Measurements of Active Messages Performance on the CM-5

Lok T. Liu
David E. Culler
Department of Electrical Engineering and Computer Science
University of California
Berkeley, CA 94720

Abstract

Active Messages is a versatile and efficient communication architecture that offers significant performance improvement over conventional message-passing approach. The Active Messages architecture achieves its efficiency by carefully integrating the hardware capabilities of the underlying machine with the software layers above it. By taking a minimalistic approach, each Active Messages primitive is highly optimized and therefore its performance is sensitive to the particular design choices being made. In this report, we compare the performance of two implementations of Active Messages on the CM-5, CMAM version 2.7 and CMAML in CMMD version 3.1.final, and investigate how the design trade-offs affect the performance of the different Active Message functions. At the same time, we also develop a new benchmark framework that can be applied to measure the performance of message passing libraries in general.

1.0 Introduction

Many of the current generation of massively parallel computers, such as the Thinking Machine CM-5 and the Intel Paragon, are message-passing multiprocessors. Even though the network hardware in these machines can provide low-latency, high-bandwidth communication, the actual communication performance obtained in real applications has been quite disappointing because of high software overhead of the message passing library. To overcome this problem, Active Messages has been demonstrated to be an efficient communication mechanism on the CM-5 and nCUBE/2 [1]. Originally developed at the University of California at Berkeley as CMAM, Active Messages are now incorporated in the CMAML layer of CMMD 3.1.final, the current version of message passing library available from Thinking Machine Corp. on the CM-5 [2].

In this report, we compare the performance of Active Messages in CMAM 2.7 and CMMD 3.1.final on the CM-5 to investigate how the design choices affect the performance of the two implementations. We also present a new benchmark framework that can be adopted to measure the performance of any message passing libraries. Our benchmark approach differs from previous work in that[3]:

1. Each benchmark is carefully designed by considering the underlying hardware capabilities and the functionalities of the primitive being measured. For example, given the low overhead of Active Messages and the low network latency of the CM-5, we use three different benchmarks to measure the send and receive overheads of the basic Active Message function in an unloaded network in Section 5.1.1.
2. Each benchmark is described in detail so that the results can be easily reproduced.
3. Each benchmark is very simple. Therefore, together with the precise description, it is relatively easy to interpret what exactly we are measuring and draw meaningful conclusions.

The rest of the report is organized as follows. First, we present an introduction on the CM-5 architecture and Active Messages. We also briefly compare the features of CMAM and CMAML. Second, we compare the performance of Active Message functions such as request, reply, and array transfer in both unloaded and loaded network conditions using the LogP model. We find that even though CMAML provides extra functionality such as message interrupts and work queues, it incurs more overhead than CMAM. In an unloaded network, sending a request message takes 1.49 µsec in CMAM whereas it takes 1.96 µsec in CMAML. Node-to-node array transfer to neighboring processor can attain a peak bandwidth of 10 MB/sec in CMAM whereas it is only 8 MB/sec in CMAML. However, in a loaded network, CMAML can provide higher bandwidth than CMAM.

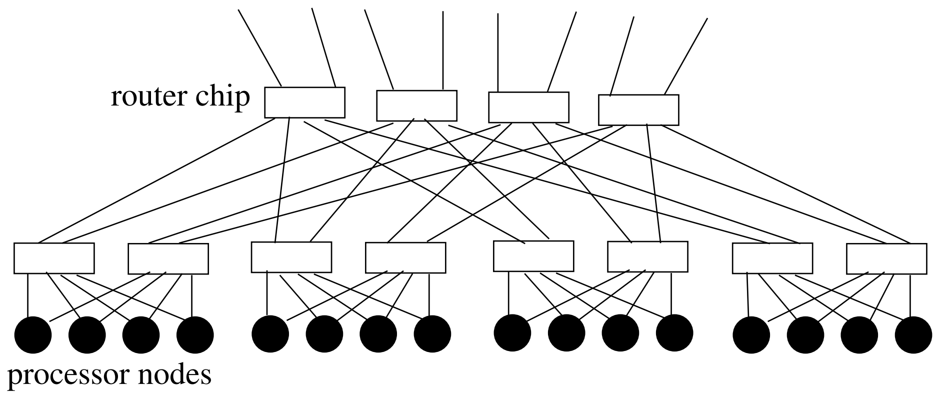
Third, we measure the overhead of receiving Active Messages by polling and by interrupt in detail because messages can only be sent as fast as they are received. Our measurements show that receiving one Active Message per poll has an overhead of 2.85 µsec in CMAM and an overhead of 3.3 µsec in CMAML. However, receiving one Active Message per interrupt has an overhead of about 19 µsec using CMMD 3.1.final and CMOST 7.2.final.

Finally, given the high overhead of message interrupt, we analyze the source code of

CMOST 7.2.final and CMMD 3.1.final to investigate the interaction between the Active Message layer and the operating system during message interrupts. Based on this analysis, we propose an efficient and flexible design for handling message interrupt on the CM-5. Overhead of as low as 9.5 μsec can be attained by a lean implementation.

2.0 Overview of the CM-5

FIGURE 1. The CM-5 data network is an incomplete 4-ary fat tree. The router chips above level 2 have 4 parents instead of 2 parents, which are not shown here.



The CM-5 is a distributed memory multiprocessor that can have up to 16K nodes [4]. The nodes are interconnected by two disjoint data networks, a diagnostic network, and a control network that supports barrier, broadcast, scan and reduce. Each node consists of a 33-MHz SPARC processor chip-set (including FPU, MMU, and 64KB cache), local DRAM, network interface (NI), and optional vector units. The nodes of the CM-5 can be divided into multiple partitions. Each partition consists of a partition manager (also called the host), a set of nodes, and dedicated portions of the data and control networks. The partition manager is a SUN workstation that executes system administration tasks and sequential user tasks. A distinguished feature of the CM-5 is that it allows user access to the network interface through memory-mapped registers and FIFO's in the user address space.

As shown in Figure 1, each of the data network is an incomplete 4-ary fat tree. In general, each router chip is connected to and four child chips and four parent chips to keep the aggregate bandwidth constant from one level to the next. However, due to packing and cost constraints, the tree is incomplete. Each processor node at the leaf level has two connections to the data network. The router chips at the first two levels are connected to two parent chips in the next higher level. All router chips above the second level have four parent connections.

Note that the CM-5 network architecture provides multiple paths from a source processor to a destination processor. To route a message from one processor to another, the message is first sent up the tree to the least common ancestor of the two processors. This is done by randomly selecting an upward connection that is not congested by other messages at each level. Then, the message is sent from the common ancestor to the destination processor via a unique downward path. The random routing algorithm balances the network load and avoids hot spots in pathological cases.

2.1 Overview of Active Messages on the CM-5

2.1.1 The Basics

The mechanism of Active Messages is very simple. The Active Message primitives send messages, which consist of a handler address and the handler arguments. Messages can be received either by polling or interrupts. The handler is invoked with the arguments on the receiving processor upon message arrival. The role of the handler is to assimilate the message from the network in the ongoing computation as fast as possible, rather than performing the actual processing on the message contents. By keeping the resource management and scheduling to a minimum in the Active Message layer, the Active Message primitives are very efficient.

2.1.2 Comparing CMAM and CMAML

Historically, Active Messages was developed at University of California at Berkeley based on the notion of an efficient communication architecture [5]. Acting as a communication abstraction on top of the underlying hardware, Active Messages allows parallel language implementations to access the network hardware with low overhead. CMAM, the Active Messages implementation on the CM-5, is used successfully in the development of the TAM compiler [6]and Split-C library [7]. Based on the Active Messages ideas in CMAM, Active Messages in now incorporated in the CMAML layer of the CMMD message passing library available from Thinking Machines Corp. As a result, CMAM and CMAML bear much resemblance in the surface even though their actual implementations are quite different.

On the CM-5, there are two main flavors of Active Messages: request messages and reply messages. They differ in their usage of the two data networks. The request functions `CMAM_4() and CMAML_request()` send the message using the request (left) data network and poll both the request (left) and reply (right) networks alternately while attempting to send. The reply functions `CMAM_reply_4() and CMAML_reply()` send the message using the reply (right) data network and poll only reply (right) networks while attempting to send. Usually, the request message invokes a handler in the destination, which replies to the sender with a reply message. By following this protocol, deadlock and unbounded handler invocations can be avoided. CMAML offers an extra function, `CMAML_rpc()`, which alternates send requests between request (left) and reply (right) networks and polls both networks alternately while attempting to send. `The CMAML_rpc()` semantics allows Active Messages to be sent in a handler without following the request-reply protocol when the number of RPC invocations can be bounded.

On the CM-5, each of the basic Active Message functions can send messages up to 5 words long. Since each node has the same code image, each node can address the handler on another node by looking up the address of the handler locally.[1] The advantage of having short messages is that the 4 handler arguments can be passed to the Active Message functions through register window on the SPARC with low overhead. However, using the basic Active Message function to transfer an array of data is not efficient because the handler address and the destination memory address would take up 2 of the 5 words.

To support array transfer, CMAM introduces the notion of a segment, which is a memory region on the receiving processor into which other processor(s) can transfer data. A segment is setup by calling CMAM_open_segment(), which specifies the base address of an array, the array size, and an end-of-transfer handler function. Using the segment identifier from the receiver, the sender(s) can then call the functions `CMAM_xfer() or CMAM_reply_xfer() to` transfer data into the segment. `CMAM_xfer()`, like a request function, sends the array data in memory through the request (left) network and polls both networks whereas `CMAM_reply_xfer()` sends and polls only the reply (right) network. The *xfer* messages do not contain a handler address. Instead, they are marked with special hardware tag and therefore, upon arrival on the receiver, a predefined handler in CMAM is called. This special handler stores the data into memory and decrements the segment counter. When the segment counter reaches zero, the end-of-transfer handler function is called at the receiver to synchronize the completion of an array transfer with the ongoing computation. By encoding the segment identifier and the offset into the segment into a word, 4 words is now available for the actual array data in a message.

Instead of segments, CMAML uses receive ports (rport). Setting a CMAML receive port is similar to setting up a CMAM segment. However, instead of calling one function, separate accessor functions in CMAML are needed to be called to specify the base address, buffer size and end-of-transfer handler function. In addition, CMAML also supports an extra synchronization mechanism on the sender by calling an end-of-transfer on the sender after the last byte is sent. There are two array transfer functions in CMAML. `CMAML_scopy()`, which calls `CMAML_rpc()`, sends data on both networks and poll both networks. `CMAML_pcopy()` is a version of `CMAML_scopy()` which transfers data between the parallel memory in the vector units.

