NAME

CMAM – Introduction to the CM-5 Active Message communication layer.

DESCRIPTION

The CM-5 Active Message layer CMAM (<see-mam>) provides a set of communication primitives intended to expose the communication capabilities of the hardware to the programmer and/or compiler at a reasonably high level. A wide variety of programming models, ranging from send&receive message passing to variants of shared memory, can be implemented efficiently on top of CMAM without performance compromise. The basic communication primitive of CMAM is the *Active Message*: a message where the first word points to the message handler. On reception of an Active Message, the message handler is called with the remainder of the message as argument. The role of the message handler is to integrate the message data into the computation, by storing it, accumulating it, or whatever makes sense in the overlaying programming model. Its execution is atomic relative to other message handlers, thus long message handlers tend to back-up the network. Note that while Active Messages might be viewed as a restricted form of RPC (Remote Procedure Call), this is not the intent: message handlers should only integrate message data into the computation or, alternatively, reply to a simple request. They should not perform arbitrary computational tasks.

The CMAM library provides primitives to send Active Messages to remote nodes, to transfer bulk data to pre-arranged communication segments, and to synchronize processors using the CM-5 control network. Functions to initialize and control the usage of hardware message tags within the CMAM library allow it to co-exist with other message layers. Over all, the CMAM library should be viewed as a "machine language for communication" and not as an end-user communication library.

Sending Active Messages

The most important communication primitive is **CMAM**(3) which sends an Active Message to a remote node. It takes a pointer to a C function to call on the remote node and a buffer to pass as argument. **CMAM** is built on top of **CMAM_4**(3) which calls a remote function with 4 32-bit words as arguments. **CMAM_4** sends exactly one CM-5 network packet and passes the arguments from registers on the source node to registers on the destination node. In contrast, **CMAM** sends a header packet and one packet per 12 bytes of buffer and requires allocation of a temporary buffer on the destination node.

Bulk data transfer

The bulk data transfer function **CMAM_xfer**(3) augments **CMAM** by providing efficient bulk data transfer to pre-arranged communication segments. Before data transfer can occur, the recipient must allocate a communications segment using **CMAM_open_segment**(3), specifying a *base address*, a *byte count* and an *end-of-transfer function*. The sender (or multiple senders) can then transfer data to memory relative to the base address. When count bytes have been transferred, the end-of-transfer function is called on the recipient node. **CMAM_xfer** transfers a contiguous byte buffer sending 16 bytes in each CM-5 packet.

"Gather-transfer-scatter" can be implemented using **CMAM_xfer_***N*i(3) (*N*=1..4) which send a packet with *N* 32-bit words of data to a segment-base + offset on the remote node. (The next version of CMAM will provide more efficient gather-scatter through two functions, **CMAM_xfer_ivec** and **CMAM_xfer_dvec** transfering 32-bit element vectors (32-bit elements) and 64-bit element vectors with stride.)

Sending Indirect Active Messages

The function **CMAM_indirect_4**(3) is nearly identical to **CMAM_4** but performs an indirect function call on the remote node: a pointer to a pointer to a function (i.e. (**fun)()) is transmitted in the message. In some circumstances this allows one pointer to refer to both the function to be invoked and its environment and thus crams more data into a CM-5 packet. The buffer version **CMAM_indirect** is provided for completeness, but does not provide a clear advantage over **CMAM**.

Polling the network

The current implementation of CMAM does not use interrupts on message arrival, thus, to receive messages, the network interface status register must be polled. All CMAM functions sending messages poll the network so that a node sending messages will handle incoming ones. The most common situation

requiring explicit polling is when a node busy-waits for the reception of specific messages. For this purpose, **CMAM_wait**(3) waits (while polling) on a synchronization flag (a **volatile int**) to reach a given value, resets the flag to zero and returns. Other wait mechanisms may be implemented using **CMAM_poll**(3).

Sending and replying

All CMAM functions sending messages come in two variants to avoid deadlock in two-way communication. The *send* variant (**CMAM**, **CMAM_4**, **CMAM_xfer**, ...) sends messages on the "left" CM-5 data network and polls both the "left" and the "right" networks. The *reply* variant (**CMAM_reply**, **CMAM_reply_4**, **CMAM_reply_xfer**, ...) sends messages on the "right" network and only polls that network. Messages sent from the computation proper should use the send variant whereas message handlers wishing to send replies *must* use the reply variant. Reply-message handlers cannot send any messages themselves. Failure to follow these rules may result in a stack overflow: a handler using the send variant to reply and finding the outgoing network backed-up will accept messages from the network to drain it, nesting the appropriate handlers on the stack. Given that these handlers may attempt to send, the stack may grow arbitrarily. By using the reply variant, only messages on the reply network need to be accepted and, given that their handlers may not reply themselves, the stack will not grow arbitrarily.

Communicating with the host (control) processor

Individual processing nodes may send messages to the host ("control processor" in CM-5 terminology) using special variants of all communication functions (e.g. **CMAM_host**(3), **CMAM_host_4**(3), **CMAM_host_reply**(3), **CMAM_host_xfer**(3), **CMAM_host_reply_xfer**(3), ...). The host can send to any processing node using the normal functions.

The difficulty in communication with the host lies in the separate address spaces: addresses of message handlers are different on the host than on the processing nodes. See **CMAM_host**(3) on possibilities to deal with the situation.

Initialization

CMAM_enable(3) must be called on the control processor before using CMAM. It initializes data structures on all nodes and sets-up default message tag usage. **CMAM_disable**(3) may be called on the control processor at the end of the program to wait for all processing nodes to return to the idle loop.

Barrier synchronization

CMAM provides access to the "synchronous global OR" of the CM-5 control network in the form of a fuzzy barrier. **CMAM_start_barrier**(3) signals the arrival of a processing node at the barrier and **CMAM_end_barrier**(3) blocks until all processing nodes have called **CMAM_start_barrier**. **CMAM_barrier**(3) implements a traditional barrier. When starting the barrier, processing nodes may enter a 1-bit value into the global OR computation which is returned at the end of the barrier. This value is sometimes useful for termination detection. Note that **CMAM_enable_barrier**(3) must be called on the host before the barrier functions can be used on the nodes.

Mixing message layers

CMAM has been designed such that it can be mixed with other message layers. However, the only other message layer available, CMMD, does not (yet?) co-operate. The node-to-node communication primitives of CMAM and CMMD cannot be mixed (although a program could alternate phases of CMAM use with phases of CMMD use). The global operations of CMMD which use the control network may be used simultaneously with CMAM node-to-node operations (in particular, the integer scan operations in CMMD are compatible with CMAM). However, the initialization of CMAM barrier operations interferes with CMMD global operations. Note that reportedly CMMD 3.0 will include Active Messages.

CMAM can share the network interface with other message layers (or with user additions to CMAM) by using different hardware message tags and by co-operating in the message tag-handler dispatch. The function **CMAM_set_tags**(3) lets the user specify which hardware message tags CMAM should use. Other message layers may use free tags provided that a handler is registered with CMAM using **CMAM_set_handler**(3) and **CMAM_set_reply_handler**(3).

USAGE

The CMAM header files assume the use of **gcc(1)** and contain ANSI-C function prototypes and gcc **inline** definitions. Conversion to **cc(1)** is certainly possible, but not provided. All programs using CMAM should (assuming U.C. Berkeley installation):

include **#include <cmam/cmam.h>**
be compiled with **gcc-2.1 -I/usr/cm5/local/include ...**
be linked with **cmld -gld2 -L/usr/cm5/local/lib ... -pe -L/usr/cm5/local/lib**
(The **-gld2** option to **cmld** loads the library necessary for programs compiled with **gcc-2.x**.)

SEE ALSO

CMAM_enable(3), **CMAM**(3), **CMAM_4**(3), **CMAM_indirect**(3), **CMAM_indirect_4**(3), **CMAM_open_segment**(3), **CMAM_xfer**(3), **CMAM_xfer_N**(3), **CMAM_xfer_4i**(3), **CMAM_barrier**(3), **CMAM_set_tags**(3), **CMAM_host**(3), **CMAM_host_4**(3), **CMAM_host_indirect**(3), **CMAM_host_indirect_4**(3), **CMAM_host_xfer**(3), **CMAM_host_xfer_N**(3), **CMAM_host_xfer_4i**(3),

Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser, *Active Messages: a Mechanism for Integrated Communication and Computation.* Proc. of the 19th Int'l Symposium on Computer Architecture, Gold Coast, Australia. May 1992. (Also available as Technical Report UCB/CSD 92/675, CS Div., University of California at Berkeley.)

AUTHOR

Thorsten von Eicken, tve@CS.Berkeley.EDU

NAME

CMAM_enable, CMAM_disable – CMAM initialization

SYNOPSIS

#include <cmam/cmam.h>

void CMAM_enable (void);

void CMAM_disable (void);

DESCRIPTION

CMAM_enable must be called on the host (''control processor'' in CM-5 terminology) before any other CMAM functions are used. It runs an initialization function on all processing nodes to set the default packet tag assignments (see **CMAM_set_tags**(3)) and to initialize other CMAM data structures.

CMAM_enable does not change the state of the network interface and therefore does not conflict with other message layers. Note that the CMAM barrier functions (e.g. **CMAM_barrier**(3)) require separate initialization (conflicting with CMMD global operations) performed by **CMAM_enable_barrier**(3).

CMAM_disable runs an echo function on all processing nodes to ensure that they are all in the ''idle loop''.

SEE ALSO

cmam(3), **CMAM_set_tags**(3), **CMAM_enable_barrier**(3), **CMAM**(3).

AUTHOR

Thorsten von Eicken, tve@CS.Berkeley.EDU

NAME

CMAM, CMAM_reply – Send Active Message

SYNOPSIS

#include <cmam/cmam.h>

typedef void CMAM_handler (void *buffer, int byte_count);

void CMAM (int node, CMAM_handler *fun, void *buffer, int byte_count);

void CMAM_reply (int node, CMAM_handler *fun, void *buffer, int byte_count);

void CMAM_set_size (int max_buffer_size);

DESCRIPTION

CMAM sends an Active Message to the remote **node**, calling the handler **fun** and passing the **buffer** as argument. The call

CMAM(node, foo, buf, buf_len);

will allocate a buffer of size **buf_len** bytes on the remote **node**, transfer the source buffer into the allocated buffer packing 12 bytes per CM-5 network packet, invoke

foo(*allocated_buffer***, buf_len)**

and finally deallocate the buffer. Note that since the buffer is deallocated after the handler returns all data must be copied to its final destination. Transferring large amounts of data is best performed using **CMAM_xfer**(3) which avoids the copy and packs more data into each CM-5 packet.

At initialization time a small number (typ. 16) of buffers of size **CMAM_SZ** are pre-allocated for **CMAM**. If needed, additional buffers are allocated using **malloc**(3). The current implementation limits the message size to the size of the pre-allocated buffers. **CMAM_set_size** may be used to change the size of the buffers used. **CMAM_set_size** must be called while no messages are partially received.

CMAM sends the Active Message on the left data network and polls both, the left and right networks. **CMAM_reply** is identical to **CMAM**, but uses the right data network and polls only the right side. **CMAM_reply** should be used within message handlers for replies to avoid the danger of stack overflow (see **cmam**(3)). Neither **CMAM** nor **CMAM_reply** may be used from within a reply-message handler.

CMAM may be used to send Active Messages from the host (''control processor'' in CM-5 terminology) to the processing nodes. However, note that the executable running on the host is different from the one running on the processing nodes, therefore the address of functions differ. The linker **cmld**(1) arranges for functions named **CMPE_...** or **cmpe_...** on the processing nodes to be resolved correctly on the host. To send Active Messages to the host from processing nodes, see **CMAM_host**(3).

PERFORMANCE

CMAM is built on top of **CMAM_4**(3) and sends three 32-bit words of data per message.

SEE ALSO

cmam(3), **CMAM_xfer**(3), **CMAM_host**(3), **CMAM_4**(3).

AUTHOR

Thorsten von Eicken, tve@CS.Berkeley.EDU

NAME

CMAM_open_segment, CMAM_open_this_segment, CMAM_shorten_segment, CMAM_query_segment, CMAM_kill_segment – Manipulate data transfer segments

SYNOPSIS

```
#include <cmam/cmam.h>

typedef int CMAM_end_xfer (void *info, void *base_address);

int CMAM_open_segment (void *base_address, int byte_count,
                       CMAM_end_xfer *end_xfer_fun, void *info);

int CMAM_open_this_segment (int segment_num, void *base_address, int byte_count,
                            CMAM_end_xfer *end_xfer_fun, void *info);

void CMAM_shorten_segment (int segment_id, unsigned int delta_count);

void CMAM_kill_segment (int segment_id);

int CMAM_query_segment (int segment_id);
```

DESCRIPTION

CMAM segments are used in conjunction with **CMAM_xfer**(3) and its many variants to transfer bulk data between nodes. A segment is opened by the recipient of the data using **CMAM_open_segment**, specifying a **base_address** relative to which received data is stored and the number of bytes to be received. **CMAM_open_segment** returns a segment identifier for the highest free segment or -1 if no segments are available (note that segment identifiers can be negative, thus check for "!= -1"). After **byte_count** bytes have been received, the **end_xfer_fun** is called with **base_address** and **info** as arguments. The **end_xfer_fun** should return a new **byte_count** or 0 if the segment is to be closed.

CMAM_open_this_segment is similar to **CMAM_open_segment**, but specifies which segment should be opened. It returns -1 if the requested segment is already in use. **CMAM_open_this_segment** can be used in all-to-all communication to open the same segment on all nodes. **CMAM_open_segment** always allocates the highest free segment, so it is best to use low segment numbers with **CMAM_open_this_segment**.

Notes: The segment identifiers returned by **CMAM_open_segment** and **CMAM_open_this_segment** encode the segment number and the alignment of the **base_address**. More precisely: **segment_id** = (**segment_num** << **CMAM_SEG_SHIFT**) | (**base_address** & 7).

Currently, there are 256 segments available.

The **base_address** of closed segments is set to -1. As a result, transferring data to a closed segment typically raises an unaligned-store exception inside of a CMAM handler.

If **CMAM_open_segment** is called with a 0 **byte_count**, **end_xfer_fun** is called immediately.

CMAM_shorten_segment reduces the remaining byte count of the specified segment. If the count becomes ≤0, **end-of-xfer** is called. **CMAM_shorten_segment** is useful, for example, when the sender cannot provide as much data as the recipient asked for.

CMAM_query_segment returns the remaining byte count of the specified segment. It returns 0 for a closed segment.

CMAM_kill_segment closes the segment *without* calling the end of transfer function. Note that due to the reordering of packets in the network it is only safe to call CMAM_kill_segment on a segment for which it is known that no packets are underway.

PERFORMANCE

The overhead in opening and closing a segment (including the call to **end-of-xfer**) is about 10us.

SEE ALSO

cmam(3), **CMAM_xfer**(3), **CMAM_xfer_4**(3), **CMAM**(3).

AUTHOR

Thorsten von Eicken, tve@CS.Berkeley.EDU

NAME

CMAM_xfer, CMAM_reply_xfer – Transfer data block to segment

SYNOPSIS

#include <cmam/cmam.h>

void CMAM_xfer (int node, int seg_addr, void *buff, int byte_count);

void CMAM_reply_xfer (int node, int seg_addr, void *buff, int byte_count);

DESCRIPTION

CMAM_xfer transfers the block of data at address **buff** and of length **byte_count** to a segment on the remote **node** previously opened with **CMAM_open_segment**(3). The destination address for the data on **node** is specified as a segment plus an unsigned byte offset. The segment and the offset are encoded in **seg_addr** which is the sum ('+') of the remote segment id (as returned by **CMAM_open_segment**) and the unsigned byte offset. This encoding limits the offset to 24 bits.

