network interfaces facilitate efficient fully protected remote memory accesses, achieving a communication performance which is competitive with that found on parallel computers [TLL93]. As a consequence they advocate separating the data and control transfer present in RPC [TLL94].

The research of efficient communication architectures is very much driven by the advances in network interface technology. The first generation of network interfaces were pure slave devices and required some involvement of the operating system (on every message or by gang scheduling) to ensure protection. Since the main processor is always involved in message reception, a single entry point is naturally formed and active messages can be implemented efficiently. The next generation goes a step further and allows fully protected user-level remote memory accesses without the involvement of main processor and the operating system (e.g., Cray T3D, Shrimp project [BLA+94]). It is difficult to implement active messages efficiently on these architectures since remote writes alone are not sufficient. It is easier on architectures which provide a freely programmable co-processor to handle communication (e.g., Meiko CS-2, *T [NPA92]). Modern shared memory machines such as the Flash architecture also integrate a fully programmable communications co-processor, but allow only system code to run on it to facilitate protection [Kea94].

8 Conclusions

This paper reports on our experience with implementing active messages on the Meiko CS-2. Active messages provide an extremely simple but efficient communication architecture. They are universal; with active messages one can support any protocol. As pointed out in [TM94], using a single underlying mechanism such as active messages to implement higher level communication protocols is useful, as it naturally allows for multiple co-existing communication primitives without the danger of deadlock or conflicting resource requirements. Many communication protocols (e.g., tagged messages) require a basic functionality similar to active messages. Thus, studying how to implement active messages efficiently is an important problem.

We examined the issues and addressed the difficulties with implementing active messages on parallel architectures with a communications co-processor. Active messages transfer both data and control, requiring the destination processor to be notified upon message arrival. We identified notification as the most difficult problem when the main processor is not directly involved in sending and receiving of messages. Efficiency considerations require a single point of entry from where the main processor can pick up messages. This is difficult to achieve when the communications co-processor autonomously handles all communication and places messages directly in the the remote memory at an address specified by the sender. Given only remote writes it is extremely difficult to obtain efficient active messages for a large number of processors.

Fortunately, the Meiko CS-2 co-processor provides other powerful network transactions besides remote writes which allow us to implement active messages relatively efficiently. We presented three possible implementations of active messages on the Meiko: MAM-W, which uses only remote writes, MAM-T, which uses a special remote test, and finally MAM-C, which uses the communication co-processor on both the sending and receiving side. As our performance measurements show, MAM-W is only suitable for a small number of processors. It also has the disadvantage that every message requires a round-trip acknowledgment. The MAM-T implementation of active messages requires only one-way messages and results in a latency of $12.3\mu s$ (including invoking the message handler), which is close to optimal for the current Meiko hardware and competitive with performance of active messages on other hardware platforms. Finally, the MAM-C implementation increases the latency as seen by the main processor because the co-processor is involved twice, on the sending and receiving side. On the other hand this implementation decreases the overhead of the main processor, especially if the co-processor also executes message handlers.

This is probably the desired solution for implementing something like Split-C. The split-phased nature of Split-C should be able to mask the increase in communication latency, while the lower overhead implies that the main processor will get more computation done. This has also the advantage that remote read requests will probably see better response times because the co-processor constantly polls for incoming messages, unlike the main processor which sometimes may ignore the network due to some large computation.

Besides the notification problem we have also identified another mismatch between the requirements of active messages and the Meiko CS-2 architecture. The CS-2 has a DMA engine which can autonomously handle bulk transfers, thus potentially freeing the main processor for other tasks. Unfortunately, this capability cannot be exploited in the current version of active messages since all forms of active message sends are blocking, i.e., logically wait until the message has left the source processor. To alleviate this mismatch future versions of active messages, particularly the Generic Active Messages definition, provide a non-blocking version of bulk send. In our current implementation of bulk transfer the main processor has to wait until the DMA engine has sent the message. This approach does not allow to overlap communication with computation but achieves the highest possible bandwidth: up to 39 MB/s as our measurements reveal. This number is again close to optimal for the Meiko CS-2 and better than the bandwidth achieved by active message implementations on other architectures.

So what are the right hardware primitives for a network interface in order to support active messages? As our discussion shows, remote writes alone are not sufficient to efficiently implement communication mechanisms such as active messages. The second generation of network interfaces are superior to the first generation because they handle protection in hardware and handle the transfer of data independently of the main processor. On the other hand, they encounter difficulties in implementing the transfer of control required by many communication protocols. To efficiently solve this problem of notification, some mechanism is required which simulates the queue behavior found in first generation network interfaces. We therefore expect the next generation of network interfaces to support this functionality without incurring round trip communication. For example, these network interfaces could provide an indirect write, which takes a message and writes it indirectly through a pointer to a remote memory location, incrementing that pointer.

References

[BLA+94] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, and E. W. Felten. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proc. of the 21st Int'l Symposium on Computer Architecture*, April 1994.

[CDG+93] D. E. Culler, A. Dusseau, S. C. Golstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proc. of Supercomputing*, November 1993.

[CGSvE93] D. E. Culler, S. C. Goldstein, K. E. Schauser, and T. von Eicken. TAM — A Compiler Controlled Threaded Abstract Machine. *Journal of Parallel and Distributed Computing*, 18, July 1993.

[CKL+94] D. Culler, K. Keeton, L. T. Liu, A. Mainwaring, R. Martin, S. Rodrigues, and K. Wright. Generic Active Message Interface Specification. UC Berkeley, November 1994.

[CKP+93] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993.

[HM93] M. Homewood and M. McLaren. Meiko CS-2 Interconnect Elan-Elite Design. In *Proc. of Hot Interconnects*, August 1993.

[Kea94] J. Kuskin and et. al. The Stanford FLASH Multiprocessor. In *Proc. of the 21st Int'l Symposium on Computer Architecture*, April 1994.

[Mar94] R. P. Martin. HPAM: An Active Message Layer for a Network of HP Workstations. In *Proc. of Hot Interconnects II*, August 1994.

[NPA92] R. S. Nikhil, G. M. Papadopoulos, and Arvind. *T: A Multithreaded Massively Parallel Architecture. In *Proc. 19th. Annual Intl. Symp. on Computer Architecture*, May 1992.

[TLL93] C. A. Thekkath, H. M. Levy, and E. D. Lazowska. Efficient Support for Multicomputing on ATM Networks. Technical Report TR 93-04-03, Dep. of Computer Science and Engineering, Univ. of Washington, April 1993.

[TLL94] C. A. Thekkath, H. M. Levy, and E. D. Lazowska. Separating Data and Control Transfer in Distributed Operating Systems. In *ASPLOS*, October 1994.

[TM94] L. W. Tucker and A. Mainwaring. CMMD: Active messages on the CM-5. *Parallel Computing*, 20(4), April 1994.

[vE93] T. von Eicken. *Active Messages: an Efficient Communications Architecture for Multiprocessors*. PhD thesis, Computer Science Division — EECS, U.C. Berkeley, December 1993.

[vEABB94] T. von Eicken, V. Avula, A. Basu, and V. Buch. Low-Latency Communication over ATM Networks using Active Messages. In *Proc. of Hot Interconnects II*, August 1994.

[vECGS92] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proc. of the 19th Int'l Symposium on Computer Architecture*, May 1992.