In CMAM, the invocation of the handler is synchronous because messages are received either implicitly when sending an Active Message or explicitly by calling a polling function. CMAML differs from CMAM in that it supports asynchronous handler execution as well. `CMAML_request()` can be called even when message interrupt is enabled. If message interrupt is enabled, the arrival of a message will trigger an interrupt. The kernel will then call the CMAML interrupt handler which dispatches the appropriate user handler. Because interrupt is disabled in the interrupt handler, subsequent messages (if any) are received by polling. When there are no more messages to be received, the ongoing computation is resumed. Since interrupt involves context switching[2] between the kernel and the user code, receiving messages by interrupt has higher overhead than by

1. In machines that do not have the same code image on all nodes, indirect Active Messages can be implemented using a handler table on the receiver.

polling. In addition, message interrupt adds to the complexity of the Active Message layer because the Active Message functions need to be re-entrant and the atomicity of the handler execution still needs to be maintained.

Another aspect that CMAML differs from CMAM is that CMAML uses a work queue internally. This is done by assigning higher priorities to the request functions and reply functions than the RPC functions, array transfer functions, handler functions, and regular user functions. If the priority of the function being invoked is lower than the current priority, the function is queued for later execution. This mechanism helps to avoid the deadlock/livelock during send failure especially because the RPC and array transfer functions send and poll on both network.

3.0 Communication Performance Model

In this report, we use the LogP model as our communication performance metric[8]. The LogP model is an machine-independent parallel computation model that is intended to facilitate development of fast, portable parallel algorithms. It captures the essential performance characteristics of the network without describing the structure of the networks. The four parameters in the LogP model are:

L: an upper bound on the latency incurred in transmitting a word (or a few words) from its source to its destination.

o: the overhead, which is defined as the time period during which the processor is engaged in sending or receiving a message; during this time, the processor cannot perform other operations.

g: the gap, which is defined as the minimum time interval between consecutive message transmissions or consecutive message receptions at a processor. The reciprocal of g gives the effective bandwidth per processor.

P: number of processors.

The LogP model also assumes that the network has finite capacity such that there can be at most $\lceil L/g \rceil$ messages in transit from any processor or to any processor. If a processor tries to send a message that would exceed this limit, it will stall until the network resource is available.

For array transfer, we complement the LogP model by a linear model

$$T = c + Bx$$

where T is the total to send or receive an array of size x, c is the start-up cost, and B is the effective bandwidth.

2. On the CM-5, this context switch is relatively light-weighted because the kernel is mapped to the same virtual pages in every process context and there is no change of address space during a message interrupt.

4.0 Benchmark Methodology

The benchmarks are performed on the 64-node partition of the 96-node CM-5 at the University of California at Berkeley running CMOST 7.2.final. In this partition, each node has 8 MB of memory and no vector units. The benchmark programs are written in Split-C 1.2.3 and compiled with the -O2 and -g flag. The results for CMAM 2.7 and CMAML in CMMD 3.1.final are reported in the following sections.

Two kinds of timers are used in the benchmarks. For benchmarks that can finish in a period shorter than 0.5 sec (the time quantum on the CM-5 nodes), the NI timer is used. The NI timer is a 32-bit counter on the network interface (NI) that is incremented every CPU cycle at 32 MHz. On the nodes, it can be read using a load instruction in 7 CPU cycles. However, it does not take into account the effect of context switching and it will wrap around every 130.15 sec. Therefore, for benchmarks that will run longer than 0.5 sec, the Split-C function, `get_seconds()` is used. `get_seconds()` uses the CMNA timers on the nodes, which are virtual timers that has a resolution of 1 µsec and measures only the time a user program is running. However, this timer has higher overhead. Each `get_seconds()` call takes about 7.8 µsec on the nodes. In all benchmarks, the timer overhead is subtracted from the measurements. Unless otherwise stated, all measurements are repeated in a loop until confidence coefficients of at least 90% are obtained. To avoid race conditions among the nodes, each benchmark is ended with a barrier in each loop iteration. In most cases, the standard deviations of the data are not reported because they are actually smaller than 0.5% of the mean values.

5.0 Active Message Benchmarks

In this section, we compare the performance of basic Active Message functions and array transfer using CMAM 2.7 and CMAML in CMMD 3.1.final. We measure the send overheads and receive overheads under both unloaded and loaded network conditions.

5.1 Basic Active Message Functions

5.1.1 Unloaded Network

In this section, we describe benchmarks and their results involving communication between 2 or 3 processors. In this case, the network is not fully utilized.

5.1.1.1 Benchmarks

In each benchmark, Active Messages are received by polling using `CMAM_wait()` or `CMAML_wait()`. Each Active Message invokes a simple handler that increments a counter in memory. For node-to-node communication, four types of benchmarks are run:

- *one-to-one*

  In this benchmark, we measure the average time to send a message from one processor to another processor by pipelining the sends across the network. This is done by sending 1024 messages in a loop and wait for a reply message from receiving processor to acknowledge that it has received all 1024 messages. Note that this measurement may include the time to send a message repeatedly when the receiving processor does not process the messages fast enough so that the network is congested.

- *one-to-two*

  This benchmark is similar to the one-to-one benchmark, except that we send messages from one processor to two other processors alternately in a loop. By sending to two different processors, the problem that the receiving processor cannot process the messages fast enough is avoided. This gives a better measurement on the send overhead of the Active Message functions.

- *two-to-one*

  In this benchmark, we measure the average time to receive an Active Message by constantly polling. This is done by first requesting two other processors to send 1024 messages each to node 0. We then measure the time for `CMAM_wait()` or `CMAML_wait()` to receive the 2048 messages on node 0. This benchmark gives the receive overhead. Notice that this measurement includes the time to execute the simple handler.

- *round-trip*

  This benchmark measures the time between a request message is sent to another processor and a reply message is received from that processor as acknowledge. This measurement includes the send and receive overheads plus the round-trip network latency. This corresponds to $T = 2L + 4o$ in the LogP model.

To measure the effect of spatial communication locality, we run each benchmark with different source and destination nodes across the partition. For node-to-host communication and host-to-node communication, only the one-to-one benchmarks are run. The NI timer is used for timing in all benchmarks.

5.1.1.2 Results and Discussions

5.1.1.2.1 Node-to-node

Table 1: Benchmark results for node-to-node Active Messages. Note that only the results between node 0 and node 1 are reported here.

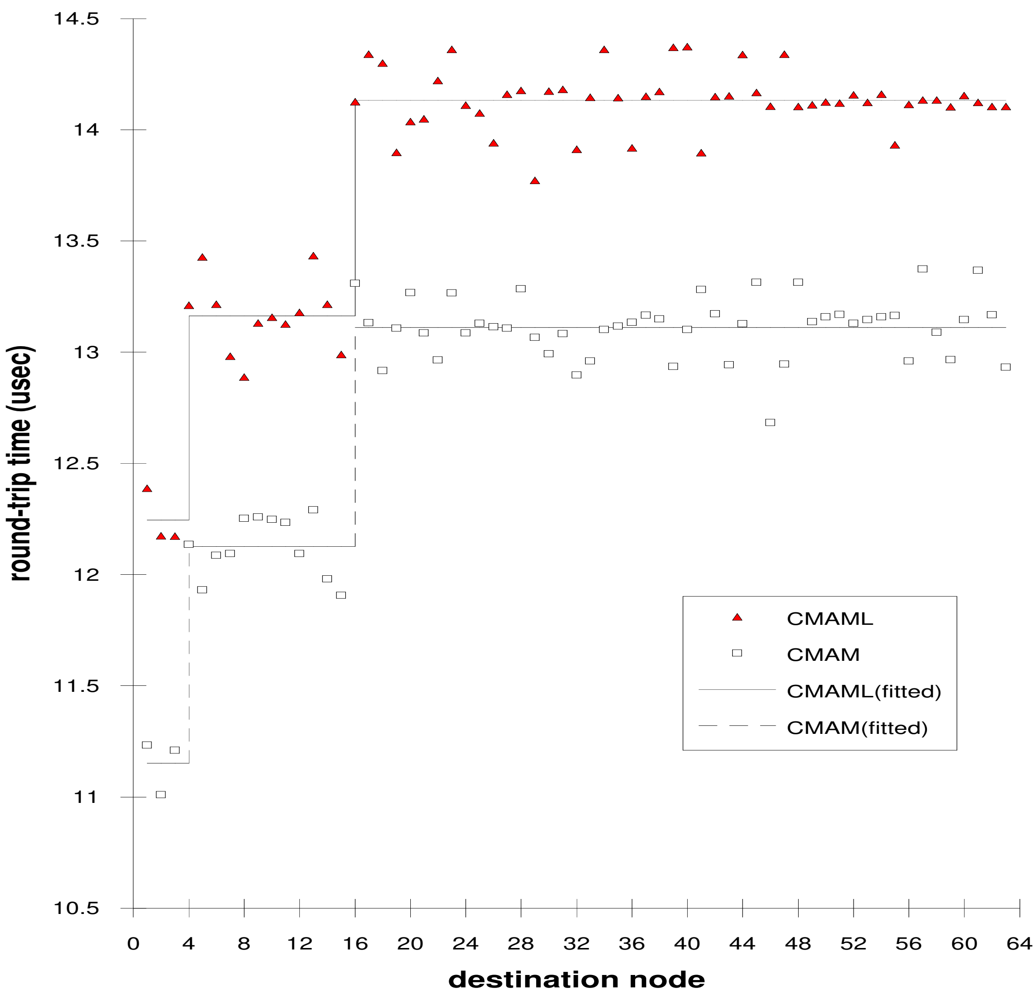
	CMAML			CMAM		
Benchmark	**function**	**cycles/op**	**μsec/op**	**function**	**cycles/op**	**μsec/op**
one-to-one	CMAML_request	64.0	2.00	CMAM_4	50.2	1.57
one-to-two		62.7	1.96		47.7	1.49
two-to-one		64.3	2.01		51.2	1.60

	CMAML			CMAM		
Benchmark	function	cycles/op	μsec/op	function	cycles/op	μsec/op
one-to-one one-to-two two-to-one	CMAML_reply	65.3 41.0 66.6	2.04 1.28 2.08	CMAM_reply_4	49.9 40.0 51.2	1.56 1.25 1.60
one-to-one one-to-two two-to-one	CMAML_rpc	60.8 40.0 61.8	1.90 1.25 1.93			
round-trip (node 0 to node 1 and back)		402.9	12.59		364.8	11.40

The results for node-to-node Active Messages are shown in Table 1. For the one-to-one, one-to-two, and two-to-one benchmarks, since we are pipelining the message across the network, the results are only slightly affected by the network latency. Therefore, only the results between node 0 and node 1 are reported in Table 1 for these benchmarks. For the round-trip benchmark, the contribution of the network latency to the timing is illustrated in Figure 2. In all benchmarks, CMAM is more efficient than CMAML. The extra functionality in CMAML, such as error checks, message interrupts and work queues, contribute to the extra overhead. The results for the one-to-one benchmarks is very close to that for the two-to-one benchmarks. However, the send overheads measured in the one-to-two benchmarks are smaller than that measured in the one-to-one benchmarks. This reflects the fact that the message send rate is limited by the message reception rate at the destination processor in the one-to-one benchmark. Reply messages has lower send overhead than request messages because only the reply (right) network is polled when sending a reply message.