On arrival at the destination node, the data is stored in memory and the byte count associated with the segment is decremented. If the count becomes zero, the end-of-xfer function is called as described in **CMAM_open_segment**(3).

CMAM_xfer sends packets on the left data network and polls both, the left and right networks. **CMAM_reply_xfer** is identical to **CMAM_xfer** but sends packets on the right data network and polls only the right side. **CMAM_reply_xfer** should be used within message handlers to send replies to avoid the danger of stack overflow (see **cmam**(3)).

CMAM_xfer can handle arbitrary block sizes and aligments of source and destination buffers. **CMAM_xfer** transfers most of the block in special XFER packets with 16 bytes of data each. These packets must be stored at a double-word boundary on the destination node. The pre-amble and post-amble necessary to reach this aligment is handled by **CMAM_xfer**.

PERFORMANCE

If the source and the destination buffers have the same aligment relative to double-words (i.e. (src_buff&7) == ((base_address+offset)&7)) most of the data is transferred with **CMAM_xfer_N** at 10Mb/s.

If the aligments differ by a word (i.e. (src_buff^(base_address+offset))&7 == 4) most of the block is transferred by **CMAM_xfer_No** at 7Mb/s.

If the source and destination buffers are badly mis-aligned (i.e. (src_buff^(base_address+offset))&3 != 0) the transfer rate is low...

SEE ALSO

cmam(3), **CMAM_open_segment**(3), **CMAM_xfer_N**(3), **CMAM_xfer_4**(3), **CMAM**(3).

AUTHOR

Thorsten von Eicken, tve@CS.Berkeley.EDU

NAME

CMAM_wait, CMAM_poll_wait, CMAM_poll, CMAM_request_poll, CMAM_reply_poll – Explicit check for pending messages

SYNOPSIS

```
#include <cmam/cmam.h>

void CMAM_wait(volatile int *flag, int value);

void CMAM_poll_wait(volatile int *flag, int value);

void CMAM_poll(void);

void CMAM_request_poll(void);

void CMAM_reply_poll(void);
```

DESCRIPTION

The current implementation of CMAM does not use interrupts on message arrival, thus, to receive messages, the network interface status register must be polled periodically. All CMAM functions sending messages poll the network so that a node sending messages automatically handles incoming ones. The most common situation requiring explicit polling is when a node busy-waits for the reception of specific messages (e.g. signalling the completion of a communication step). For this purpose, **CMAM_wait** polls both (left and right) data networks while waiting on a synchronization **flag** to reach **value** (i.e. ***flag** ≥ **value**), then subtracts **value** from **flag** before returning. (This scheme supposes that, in general, handlers increment a flag to signal when they have executed.) The difference between **CMAM_wait** and **CMAM_poll_wait** is that the former does not poll at all if ***flag** ≥ **value** at entry, while the latter polls at least once.

Other wait mechanisms may be implemented using **CMAM_poll** which polls both data networks, handling all pending messages. **CMAM_request_poll** and **CMAM_reply_poll** are similar, but poll only one network.

PERFORMANCE

SEE ALSO

cmam(3), **CMAM**(3), **CMAM_xfer**(3).

AUTHOR

Thorsten von Eicken, tve@CS.Berkeley.EDU

NAME

CMAM_enable_barrier, CMAM_disable_barrier, CMAM_start_barrrier, CMAM_end_barrier, CMAM_barrier – Global barrier synchronization

SYNOPSIS

#include <cmam/cmam.h>

void CMAM_enable_barrier (void);

void CMAM_disable_barrier (void);

void CMAM_start_barrier (int bit);

int CMAM_end_barrier (void);

int CMAM_query_barrier (void);

int CMAM_barrier (int bit);

DESCRIPTION

These functions provide access to the "synchronous global OR" of the CM-5 control network in the form of a fuzzy barrier. Before using the CMAM barrier functions, the network interface must be initialized from the host ("control processor" in CM-5 terminology) by calling **CMAM_enable_barrier**. Note that this conflicts with the initialization required by CMMD's global operations (CMAM turns host participation off whereas CMMD turns it on by default). **CMAM_disable_barrier** reverts the initialization performed by **CMAM_enable_barrier**.

Calling **CMAM_start_barrier** signals to all other nodes that the current node "entered the barrier". A call to **CMAM_end_barrier** blocks until all nodes have called **CMAM_start_barrier**. While busy-waiting on the barrier, **CMAM_end_barrier** polls and services the network.

CMAM_query_barrier returns true if all nodes have called **CMAM_start_barrier** (i.e. if a call to **CMAM_end_barrier** would not block). **CMAM_query_barrier** does not poll the network.

CMAM_barrier provides a traditional barrier: it blocks until all other nodes call **CMAM_barrier** as well.

Both, fuzzy and traditional barriers provide a one-bit global OR in addition to the synchronization. When starting the barrier, each processor injects a 1-bit value (the LSB of **bit**). When ending the barrier, all processors receive the global OR of all 1-bit values.

Note that the host is not involved in any of these barriers.

PERFORMANCE

A call to **CMAM_barrier** when all processors are already in synch takes 5us.

SEE ALSO

cmam(3).

AUTHOR

Thorsten von Eicken, tve@CS.Berkeley.EDU

NAME

CMAM_4, CMAM_reply_4 – Send single-packet Active Messages

SYNOPSIS

#include <cmam/cmam.h>

void CMAM_4 (int node, void (*fun)(), ...);

void CMAM_reply_4 (int node, void (*fun)(), ...);

DESCRIPTION

CMAM_4 sends a single-packet Active Message to the remote **node**, calling the handler **fun** and passing up to 4 32-bit word arguments. The call

CMAM(node, foo, i1, i2, i3, i4);

sends a single CM-5 network packet to **node** and causes

foo(i1, i2, i3, i4);

to be invoked. **CMAM_4** is declared as a varargs function, but really takes up to four words of arguments in addition to **node** and **fun**, and passes them into the remote function. Due to SPARC calling conventions, these arguments can be any combination of integer and floating-point values (i.e., passed in registers i2 through i5). Note that **CMAM_4** always transmits a full CM-5 packet, thus there is no performance advantage in passing less than four words of arguments to the remote function.

CMAM_4 sends the Active Message on the left data network and polls both, the left and right networks. **CMAM_reply_4** is identical to **CMAM_4**, but uses the right data network and polls only the right side. **CMAM_reply_4** should be used within message handlers for replies to avoid the danger of stack overflow (see **cmam**(3)). Neither **CMAM_4** nor **CMAM_reply_4** may be used from within a reply-message handler.

CMAM may be used to send Active Messages from the host (''control processor'' in CM-5 terminology) to the processing nodes. However, note that the executable running on the host is different from the one running on the processing nodes, therefore the address of functions differ. The linker **cmld**(1) arranges for functions named **CMPE_...** or **cmpe_...** on the processing nodes to be resolved correctly on the host. To send Active Messages to the host from processing nodes, see **CMAM_host**(3).

PERFORMANCE

CMAM_4 takes 2us, **CMAM_reply_4** takes 1.6us. Handling either one at arrival takes 2us.

SEE ALSO

CMAM(3) extends **CMAM_4** in that it sends an Active Message of arbitrary length, passing a buffer as argument. **CMAM_indirect_4**(3) is similar to **CMAM_4** but may cram an additional word of information into the CM-5 packet in certain circumstances.

cmam(3), **CMAM**(3), **CMAM_xfer**(3), **CMAM_host**(3), **CMAM_indirect_4**(3).

AUTHOR

Thorsten von Eicken, tve@CS.Berkeley.EDU

NAME

CMAM_xfer_N, CMAM_reply_xfer_N, CMAM_xfer_No, CMAM_reply_xfer_No, CMAM_xfer_Nb, CMAM_reply_xfer_Nb – Transfer aligned data block to segment

SYNOPSIS

#include <cmam/cmam.h>

void CMAM_xfer_N (int node, int seg_addr, void *buff, int byte_count);

void CMAM_xfer_No (int node, int seg_addr, void *buff, int byte_count);

void CMAM_xfer_Nb (int node, int seg_addr, void *buff, int byte_count);

void CMAM_reply_xfer_N (int node, int seg_addr, void *buff, int byte_count);

void CMAM_reply_xfer_No (int node, int seg_addr, void *buff, int byte_count);

void CMAM_reply_xfer_Nb (int node, int seg_addr, void *buff, int byte_count);

DESCRIPTION

CMAM_xfer_N transfers the block of *quadwords* at address **buff** and of length **byte_count** to a segment on the remote **node** previously opened with **CMAM_open_segment**(3). The destination address for the data on **node** is specified as a segment plus an unsigned byte offset. The segment and the offset are encoded in **seg_addr** which is the sum ('+') of the remote segment id (as returned by **CMAM_open_segment**) and the unsigned byte offset. This encoding limits the offset to 24 bits.

On arrival at the destination node, the data is stored in memory and the byte count associated with the segment is decremented. If the count becomes zero, the end-of-xfer function is called as described in **CMAM_open_segment**(3).

On arrival at the destination node, the data is stored in memory and the byte count associated with the segment is decremented. If the count becomes zero, the end-of-xfer function is called as described in **CMAM_open_segment**(3).

CMAM_xfer_N sends packets on the left data network and polls both, the left and right networks. **CMAM_reply_xfer_N** is identical to **CMAM_xfer_N** but sends packets on the right data network and polls only the right side. **CMAM_reply_xfer_N** should be used within message handlers to send replies to avoid the danger of stack overflow (see **cmam**(3)).

CMAM_xfer_N requires the source and the destination block the be doubleword aligned (i.e. (addr&7) == 0) and transfers floor(**byte_count**/16) quadword packets.

CMAM_xfer_No and **CMAM_reply_xfer_No** are similar to **CMAM_xfer_N** respectively **CMAM_reply_xfer_N** but require only word-aligment of the source data block (i.e. (**buff**&3) == 0).

CMAM_xfer_Nb and **CMAM_reply_xfer_Nb** transfer up to 8 bytes of any aligment to a remote segment.

All six function described in this man page are intended to be building blocks for more convenient block transfer functions such as **CMAM_xfer**(3).

PERFORMANCE

CMAM_xfer_N transfers data at up to 10Mb/s. A node sending and receiving xfers at the same time typically sends at 5Mb/s and receives at 5Mb/s. A node receiving two xfers to different segments accepts data at about 7.5Mb/s due to overhead incurred when switching between the two segments.

CMAM_xfer_No transfers data at most at 7Mb/s.

SEE ALSO

cmam(3), **CMAM_open_segment**(3), **CMAM_xfer**(3), **CMAM_xfer_4**(3), **CMAM**(3)

AUTHOR

Thorsten von Eicken, tve@CS.Berkeley.EDU

NAME

CMAM_xfer_[1234]i, CMAM_reply_xfer_[1234]i, CMAM_xfer_[12]d, CMAM_reply_xfer_[12]d – Transfer single data packet to segment

SYNOPSIS

#include <cmam/cmam.h>

void CMAM_xfer_1i (int node, int segment_id, unsigned int offset, int d1);

void CMAM_xfer_2i (int node, int segment_id, unsigned int offset, int d1, int d2);

void CMAM_xfer_3i (int node, int segment_id, unsigned int offset, int d1, int d2, int d3);

void CMAM_xfer_4i (int node, int segment_id, unsigned int offset, int d1, int d2, int d3, int d4);

void CMAM_xfer_1d (int node, int segment_id, unsigned int offset, double d1);

void CMAM_xfer_2d (int node, int segment_id, unsigned int offset, double d1, double d2);

void CMAM_reply_xfer_1i (int node, int segment_id, unsigned int offset, int d1);

void CMAM_reply_xfer_2i (int node, int segment_id, unsigned int offset, int d1, int d2);

void CMAM_reply_xfer_3i (int node, int segment_id, unsigned int offset, int d1, int d2, int d3);

void CMAM_reply_xfer_4i (int node, int segment_id, unsigned int offset, int d1, int d2, int d3, int d4);

void CMAM_reply_xfer_1d (int node, int segment_id, unsigned int offset, double d1);

void CMAM_reply_xfer_2d (int node, int segment_id, unsigned int offset, double d1, double d2);

DESCRIPTION

These functions transfer one CM-5 packet with 1 to 4 32-bit words of data to a segment on the remote **node** previously opened with **CMAM_open_segment**(3). The destination address for the data on **node** is specified as a segment plus an unsigned byte offset: **segment_id** (as returned by **CMAM_open_segment**) specifies the segment and **offset** specifies an unsigned 24-bit byte offset within the segment.

On arrival at the destination node, the data is stored in memory and the byte count associated with the segment is decremented. If the count becomes zero, the end-of-xfer function is called as described in **CMAM_open_segment**(3).

The **CMAM_xfer_*** functions send packets on the left data network and poll both, the left and right networks. The **CMAM_reply_xfer_*** functions are identical, but send packets on the right data network and poll only the right network. **CMAM_reply_xfer_*** should be used within message handlers to send replies to avoid the danger of stack overflow (see **cmam**(3)).

CMAM_xfer_4i and **CMAM_xfer_2d** require the destination to be doubleword aligned (i.e. (addr&7) == 0). The other functions transfering less than 4 32-bit words require word alignment of the destination.

PERFORMANCE

SEE ALSO

cmam(3), **CMAM_open_segment**(3), **CMAM_xfer**(3), **CMAM_xfer_N**(3), **CMAM**(3).

AUTHOR

Thorsten von Eicken, tve@CS.Berkeley.EDU

NAME

CMAM_host, CMAM_host_reply, CMAM_host_4, CMAM_host_reply_4 – Send Active Messages to host

SYNOPSIS

#include <cmam/cmam.h>

void CMAM_host (void (*fun)(), void *buffer, int byte_count);

void CMAM_host_reply (void (*fun)(), void *buffer, int byte_count);

void CMAM_host_4 (void (*fun)(), ...);

void CMAM_host_reply_4 (void (*fun)(), ...);

DESCRIPTION

CMAM_host, **CMAM_host_reply**, **CMAM_host_4**, and **CMAM_host_reply_4** are similar to **CMAM**(3), **CMAM_reply**(3), **CMAM_4**(3), respectively **CMAM_reply_4**(3), but send an Active Message to the host processor (''control processor'' in CM-5 terminology) instead of a processing node.

Note that the executable running on the host is different from the one running on the processing nodes and therefore the address of functions differ. While the linker **cmld**(1) arranges for functions named **CMPE_...** or **cmpe_...** on the processing nodes to be resolved correctly on the host, no similar symbol resolution is provided the other way around. One solution is to broadcast the address of all host handlers to the processing nodes at the beginning of a program. Alternatively, a table of handlers may be set-up on the host, its base address broadcast to the processing nodes and **CMAM_host_indirect**(3) used to send Active Messages to the host. Finally, the **cmld** shell script could be modifed to include resolution of host symbols within the processing node executable.

PERFORMANCE

These functions are very slow because an O.S. trap is required for every packet.

SEE ALSO

cmam(3), **CMAM**(3), **CMAM_4**(3), **CMAM_indirect**(3).

AUTHOR

Thorsten von Eicken, tve@CS.Berkeley.EDU

NAME

CMAM_host_xfer, CMAM_host_reply_xfer – Transfer data block to segment on host

SYNOPSIS

#include <cmam/cmam.h>

void CMAM_host_xfer (int seg_addr, void *buff, int byte_count);

void CMAM_host_reply_xfer (int seg_addr, void *buff, int byte_count);

DESCRIPTION

CMAM_host_xfer and **CMAM_host_reply_xfer** are similar to **CMAM_xfer**(3) respectively **CMAM_reply_xfer**(3) but transfer data to the host processor (''control processor'' in CM-5 terminology) instead of a processing node.

PERFORMANCE

These functions are very slow because an O.S. trap is required for every packet.

SEE ALSO

cmam(3), **CMAM_xfer**(3), **CMAM_host**(3).

AUTHOR

Thorsten von Eicken, tve@CS.Berkeley.EDU

NAME

CMAM_host_xfer_N, CMAM_reply_host_xfer_N, CMAM_host_xfer_No, CMAM_reply_host_xfer_No, CMAM_host_xfer_Nb, CMAM_reply_host_xfer_Nb – Transfer aligned data block to segment on host

SYNOPSIS

#include <cmam/cmam.h>

void CMAM_host_xfer_N (int seg_addr, void *buff, int byte_count);

void CMAM_host_xfer_No (int seg_addr, void *buff, int byte_count);

void CMAM_host_xfer_Nb (int seg_addr, void *buff, int byte_count);

void CMAM_host_reply_xfer_N (int seg_addr, void *buff, int byte_count);

void CMAM_host_reply_xfer_No (int seg_addr, void *buff, int byte_count);

void CMAM_host_reply_xfer_Nb (int seg_addr, void *buff, int byte_count);

DESCRIPTION

CMAM_host_xfer_N, **CMAM_host_xfer_No**, **CMAM_host_xfer_Nb**, **CMAM_host_reply_xfer_N**, **CMAM_host_reply_xfer_No** and **CMAM_host_reply_xfer_Nb** are similar to **CMAM_xfer_N**(3), **CMAM_xfer_No**(3), **CMAM_xfer_Nb**(3), **CMAM_reply_xfer_N**(3), **CMAM_reply_xfer_No**(3) respectively **CMAM_reply_xfer_Nb**(3) but transfer data to the host processor (''control processor'' in CM-5 terminology) instead of a processing node.

PERFORMANCE

These functions are very slow because an O.S. trap is required for every packet.

SEE ALSO

cmam(3), **CMAM_xfer_N**(3), **CMAM_host**(3).

AUTHOR

Thorsten von Eicken, tve@CS.Berkeley.EDU

NAME

CMAM_host_xfer_[1234]i, CMAM_reply_host_xfer_[1234]i, CMAM_host_xfer_[12]d, CMAM_reply_host_xfer_[12]d – Transfer single data packet to segment on host

SYNOPSIS

```
#include <cmam/cmam.h>

void CMAM_host_xfer_1i (int segment_id, unsigned int offset, int d1);
void CMAM_host_xfer_2i (int segment_id, unsigned int offset, int d1, int d2);
void CMAM_host_xfer_3i (int segment_id, unsigned int offset, int d1, int d2, int d3);
void CMAM_host_xfer_4i (int segment_id, unsigned int offset, int d1, int d2, int d3, int d4);

void CMAM_host_xfer_1d (int segment_id, unsigned int offset, double d1);
void CMAM_host_xfer_2d (int segment_id, unsigned int offset, double d1, double d2);

void CMAM_host_reply_xfer_1i (int segment_id, unsigned int offset, int d1);
void CMAM_host_reply_xfer_2i (int segment_id, unsigned int offset, int d1, int d2);
void CMAM_host_reply_xfer_3i (int segment_id, unsigned int offset, int d1, int d2, int d3);
void CMAM_host_reply_xfer_4i (int segment_id, unsigned int offset, int d1, int d2, int d3, int d4);

void CMAM_host_reply_xfer_1d (int segment_id, unsigned int offset, double d1);
void CMAM_host_reply_xfer_2d (int segment_id, unsigned int offset, double d1, double d2);
```

DESCRIPTION

CMAM_host_xfer_*[1..4]***i**, **CMAM_host_xfer_***[1..2]***d**, **CMAM_host_reply_xfer_***[1..4]***i** and **CMAM_host_reply_xfer_***[1..2]***d** are similar to **CMAM_xfer_***[1..4]***i**(3), **CMAM_xfer_***[1..2]***d**(3), **CMAM_reply_xfer_***[1..4]***i**(3) respectively **CMAM_reply_xfer_***[1..2]***d**(3) but transfer data to the host processor ("control processor" in CM-5 terminology) instead of a processing node.

PERFORMANCE

These functions are very slow because an O.S. trap is required for every packet.

SEE ALSO

cmam(3), **CMAM_xfer**(3), **CMAM_host**(3).

AUTHOR

Thorsten von Eicken, tve@CS.Berkeley.EDU

NAME

CMAM_set_tags, CMAM_set_handler, CMAM_set_reply_handler – Control hardware message tag usage

SYNOPSIS

#include <cmam/cmam.h>

void CMAM_set_tags (int CMAM_tag, int CMAM_indirect_tag, int CMAM_xfer_tag);

typedef void CMAM_tag_handler (void *NI_base);

void CMAM_set_handler (int tag, CMAM_tag_handler *handler);

void CMAM_set_reply_handler (int tag, CMAM_tag_handler *handler);

DESCRIPTION

CMAM has been designed such that it can be mixed with other message layers or extended by the user: CMAM can be directed to use any of the available hardware packet tags and handlers for non-CMAM packets can be registered with CMAM.

The CMAM initialization function **CMAM_enable**(3) sets CMAM up to use hardware tags 1, 2, and 3 for CMAM, CMAM_indirect, and CMAM_xfer, respectively, packets. After initialization, these tag assignments can be changed by calling **CMAM_set_tags** on the host (''control processor'' in CM-5 terminology) while the processing nodes are idle. As of version 7.1.4 of CMOST, only tags 0 through 3 are available to user message handlers. The remaining tags 4 through 15 are reserved (but not all used) by CMOST. Please complain to Thinking Machines Corp. for not leaving more tags to the user!

For multiple message layers to operate concurrently, a certain amount of coordination in handling incoming packets is necessary. Whenever CMAM polls the network and discovers the arrival of a packet, it dispatches to the appropriate tag-handler based on the hardware packet tag. The dispatch is performed by branching into a table indexed by the tag, each table ''entry'' holding 64 SPARC instructions. **CMAM_set_handler** takes a pointer to a tag-handler and copies 64 instructions into the appropriate entry in the left data network table. Similarly, **CMAM_set_reply_handler** copies into the right data network table. While copying the instructions, both functions adjust the PC-relative offsets of CALL instructions. The offsets of branch instructions are not modified (the offsets are too small to be adjustable in most cases). When called, the tag-handler receives the base address of the Network Interface chip (e.g. 0x20000000 on the PNs) as argument.

CMAM does not export any of its handlers for other message layers to call when they receive a CMAM packet. Several existing functions could be used, please contact the author if you need this type of functionality.

SEE ALSO

cmam(3), **CMAM_enable**(3), **CMAM**(3), **CMAM_indirect**(3), **CMAM_xfer**(3), **CMAM_host**(3).

AUTHOR

Thorsten von Eicken, tve@CS.Berkeley.EDU

Active Messages: a Mechanism for Integrated Communication and Computation [1]

Thorsten von Eicken
David E. Culler
Seth Copen Goldstein
Klaus Erik Schauser

{tve,culler,sethg,schauser}@cs.berkeley.edu

Report No. UCB/CSD 92/#675, March 1992
Computer Science Division — EECS
University of California, Berkeley, CA 94720

Abstract

The design challenge for large-scale multiprocessors is (1) to minimize communication overhead, (2) allow communication to overlap computation, and (3) coordinate the two without sacrificing processor cost/performance. We show that existing message passing multiprocessors have unnecessarily high communication costs. Research prototypes of message driven machines demonstrate low communication overhead, but poor processor cost/performance. We introduce a simple communication mechanism, *Active Messages*, show that it is intrinsic to both architectures, allows cost effective use of the hardware, and offers tremendous flexibility. Implementations on nCUBE/2 and CM-5 are described and evaluated using a split-phase shared-memory extension to C, *Split-C*. We further show that active messages are sufficient to implement the dynamically scheduled languages for which message driven machines were designed. With this mechanism, latency tolerance becomes a programming/compiling concern. Hardware support for active messages is desirable and we outline a range of enhancements to mainstream processors.

1 Introduction

With the lack of consensus on programming styles and usage patterns of large parallel machines, hardware designers have tended to optimize along specific dimensions rather than towards general balance. Commercial multiprocessors invariably focus on raw processor performance, with network performance in a secondary role, and the interplay of processor and network largely neglected. Research projects address specific issues, such as tolerating latency in dataflow architectures and reducing latency in cache-coherent architectures, accepting significant hardware complexity and modest processor performance in the prototype solutions. This paper draws on recent work in both arenas to demonstrate that the utility of exotic message-driven processors can be boiled down to a simple mechanism and that this mechanism can be implemented efficiently on conventional message passing machines. The basic idea is that the control information at the head of a message is the address of a user-level instruction sequence that will extract the message from the network and integrate it into the on-going computation. We call this *Active Messages*. Surprisingly, on commercial machines this mechanism is an order of magnitude more efficient than the message passing primitives that drove the original hardware designs. There is considerable room for improvement with direct hardware support, which can be addressed in an evolutionary manner. By smoothly integrating communication with computation, the overhead of communication is greatly reduced and an overlap of the

[1]This report first appeared in the Proceedings of the 19th International Symposium on Computer Architecture, ACM Press, May 1992, Gold Coast, Australia.

two is easily achieved. In this paradigm, the hardware designer can meaningfully address what balance is required between processor and network performance.

1.1 Algorithmic communication model

The most common cost model used in algorithm design for large-scale multiprocessors assumes the program alternates between computation and communication phases and that communication requires time linear in the size of the message, plus a start-up cost[9]. Thus, the time to run a program is $T = T_{compute} + T_{communicate}$ and $T_{communicate} = N_c(T_s + L_c T_b)$, where T_s is the start-up cost, T_b is the time per byte, L_c is the message length, and N_c is the number of communications. To achieve 90% of the peak processor performance, the programmer must tailor the algorithm to achieve a sufficiently high ratio of computation to communication that $T_{compute} \geq 9T_{communicate}$. A high-performance network is required to minimize the communication time, and it sits 90% idle!

If communication and computation are overlapped the situation is very different. The time to run a program becomes $T = \max(T_{compute} + N_c T_s,\ N_c L_c T_b)$. Thus, to achieve high processor efficiency, the communication and compute times need only balance, and the compute time need only swamp the communication overhead, *i.e.*, $T_{compute} \gg N_c T_s$. By examining the average time between communication phases ($T_{compute}/N_c$) and the time for message transmission, one can easily compute the per-processor bandwidth through the network required to sustain a given level of processor utilization. The hardware can be designed to reflect this balance. The essential properties of the communication mechanism are that the start-up cost must be low and that it must facilitate the overlap and co-ordination of communication with on-going computation.

1.2 Active Messages

Active Messages is an asynchronous communication mechanism intended to expose the full hardware flexibility and performance of modern interconnection networks. The underlying idea is simple: each message contains at its head the address of a user-level handler which is executed on message arrival with the message body as argument. The role of the handler is to get the message out of the network and into the computation ongoing on the processing node. The handler must execute quickly and to completion. As discussed below, this corresponds closely to the hardware capabilities in most message passing multiprocessors where a privileged interrupt handler is executed on message arrival, and represents a useful restriction on message driven processors.

Under Active Messages the network is viewed as a pipeline operating at a rate determined by the communication overhead and with a latency related to the message length and the network depth. The sender launches the message into the network and continues computing; the receiver is notified or interrupted on message arrival and runs the handler. To keep the pipeline full, multiple communication operations can be initiated from a node, and computation proceeds while the messages travel through the network. To keep the communication overhead to a minimum, Active Messages are not buffered except as required for network transport. Much like a traditional pipeline, the sender blocks until the message can be injected into the network and the handler executes immediately on arrival.

Tolerating communication latency has been raised as a fundamental architectural issue[1]; this is not quite correct. The real architectural issue is to provide the ability to overlap communication and computation, which, in-turn, requires low-overhead asynchronous communication. Tolerating latency then becomes a programming problem: a communication must be initiated sufficiently in advance of the use of its result. In Sections 2 and 3 we show two programming models where the programmer and compiler, respectively, have control over communication pipelining.

Active Messages is not a new parallel programming paradigm on par with send/receive or shared-memory: it is a more primitive communication *mechanism* which can be used to implement these paradigms (among others) simply and efficiently. Concentrating hardware design efforts on implementing fast Active Messages is more versatile than supporting a single paradigm with special hardware.

1.3 Contents

In this paper, we concentrate on message-based multiprocessors and consider machines of similar base technology representing the architectural extremes of processor/network integration. Message passing machines, including the nCUBE/2, iPSC/2, iPSC/860 and others, treat the network essentially as a fast I/O device. Message driven architectures, including Monsoon[18, 17] and the J-Machine[5], integrate the network deeply into the processor. Message reception is part of the basic instruction scheduling mechanism and message send is supported directly in the execution unit.

Section 2 examines current message passing machines in detail. We show that send/receive programming models make inefficient use of the underlying hardware capabilities. The raw hardware supports a simple form of Active Messages. The utility of this form of communication is demonstrated in terms of a fast, yet powerful asynchronous communication paradigm. Section 3 examines current message driven architectures. We show that the power of message driven processing, beyond that of Active Messages, is costly to implement and not required to support the implicitly parallel programming languages for which these architectures were designed. Section 4 surveys the range of hardware support that could be devoted to accelerating Active Messages.

2 Message passing Architectures

In this section we examine message passing machines, the one architecture that has been constructed and used on a scale of a thousand high-performance processors. We use the nCUBE/2 and the CM-5 as primary examples.

The nCUBE/2 has up to a few thousand nodes interconnected in a binary hypercube network. Each node consists of a CPU-chip and DRAM chips on a small double-sided printed-circuit board. The CPU chip contains a 64-bit integer unit, an IEEE floating-point unit, a DRAM memory interface, a network interface with 28 DMA channels, and routers to support cut-through routing across a 13-dimensional hypercube. The processor runs at 20 Mhz and delivers roughly 5 MIPS or 1.5 MFLOPS.

The CM-5 has has up to a few thousand nodes interconnected in a "hypertree" (an incomplete fat tree). Each node consists of a 33 Mhz Sparc RISC processor chip-set (including FPU, MMU and cache), local DRAM memory and a network interface to the hypertree and broadcast/scan/prefix control networks. In the future, each node will be augmented with four vector units.

We first evaluate the machines using the traditional programming models. Then we show that Active Messages are well-suited to the machines and support more powerful programming models with less overhead.

2.1 Traditional programming models

In the traditional programming model for message passing architectures, processes communicate by matching a *send* request on one processor with a *receive* request on another. In the synchronous, or crystalline[9] form, send and receive are blocking — the send blocks until the corresponding receive is executed and only then is data transferred. The main advantage of the blocking send/receive model is its simplicity. Since data is only transferred after both its source and destination addresses are known, no buffering is required at the source or destination processors.