From Figure 2, we can see that the curves for the round-trip time are like step functions even though the data is quite noisy and the standard deviation is as high as 0.7 μsec for some data points. The first jump of the curve is at destination node 4 and the second jump is at node 16. This is exactly what we would expect from the 4-ary fat-tree network. Each jump is about 1 μsec, which indicates that the delay through one router chip is about 0.25 μsec. The round-trip time for CMAML is about 1 μsec higher than that for CMAM because CMAML has higher send and receive overheads for both the request and reply functions. Subtracting the send and receive overheads from the results of the round-trip benchmark, we can deduce that the round-trip network latency between node 0 and node 63 is 6.8 μsec using the data from CMAML and 7.17 μsec using the data from CMAM.

Figure 2. Round-trip times from node 0 to node 1 through node 64. Note that the round-trip time is the sum of the send and receive overheads and the round-trip network latency



5.1.1.2.2 Node-to-host

Table 2: One-to-one benchmark results for node-to-host Active Messages

CMAML function	cycles/op	µsec/op	CMAM function	cycles/op	µsec/op
CMAML_request_tohost	12774	399.2	CMAM_host_4	12784	399.5
CMAML_reply_tohost	12461	389.4	CMAM_host_reply_4	12589	393.4
CMAML_rpc	12438	388.7			

The results for node-to-host Active Messages are shown in Table 2. Since there is a high variation in the timing, the average time to send a message reported here is calculated by sending 1024 messages instead of repeating the measurements until a 90% confidence coefficient is reached. In this benchmark, there is no significant difference between the results for CMAML and that for CMAM. Sending reply or RPC messages is still faster then sending request messages. However, the time to send an Active Message from one node to the host is 200 times longer than from one node to another node. There are two reasons for the poor performance:

1. Since the host lies outside the address space of the partition it manages, it can only be addressed using its physical data network address, rather than its relative address in the partition. This requires a system call into the CMOST kernel on the nodes and adds to the overhead.
2. All CM-5 programs comprises of two parts: the host program that runs on the host and the node program that runs on the nodes. Even though the node program is scheduled synchronously (coscheduled) on the nodes, there is no coordination between the scheduling of the host program and the node program. As a result, if the host program is not scheduled on the host when a node tries to send messages to the host, that node is simply blocked because the host is not receiving messages and the network link is congested. This adds to the indeterminacy in the timing for node-to-host communication.

5.1.1.2.3 Host-to-node

Table 3: One-to-one benchmark results for host-to-node Active Messages

CMAML function	cycles/op	μsec/op	CMAM function	cycles/op	μsec/op
CMAML_request	155.8	4.87	CMAM_4	160.3	5.01
CMAML_reply	138.2	4.32	CMAM_reply_4	139.2	4.35
CMAML_rpc	141.1	4.41			

The results for host-to-node Active Messages are shown in Table 3. The average time to send a message reported here is calculated by sending 1024 messages instead of repeating the measurements until a 90% confidence coefficient is reached. We can see that the time to send an Active Message from the host to the node is about 3 times slower than between the nodes. This is because the network interface is accessed through the S-Bus on the host rather than through the M-Bus as on the nodes. The S-Bus runs at the half the CPU clock rate and is only 32-bit wide. In comparison, the M-Bus can run at the same clock speed of the CPU up to 40 MHz and is 64-bit wide. Moreover, data need to pass through extra bus adaptor to access the NI on the host and this adds to the overhead. For example, the NI timer register can be read in 7 cycles on the nodes. However, this will take about 21.4 cycles on the host.

5.1.2 Loaded Network

In this section, we present results for benchmarks in which all processors are sending and receiving at the same time.

5.1.2.1 Benchmarks

To get an idea on how the Active Message functions perform in a loaded network, we use the following benchmarks:

- *ring*

  In this benchmark, we arrange the processors into a linear ring. Each node is to send 1024 messages to its right neighbor while receiving 1024 messages from its left neighbor. This is done by calling `CMAM_4()` or `CMAML_request()` 1024 times in a loop and then call the `CMAM_poll_wait()` or `CMAML_poll_wait()` to make sure that all 1024 messages from the left neighbor are received. We measure the average time for node i to send a request message to node (i+1) mod P while receiving a message from node (i-1) mod P, where P is the total number of nodes. The communication pattern is very localized in this benchmark.

- *traverse*

  This benchmark consists of P steps, where P is the total number of nodes. In step i, node j and node ($j \oplus i$) exchange 1024 messages with each other, where ($j \oplus i$) means j exclusive-or i. In a hypercubic network, the communication pattern at step i can be interpreted as a permutation in which a packet is routed from node j to node ($j \oplus i$) by traversing the dimensions of the hypercube in which there is a 1 at that bit position in the binary value of i. For the incomplete 4-ary fat tree of the CM5, in step i, a packet needs to travel up to the common ancestor in level $\lfloor \log_4 i \rfloor + i$ of the network before it follows the downward path to the destination node. Again, we measure the average time to send and receive a request message in this benchmark. This benchmark measures how the hierarchical structure and bisection bandwidth of the network affect the performance.

Note that we are measuring the sum of the gap (g) and the receive overhead (o) of the LogP model in these benchmarks. Polling is used in both benchmarks.

5.1.2.2 Results and Discussions

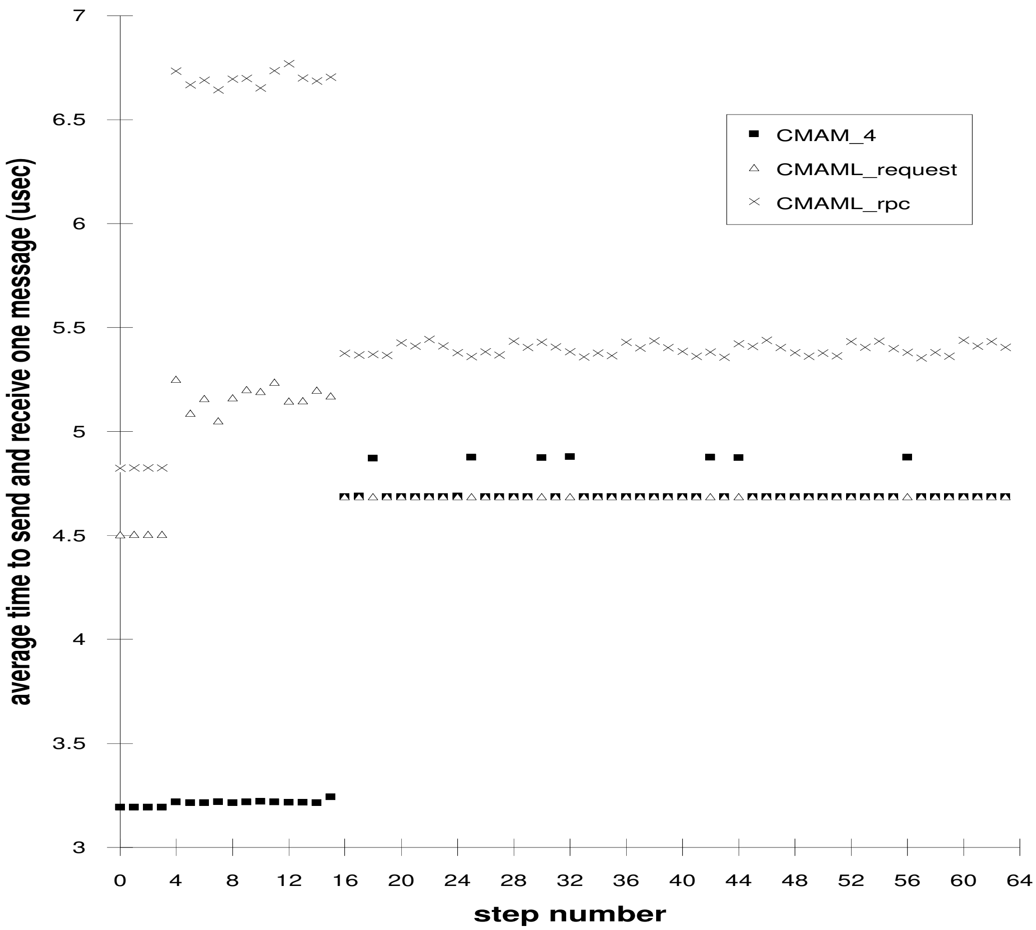
5.1.2.2.1 ring

Table 4: Results for the Active Messages ring benchmark

function	measured (μsec/op)	expected (μsec/op)
CMAM_4	3.18	3.09
CMAML_request	4.59	3.97

In this benchmark, the average times to exchange a request message are very close among the nodes and the average time among the nodes are reported in Table 4. Since the communication is very localized, we would expect the measured time to be the sum of the send overhead and receive overhead we obtain in Section 5.1.1.2.1. Indeed, the measured times are pretty close to the expected time. However, the deviation for CMAML is greater than that for CMAM probably because CMAML is not fast enough to handle the messages and the network starts to be congested.