Blocking send/receive communication exacerbates the effects of network latency on communication latency[2]: in order to match a send with a receive a 3-phase protocol, shown in Figure 1, is required: the sender first transmits a request to the receiver which returns an acknowledgement upon executing a matching receive operation and only then is data transferred. With blocking send/receive, it is impossible to overlap communication with computation and thus the network bandwidth cannot be fully utilized.

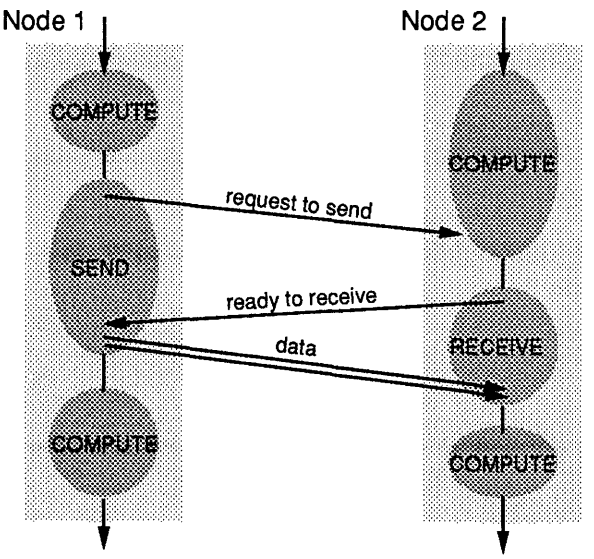


Figure 1: *Three-phase protocol for synchronous send and receive. Note that the communication latency is at best three network trips and that both send and receive block for at least one network round-trip each.*

To avoid the three-phase protocol and to allow overlap of communication and computation, most message passing implementations offer non-blocking operation: *send* appears instantaneous to the user program. The message layer buffers the message until the network port is available, then the message is transmitted to the recipient, where it is again buffered until a matching *receive* is executed. As shown in the ring communication example in Figure 2, data can be exchanged while computing by executing all sends before the computation phase and all receives afterwards.

Table 1 shows the performance of send/receive on several current machines. The start-up costs are on the order of a thousand instruction times. This is due primarily to buffer management. The CM-5 is blocking and uses a three-phase protocol. The iPSC long messages use a three-phase protocol to ensure that enough buffer space is available at the receiving processor. However, the start-up costs alone prevent overlap of communication and computation, except for very large messages. For example, on the nCUBE/2 by the time a second send is executed up to 130 bytes of the first message will have reached the destination. Although the network bandwidth on all these machines is limited, it is difficult to utilize it fully, since this requires multiple simultaneous messages per processor.

[2]We call communication latency the time from initiating a send in the user program on one processor to receiving the message in the user program on another processor, *i.e.*, the sum of software overhead, network interface overhead and network latency.

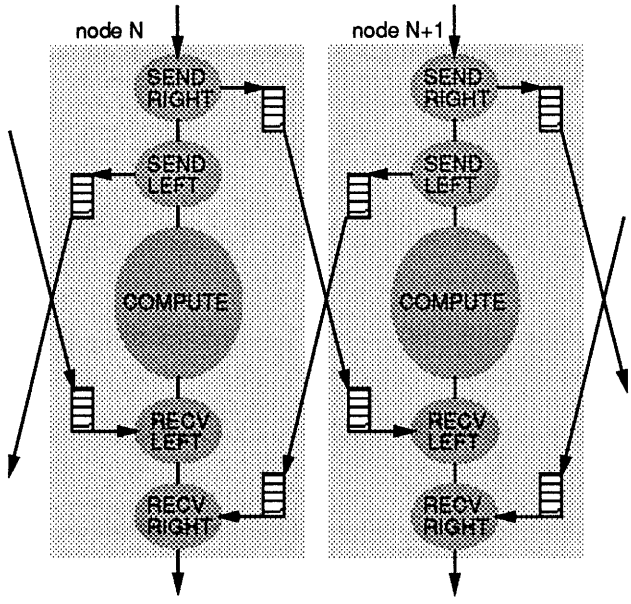


Figure 2: *Communication steps required for neighboring processors in a ring to exchange data using asynchronous send and receive. Data can be exchanged while computing by executing all sends before the computation phase and all receives afterwards. Note that buffer space for the entire volume of communication must be allocated for the duration of the computation phase!*

Machine	T_s [μs/mesg]	T_b [μs/byte]	T_{fp} [μs/flop]
iPSC[8]	4100	2.8	25
nCUBE/10[8]	400	2.6	8.3
iPSC/2[8]	700	0.36	3.4
	390†	0.2	
nCUBE/2	160	0.45	0.50
iPSC/860[13]	160	0.36	0.033[7]
	60†	0.5	
CM-5‡	86	0.12	0.33[7]

†: messages up to 100 bytes

‡: blocking send/receive

Table 1: *Asynchronous send and receive overheads in existing message passing machines.* T_s *is the message start-up cost (as described in Section 1.1),* T_b *is the per-byte cost and* T_{fp} *is the average cost of a floating-point operation as reference point.*

2.2 Active Messages

Although the hardware costs of message passing machines are reasonable, the effectiveness of the machine is low under traditional send/receive models due to poor overlap of communication and computation, and due to high communication overhead. Neither of these shortcomings can be attributed to the base hardware: for example, initiating a transmission on the nCUBE/2 takes only two instructions, namely to set-up the DMA[3]. The discrepancy between the raw hardware message initiation cost and the observed cost can be explained by a mismatch between the programming model and the hardware functionality. Send and receive is not native to the hardware: the hardware allows one processor to send a message to another one and cause an interrupt to occur at arrival. In other words the hardware model is really one of launching messages into the network and causing a handler to be executed asynchronously upon arrival. The only similarity between the hardware operation and the programming model is in respect to memory address spaces: the source address is determined by the sender while the destination address is determined by the receiver[4].

Active Messages simply generalize the hardware functionality by allowing the sender to specify the address of the handler to be invoked on message arrival. Note that this relies on a uniform code image on all nodes, as is commonly used (the SPMD programming model). The handler is specified by a user-level address and thus traditional protection models apply. Active Messages differ from general remote procedure call (RPC) mechanisms in that the role of the Active Message handler is not to perform computation on the data, but to extract the data from the network and integrate it into the ongoing computation with a small amount of work. Thus, concurrent communication and computation is fundamental to the message layer. Active Messages are not buffered, except as required for network transport. Only primitive scheduling is provided: the handlers interrupt the computation immediately upon message arrival and execute to completion.

The key optimization in Active Messages compared to send/receive is the elimination of buffering. Eliminating buffering on the receiving end is possible because either storage for arriving data is pre-allocated in the user program or the message holds a simple request to which the handler can immediately reply. Buffering on the sending side is required for the large messages typical in high-overhead communication models. The low overhead of Active Message makes small messages more attractive, which eases program development and reduces network congestion. For small messages, the buffering in the network itself is typically sufficient.

Deadlock avoidance is a rather tricky issue in the design of Active Messages. Modern network designs are typically deadlock-free provided that nodes continuously accept incoming messages. This translates into the requirement that message handlers are not allowed to block, in particular a reply (from within a handler) must not busy-wait if the outgoing channel is backed-up.

2.2.1 Active Messages on the nCUBE/2

The simplicity of Active Messages and its closeness to hardware functionality translate into fast execution. On the nCUBE/2 it is possible to send a message containing one word of data in 21 instructions taking $11\mu s$. Receiving such a message requires 34 instructions taking $15\mu s$, which includes taking an interrupt on message arrival and dispatching it to user-level. This near order of magnitude reduction ($T_c = 30\mu s$, $T_b = 0.45\mu s$) in send overhead is greater than that achieved by a hardware generation. Table 2 breaks the instruction counts down into the various tasks performed.

The Active Message implementation reduces buffer management to the minimum required for actual data transport. On the nCUBE/2 where DMA is associated with each network channel, one memory buffer

[3]On the nCUBE/2, each of the 13 hypercube channels has independent input and output DMAs with a base-address and a count register each. Sending or receiving a message requires loading the address and the count.

[4]Shared-memory multiprocessor advocates argue that this is the major cause of programming difficulty of these machines.

Task	Instruction count send	receive
Compose/consume message	6	9
Trap to kernel	2	–
Protection	3	–
Buffer management	3	3
Address translation	1	1
Hardware set-up	6	2
Scheduling	–	7
Crawl-out to user-level	–	12
Total	21	34

Table 2: *Breakdown into tasks of the instructions required to send and receive a message with one word of data on the nCUBE/2. "Message composition" and "consumption" include overhead for a function call and register saves in the handler. "Protection" checks the destination node and limits message length. "Hardware set-up" includes output channel dispatch and channel ready check. " Scheduling" accounts for ensuring handler atomicity and dispatch. "Crawling out to user-level" requires setting up a stack frame and saving state to simulate a return-from-interrupt at user-level.*

per channel is required. Additionally, it is convenient to associate two buffers with the user process: one to compose the next outgoing message and one for handlers to consume the arrived message and compose eventual replies. This set-up reduces buffer management to swapping pointers for a channel buffer with a user buffer. Additional buffers must be used in exceptional cases to prevent deadlock: if a reply from within a handler blocks for "too long", it must be buffered and retried later so that further incoming messages can be dispatched. This reply buffering is not performed by the message layer itself, rather REPLY returns an error code and the user code must perform the buffering and retry. Typically the reply (or the original request) is saved onto the stack and the handlers for the incoming messages are nested within the current handler.

The breakdown of the 55 instructions in Table 2 shows the sources of communication costs on the nCUBE/2. A large fraction of instructions (22%) are used to simulate user-level interrupt handling. Hardware set-up (15%) is substantial due to output channel selection and channel-ready checks. Even the minimal scheduling and buffer management of Active Messages is still significant (13%). Note however, that the instruction counts on the nCUBE/2 are slightly misleading, in that the system call/return instructions and the DMA instructions are far more expensive than average.

The instruction breakdown shows clearly that Active Messages are very close to the absolute minimal message layer: only the crawl-out is Active Message specific and could potentially be replaced. Another observation is that most of the tasks performed here in software could be done easily in hardware. Hardware support for active messages could significantly reduce the overhead with a small investment in chip complexity.

2.2.2 Active Messages on the CM-5

The Active Messages implementation on the CM-5 differs from the nCUBE/2 implementation for five reasons[5]:

[5]The actual network interface is somewhat more complicated than described below, we only present the aspects relevant to this discussion.

1. The CM-5 provides user-level access to the network interface and the node kernel time-shares the network correctly among multiple user processes.

2. The network interface only supports transfer of packets of up to 24 bytes (including 4 bytes for the destination node) and the network routing does not guarantee any packet ordering.

3. The CM-5 has two identical, disjoint networks. The deadlock issues described above are simply solved by using one network for requests and the other for replies. One-way communication can use either.

4. The network interface does not have DMA. Instead, it contains two memory-mapped FIFOs per network, one for outgoing messages and one for incoming ones. Status bits indicate whether incoming FIFOs hold messages and whether the previous outgoing message has been successfully sent by the network interface. The network interface discards outgoing messages if the network is backed-up or if the process is time-sliced during message composition. In these cases the send has to be retried.

5. The network interface generally does not use interrupts in the current version due to their prohibitive cost. (The hardware and the kernel do support interrupts, but their usefulness is limited due to the cost.) For comparison, on the nCUBE/2 the interrupt costs the same as the system call which would have to be used instead since there is no user-level access to the network interface.

Sending a packet-sized Active Message amounts to stuffing the outgoing FIFO with a message having a function pointer at its head. Receiving such an Active Message requires polling, followed by loading the packet data into argument registers, and calling the handler function. Since the network interface status has to be checked whenever a message is sent (to check the send-ok status bit), servicing incoming messages at send time costs only two extra cycles. Experience indicates that the program does not need to poll explicitly unless it enters a long computation-only loop.

Sending multi-packet messages is complicated by the potential reordering of packets in the network. For large messages, set-up is required on the receiving end. This involves a two-phase protocol for GET, and a three-phase protocol for PUT (discussed below). Intermediate-sized messages use a protocol where each packet holds enough header information (at the expense of the payload) that the arrival order is irrelevant.

The performance of Active Message on the CM-5 is very encouraging: sending a single-packet Active Message (function address and 16 bytes of arguments) takes 1.6μs ($\approx$ 50 cycles) and the receiver dispatch costs 1.7μs. The largest fraction of time is spent accessing the network interface across the memory bus. A prototype implementation of blocking send/receive on top of Active Messages compares favorably with the (not yet fully optimized) vendor's library: the start-up cost is $T_c = 23\mu s$ (vs. $86\mu s$) and the per byte cost is $T_b = 0.12\mu s$ (identical). Note that due to the three-phase protocol required by send/receive, T_c is an order of magnitude larger than the single packet send cost. Using different programming models such as Split-C, the cost off communication can be brought down to the Active Message packet cost.

2.3 Split-C: an experimental programming model using Active Messages

To demonstrate the utility of Active Messages, we have developed a simple programming model that provides split-phase remote memory operations in the C programming language. The two split-phase operations provided are PUT and GET: as shown in Figure 3a, PUT copies a local memory block into a remote memory at an address specified by the sender. GET retrieves a block of remote memory (address specified by sender) and makes a local copy. Both operations are non-blocking and do not require explicit coordination with the remote processor (the handler is executed asynchronously). The most common versions of PUT and GET increment a separately specified flag on the processor that receives the data. This allows simple

synchronization through checking the flag or busy-waiting. Operating on blocks of memory can yield large messages which are critical to performance on current hardware as seen below.

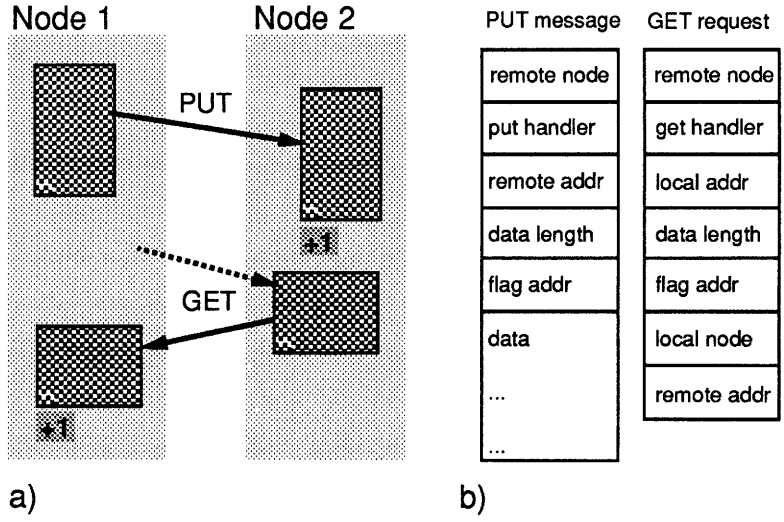


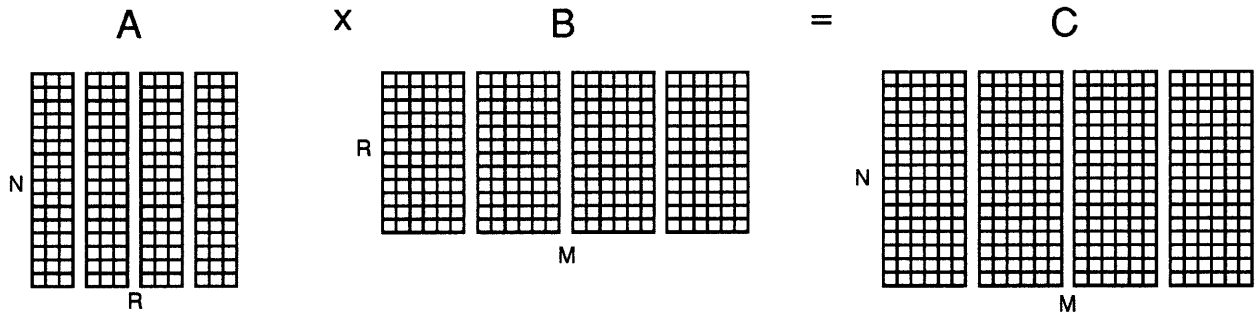
Figure 3: *Split-C* PUT *and* GET *perform split-phase copies of memory blocks to/from remote nodes. Also shown are the message formats.*

The implementations of PUT and GET consist of two parts each: a message formatter and a message handler. Figure 3b shows the message formats. PUT messages contain the instruction address of the PUT handler, the destination address, the data length, the completion-flag address, and the data itself. The PUT handler simply reads the address and length, copies the data and increments the flag. GET requests contain the information necessary for the GET handler to reply with the appropriate PUT message. Note that it is possible to provide versions of PUT and GET that copy data blocks with a stride or any other form of gather/scatter[6].

To demonstrate the simplicity and performance of Split-C, Figure 4 shows a matrix multiply example that achieves 95% of peak performance on large nCUBE/2 configurations. In the example, the matrices are partitioned in blocks of columns across the processors. For the multiplication of $C = A \times B$ each processor GETs one column of A after another and performs a rank-1 update (DAXPY) with the corresponding elements of its own columns of B into its columns of C. To balance the communication pattern, each processor first computes with its own column(s) of A and then proceeds by getting the columns of the next processor. Note that this algorithm is independent of the network topology and has a familiar shared-memory style. The remote memory access and its completion are made explicit, however.

The key to obtaining high performance is to overlap communication and computation. This is achieved by GETting the column for the next iteration while computing with the current column. It is now necessary to balance the latency of the GET with the time taken by the computation in the inner loops. Quantifying the computational cost is relatively easy: for each GET the number of multiply-adds executed is Nm (where m is the number of local columns of B and C) and each multiply-add takes 1.13μs. To help understand the latency of the GET, Figure 6 shows a diagram of all operations and delays involved in the unloaded case.

[6]Split-C exposes the underlying RPC mechanism the programmer as well, so that specialized communication structures can be constructed, e.g., enqueue record.



The matrices are partitioned in blocks of columns across the processors. For the multiplication of $C_{N\times M} = A_{N\times R} \times B_{R\times M}$ *each processor* GETs *one column of* A *after another and performs a rank-1 update (DAXPY) with its own columns of* B *into its columns of* C*. To balance the communication pattern, each processor first computes with its own column(s) of* A *and then proceeds by getting the columns of the next processor. This network topology independent algorithm achieves 95% of peak performance on large nCUBE/2 configurations.*

```
int N, R, M;                /* matrix dimensions (see figure) */
double A[R/P][N], B[M/P][R], C[M/P][N]; /* matrices */
int i, j, k;                /* indices */
int j0, dj, nj;             /* initial j, delta j (j=j0+dj), next j */
int P, p;                   /* number of processors, my processor */
int Rp = R/P;
double V0[N], V1[N];        /* buffers for getting remote columns */
double *V=V0, *nV=V1, *tV; /* current column, next column, temp column */
static int flag = 0;        /* synchronization flag */
extern void get(int proc, void *src, int size, void *dst, int &flag);

j0 = p * Rp;                              /* starting column */
get(p, &A[0][0], N*sizeof(double), nV, &flag); /* get first column of A  */
for(dj=0; dj<R; dj++) {                   /* loop over all columns of A */
    j = (j0+dj)%R; nj = (j0+dj+1)%R;      /* this&next column index */
    while(!check(1, &flag)) ;             /* wait for previous get */
    tV=V; V=nV; nV=tV;                    /* swap current&next column */
    if(nj != j0)                          /* if not done, get next column */
        get(nj/Rp, &A[nj%Rp][0], N*sizeof(double), nV, &flag);
    for(k=0; k<M/P; k++)                  /* accum. V into every col. with scale */
        for(i=0; i<N; ++)                 /* unroll 4x (not shown) */
            C[i][k] = C[i][k] + V[i]*B[j][k];
}
```

Figure 4: Matrix multiply example in Split-C.

The two top curves in Figure 5 show the performance predicted by the model and measured on a 128 node nCUBE/2, respectively, as the number of columns per processor of A is varied from 1 to 32. N is kept constant ($N = 128$) and R is adjusted to keep the total number of arithmetic operations constant ($R = 262144/M$). The matrix multiply in the example is computation bound if each processor holds more than two columns of A (*i.e.*, $m > 2$). The two bottom curves show the predicted and measured network utilization. The discrepancy between the model and the measurement is due to the fact that network contention is not modeled. Note that while computational performance is low for small values of m, the joint processor and network utilization is relatively constant across the entire range. As the program changes from a communication to a computation problem the "overall performance" is stable.

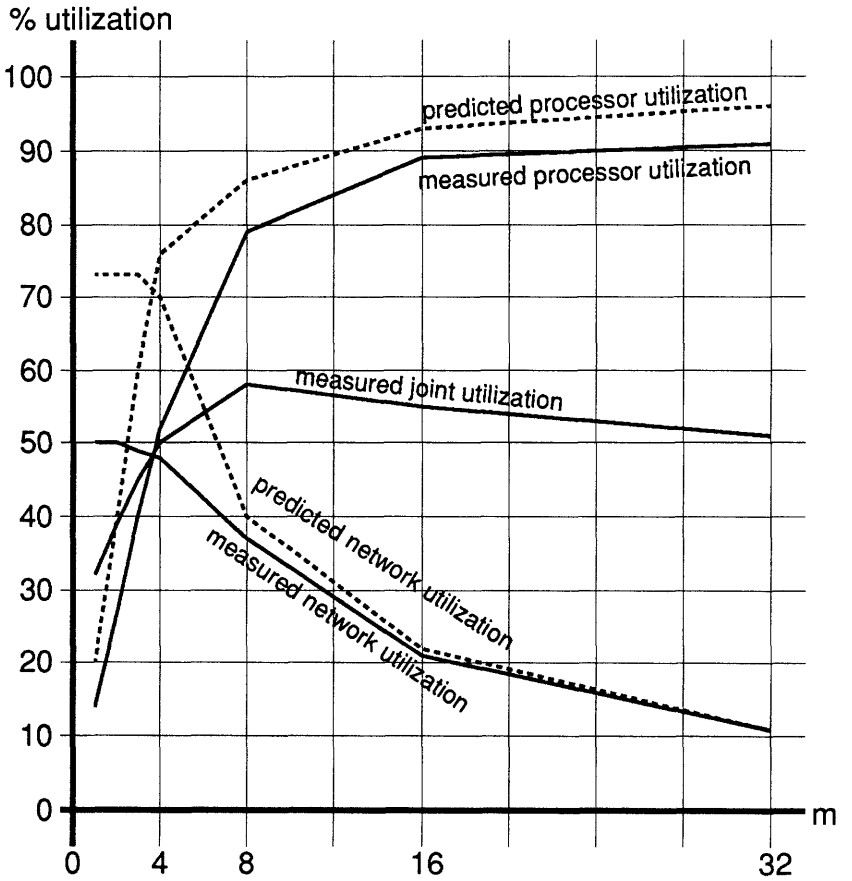


Figure 5: *Performance of Split-C matrix multiply on 128 processors compared to predicted performance using the model shown in Figure 6.*

2.4 Observations

Existing message passing machines have been criticized for their high communication overhead and the inability to support global memory access. With Active Messages we have shown that the hardware is capable of delivering close to an order of magnitude improvement today if the right communication mechanism is used, and that a global address space may well be implemented in software. Split-C is an

example of how Active Messages can be incorporated into a coarse-grain SPMD (single-program multiple-data) programming language. It generalizes shared memory read/write by providing access to blocks of memory including simple synchronization. It does not, however, address naming issues.

Using Active Messages to guide the design, it is possible to improve current message passing machines in an evolutionary, rather than revolutionary, fashion. In the next section, we examine research efforts to build hardware which uses a different approach to provide another magnitude of performance improvement.

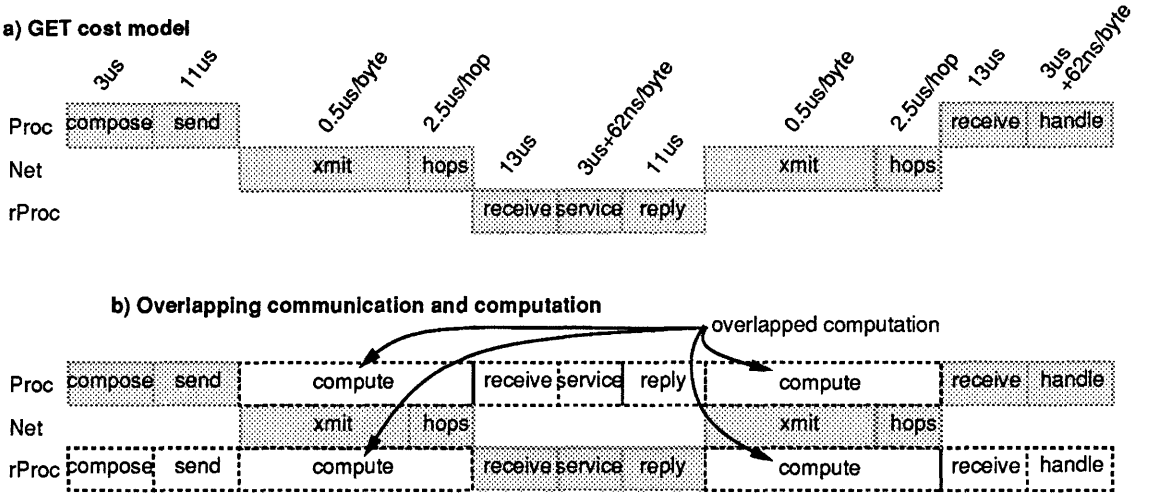


Figure 6: *Performance model for* GET. Compose *accounts for the time to set-up the request.* Xmit *is the time to inject the message into the network and* hops *is the time taken for the network hops.* Service *includes for copying the data into the reply buffer and* handle *for the time to copy the data into the destination memory block.*

3 Message driven architectures

Message driven architectures such as the J-Machine and Monsoon expend a significant amount of hardware to integrate communication into the processor. Although the communication performance achieved by both machines is impressive, the processing performance is not. At first glance this seems to come from the fact that the processor design is intimately affected by the network design and that the prototypes in existence could not utilize traditional processor design know-how. In truth, however, the problem is deeper: in message driven processors a context lasts only for the duration of a message handler. This lack of locality prevents the processor from using large register sets. In this section, we argue that the hardware support for communication is partly counter-productive. Simpler, more traditional, processors can be built without unduly compromising either the communication or the processing performance.

3.1 Intended programming model

The main driving force behind message driven architectures is to support languages with dynamic parallelism, such as Id90[15], Multilisp[10], and CST[12]. Computation is driven by messages, which contain the name of a handler and some data. On message arrival, storage for the message is allocated in a scheduling

queue. When the message reaches the head of the queue, the handler is executed with the data as arguments. The handler may perform arbitrary computation, in particular it may synchronize and suspend. This ability to suspend requires general allocation and scheduling on message arrival and is the key difference with respect to Active Messages.

In the case of the J-Machine, the programming model is put forward in object-oriented language terms[6]: the handler is a method, the data holds the arguments for the method and usually one of them names the object the method is to operate on. In a functional language view, the message is a closure with a code pointer and all arguments of the closure. Monsoon is usually described from the dataflow perspective[18] and messages carry tokens formed of an instruction pointer, a frame pointer and one piece of data. The data value is one of the operands of the specified instruction, the other is referenced relative to the frame pointer.

The fundamental difference between the message driven model and Active Messages is where computation-proper is performed: in the former, computation occurs in the message handlers whereas in the latter it is in the "background" and handlers only remove messages from the network transport buffers and integrate them into the computation. This difference significantly affects the nature of allocation and scheduling performed at message arrival.

Because a handler in the message driven model may suspend waiting for an event, the lifetime of the storage allocated in the scheduling queue for messages varies considerably. In general, it cannot be released in simple FIFO or LIFO order. Moreover, the size of the scheduling queue does not depend on the rate at which messages arrive or handlers are executed, but on the amount of excess parallelism in the program[4]. Given that the excess parallelism can grow arbitrarily (as can the conventional call stack) it is impractical to set aside a fraction of memory for the message queue, rather it must be able to grow to the size of available memory.

Active Message handlers, on the other hand, execute immediately upon message arrival, cannot suspend, and have the responsibility to terminate quickly enough not to back-up the network. The role of a handler is to get the message out of the network transport buffers. This happens either by integrating the message into the data structures of the ongoing computation or, in the case of remote service requests, by immediately replying to the requester. Memory allocation upon message arrival occurs only as far as is required for network transport (*e.g.* if DMA is involved) and scheduling is restricted to interruption of the ongoing computation by handlers. Equivalently, the handlers could run in parallel with the computation on separate dedicated hardware.

3.2 Hardware Description

The Monsoon and J-Machine hardware is designed to support the message driven model directly. The J-Machine has a 3-D mesh of processing nodes with a single-chip CPU and DRAM each. The CPU has a 32-bit integer unit with a closely integrated network unit, a small static memory and a DRAM interface (but no floating-point unit). The hardware manages the scheduling queue as a fixed-size ring buffer in on-chip memory. Arriving messages are transferred into the queue and serviced in FIFO order. The first word of each message is interpreted as an instruction pointer and the message is made available to the handler as one of the addressable data segments. The J-Machine supports two levels of message priorities in hardware and two independent queues are maintained. Each message handler terminates by executing a SUSPEND instruction that causes the next message to be scheduled.

In Monsoon, messages arrive into the token queue. The token queue is kept in a separate memory proportional in size to the frame store. It provides storage for roughly 16 tokens per frame on average[7]. The queuing policy allows both FIFO and LIFO scheduling. The ALU pipeline is 8-way interleaved, so eight handlers can be active simultaneously. As soon as a handler terminates or suspends by blocking on a

[7] A token queue store of 64K tokens for 256K words of frame store and an expected average frame size of 64 words.

synchronization event, a token is popped from the queue and a new handler starts executing in the vacated pipeline interleave.

A common characteristic of both machines is that the amount of state available to an executing handler is very small: four data and three address registers in the J-Machine, an accumulator and three temporary registers in Monsoon. This reflects the fact that the computation initiated by a single message is small, typically less than ten arithmetic operations. This small amount of work cannot utilize many registers and since no locality is preserved from one handler to the next, no useful values could be carried along.

It is interesting to note that the J-Machine hardware does not actually support the message driven programming model fully in that the hardware message queue is managed in FIFO order and of fixed size. If a handler does not run to completion, its message must be copied to an allocated region of non-buffer memory by software. This happens for roughly $1/3$ of all messages. The J-Machine hardware does support Active Messages, however, in which case the message queue serves only as buffering. Close to $1/3$ of the messages hold a request to which the handler immediately replies and general allocation and scheduling is not required.