Figure 3. Results for the Active Messages traverse benchmark



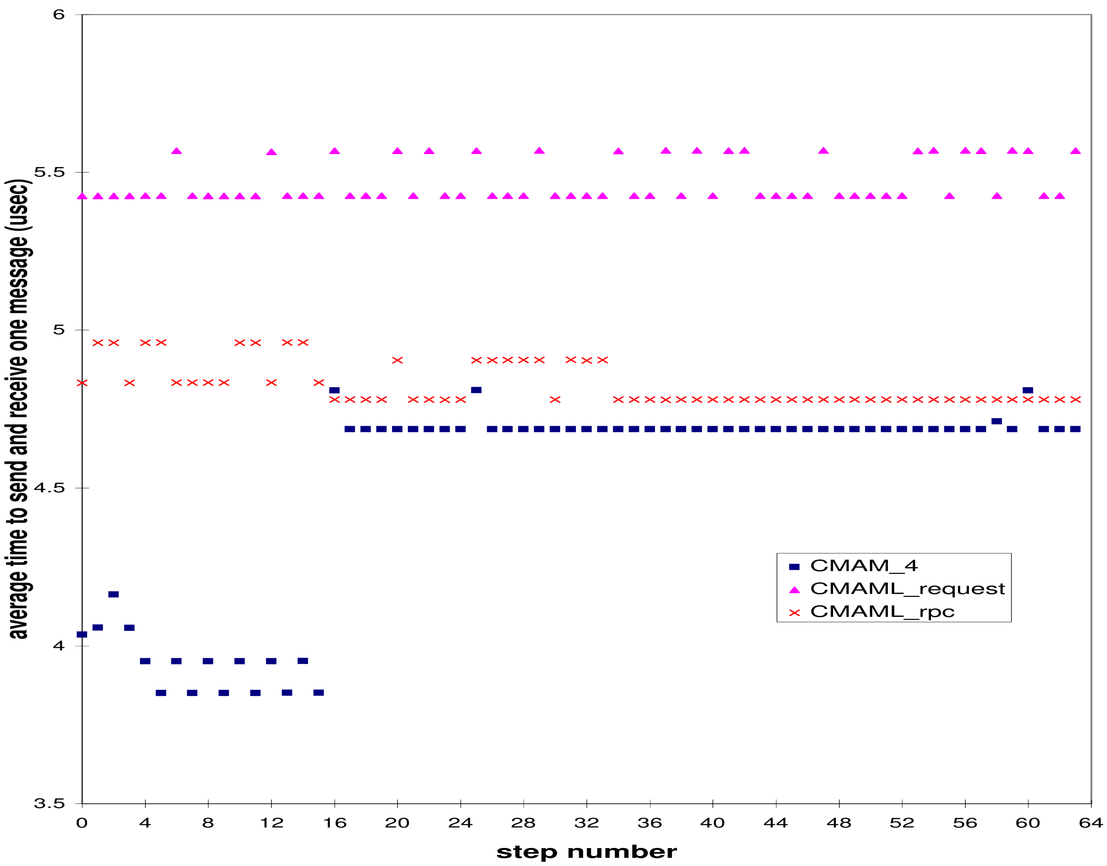
5.1.2.2.2 traverse

In this benchmark, the variation of the average times to exchange a message is quite high among the nodes. For some data points the standard deviation is as high as 0.5 µsec. The 0.5 µsec standard deviation is expected because, as we will see in Section 6.1.3.1, each iteration in the dispatch loop is roughly 1 µsec and, on average, a message will arrive at the midpoint of the loop. In Figure 3, the average time to exchange a message among the nodes in each step is plotted against the step number for the different functions. It can be seen that the CMAM and CMAML functions behave differently. For `CMAM_4()`, the curve is like a step function with a jump of about 1.4 µsec at step 16. For CMAML, even though `CMAML_rpc()` has lower overhead than `CMAML_request()` in a unloaded network, its performance is worse than `CMAML_request()`. Nevertheless, the curves for both CMAML_request() and CMAML_rpc() exhibit an interesting behavior. Between step 0 and step 3, their curves are quite flat. Then, there is a jump at step 4 in both curves and the curves stay quite flat between step 4 and step 15. For CMAML_rpc(), the jump is about 1.9 µsec, which is much higher than the jump of about 0.7 µsec for CMAML_request(). Interestingly, both curves jump back down at step 16 and stay flat between step 16 and step 63 around a value which is still higher than that between step 0 and step 3. In fact, the average time to exchange a message is almost the same for `CMAM_4()` and CMAML_request() between step 16 and step 63.

In general, we would expect the measured time to increases with the step number because more levels of the network need to traversed and the nodes need to compete for the limited bandwidth between each level of the network. Since `CMAM_4()` has low receive overhead, there is no severe backlog from the network between step 0 and step 15 and therefore our measurements are very close to the sum of the send overhead and receive overhead. From step 16 and on, bandwidth limitation comes into place and the network is congested. Hence, the network backlog causes a jump in the average time to send a message at step 16. However, for CMAML_request() and CMAML_rpc(), it is not clear why the measurements between step 4 and step 15 have higher values than those between step 16 and step 63. In addition, it is also not clear why CMAML_rpc() has a higher jump than CMAML_request() at step 4. This may be related to the fact that CMAML_rpc() has much lower send overhead than CMAML_request(). Therefore, CMAML_rpc() may keep attempting to re-send the packet at a higher rate when the network is congested. This, in turn, can aggravate the network congestion. Since the interaction between the sender and the receiver is highly coupled in the CM-5 network, the phenomena we observe in Figure 3 may be caused by the particular way in which messages are received in the request and RPC functions. To probe the matter further, we insert an extra polling function call in the inner loop of the traverse benchmark and the results are shown in Figure 4. Comparing the results in Figure 3 and Figure 4, the following observations can be made:

1. For `CMAM_4()`, in steps 0 through 15, the average time to send and receive one message in Figure 4 is higher than that in Figure 3 by about 0.8 µsec, which is approximately the time for an unsuccessful `CMAM_poll()` that we will show in Section 6.1.3.1. In steps 16 through 63, both benchmarks have the same results. This seems to confirm the hypothesis that the performance of `CMAM_4()` is limited by the network bandwidth in this region.
2. For `CMAML_request()` and `CMAML_rpc()`, the jumps in steps 4 through 15 disappear in Figure 4. In Figure 4, the graph for `CMAML_request()` is relatively flat at around 5.4 µsec, which is almost the same as the value for `CMAML_rpc()` between steps 16 and 63 in Figure 3. This value is also about 0.8 µsec higher than that for `CMAML_request()` in Figure 3 between steps 16 and 63. However, as we will see in Section 6.1.3.1, the time for an unsuccessful `CMAML_poll()` is about 1.3 µsec. Another interesting point is that, in Figure 4, the graph for `CMAML_rpc()` is relatively flat at 4.68 µsec, which is almost the same value for `CMAM_4()` between steps 16 and 63 and almost the same as its initial value in Figure 3. It seems that by adding an extra poll to slow down the send rate and poll both networks has helped to reduce the congestion in this case.

Figure 4. Results for the Active Messages traverse benchmark with an extra polling function call in the inner loop



5.2 Array Transfer Functions

In this section, we measure the performance of bulk data transfer using Active Messages in both unloaded and loaded network conditions.

5.2.1 Unloaded Network

5.2.1.1 Benchmarks

In this section, we measure the performance of array transfer using `CMAM_xfer()` in CMAM and `CMAML_scopy()` in CMAML by transmitting arrays of sizes from 2 bytes to 64K bytes. We also vary the alignment of the destination buffer addresses among byte, word, and double-word boundaries. Since the SPARC processor has a 64KB unified cache, we would expect our results not to be affected by the cache misses in most cases. However, we do not study how the cache performance affects the data transfer rate in these benchmarks. For node-to-node and node-to-host array transfer, the Split-C function, `get_seconds()` is used for timing. For host-to-node array transfer, the NI timer on the host is used. The results exclude the time to allocate receive ports in CMAML or to allocate segments in CMAM. Polling is used in all benchmarks.

For node-to-node communication, four types of benchmarks are run:

- *one-to-one*

  This measures the average time to send an array of size x from node 0 to node 1. For blocking send, the result includes the time to wait for a reply message from node 1 to signal that the whole array is received. For non-blocking send, we stop the timer once the array transfer function returns. Then, we wait for an acknowledge from node 1 before we proceed to send the next array.

- *two-to-one, one segment*

  This measures the average time to receive an array of size x by having node 1 and node 2 sending to two different halves of the same buffer on node 0 at the same time, i.e. same segment or same receive port.

- *two-to-one, two segments*

  This measures the average time to receive two arrays of size x/2 by having node 1 and node 2 sending to two different buffers on node 0 at the same time, i.e. two different segments or receive ports.

- *swap*

  This measures the average time for two processors to swap an array of size x simultaneously. In the CMAM version of the benchmark, node 0 requests node 1 to use `CMAM_reply_xfer()` to send it an array before it starts to send an array to node 1 using `CMAM_xfer()`. We measure the total time for node 0 to send an array to node 1, receive an array from node 1, and wait for an acknowledgement from node 1 that it has received the array. In the CMMD version of the benchmark, since there is no reply version of `CMAML_scopy()`, `CMAML_scopy()` is used on both node 0 and node 1.

For node-to-host and host-to-node array transfer, only the non-blocking one-to-one benchmark is run. In these cases, we are transmitting data between node 0 and the host.

5.2.1.2 Results and Discussions

5.2.1.2.1 Node-to-node

Table 5: Benchmark results for node-to-node array transfer

		CMAM			**CMAML**		
Benchmark	**Alignment**	**4 words (μsec)**	**start-up cost (μsec)**	**bandwidth (MB/sec)**	**4 words (μsec)**	**start-up cost (μsec)**	**bandwidth (MB/sec)**
one-to-one (blocking)	byte	21.6	28.7	5.76	33.1	40.6	3.81
	word	21.6	30.0	8.42	31.9	38.5	8.04
	double-word	23.8	21.1	10.11	31.7	30.1	8.12

Benchmark	Alignment	CMAM 4 words (μsec)	CMAM start-up cost (μsec)	CMAM bandwidth (MB/sec)	CMAML 4 words (μsec)	CMAML start-up cost (μsec)	CMAML bandwidth (MB/sec)
one-to-one (non-blocking)	byte	8.6	13.1	5.76	19.8	25.0	3.81
	word	8.6	12.7	8.43	19.7	21.0	8.12
	double-word	10.7	8.0	10.09	19.8	10.6	8.09
two-to-one, one segment	byte	17.8	35.4	10.06	22.5	31.6	7.63
	word	18.5	36.5	10.06	22.5	32.3	8.07
	double-word	18.9	16.6	10.12	22.7	19.7	8.12
two-to-one, two segments	byte	20.1	35.9	7.31	26.7	42.6	6.98
	word	20.2	33.9	7.36	26.7	31.3	6.94
	double-word	20.9	14.7	7.39	26.6	24.0	7.11
swap	byte	21.2	32.0	2.98	38.8	49.7	2.81
	word	21.0	46.7	4.48	37.0	57.2	5.34
	double-word	31.2	25.0	5.06	37.4	59.8	5.66

The results for node-to-node array transfer are shown in Table 5. For each benchmark, we show the time to send or receive 4 words so that we can compare the overhead of the array transfer functions with that of the basic Active Message functions. We also report the start-up cost and bandwidth from a least-square fit of the data points using a linear model. Note that the start-up cost value from the linear model is affected by the non-linearity and the noise of the data. Therefore, it may not reflect the actual start-up cost accurately.