In Monsoon, the fact that tokens are popped from the queue means that the storage allocated for an arriving message is deallocated upon message handler execution. If a handler suspends, all relevant data is saved in pre-allocated storage in the activation frame thus, unlike the J-Machine, Monsoon does implement the message driven model, but at the cost of a large amount of high-speed memory.

3.3 TAM: compiling to Active Messages

So far, we have argued that the message driven execution model is tricky to implement correctly in hardware due to the fact that general memory allocation and scheduling are required upon message arrival. Using hardware that implements Active Messages, it is easy to simulate the message driven model by performing the allocation and scheduling in the message handler. Contrary to expectation this does not necessarily result in lower performance than a direct hardware implementation because software handlers can exploit and optimize special cases.

TAM[3] (Threaded Abstract Machine), a fine-grain parallel execution model based on Active Messages, goes one step further and requires the compiler to help manage memory allocation and scheduling. It is currently used as a compilation target for implicitly parallel languages such as Id90. When compiling for TAM, the compiler produces sequences of instructions, called *threads*, performing the computation proper. It also generates handlers, called *inlets*, for all messages to be received by the computation. Inlets are used to receive the arguments to a function, the results of called (child) functions, and the responses of global memory accesses. All accesses to global data structures are split-phase, allowing computation to proceed while requests travel through the network.

For each function call, an *activation frame* is allocated. When an inlet receives a message it typically stores the data in the frame and schedules a thread within the activation. Scheduling is handled efficiently by maintaining the scheduling queue within the activation frame: each frame, in addition to holding all local variables, contains counters used for synchronizing threads and inlets, and provides space for the *continuation vector* — the addresses of all currently enabled threads of the activation. Enabling a thread simply consists of pushing its instruction address into the continuation vector and possibly linking the frame into the ready queue. Figure 7 shows the activation tree data structure.

Service requests, such as remote reads, can typically be replied-to immediately and need no memory allocation or scheduling beyond what Active Messages provides. However, in exceptional cases requests must be delayed either for a lack of resources or because servicing inside the handler is inadequate. To amortize memory allocation, these requests are of fixed size and queue space is allocated in chunks.

Maintaining thread addresses in frames provides a natural two-level scheduling hierarchy. When a frame is scheduled (*activated*), enabled threads are executed until the continuation vector is empty. When

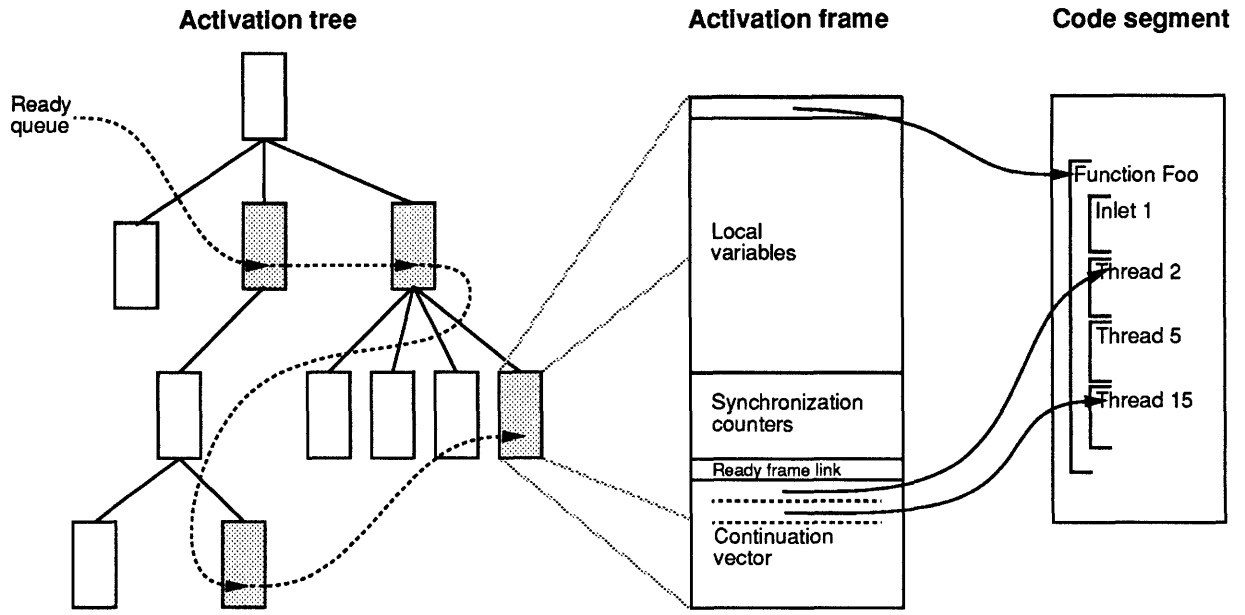


Figure 7: *TAM activation tree and embedded scheduling queue. For each function call, an activation frame is allocated. Each frame, in addition to holding all local variables, contains counters used to synchronize threads and inlets, and provides space for the* continuation vector *— the addresses of all currently enabled threads of the activation. On each processor, all frames holding enabled threads are linked into a ready queue. Maintaining the scheduling queue within the activation keeps costs low: enabling a thread simply consists of pushing its instruction address into the continuation vector and sometimes linking the frame into the ready queue. Scheduling the next thread within the same activation is simply a pop-jump.*

a message is received, two types of behavior can be observed: either the message is for the currently active frame and the inlet simply feeds the data into the computation, or the message is for a dormant frame in which case the frame may get added to the ready queue, but the ongoing computation is otherwise undisturbed.

Using the TAM scheduling hierarchy, the compiler can improve the locality of computation by synchronizing in message handlers and enabling computation only when a group of messages has arrived (one example is when all prerequisite remote fetches for an inner loop body have completed). This follows the realization that while the arrival of one message enables only a small amount of computation, the arrival of several closely related messages can enable a significant amount of computation. In cases beyond the power of compile-time analysis, the run-time scheduling policy dynamically enhances locality by servicing a frame until its continuation vector is empty.

As a result of the TAM compilation model, typically no memory allocation is required upon message arrival. Dynamic memory allocation is only performed in large chunks for activation frames and for global arrays and records. Locality of computation is enhanced by the TAM scheduling hierarchy. It is possible to implement TAM scheduling well even without any hardware support: on a uniprocessor[8] the overall cost for dynamic scheduling amounts to doubling the number of control-flow instructions relative to languages such as C. However, the overall performance depends critically on the cost of Active Messages. Table 3 summarizes the frequency of various kinds of messages in the current implementation. On average, a

[8]Id90 requires dynamic scheduling even on uniprocessors.

message is sent and received every eight TAM instructions (equivalent to roughly 20 RISC instructions). Note that these statistics are sensitive to optimizations. For example, significant changes can be expected from a software cache for remote arrays.

Message types	data words	frequency
Frame-frame	0	1%
	1	10%
	2	1%
Store request	1	8%
Fetch request	0	40%
Fetch reply	1	40%

Table 3: *Frequency of various message types and sizes (represented by the number of data values transmitted) in the current implementation of TAM. On average, a message is sent and received every 8 TAM instructions. These statistics are sensitive to compiler optimizations and, in some sense, represent a worst case scenario.*

4 Hardware support for Active Messages

Active messages provide a precise and simple communication mechanism which is independent of any programming model. Evaluating new hardware features can be restricted to evaluating their impact on Active Messages. The parameters feeding into the design are the size and frequency of messages, which depend on the expected workload and programming models.

Hardware support for active messages falls into two categories: improvements to network interfaces and modifications to the processor to facilitate execution of message handlers. The following subsections examine parts of the design space for each of these points of view.

4.1 Network interface design issues

Improvements in the network interface can significantly reduce the overhead of composing a message. Message reception benefits from these improvements as well, but also requires initiation of the handler.

Large messages: The support needed for large messages is a superset of that for small messages. To overlap computation with large message communication, some form of DMA transfer must be used. To set-up the DMA on the receiving side, large messages must have a header which is received first. Thus, if small messages are well supported, a large message should be viewed as a small one with a DMA transfer tacked-on.

Message registers: Composing small messages in memory buffers is inefficient: much of the information present in a small message is related to the current processor state. It comes from the instruction stream, processor registers and sometimes from memory. At the receiving end, the message header is typically moved into processor registers to be used for dispatch and to address data. Direct communication between the processor and the network interface can save instructions and bus transfers. In addition, managing the memory buffers is expensive.

The J-Machine demonstrates an extreme alternative for message composition: in a single SEND instruction the contents of two processor registers can be appended to a message. Message reception,

however, is tied to memory buffers (albeit on-chip). A less radical approach is to compose messages in registers of a network coprocessor.

Reception can be handled similarly: when received, a message appears in a set of registers. A (coprocessor) receive instruction enables reception of the next message. In case a coprocessor design is too complex, the network interface can also be accessed as a memory mapped device (as is the case in the CM-5).

Reuse of message data: Providing a large register set in the network interface, as opposed to network FIFO registers, allows a message to be composed using portions of other messages. For example, the destination for a reply is extracted from the request message. Also, multiple requests are often sent with mostly identical return addresses. Keeping additional context information such as the current frame pointer and a code base pointer in the network interface can further accelerate the formatting of requests.

The NIC[11] network interface contains 5 input and 5 output registers which are used to set-up and consume messages. Output registers retain their value after a message is sent, so that consecutive messages with identical parts can be sent cheaply. Data can be moved from input to an output registers to help re-using data when replying or forwarding messages.

Single network port: Multiple network channels connected to a node should not be visible to the message layer. On the nCUBE/2, for example, a message must be sent out on the correct hypercube link by the message layer, even though further routing in the network is automatic. The network interface should allow at least two messages to be composed simultaneously or message composition must be atomic. Otherwise, replies within message handlers may interfere with normal message composition.

Protection: User-level access to the network interface requires that protection mechanisms be enforced by the hardware. This typically includes checking the destination node, the destination process and, if applicable, the message length. For most of these checks a simple range check is sufficient. On reception, the message head (*i.e.*, the handler address and possibly a process id) can be checked using the normal memory management system.

Frequent message accelerators: A well-designed network interface allows the most frequent message types to be issued quickly. For example in the *T[16] proposal, issuing a global memory fetch takes a single store double instruction (the network interface is memory mapped). The 64-bit data value is interpreted as a global address and translated in the network interface into a node/local-address pair. For the return address the current frame pointer is cached in the network interface and the handler address is calculated from the low-order bits of the store address.

4.2 Processor support for message handlers

Asynchronous message handler initiation is the one design issue that cannot be addressed purely in the network interface: processor modifications are needed as well. The only way to signal an asynchronous event on current microprocessors is to take an interrupt. This not only flushes the pipeline, but enters the kernel. The overhead in executing a user-level handler includes a crawl-out to the handler, a trap back into the kernel, and finally the return to the interrupted computation[9]. Super-scalar designs tend to increase the cost of interrupts.

Fast polling: Frequent asynchronous events can be avoided by relying on software to poll for messages. In execution models such as TAM where the message frequency is very high, polling instructions can

[9]It may be possible for the user-level handler to return directly to the computation.

be inserted automatically by the compiler as part of thread generation. This can be supported with little or no change to the processor. For example, on Sparc or Mips a message-ready signal can be attached to the coprocessor condition code input and polled using a branch on coprocessor condition instruction.

User-level interrupts: User-level traps have been proposed to handle exceptions in dynamically typed programming languages[14] and floating-point computations. For Active Messages, user-level interrupts need only occur between instructions. However, an incoming message may not be for the currently running user process and the network interface should interrupt to the kernel in this case.

PC injection: A minimal form of multithreading can be used to switch between the main computational thread and a handler thread. The two threads share all processor resources except for the program counter (PC). Normally instructions are fetched using the computation PC. On message arrival, instruction fetch switches to use the handler PC. The handler suspends with a `swap` instruction, which switches instruction fetch back to the computation PC. In the implementation the two PCs are in fact symmetrical. Switching between the two PCs can be performed without pipeline bubbles, although fetching the `swap` instruction costs one cycle. Note, that in this approach the format of the message is partially known to the network interface, since it must extract the handler PC from the message.

Dual processors: Instead of multiplexing the processor between computation threads and handlers, the two can execute concurrently on two processors, one tailored for the computation and a very simple one for message handlers (*e.g.*, it may have no floating-point). The crucial design aspect is how communication is handled between the two processors. The communication consists of the data received from the network and written to memory, *e.g.*, into activation frames, and the scheduling queue.

A dual-processor design is proposed for the MIT *T project. It uses an MC88110 for computation and a custom message processor. In the *T design, the two processors are on separate die and communicate over a snooping bus. If the two processors were integrated on a single die, they could share the data cache and communication would be simpler. The appealing aspect of this design is that normal uniprocessors can be used quite successfully.

For coarse-grain models, such as Split-C, it is most important to overlap computation with the transmission of messages into the network. An efficient network interface allows high processor utilization on smaller data sets. On the other extreme, implicitly parallel language models that provide word-at-a-time access to globally shared objects are extremely demanding of the network interface. With modest hardware support, the cost of handling a simple message can be reduce to a handful of instructions, but not to one. Unless remote references are infrequent, the amount of resources consumed by message handling is significant. Whether dual processors or a larger number of multiplexed processors is superior depends on a variety of engineering issues, but neither involves exotic architecture. The resources invested in message handling serve to maintain the efficiency of the background computation.

5 Related work

The work presented in this paper is similar in character to the recent development of optimized RPC mechanisms in the operating system research community[19, 2]. Both attempt to reduce the communication layer functionality to the minimum required and carefully analyze and optimize the frequent case. However, the time scales and the operating system involvement are radically different in the two arenas.

The RPC mechanisms in distributed systems operate on time-scales of 100s of microseconds to milliseconds, and operating system involvement in every communication operation is taken for granted. The

optimizations presented reduce the OS overhead for moving data between user and system spaces, marshaling complex RPC parameters, context switches and enforcing security. Furthermore, connecting applications with system services is a major use of operating system RPCs, so the communication partners must be protected from one another.

In contrast, the time scale of communication in parallel machines is measured in tens of processor clock cycles (a few μs) and the elimination of all OS intervention is a central issue. Security is less of a concern given that the communication partners form a single program.

Another difference is that in the distributed systems arena the communication paradigm (RPC) is stable, whereas we propose a new mechanism for parallel processing and show how it is more primitive than and subsumes existing mechanisms.

6 Conclusions

Integrated communication and computation at low cost is the key challenge in designing the basic building block for large-scale multiprocessors. Existing message passing machines devote most of their hardware resources to processing, little to communication and none to bringing the two together. As a result, a significant fraction of the processor is lost to the layers of operating system software required to support message transmission. Message driven machines devote most of their hardware resources to message transmission, reception and scheduling. The dynamic allocation required on message arrival precludes simpler network interfaces. The message-by-message scheduling inherent in the model results in short computation run-lengths, limiting the processing power that can be utilized.

The fundamental issues in designing a balanced machine are providing the ability to overlap communication and computation and to reduce communication overhead. The active message model presented in this paper minimizes the software overhead in message passing machines and utilizes the full capability of the hardware. This model captures the essential functionality of message driven machines with simpler hardware mechanisms.

Under the active message model each node has an ongoing computational task that is punctuated by asynchronous message arrival. A message handler is specified in each message and serves to extract the message data and integrate it into the computation. The efficiency of this model is due to elimination of buffering beyond network transport requirements, the simple scheduling of non-suspensive message handlers, and arbitrary overlap of computation and communication. By drawing the distinction between message handlers and the primary computation, large grains of computation can be enabled by the arrival of multiple messages.