- *Overhead for array transfer*

  From the results of the non-blocking one-to-one benchmark and the two-to-one benchmark, we can see that the overhead of the array transfer functions is much higher than that of the basic Active Message functions. For the basic Active Message functions, all arguments can fit into a register window in the SPARC processor. Therefore, the compiler can pass the arguments to the Active Message functions and the handler functions through registers instead of memory. However, for array transfer, data has to be copied from memory to the NI at the sender and again from the NI back to memory at the receiver. In addition, on the sender side, the array transfer functions need to packetize the array into 5-word messages and to handle data at different alignments. This slows down the send rate. On the receiver side, the segment or receive port data structure also needs to be accessed to figure out where to store the data in memory. The extra overhead in maintaining the segment or receive port data structure also slows down the receive rate.

- *Comparing the overhead of CMAM and CMAML*

  `CMAML_scopy()` also has higher overhead than `CMAM_xfer()`. One reason is that `CMAML_scopy()` requires 8 arguments which cannot fit into a register window. Therefore, the compiler has to pass the arguments through the stack rather than registers. The extra functionality that a handler function can be invoked when the transfer is completed at the sender incurs a cost of 2 extra arguments for `CMAML_scopy()`. In addition, we can also observe that CMAML can only achieve a peak bandwidth of about 8 MB/sec, which is 20% lower than the 10 MB/sec bandwidth of CMAM. This is due to the fact that the message dispatch loop in the polling function and the data packet handler have higher instruction counts in CMAML than those in CMAM. Note that this 4:5 ratio can also be observed in the receive overhead measured in Section 5.1.1.2.1 for the basic Active Message functions.

- *Effect of buffer address alignment*

  The relative alignment of the source buffer address and destination buffer address also affects the start-up cost and the peak bandwidth. If both source and destination buffer addresses are double-word aligned, double-word loads and stores can be used and the array can be easily transferred 16 bytes at a time. Therefore, the start-up cost is usually lower for double-word aligned buffer and the peak bandwidth can be obtained.

- *Blocking vs. Non-blocking*

  Comparing the results for the blocking and non-blocking benchmarks, the attainable bandwidths are the same. However, the non-blocking sends have lower latency than the blocking sends because there is no need to wait for an acknowledgement in the non-blocking case. The difference in latency between the blocking case and non-blocking case is about the round-trip time we measure in Section 5.1.1.2.1.

- *Comparing the two-to-one benchmark results*

  To avoid the higher latency to access the segment or receive port data in memory, both CMAM and CMAML cache these data in registers, assuming that packets destined for the same segment or receive port is likely to arrive in a contagious stream. In the two-to-one, 2 segments benchmark, the receiving processor needs to demultiplex the incoming packet stream into two different segments or receive ports. As a result, the caching scheme becomes less effective and there is higher overhead than in the two-to-one, 1 segment benchmark. The receive bandwidth in CMAM drops from 10 MB/sec to 7.3 MB/sec when two segments are used, which is comparable to that for CMAML. The packet receiving code in CMAM is highly optimized for the case that there is only one sender.

- *Swapping data*

  In this case, since the processor is sending while receiving, the bandwidth drops from 10 MB/sec to 5 MB/sec in CMAM. However, in CMAML, the bandwidth drops less significantly and is higher than that for CMAM because `CMAML_scopy()` uses both networks to send data.

5.2.1.2.2 Node-to-host

Table 6: Non-blocking one-to-one benchmark results for node-to-host array transfer

Alignment	CMAM			CMAML		
	4 words (µsec)	**start-up cost(µsec)**	**bandwidth (MB/sec)**	**4 words (µsec)**	**start-up cost(µsec)**	**bandwidth (MB/sec)**
byte	425	505	0.0397	421	1040	2.49
word	424	341	0.0403	421	1001	3.17
double-word	430	391	0.0405	421	77	3.36

The results for node-to-host array transfer are shown in Table 6. In this set of benchmarks, the data for CMAML in the double-word aligned case fails to reach the 90% confidence coefficients before 65536 iterations are run. Therefore, the average results from the 65536 iterations are shown for the double-word aligned case. We can see that the latency to send a 4-word is almost the same for both CMAM and CMAML. In fact, it is comparable to that for sending a request Active Message to the host. However, the bandwidth for CMAML is two-order of magnitude higher than that for CMAM.[3] The start-up costs for CMAML in the byte aligned and word aligned cases are 2 or 3 times higher than those for CMAM. However, the start-up cost in the double-word aligned case in CMAML is suspiciously low. An examination of the raw data reveals that the latency drops from 439 µsec for an array of 8 bytes to 56 µsec for an array of 16 bytes and then increases monotonically. The non-linearity of the data leads to an inaccurate start-up cost estimate from the linear model. The S-Bus on the host still limits the bandwidth in CMAML.

5.2.1.2.3 Host-to-node

Table 7: Non-blocking send benchmark results for host-to-node array transfer

Alignment	CMAM			CMAML		
	4 words (µsec)	**start-up (µsec)**	**bandwidth (MB/sec)**	**4 words (µsec)**	**start-up (µsec)**	**bandwidth (MB/sec)**
byte	7.5	48	2.6	21.7	35.2	2.37
word	7.4	45	3.0	21.7	38.3	3.58
double-word	9.7	33	3.3	21.7	14.8	3.62

The results for host-to-node array transfer are shown in Table 7. In this case, the time to send a 4-word array is larger than that to send an Active Message because of the extra

3. In CMMD 3.0, the node-to-host bandwidth is almost the same as that for CMAM, which is about 40 KB/sec.

overhead to access the segment or receive port. CMAML achieves a slightly higher bandwidth than CMAM. The S-Bus on the host is still the bottleneck.

5.2.2 Loaded Network

5.2.2.1 Benchmarks

In this section, we use the ring and traverse benchmarks from Section 5.1.2 to measure the performance of array transfer in a highly utilized network. In this case, instead of sending individual Active Messages, we measure the time to exchange arrays of size 2 bytes to 64 KB and fit the data into a linear model to obtain the effective bandwidth.

5.2.2.2 Results and Discussions

5.2.2.2.1 ring

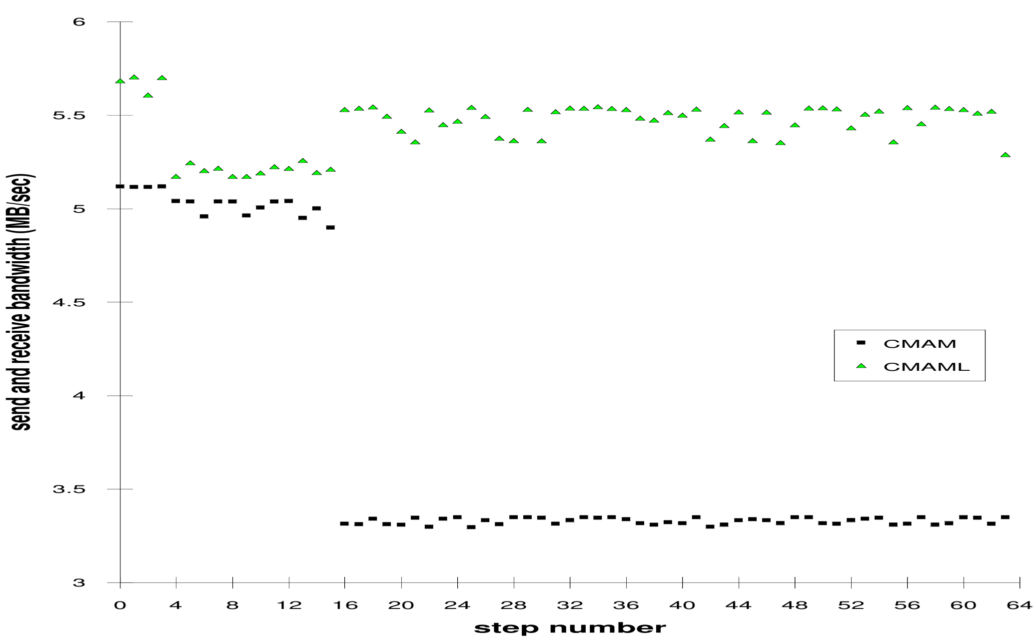
Table 8: Results of the ring benchmark for array transfer

function	bandwidth (MB/sec)
CMAM_xfer	5.02
CMAML_scopy	5.7

Since the communication is very localized in this benchmark, the measured bandwidth obtained here is almost the same as that for the swap benchmark in Section 5.2.1.2.1.

5.2.2.2.2 traverse

Figure 5. Results of the traverse benchmark for array transfer



In this benchmark, we can see that CMAML outperforms CMAM. Since `CMAML_scopy()` calls `CMAML_rpc()`, which sends data down both networks, its performance is limited by the overhead to send and receive the data packets, rather than by the network bisection bandwidth. Therefore, the effective bandwidth stays above 5.4 MB/sec in most steps, except that in step 4 through step 15, the bandwidth dips a little bit to about 5.2MB/sec. This echoes the results we obtain in Section 5.1.2.2.2 for `CMAML_rpc()`. For `CMAM_xfer()`, its low overhead enables it to sustain bandwidth above 5 MB/sec in step 0 through step 15. From step 16 and on, its performance is limited by the network bisection bandwidth and drops to about 3.3 MB/sec.

6.0 Message Reception Benchmarks

Since messages can only be sent as fast as they can be received, the overhead to receive a message is as important as that to send a message in determining the performance of Active Messages. In this section, we measure the overhead to receive messages using polling and interrupt in details.

6.1 Polling

6.1.1 Basic Polling Mechanism

Active Messages can be received by explicitly checking the status bits on the NI. If a message is present, the message is read in and the corresponding handler is invoked. This process is repeated until no more messages are available. The basic functions that implements this dispatch loop are `CMAM_poll()` and `CMAML_poll()`, which poll both the request (left) and reply (right) data networks. The functions `CMAM_request_poll()` and `CMAML_request_poll()` poll only the request (left)data network while the functions `CMAM_reply_poll()` and `CMAML_reply_poll()` poll only the reply (right) data network. `CMAM_wait()` and `CMAML_wait()` are variants of `CMAM_poll()` and `CMAML_poll()` that poll until a flag is set to a certain value. `CMAM_poll_wait()` and `CMAML_poll_wait()` differ from `CMAM_wait()` and `CMAML_wait()` in that they poll at least once before checking the flag.

6.1.2 Benchmarks

In this section, we measure the overhead of both unsuccessful polls and successful polls. To measure the overhead of an unsuccessful poll, we make sure that no processors are sending messages and measure the average time to call the polling functions 1024 times. Essentially, this measures the time for the polling functions to poll the network once or twice and return to the caller because no message is received. The NI timer is used for timing in this benchmark.

To measure the overhead of a successful poll, we send 1024 messages from node 1 to node 0. Each message invokes a simple handler that increments a counter in memory. Since multiple messages can be received in one call to the polling function, we vary the time intervals between which a message is sent by inserting delay loop between each call to `CMAM_4()` or `CMAML_request()`. We record the time to receive the messages and the number of calls to `CMAM_poll()` or `CMAML_poll()` required to receive the messages. To interpret the data, we amortize the time to receive the messages using the number of messages received and using the number of calls to the polling functions. By plotting the average time per message and average time per poll against the message send interval, the intersection of the two curves will give us the time to successfully receive one message per poll. Notice that this measurement includes the time to execute the simple handler. `get_seconds()` is used for timing in all these benchmarks.

6.1.3 Results and Discussions

6.1.3.1 Unsuccessful poll

Table 9: Benchmark results for unsuccessful polls

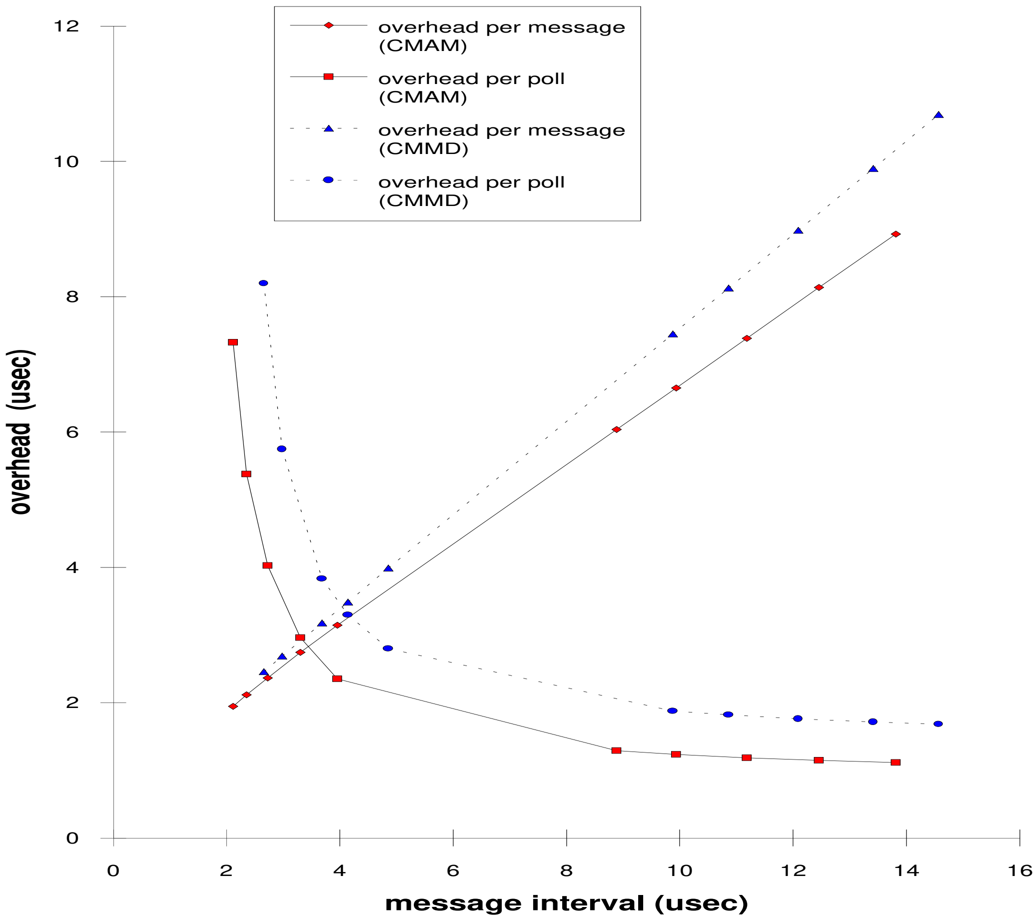
CMAML function	cycles/op	μsec/op	CMAM function	cycles/op	μsec/op
CMAML_poll	42.6	1.33	CMAM_poll	27.2	0.85
CMAML_request_poll	27.8	0.87	CMAM_request_poll	11.2	0.35
CMAML_reply_poll	22.1	0.69	CMAM_reply_poll	16.3	0.51
CMAML_wait	13.4	0.42	CMAM_wait	22.1	0.69
CMAML_poll_wait	53.1	1.66	CMAM_poll_wait	43.5	1.36

The time of an unsuccessful poll for various polling functions are shown in Table 9. The polling functions in CMAM have lower overhead than those in CMAML except for `CMAM_wait()`. The overheads for `CMAM_wait()` and `CMAML_wait()` are lower than expected because, in this case, the functions check the flag and return without polling the NI.

6.1.3.2 Successful poll

We plot the average time to receive a message on a per message basis and on a per poll basis versus the message send interval in Figure 5. The curve for the average time per poll decreases monotonically because more polls are unsuccessful as the message send interval increases. The curve for the average time per message increases linearly because the time to receive a message increases when more time is wasted in unsuccessful polls. From the intersection of the two curves, we can find that the time to receive one message per poll is about 2.85 μsec for CMAM and about 3.3 μsec for CMAML.

Figure 6. Benchmark results for successful polls. We average the time to receive a message on a per poll basis and on a per message basis.



6.2 Interrupts

6.2.1 Basic Message Interrupt Handling Mechanism

Messages can also be received via interrupt instead of polling. If message interrupt is enabled on the NI, the NI will generate an interrupt to the CPU upon message arrival. If the CPU interrupt is also enabled, the CPU will suspend the ongoing computation and jump to a message interrupt handler through the kernel. In general, the message interrupt handler performs the following functions:

- disable message interrupt on the NI
- save any necessary states
- call a polling function to dispatch all incoming messages

When there are no more messages to be received, the interrupt handler will restore the saved states, enable message interrupt, and return to the interrupted computation.

6.2.2 Benchmarks

In this section, we measure the overhead of using interrupt to receive messages. Since message interrupt is not supported in CMAM, all measurements are done using CMAML.

First, we measure the overhead of enabling and disabling message interrupt using `CMAML_enable_interrupts()` and `CMAML_disable_interrupts()`. These functions are useful in forming critical sections. In the pipeline benchmark, we measure the average time to call these functions in a loop. This measures the minimum time taken by these functions without changing the interrupt enable status. In this case, it only involves checking the value of a global variable. In the toggle benchmark, we measure the average time to enable interrupt when interrupt is currently disabled and vice versa. We also measure the time taken by a pair of calls to `CMAML_enable_interrupts()` and `CMAML_disable_interrupts()`. The NI timer is used for timing in these benchmarks.

Second, we measure the overhead to receive messages by interrupts. In this benchmark, we enable interrupt on node 0 and then call a function to perform a fixed amount of computation on node 0. At the same time, we send 2500 messages from node 1 to node 0 at a certain time interval and measure the total time taken by node 0 to perform the computation while receiving messages using interrupt. Then, we measure the time taken to perform the same computation without receiving any messages. The difference of the two time measurements gives us the time to receive the 2500 messages using interrupt. By subtracting the time to execute the handler, we can obtain the overhead to receive the 2500 messages by interrupt. `get_seconds()` is used for timing in these benchmarks.

6.2.3 Results and Discussions

6.2.3.1 Enabling/disabling message interrupt

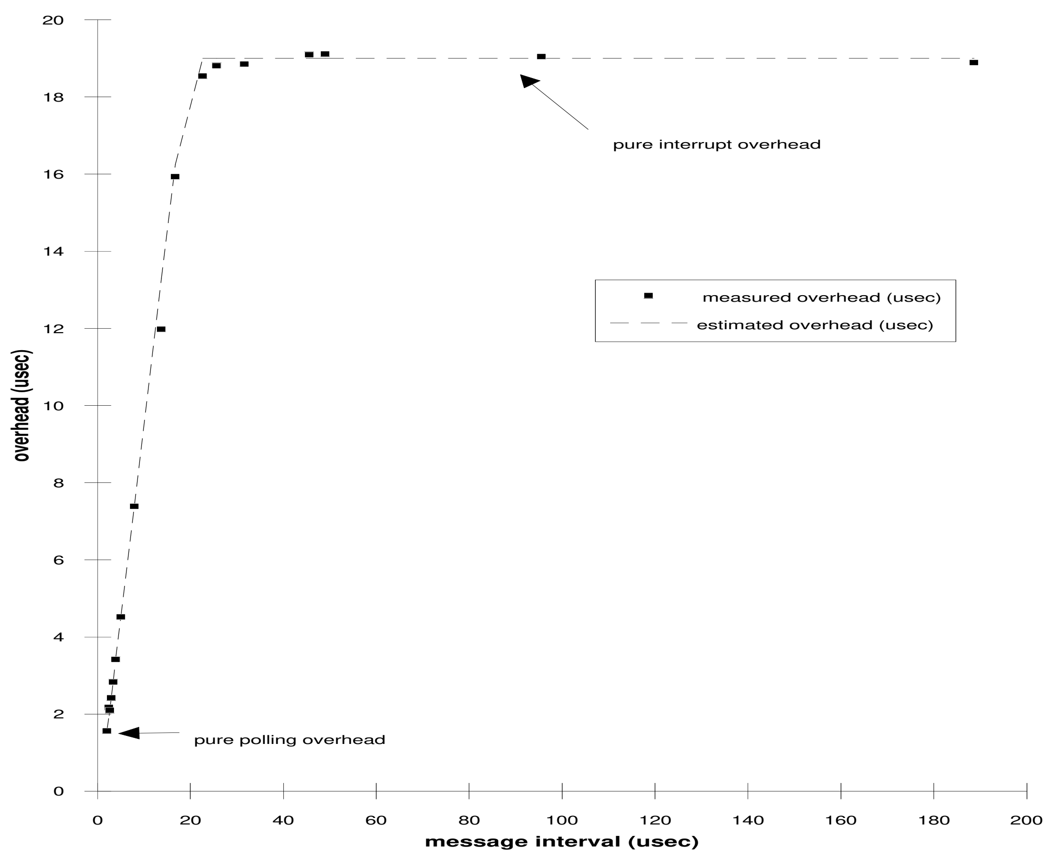
Table 10: Benchmark results for interrupt enable and disable

CMAML function	cycles/op	µsec/op
CMAML_enable_interrupts (pipeline)	13.4	0.42
CMAML_enable_interrupts (toggle)	156.8	4.90
CMAML_disable_interrupts (pipeline)	13.4	0.42
CMAML_disable_interrupts (toggle)	120.0	3.75
CMAML_enable_interrupts/CMAML_disable_interrupts	267.8	8.37

From the toggle benchmark results in Table 10, we can see that the overhead to enable and disable message interrupt is quite high. The reason is that the interrupt enable bit in the NI can only be changed in supervisor mode. Therefore, a system call to the CMOST kernel on the node is required. For `CMAML_enable_interrupts()`, after interrupt is enabled, it needs to check whether a message has moved to the head of the receive FIFO while the interrupt enable bit is set. Otherwise, we may miss an interrupt for that packet. As a result, the overhead of `CMAML_enable_interrupts()` is higher.