Active messages are sufficient to support a wide range of programming models and permit a variety of implementation tradeoffs. The best implementation strategy for a particular programming model depends on the usage patterns typical in the model such as message frequency, message size and computation grain. Further research is required to characterize these patterns in emerging parallel languages and compilation paradigms. The optimal hardware support for active messages is an open question, but it is clear that it is a matter of engeneering tradeoffs rather than architectural revolution.

Acknowledgements

We thank our good friends at MIT for the work on Id90, Monsoon and the J-Machine, which stimulated this investigation. nCUBE/2 time was gratefully provided by Sandia National Laboratories and nCUBE Corp. nCUBE also provided access to Vertex kernel sources and a mini nCUBE/2 configuration to reboot at leisure. Guidance and encouragement from hard working folks at Thinking Machines Corp. was instrumental throughout the development of Active Messages on the CM-5.

This work was supported by National Science Foundation PYI Award (CCR-9058342) with matching funds from Motorola Inc. and the TRW Foundation. Thorsten von Eicken is supported by the Semiconductor Research Corporation Grant 91-DC-008. Seth Copen Goldstein is supported by Sandia National Laboratories Contract 87-0212. Klaus Erik Schauser is supported by an IBM Graduate Fellowship. CM-5 computational resources were provided by the NSF Infrastructure Grant CDA-8722788.

References

[1] Arvind and R. A. Iannucci. Two Fundamental Issues in Multiprocessing. In *Proc. of DFVLR - Conf. 1987 on Par. Proc. in Science and Eng.*, Bonn-Bad Godesberg, W. Germany, June 1987.

[2] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight Remote Procedure Call. *ACM Trans. on Computer Systems*, 8(1), February 1990.

[3] D. Culler, A. Sah, K. Schauser, T. von Eicken, and J. Wawrzynek. Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine. In *Proc. of 4th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, Santa-Clara, CA, April 1991. (Also available as Technical Report UCB/CSD 91/591, CS Div., University of California at Berkeley).

[4] D. E. Culler and Arvind. Resource Requirements of Dataflow Programs. In *Proc. of the 15th Ann. Int. Symp. on Comp. Arch.*, pages 141–150, Hawaii, May 1988.

[5] W. Dally and et al. Architecture of a Message-Driven Processor. In *Proc. of the 14th Annual Int. Symp. on Comp. Arch.*, pages 189–196, June 1987.

[6] W. Dally and et al. The J-Machine: A Fine-Grain Concurrent Computer. In *IFIP Congress*, 1989.

[7] J. J. Dongarra. Performance of Various Computers Using Standard Linear Equations Software. Technical Report CS-89-85, Computer Science Dept., Univ. of Tennessee, Knoxville, TN 37996, December 1991.

[8] T. H. Dunigan. Performance of a Second Generation Hypercube. Technical Report ORNL/TM-10881, Oak Ridge Nat'l Lab, November 1988.

[9] G. Fox. *Programming Concurrent Processors.* Addison Wesley, 1989.

[10] R. H. Halstead, Jr. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[11] Dana S. Henry and Christopher F. Joerg. The Network Interface Chip. Technical Report CSG Memo, 331, MIT Lab for Comp. Sci., 545 Tech. Square, Cambridge, MA, June 1991.

[12] W. Horwat, A. A. Chien, and W. J. Dally. Experience with CST: Programming and Implementation. In *Proc. of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, 1989.

[13] Intel. Personal communication, 1991.

[14] D. Johnson. Trap Architectures for Lisp Systems. In *Proc. of the 1990 ACM conf. on Lisp and Functional Programming*, June 1990.

[15] R. S. Nikhil. The Parallel Programming Language Id and its Compilation for Parallel Machines. In *Proc. Workshop on Massive Parallelism, Amalfi, Italy, October 1989*. Academic Press, 1991. Also: CSG Memo 313, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA.

[16] R. S. Nikhil, G. M. Papadopoulos, and Arvind. *T: A Killer Micro for A Brave New World. Technical Report CSG Memo 325, MIT Lab for Comp. Sci., 545 Tech. Square, Cambridge, MA, January 1991.

[17] G. M. Papadopoulos. Implementation of a General Purpose Dataflow Multiprocessor. Technical Report TR432, MIT Lab for Comp. Sci., 545 Tech. Square, Cambridge, MA, September 1988. (PhD Thesis, Dept. of EECS, MIT).

[18] G. M. Papadopoulos and D. E. Culler. Monsoon: an Explicit Token-Store Architecture. In *Proc. of the 17th Annual Int. Symp. on Comp. Arch.*, Seattle, Washington, May 1990.

[19] A. Thekkath and H. M. Levy. Limits to Low-Latency RPC. Technical Report TR 91-06-01, Dept. of Computer Science and Engineering, University of Washington, Seattle WA 98195, 1991.

Building Communication Paradigms with the CM-5 Active Message layer (CMAM)

Thorsten von Eicken
David E. Culler

Computer Science Division — EECS
University of California, Berkeley
Berkeley, CA 94720
(510) 642-8299

July 3, 1992

1 Introduction

The CM-5 Active Message layer (*CMAM*) provides a simple, general purpose communication primitive as a thin veneer over the raw hardware. It is intended to serve as a substrate for building libraries that provide higher-level communication abstractions and for generating communication code from a parallel-language compiler, rather than for direct use by programmers. *CMAM* is currently in use at UC Berkeley under the Split-C and Id compilers, as well as in a variety of libraries. (Of course, some people insist on writing in *CMAM* directly. No accounting for taste.) This document describes the basic concepts embodied in *CMAM* and illustrates how various abstractions can be constructed on this substrate. For specifics on the interface, consult the manual pages (appendix A). Experience with the CM-5 is assumed.

The basic communication primitive provided by *CMAM* is an Active Message: a message with an associated small amount of computation (in the form of a handler) at the receiving end. In *CMAM*, the first word of a message points to the handler for that message. On message arrival, the computation on the node is interrupted and the handler is executed. The role of the handler is to get the message out of the network, either by integrating it into the ongoing computation or by replying, in the case of a remote request. The buffering and scheduling provided in *CMAM* are extremely primitive and thereby fast. The only buffering is that involved in actual transport and the only scheduling is that required to activate the handler. This is sufficient for many applications. More general buffering and scheduling can be easily constructed in layers above *CMAM*. This minimalist approach avoids paying a performance penalty for unneeded funtionality. The power of Active Messages comes from the ability to customize message formatters and handlers, and from the simplicity (hence efficiency) of the implementation. Currently a five word *CMAM* packet can be sent in 1.4us and dispatched to an arbitrary C function on the receiving end in 1.6us.

A few aspects of active messages and the current implementation on the CM-5 are worth noting at the outset.

- The role of message handlers is to service the network, i.e., to reply to requests or to store the data of incoming messages into memory in a reasonable way. Message handlers are not intended to perform computation as such, but are not prevented from doing some reasonable amount of work. In this sense the active message model is fundamentally different from message-driven models of computation (e.g., dataflow, concurrent smalltalk, concurrent agregates, etc.). In some cases, it is cheaper to process the data on-the-fly than to copy it to a buffer and handlers are often specialized in this way. It is generally best if all storage required to handle a message is preallocated.

- The current *CMAM* implementation polls the network in order to accept messages and execute the appropriate handlers. This is buried within the various *CMAM* operations, so it is generally invisible. However, the `CMAM_wait` primitive, or an equivalent poll loop, should be used for busy waiting. Using interrupts to receive messages is more natural to active messages, but the costs on the CM-5 are rather steep. The interval between polls increases the latency of remote requests which affects the prefetching distance or multithreading required. Failure to service the network for long periods may cause messages to back-up into the routers. It is anticipated that later versions will utilize interrupts.

- The size of message packets on the CM-5 is limited to five words. *CMAM* provides variable length active messages as the primary interface, but also exposes an interface that reflects this physical limit. The CM-5 network allows packets to be re-ordered along a point-to-point connection, so there is some overhead to forming multipacket messages. *CMAM* provides *communication segments* to facilitate bulk transfer and all-to-all personalized communication. When a certain amount of data is deposited into the segment the handler is activated.

- Deadlock avoidance is a key aspect of the design of the *CMAM* layer, but still needs to be understood when constructing communication layers on *CMAM*, especially when handlers reply to remote requests as in a shared memory model, for example. Networks are usually "deadlock-free" as long as processors absorb messages. One-way communication is straight-forward; the sender blocks if the network cannot accept packets, but continues to receive and handle incoming packets. Two-way communication is trickier since handling an incoming request involves sending a reply. Accepting requests while replies are blocked is likely to lead to stack overflow. In general, requests and replies must have different priority levels, so this distinction is explicit in the *CMAM* interface. On the CM-5 the priorities are established by the usage of the two data networks; one is reserved for requests and the other one for replies. The processor must always accept packets from the reply network, but can stop listening to the request network while servicing a request, even if the request blocks. (The implementation of active messages on other architectures achieves this deadlock avoidance through other means.) For one-way communication either network can be used and the *CMAM* interface provides calls to inject packets into either network.

The core of *CMAM* is in three parts. The message sending and reception primitives are hand-coded in assembler. Header files contains ANSI-C function prototypes and GCC inline functions to poll the network and to format simple requests. Several communication abstractions, including shared memory and traditional message passing, are provided as formatters and handlers are written in C. As the examples below show, the *CMAM* user is permitted, even

encouraged, to extend these with specialized format and handler functions which can be far more effective than the standard ones.

The current header files assume that GCC is being used. Converting the prototypes and using macros instead of inline functions is, in principle, possible. Note that there is no special compiler for the CM-5, any Sun-4 compiler can be used. Only the linker is special and produces two executables, one for the CN (control node, scalar, or front-end) and one for the PNs (processing nodes).

In general, a communication operation built on top of *CMAM* includes a formatter and a handler. Usually, the formatter is trivial and simply uses *CMAM* to call the handler function on a remote processor. For two-way communications, the request handler will use `CMAM_reply` to call a function on the requesting processor. This is illustrated by example in the following sections.

2 Hello world

This section describes a trivial program which pings all processors using *CMAM* and prints a hello string. A minimal *CMAM* program consists of three files: the program running each of the nodes, the program running on the host and a C-prototype file to link the two together. This is identical to programs written using the CMMD message passing library and is described in further detail in the "CMMD User's Guide".

2.1 Host program

Figure 1 shows the host code for our first program. When execution begins `main` is called on the host (as in any standard C program) while all processing nodes (PNs) enter a dispatch loop. `Main` first enables *CMAM* and the *CMAM* synchronization barrier and then calls `do_pings` on the PNs. `Do_pings` really is a stub generated from the prototype file in Figure 2 and causes the dispatcher on each PN to call `CMPE_do_pings` of the node program. The argument to `do_pings` is the handler to be called by PN 0 to return the total execution time. The call to `do_pings` is asynchronous, i.e. the host program continues to execute without waiting for `do_pings` to complete. After starting `do_pings`, the host program calls `CMAM_wait` to wait for the `done` flag. `CMAM_wait(&flag, value)` busy-waits for the `flag` to reach `value`. While busy-waiting, it polls the network (otherwise nothing would ever happen) and before returning, it resets `flag` by subtracting `value`. After printing a "Hello World" with the timing information, `main` disables *CMAM* and terminates. `CMAM_disable` waits for all PNs to return to the idle loop before disabling *CMAM*. The call to `exit` not only terminates the host programs but also kills all PN programs.

Observe that *CMAM* operations work between the host and the nodes as they do between nodes. However, there are some underlying differences. The nodes all have identical code address spaces. Thus, the local address of a handler is also its remote address. This uniformity does not hold for the host, since it has a completely different address space. This is why the host handler address is passed as an argument to the node code. Secondly, the performance of node-to-host operations is much worse than other operations because the node operating system is involved.

`CMAM_barrier` is enabled separately from the rest of *CMAM* because it changes the host participation in global synchronization operations vis a vis CMMD. Under CMMD the host participates, under *CMAM* it does not. Mixing CMMD and *CMAM* is somewhat subtle and

```
#include <stdio.h>
#include <cmam/cmam.h>

/* receive result from PNs */
static volatile int done = 0;                          /* flag set when PN part finished */
static volatile double time = 0;                       /* time taken by PN part */
static void set_time(double t) { time = t; done++; }   /* handler for PN 0 to call */

/* main program */
void main(int argc, char **argv)
{
        CMAM_enable();                                 /* enable CMAM */
        CMAM_enable_barrier();                         /* enable barrier separately */

        do_pings(set_time);                            /* run PN code */
        CMAM_wait(&done, 1);                           /* wait for PN code to complete */

        fprintf(stderr, "Hello world from %d nodes.  Pings took %.1fus each.\n",
                CMAM_partition_size, (time*1E6)/CMAM_partition_size);

        CMAM_disable();                                /* sync all PNs and disable CMAM */
        exit(0);
}
```

Figure 1: *Host program for the "Hello World" example.* File `ping_host.c`.

discussed further in Section 6.

```
void do_pings(void (*set_time)());
```

Figure 2: *Stub prototypes for the "Hello World" example.* File `ping.proto`.

A prototype file, shown in Figure 2, connects the host call to the node routine, shown in Figure 3. Note, however, that the routine called on the node must have a `CMPE_` prefix, *i.e.*, calling `do_pings` on the host goes through a stub and invokes `CMPE_do_pings` on all the nodes. The node program shows the three parts of a two-way communication operation: `ping`, `pong`, and `poof`. Ping uses the machine specific `CMAM_4` call to invoke `pong` on `dest_node` passing it the requesting node number and the address of a flag on the requesting node. Then `ping` busy waits on this flag. The *CMAM* call invokes the function specified by its second argument on the node specified by its first argument with the remaining arguments as input. The CMNA timer registers are used in the example to time this loop. These timers provide elasped time in processor cycles (33MHz) including timesharing gaps.

The `pong` function is invoked by *CMAM* on the remote node with the return node and flag address as arguments. It simply invokes `poof` on the requesting processor using `CMAM_reply`. The reply version must be used within request handlers. Response handlers are not permitted to send additional messages. `poof` simply increments the flag specified in its argument. This is the flag that ping is busy waiting on, so ping will return from the `CMAM_wait` call after poof is invoked. The top level node routine exercises the ping-pong-poof from node zero to each of the other nodes. It then delivers the aggregate time to the host. Meanwhile, all the other nodes

are waiting in the **CMAM_barrier**. Nodes continue to services requests while in the barrier. After all nodes have execute the **CMAM_barrier** call, each is allowed to proceed. In this case, they all complete and return to the dispatch loop, whereupon the program terminates.

```
#include <cmam/cmam.h>
#include <cm/timers.h>                                  /* access to hardware timer */
static double time(int t)                                 /* turn tick into seconds */
{ CM_TIME ct = {0, t}; return CM_ni_time_in_sec(ct); }

void CMPE_do_pings(void (*set_time)())
{
        int me, dest;
        double time;
        static double ping(int dest_node);

        me = CMNA_self_address;

        if(me == 0) {
                time = 0.0;
                for(dest=CMNA_partition_size-1; dest≥0; dest--) {
                        time += ping(dest);
                }
                CMAM_host_4(set_time, time);
        }

        CMAM_barrier(0);
}

static void poof(volatile int *flag) { (*flag)++; }

static void pong(int return_node, volatile int *return_flag)
{ CMAM_reply_4(return_node, poof, return_flag); }

static double ping(int dest_node)
{
        int t;
        volatile int done = 0;

        t = *(volatile int *)time_reg;
        CMAM_4(dest_node, pong, CMNA_self_address, &done);
        CMAM_wait(&done, 1);
        t = *(volatile int *)time_reg - t;
        return time(t);
}
```

Figure 3: *Node program for the "Hello World" example.* File **ping.c**.