6.2.3.2 Overhead of receiving messages using interrupt

Figure 7. Overhead of receiving messages using interrupt in CMAML



The average overhead of receiving a message using interrupt in CMAML is plotted against the message interval in Figure 6. The expected overhead per message is also plotted in the same graph. Note that the leftmost point in Figure is below the 2 µsec we measure in Section 5.1.1.2.1 because we have deducted the 0.5 µsec per call to execute the handler here.

The plateau-shaped curve can be explained as follows. The cost of receiving a message via interrupt is higher than that for polling because the kernel is involved. When messages are sent frequently, the first message arrival will trigger a message interrupt but subsequent messages are received by polling in the dispatch loop. Therefore, the average cost per message is quite low and is very close to that for polling. When messages are sent less frequently, the number of interrupts increases and the number of messages received per interrupt decreases. As a result, the average cost per message increases until the message interval is equal to the overhead to receive one message per interrupt. Beyond this point, every message arrival will trigger an message interrupt and no message is received by polling. Since there is no wasted effort to poll for messages, the curve stays flat at the overhead of receiving one message per interrupt, which is about 19 μsec.

The expected time per message can be calculated using the following model. Suppose n messages are sent at a certain message interval, x. Given the time to receive a message per interrupt(T_i) and the time to receive a message by polling (T_p), the time to receive n messages is $T_i + (n - 1)T_p$ Therefore, the average time to receive a message is $\frac{T_i + (n-1)T_p}{n}$. However, we can only send message as fast as we can receive. Hence, we get the following relation:

$$x = \frac{T_i + (n-1)T_p}{n}$$

As a result, the expected average overhead excluding the 0.5 μsec handler execution time is:

$$\begin{aligned} y &= x - 0.5 \\ &= \frac{T_i + (n-1)T_p}{n} - 0.5 \quad \text{for} \quad x < T_i \\ &= T_i - 0.5 \quad \text{for} \quad x \ge T_i \end{aligned}$$

Our measured results match the expected results very closely in Figure 6. The uphill portion of the graph is just a manifestation of the fact that the average message send rate can only be as fast as the average message receive rate. In this region, the CPU is spending most of its time receiving messages rather than performing computation.

7.0 Message Interrupt Handling

In this section, we analyze the message interrupt path in CMOST 7.2.final and CMMD 3.1.final. Based on this analysis, we propose an efficient design for the interrupt path and show an example implementation that has lower overhead than CMOST.

7.1 Message Interrupt Handling in CMOST and CMMD

The message interrupt handler in CMOST 7.2.final and CMMD 3.1.final is actually more complex than what we have described in Section 6.2.1. If message interrupt is enabled, the NI will generate a Green interrupt to the SPARC processor upon message arrival. If CPU interrupt is enabled, the CPU will automatically perform the following actions:

- suspend the ongoing computation
- switch to supervisor mode and update to the processor status register(PSR)
- activate the trap register window

- save the PC and nPC in the trap register window
- jump to SYS_GREEN, the Green interrupt entry in the trap table

Since there can be only 4 instructions in each entry of the trap table, the program must jump from the trap table entry, SYS_GREEN, to the actual green interrupt handler, sys_green.

sys_green is the kernel-level interrupt handler that handles both system messages (e.g. I/O messages) and user messages. When receiving user messages, it performs the following functions:

1. update the variables _num_green and _green_int_cause (probably for debugging purpose)
2. check the interrupt cause register on the NI to see whether the interrupt is caused by an alarm condition
3. save the global registers %g6 and %g7 into memory because they are used in the handler
4. clear the interrupt cause register on the NI
5. poll both the left data network (LDR) and right data network (RDR) to see whether there is any incoming message
6. check the tag of the message to see whether it is a system message or user message
7. check the tag of the user message to see whether it is supposed to trigger an interrupt
8. restore %g6 and %g7
9. disable message interrupt on the NI by resetting the interrupt mask
10. get the user-level message handler from process control block (PCB)
11. save the processor status register (PSR), program counters(PC and nPC), the multiply-step register (Y), the stack pointer (SP), and global registers(%g1 through %g4) to the PCB.
12. check the access permission of the user-level message handler
13. return from the trap and jump to the user-level message handler

In this case, the user-level message handler is CM_message_func, which is a wrapper function that calls the CMAML message interrupt handler, cmaml_interrupt_handler. CM_message_func is invoked to make sure that the user-level message handler trap back to the kernel to restore the previous states.

cmaml_interrupt_handler performs the following functions:

1. update two flags _CMAML_InterruptsEnabledP and _CMAML_InsideInterruptHandlerP
2. save the floating-point status register (FSR) in memory
3. call CMAML_poll() to dispatch all incoming messages

Note that CMAML_poll is used to receive messages here. But, rather than calling it directly, we need to crawl through the kernel first.

When CMAML_poll has no message to receive, it returns to cmaml_interrupt_handler, which restores the registers and updates the flags again. Then, CM_message_func traps back to the kernel through SYS_MRESTORE. The trap handler, _sys_mrestore, restores the PSR, PC, nPC, SP, Y, and %g1 through %g4 from the PCB. It also enables the user message interrupt by restoring the NI interrupt mask from the PCB. Since interrupt is disabled in the interrupt handler, some messages may have arrived after returning from CMAML_poll. Therefore, _sys_mrestore needs to check that there is no more incoming system or user message. As a result, it jumps back to the _io_interrupt section of _sys_green before it returns from the trap to the interrupted computation.

From the description above, we can see that the message interrupt path in CMOST and CMMD is not efficient for receiving user messages. For example, we can make the following observations:

- _sys_green gives priority to system messages over user messages. User messages are handled only if there are no pending system messages. As a result, the path to the user message handler is lengthened.
- Since CMOST allows users to set their own message handlers, it is very cautious in saving the process state and error checking. For example, it saves the global registers %g1 through %g4 even though the user message handler may not use them. Unfortunately, together with saving the PSR, PC, nPC, Y, and SP, this will cause the 4-double-word write buffer to overflow in the default write-through cache mode.

7.2 An Efficient Message Interrupt Handling Design

Based on the analysis in the last section, we propose an efficient design for handling message interrupt. Contrary to the circumspect approach taken by CMOST, we take a lazy approach. Our design differs from the CMOST design in that:

- We assume that handling user messages is the common case. Therefore, there should be a fast path through the kernel to the user message handler. Since system messages such as alarms are infrequent and system messages for I/O have high overhead already, we can trade off the overhead of system messages for faster processing of user messages.
- We believe that the kernel should be as lean as possible so that no extra overhead will be incurred for features that are not frequently used. As long as we define the interface between kernel code and user code clearly and follow the SPARC ABI convention, we can rely on the user to save any necessary state with the callee save convention. Therefore, the kernel can save the minimal amount of state.

To illustrate this approach, we have modified the message interrupt path in the CMOST kernel to implement a fast path for handling user messages. In the modified kernel, we insert code at the beginning of _sys_green to determine whether user messages are received. If so, the code falls through the fast path to the user message handler. Otherwise, if a system message is received, the code jumps back to the original CMOST code. The fast path for user messages only saves and restores the PC, nPC, and PSR, but not the Y, SP and global registers. This avoids overflowing the write buffers in the default write-through cache mode. The user message handler should save and restore the global registers and floating point registers as needed.

In addition, we streamline the message handler for user messages as follows:

- No global registers are used in the fast path for handling user messages in the kernel. As a result, %g6 and %g7 do not need to be saved and restored.
- In the original CMOST code, the NI is accessed a lot of times using 3 instructions like:

```
sethi   %hi(NI_REC_INTERRUPT_MASK_A),%l4          ! set high bits of absolute address
or      %l4,%lo(NI_REC_INTERRUPT_MASK_A),%l4      ! set low bits of absolute address
ld      [%l4],%l7                                 ! load the interrupt mask from NI
```

  In the fast path, we keep the base address of the NI in the local register %l5, and do the load/store using the offset with one instruction, e.g.:

```
ld      [%l5+NI_REC_INTERRUPT_MASK_O],%l7         ! use the offset from the NI base
```

- In `_sys_mrestore`, instead of jumping back to `_io_interrupt`, the check for more messages is done right away by checking the NI status.
- Users can still set the user message handler. However, instead of checking the access permission of the user message handler every time it is invoked, we assume it is checked only once when it is set using the `CMOS_SET_MESSAGE_HANDLER` syscall.
- The variables `_num_green` and `_green_int_cause` do not seem to be used in anywhere else in the kernel. So, they are not updated in the fast path.

To find out what is the lowest overhead achievable, we have also implemented a user message handler by merging CM_message_func, cmaml_interrupt_handler, and CMAM_dispatch into one function. This handler does not update any flags and does not use the work queue and the global registers. It is just a minimal handler to dispatch request or reply messages.

7.3 Performance Comparison of the Two Implementations

Table 11 shows the breakdown of the instruction counts and cycle counts for the message interrupt path in CMOST 7.2.final and CMMD 3.1.final. A similar breakdown is shown in Table 12 for the kernel fast path and the minimal user message handler we have implemented. The following assumptions are made to estimated the cycle counts:

- We are receiving one user message per interrupt from the request (left) data network. There are no system messages.
- All instructions are in the cache.
- For a 32-bit word, NI register reads take 7 cycles and NI register writes take 3 cycles.
- All non-NI data are in the cache, i.e. 32-bit word loads take 2 cycles and 32-bit word stores take 3 cycles.

To compare the actual performance of the two implementations, we also measure the interrupt overhead under various conditions using the benchmark we use in Section 6.2.2. The results are shown in Table 13.

Based on assumptions above, the estimated overhead of receiving one message per interrupt using CMOST 7.2.final and CMMD 3.1.final is 12.6 μsec and the measured overhead is 19 μsec. For the kernel fast path and minimal user message handler, the estimated overhead is 6.2 μsec and the measured overhead is 9.6 μsec. Note that the estimated time is always smaller than the measured time. One possible reason is that there are actually cache misses since the SPARC processor has only a 64 KB direct-mapped unified cache and the CM-5 uses a write-through cache by default. However, by examining the code and data addresses in the symbol table, we fail to identify any possible sources of cache conflict misses. The discrepancy between the estimated and measured values shows that there are certain factors that we have neglected when we estimate the overhead.

By turning on the copy-back cache, we can see that the overhead drops from 19 μsec to 15 μsec when using CMOST 7.2.final and CMMD 3.1.final. When the modified kernel or the minimal user handler is used, the decrease is less substantial. In addition, from the measurements, we find that saving the global register %g1 through %g4 costs about 2 μsec in the write-through cache mode because this causes the write buffers to overflow. If the global registers are saved later in the user message handler, we can avoid this extra overhead.

In summary, we can see that taking a minimalistic approach can substantially cut down the overhead of receiving messages by interrupt.

Table 11: Breakdown of cycle counts for message interrupt path in CMOST 7.2.final and CMMD 3.1.final

Function/Label	Description	# of instr	# of cycles
SYS_GREEN	Trap table entry for message interrupt	4	10
sys_green	Green interrupt handler update _num_green and _green_int_cause check whether message tagged as alarm	 8 6	 13 6
_io_interrupt	I/O interrupt handler save %g6 and %g7; initialize variables	 5	 8
check_dr	clear interrupt cause; receive message on LDR? Yes message tagged as I/O? No	10 7	18 7
check_other_dr	receive message on RDR? No	5	11
_check_for_user_message	get user interrupting message? Yes	13	19
_sys_green_user_msg	restore %g6 and %g7; disable user message interrupt get message handler address; save states in PCB check access permission of handler address return from trap to CM_message_func	8 12 4 2	18 30 4 4
			subtotal: 148 cycles(4.6 µsec)
CM_message_func	user message handler wrapper save frame; call cmaml_interrupt handler	 6	 7
cmaml_interrupt_handler	update flags; save FSR in memory; save %o6, %o7, and %g[5-7] in %l[3-7]; call _CMAML_dispatch	13	19
CMAML_poll	save frame; initialize work queue and variables poll left	10 4	10 10
INTERNAL_DISPATCH	jump to handler table pull message out of NI	4 8	5 29
			subtotal: 80 cycles(2.5 µsec)
Execute user handler			
CMAML_poll	poll right poll left poll right check rport check work queue and return	4 5 5 3 6	10 11 11 3 9
cmaml_interrupt_handler	restore registers; update flags; return	14	20

Function/Label	Description	# of instr	# of cycles
CM_message_func	trap to SYS_MRESTORE	3	6
subtotal: 69 cycles(2.2 μsec)			
SYS_MRESTORE	Trap table entry	4	5
_sys_mrestore	restore PSR restore registers from PCB; enable user message interrupt	5 14	6 26
_io_interrupt	Go back to check for I/O messages again save %g6 and %g7; initialize	 5	 8
check_dr	clear interrupt; receive message on LDR? No	10	18
check_other_dr	receive message on RDR? No	5	11
_check_for_user_message	get user message? No is tag bad? No	13 4	19 4
_io_read_done	restore %g6, %g7, and PSR; return from trap	6	10
subtotal: 107 cycles(3.3 μsec)			
Total: 12.6 μsec (235 instructions; 404 cycles)			

Table 12: Breakdown of cycle counts for message interrupt path in the modified CMOST kernel and a minimal user message handler

Function/Label	Description	# of instr	# of cycles
SYS_GREEN	Interrupt vector	4	10
sys_green	Trap table entry for message interrupt check whether message tagged as alarm	 5	 5
_io_interrupt	I/O interrupt handler get LDR and RDR status	 2	 14
_io_interrupt_more	clear interrupt cause receive message on LDR? Yes set tag mask message tagged as I/O? No receive message on RDR? No	1 2 2 4 2	3 2 2 4 2

Function/Label	Description	# of instr	# of cycles
_sys_green_user_msg_fast	load interrupt mask (in delay slot)	1	7
	user interrupting message? Yes	8	8
	disable user interrupt	2	4
	get message handler addr	3	5
	save PC, nPC, and PSR	2	7
	return from trap to my_message_func	2	4
			subtotal: 77 cycles(2.4 μsec)
my_message_func	save frame; initialize registers	5	5
	poll left	4	10
	jump to handler table	4	5
	poll message out of NI and jump to handler	4	25
			subtotal: 45 cycles(1.4 μsec)
Execute user handler			
my_message_func	poll right	4	10
	poll left	5	11
	poll right	5	11
	trap to SYS_MRESTORE	3	6
			subtotal: 38 cycles(1.2 μsec)
SYS_MRESTORE	Trap table entry	4	5
_sys_mrestore	load tag mask, PC, NPC, and PSR	3	7
	enable user message interrupt	2	4
	check LDR and RDR	2	14
	any more message? No	3	3
	restore PSR and return from trap	3	5
			subtotal: 38 cycles(1.2 μsec)
			Total: 6.2 μsec (91 instructions; 198 cycles)

Table 13: A comparison of interrupt overheads using a combination of CMOST 7.2.final, CMMD 3.1.final, a modified CMOST kernel with fast user message path, and a minimal user message handler. Note that, by default, the CMOST 7.2.final kernel saves and restores the SP and global registers %g1 through %g4 but the modified kernel does not save and restores these registers.

Code	Options	Estimated(μsec)	Measured(μsec)
CMOST 7.2.final with CMAML interrupt handler	write-through cache copy-back cache	12.6 12.6	19 15
modified kernel with CMAML interrupt handler	write-through cache copy-back cache	8.3 8.3	13.4 11.5
CMOST 7.2.final with minimal interrupt handler	write-through cache don’t save/restore %g’s and SP save/restore %g[1-4] and SP save %g[1-4] and SP copy-back cache save/restore %g[1-4] and SP	 9.9 10.6 10.2 10.6	 13 15.1 14.8 12.8
modified kernel with minimal interrupt handler	write-through cache don’t save/restore %g[1-4] save/restore %g[1-4] save %g[1-4] copy-back cache don’t save/restore %g[1-4]	 6.2 6.7 6.5 6.2	 9.6 11.4 10.9 9.4
CMOST 7.2.final	write-through cache	8	12
CMAML interrupt handler	write-through cache	4.7	7
modified kernel	write-through cache	3.6	6.6
minimal interrupt handler	write-through cache	2.6	3

8.0 Conclusion

In this report, we compare the performance of Active Messages on the CM-5 using CMAML in CMMD 3.1.final and CMAM 2.7. In addition, we develop a new benchmark framework for measuring the performance of message passing primitives. From our experience, we find that it is important to pay close attention to the hardware and software details when designing the benchmarks. For example, the simple one-to-one benchmark in Section 5.1.1 fails to distinguish the difference between the actual send overhead and the receive overheads of the basic Active Message functions. By using the two-to-one and one-to-two benchmarks, we find out that the results in the one-to-one benchmark are actually limited by the receive overheads. In addition, it is important to describe the benchmarks in details so that readers can verify the results and draw meaningful conclusions on their own.

Comparing the CMAM and CMAML implementations, we can see that there is a trade-off between functionality and performance. CMAML offers extra functionality such as message interrupt and work queue. But, it also has higher overhead than CMAM. Our benchmarks show that CMAM outperforms CMAML in both the basic Active Message functions and nearest-neighbor array transfer. In a loaded network, the use of the work queue enables the CMAML functions to make use of both networks to send data. Therefore, `CMAML_scopy()` has higher effective bandwidth than `CMAM_xfer()`. In other words, CMAML is sacrificing the short message performance for better performance in sending large arrays of data. While our benchmarks shows that adding message interrupt and the work queue leads to the higher overhead of CMAML, our benchmarks do not measure any specific cases in which these features can be useful. Whether this will benefit an application or not will depends on the communication characteristics of the particular application.

From the benchmarks to measure the overhead of receiving messages by polling and interrupt, we can see that it is important to balance the send overhead and receive overhead when designing a message layer. Since the network is a closed system, the message injection rate must be equal to the message extraction rate at equilibrium. To reduce the overhead of receiving messages by interrupt, we propose that there should be a fast path in the kernel to handle user message interrupt. In addition, the kernel should save the minimal amount of state so that no extra overhead will be incurred. This provides a flexible interface for the user message handler to implement any functions it want.

9.0 Acknowledgement

This research is supported by NFS Presidential Faculty Fellowship (CCR-9253705). CM-5 computational resource is provided by NFS Infrastructure Grant (CDA-8722788) and the Advanced Computing Laboratory of Los Alamos National Laboratory. The author would like to thank Alan Mainwaring of Thinking Machine Corp. for providing valuable information and discussions about CMMD, and Eric Allman and Eric Fraser, our CM-5 system administrators, for their kind assistance. The statistical analysis routines used in the benchmarks are originally written by Andrea Dusseau.

References

[1] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Eric Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture* (Gold Coast, Australia, May 1992).

[2] Thinking Machine Corp. *CMMD Reference Manual Version 3.0*. May 1993.

[3] Thomas T. Kwan, Brian K. Totty, and Daniel A. Reed. Communication and Computation Performance of the CM-5. In *Proceedings of Supercomputing'93* (Portland, OR, November 1993).

[4] Charles E. Leiserson et al. The Network Architecture of the Connection Machine CM-5. In *Symposium on Parallel Algorithms and Architectures* (April 1992).

[5] Thorsten von Eicken. Active Messages: an Efficient Communication Architecture for Multiprocessors. Ph.D. Dissertation, University of California at Berkeley, November 1993.

[6] David E. Culler, Anurag Sah, Klaus Erik Schauser, Thorsten von Eicken, and John Wawrzynek. Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine. In *Proceedings of 4th International Conference on Architectural Supports for Programming Languages and Operating Systems* (Santa Clara, CA, April 1991).

[7] David E. Culler, Andrea Dusseau, Seth C. Goldstein, Arvind Krishnamurthy, Steve Lumetta, Thorsten von Eicken, and Kathy Yelick. Parallel Programming in Split-C. In *Proceedings of Supercomputing'93* (Portland, OR, November 1993).

[8] David E, Culler, Richard Karp, Dave Paterson, Abhijit Sahay, Klaus Eril Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of the 4th ACN SIGPLAN Symposium on Principles and Practices of Parallel Programming* (San Diego, CA, May 1993).