The **Makefile** for the "Hello World" program is shown in Figure 4. The first two lines define the location of the *CMAM* libraries and *include* files. This may vary depending on the installation. *CMAM* requires the use of the GNU C compiler, preferably version 2.0 or later. If you use GCC 2.xx, you must load the gcc library with both the host executable and the node executable, and the C compiler flags must include `-gstabs` to allow `pndbx` (the CM-5 node debugger) to read the debugging information generated by GCC. The additional `-Dpe_obj` and `-DPE_CODE` flags are required for compiling the node program. The last rule in the makefile converts the prototype in `ping.proto` into a stub to be loaded with the host program.

```
CMAM_LIB=/usr/cm5/local/lib
CMAM_INC=/usr/cm5/local/include/cmam

CC=gcc-2
CFLAGS=-gstabs -O2 -I/usr/cm5/include -I$(CMAM_INC) -DCM5
PNFLAGS=$(CFLAGS) -Dpe_obj -DPE_CODE
GCC2LIB=/usr/local/lib/gcc-lib/libgcc.a

CP_LIBS=-L$(CMAM_LIB) -lcmam -lcmna_sp -lm $(GCC2LIB)
PN_LIBS=-L$(CMAM_LIB) -lcmam_pe -lcmna_pe -lm $(GCC2LIB)

PGM_CP=ping_host.o ping_proto.o
PGM_PN=ping.o

ping: $(PGM_CP) $(PGM_PN)
        cmld -o ping $(PGM_CP) $(CP_LIBS) -pe $(PGM_PN) $(PN_LIBS)

.c.o:
        $(CC) $(PNFLAGS) -c $*.c

ping_host.o: ping_host.c
        $(CC) $(CFLAGS) -c $*.c
ping.o: ping.c

ping_proto.o: ping.proto
        sp-pe-stubs <ping.proto >ping_proto.c
        $(CC) $(CFLAGS) -c ping_proto.c
```

Figure 4: *Makefile for the "Hello World" example.* File **`Makefile`**.

3 Shared Memory

This section shows how shared memory primitives can be built on *CMAM*This is essentially like the ping-pong example in the previous section. The implementation of global read and write function is shown in Figure 5. A counter `sm_flag` is used to keep track of the number of outstanding shared memory operations. For a simple shared memory model this will be one or zero. The `release` routine waits until all previous shared memory operations have completed, *i.e.*, until the counter reaches zero. Observe that it calls `CMAM_poll` to ensure that incoming messages are received and handled.

The read operation has three parts. `read_i` waits for previous requests to complete and then sends an Active Messageto read the desired location on the remote processor. The counter is incremented to reflect that a request is outstanding. The handler reads the location and replies to the originating processor. The response handler deposits data in a buffer and decrements the counter, so the original `read_i` observes the completion and can return the data.

The write operation similarily has three parts in order to provide sequential consistency. The request contains the address and the data. The handler deposits the data in the desired location on the remote node and replies with an acknowedgement. The `write_i` does not wait for the acknowledgement, but it does wait to ensure that all previous acknowledgements have been received before initating the write. Thus, there can be at most one outstanding write and the writes from each processor occur in the order that they are issued.

We have elected to wrap a function call around the increment of the outstanding counter to highlight the issue of atomicity between the main computation and the handlers. Handlers execute atomically with respect to other handlers, but may interupt computation. Thus, updates to shared variables at the computation level should be protected relative to handlers. Since the current version of *CMAM* uses polling, this concern does not arise. However, it is anticipated that interupts will be enabled in the future.

Figure 6 shows a weaker set of shared memory primitives based on the release consistency model. `write_iw` initates the write without waiting for previous writes to complete. `release` can be used directly to ensure that all previous writes have completed, or `write_i` will implicitly perform a release.

The two phases of `read_i` have been split apart in `prefetch_i` and `accept_i` so that global reads can be overlapped with computation. Notice that the previous shared memory primitives can be mixed freely with the weaker forms, except that no reads can occur between a prefetch and an accept. This restriction could be eliminated by providing separate buffers or, better yet, providing the buffer address as an argument to the prefetch. This idea is carried forward more thoroughly in the distributed memory model discussed in the next section.

```
/* Implementation of shared memory operations on CMAM */

volatile int sm_flag = 0;
volatile int read_buf;
inline void release() {while (sm_flag ≠ 0) CMAM_poll();}
inline void outstanding() {sm_flag++; };

/* Read INT from a global address: <proc,addr> */
void read_i_resp_handler(int data)
{
        read_buf = data;
        sm_flag--;
}

void read_i_handler(int ret_node, int *local_addr)
{
        CMAM_reply_4(ret_node, read_i_resp_handler, *local_addr);
}

inline int read_i(int node, int *addr)
{
        release();
        outstanding();
        CMAM_4(node, read_i_handler, CMAM_self_address, addr);
        release();
        return(int_buf);
}

/* Write INT to a global address: <proc,addr> */

void write_ack_handler() { sm_flag--; }

void write_i_handler(int ret_node, int *local_addr, int data1)
{
        *local_addr = data1;
        CMAM_reply_4(ret_node, write_ack_handler);
}

inline void write_i(int node, int *addr, int data)
{
        release();
        sm_flag++;
        CMAM_4(node, write_i_handler, CMAM_self_address, addr, data);
}
```

Figure 5: *Shared Memory Implmentation.* File `shared-mem.c`.

```
/* Weaker forms of write and read overlap of communication and computation */

inline void write_iw(int node, int *addr, int data)
{
        sm_flag++;
        CMAM_4(node, write_i_handler, CMAM_self_address, addr, data);
}

inline int prefetch_i(int node, int *addr)
{
        release();
        sm_flag++;
        CMAM_4(node, read_i_handler, CMAM_self_address, addr);
}

inline int accept_i(int node, int *addr)
{
        release();
        return(int_buf);
}
```

Figure 6: *Weak shared memory operations.* File `weak-mem.c`.

4 Distributed Memory

This section describes a novel communication paradigm that falls somewhere between shared memory and message passing. It retains the global address space of shared memory, but exposes the split-phase nature of accessing data across a network. In addition, bulk transfer operations are provided. The idea basically is to allow the program to initate tranfers and test explicitly for their completion. Get generalizes read and put generalizes write. For integers, these are implemented as in Figure 7. This is part of the shared memory utility provided with the *CMAM* release as `CMAM_shmem.h`.

Note that the put handler is the same as the get response handler. The blocked version of get illustrates the use of communication segments to support bulk communication.

Figure 8 shows how the put operation can be used in supporting all-to-all personalized communication. The program is a generalization of a matrix transpose operation and forms of a key step in many fast parallel algorithms. The idea is that the layout of an array is specified by a mapping function from indexes to < processor, offset > pairs and the inverse. An array is provided under one layout and a new array is required under a different layout. In this case, mapping function A specifies a cyclic mapping and mapping function B specifies a blocked mapping. Each processor simply puts each datum where it goes. It is complete when all the data it requires has been put to it by others. There is no need to retaining consistency of write ordering within the xpose, no need for acknowledgments, and no barrier.

An alternative way of writing this routine would be to open a specific communication segment on all processors using `CMAM_open_this_segment`. The puts are then replaced by `CMAM_xfer` operations which can pack more data in a network packet, since the offset is relative to the base of the communication segment and no handler address field is needed. By organizing the remap to operate on quad words the full network bandwidth can be utilized.

```
/* Get data from remote processor and increment local flag */
inline void CMAM_get_i(int node, int *remote_addr, int *local_addr, volatile int *flag)
{
        extern void CMAM_get_i_handler();
        CMAM_4(node, CMAM_get_i_handler,
                CMNA_self_address, remote_addr, local_addr, flag);
}

/* Get block of data from remote processor and increment local flag */
inline void CMAM_get_N(int node, void *addr, int count, void *local_addr,volatile int *flag)
{
        extern void CMAM_get_N_handler();
        extern int CMAM_get_N_end();
        CMAM_4(node, CMAM_get_N_handler, CMNA_self_address, addr, count,
                CMAM_open_segment(local_addr, count, CMAM_get_N_end, (void *)flag));
}

/* Put data to remote processor and increment flag there */
inline void CMAM_put_i(int node, int *addr, int data, volatile int *flag)
{
        extern void CMAM_put_i_handler();
        CMAM_4(node, CMAM_put_i_handler, addr, data, flag);
}

void CMAM_put_i_handler(int *addr, int data, int *flag)
{
        addr[0] = data; (*flag)++;
}

void CMAM_get_i_handler(int ret_node, int *addr, int *ret_addr, int *flag)
{
        CMAM_reply_4(ret_node, CMAM_put_i_handler, ret_addr, *addr, flag);
}

void CMAM_get_N_handler(int node, void *addr, int count, int seg_addr)
{
        CMAM_reply_xfer(node, seg_addr, addr, count);
}
int CMAM_get_N_end(volatile int *flag, void *local_addr)
{
        (*flag)++; return 0;
}
```

Figure 7: *Distributed Memory Implmentation.* File `dist-mem.c`.

```
/* xpose.c - Generalized Transpose
 *
 * The layout of an array is specified by a pair of decode functions which
 * map the logical index (i) to a processor (p) and a processor local
 * offset (o).  The reverse mapping from (p,o) to i is given by an encode
 * function.
 *
 * Given two arrays specifed by arbitrary layout functions, move one array
 * into the other.
 *
 * Algorithm: scan through local elements, starting from a random offset,
 * and put them where they belong. Stop when have received a full set of puts.
 */

#include <cmam/cmam.h>

#define SIZE (1024)                          /* hard coded array size for now */

/* A Layout: wrap indexes around the processors. */
static double A[SIZE];
#define decode_a_p(i) ((int) (i % num_procs))
#define decode_a_o(i) ((int) (i / num_procs))
#define encode_a(p,o) ((int) (o * num_procs + p))

/* B Layout: block indexes onto processors. */
static double B[SIZE];
#define decode_b_p(i) ((int) (i / SIZE))
#define decode_b_o(i) ((int) (i % SIZE))
#define encode_b(p,o) ((int) (p * SIZE + o))

void CMPE_xpose(void)
{
        int me, num_procs, a, a0, r;
        static int count = 0;

        me = CMNA_self_address;
        num_procs = CMNA_partition_size;
        srandom(me);

        a = a0 = random()%SIZE;
        do {
                int vi = encode_a(me,a);
                CMAM_put_d(decode_b_p(vi), B+decode_b_o(vi), A[a], &count);
                a++; if(a == SIZE) a = 0;
        } while(a ≠ a0);
        CMAM_wait(&count, SIZE);
}
```

Figure 8: *Node program for the generalized array transpose example.*

5 Send and Receive

For those wedded to send&receive, here's a first-cut implementation: it only supports blocking send&receive (i.e. SEND blocks until the corresponding RECEIVE is executed, and vice-versa) and it only supports receiving from a specific node (i.e. cannot RECEIVE from "any node").

This implementation uses a standard three-way handshake: the sender sends a request to the receiver, which replies by acknowledgement when ready, and finally the sender transfers the data. The protocol state is maintained by the receiver in a per-processor table. `CMAM_send` sends a request Active Message to the receiver and, depending on whether the corresponding `CMAM_receive` has been executed, the request handler either enters the request into the state table or directly replies with the acknowledgement. In the meantime, `CMAM_send` waits for the acknowledgement before sending the data itself. Similarly, `CMAM_receive` checks the state table and, if the corresponding request has been received, sends the acknowledgement.

```
/* State of pending sends and receives */
#define MAX_PROC 1024                          /* max num of processors */
static int state[MAX_PROC];                                  /* state */
#define IDLE -1                                      /* processor idle */
#define SEND_PEND(state) ((state) ≥ 0)     /* he's waiting to send <state> bytes */
#define RECV_PEND -2                        /* we're waiting for him to send */

/* State of current SEND */
static volatile int send_seg = -1;                   /* segment to send to */
static volatile int send_count = 0;                /* agreed message length */

/* State of current RECEIVE */
static volatile int recv_flag = 0;                   /* receive-complete flag */
static int recv_done(void *info, void *base) { recv_flag++; return 0; }
static int recv_from = 0;                                 /* source processor */
static int recv_count = 0;                                        /* length */
static int recv_seg = 0;                                         /* segment */

/* CMAM_send - SEND to remote node */
void CMAM_send(int node, void *buff, int byte_count)
{
        /* send request to remote node - wait for ack - send data */
        send_seg = -1;                                       /* no segment yet */
        CMAM_4(node, send_handler, CMAM_self_address, count);      /* send REQ */
        while(send_seg == -1) CMAM_poll();                  /* wait to get ACK */
        CMAM_xfer(node, send_seg, buff, send_count);              /* send data */
}
```

Figure 9: *Implementation of blocking send&receive (part 1).*

```
/* Handle a send request */
static void send_handler(int requesting_node, int send_count)
{
        if(RECV_PEND(state[requesting_node])) {                    /* is receiver ready? */
                recv_from = requesting_node;
                /* message length is MIN(send_count, recv_count) */
                if(send_count < recv_count) {
                        CMAM_shorten_segment(recv_seg, recv_count-send_count);
                        recv_count = send_count;
                }
                /* send ACK */
                CMAM_reply_4(requesting_node, send_set_seg, recv_seg, recv_count);
                state[requesting_node] = IDLE;                              /* ready for next */
        } else {
                state[requesting_node] = send_count;        /* .. not ready, record request */
        }
}

/* Handle send acknowledgement */
static void send_set_seg(int seg, int count)
{
        send_count = count; send_seg = seg;
}

/* CMAM_recv - RECEIVE from remote node */
void CMAM_recv(int node, void *buff, int count)
{
        /* allocate a segment */
        recv_count = count;
        recv_seg = CMAM_open_segment(buff, count, recv_done, 0);
        if(recv_seg == -1) CMPN_panic("no segment left");
        if(SEND_PEND(state[node])) {
                /* sender is already there */
                recv_from = node;
                /* message length is MIN(send_count, recv_count) */
                if(state[node] < count) {
                        CMAM_shorten_segment(recv_seg, count-state[node]);
                        recv_count = state[node];
                }
                /* send ACK */
                CMAM_reply_4(node, send_set_seg, recv_seg, recv_count);
                state[node] = IDLE;                                         /* ready for next */
        } else {
                state[node] = RECV_PEND;
        }
        /* wait for completion of receive */
        CMAM_wait(&recv_flag, 1);
}
```

Figure 10: *Implementation of blocking send&receive (part 2).*

6 Mixing CMAM and CMMD

While not recommended, it is possible to mix both *CMAM* and CMMD in the same program with some care:

- CMAM and CMMD node-node communication cannot occur a the same time: CMAM cannot deal with the arrival of a CMMD packet and vice-versa. It is possible to use either one in well separated phases of the program though.
- The CMMD operations using the CM-5 control network are compatible with CMAM except for `CMAM_barrier`. Enabling the barrier changes the "host participation" in control network operations and interferes with CMMDs use of the control network.

A Manual Pages