4 Active Messages Communication Architectures

The communication architecture plays a key role in the design of a parallel machine in that it is part of the interface between the machine designer and its users, in particular the language system developers. As such, the communication architecture must take many influences into account, from concrete constraints imposed by hardware technology to the more abstract requirements of parallel language systems. The central concern in Active Message is to address each of the four key communications issues (data transfer, send failure, synchronization, and network virtualization) at the right level of the system. The Active Messages approach is characterized by a particular choice of which aspects of communication are exposed to the language system and which are hidden within the architecture itself. Active Messages arrives at more efficient and versatile communication architectures that traditional approaches because communication and compiler optimizations are coupled more tightly than before.

It is tempting to solve the key communications issues "once and for all" within the communication architecture and to expose a conceptually simple interface to the language system. As argued in Chapter 5, this is in essence what traditional communication architectures attempt, and, unfortunately, it compromises the efficiency of communication because it requires either the complex general case to be solved at a low level or the programming models to be restricted. Active Messages takes a different approach and explores how the communication architecture can expose the key issues to the compiler such that a combination of low-level primitives (potentially accelerated in hardware) and high-level optimizations can be used.

Exposing more of the communication micro-architecture to the language system requires a change of mind-set in the definition of the communication architecture (in particular with respect to message passing). Active Messages is a framework which defines a *class of communication architectures* joined by a common central mechanism and a common set of trade-offs; Active Messages is specifically not a single standard communication architecture which remains invariant across all platforms, because such a standard cannot offer the desired versatility and efficiency.

The concept of Active Messages as a class of communication architectures is analogous to that of load/store architectures forming a class of instruction set architectures. Load/store architecture are predicated on a layering model very similar to that of Active Messages. In particular, both assume the use of high-level languages which are mapped to the architecture using a compiler and run-time substrate. When porting a HLL from one load/store architecture to another it is assumed that the code generator and run-time substrate require modification, although the essential algorithms, such as register allocation, remain the same. With Active Messages the notion of portability is similar: the algorithms to implement higher-level communication operations, such as fetch&add or a remote task enqueue) are portable while their implementation must be adapted to the peculiarities of the new platform. The le-

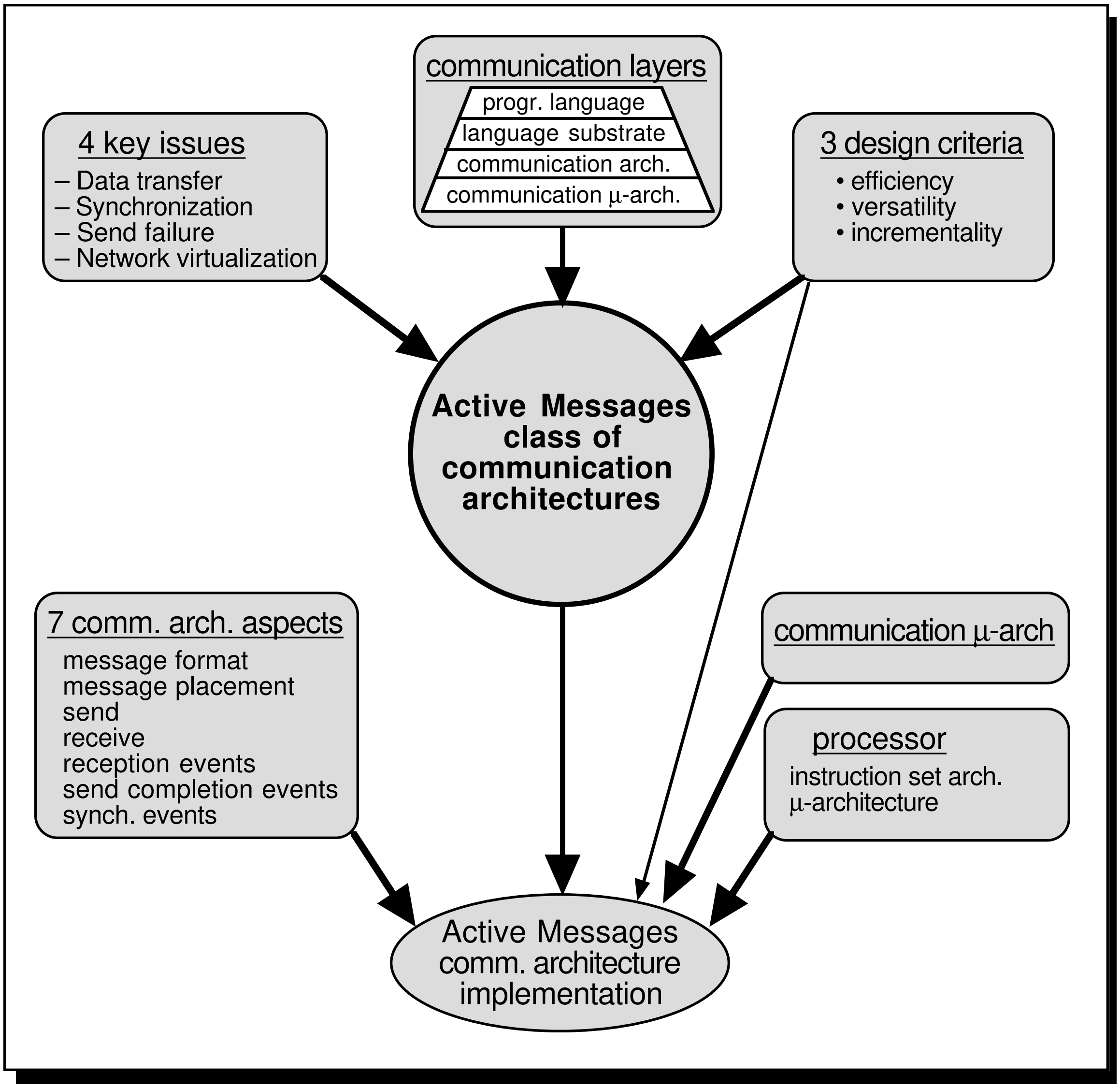


Figure 4-1: Influences on Active Messages communication architectures.

verage offered by Active Messages is a consistent view of communication from higher software layers that is portable and efficient.

The relationship among the various components and concepts influencing Active Messages is depicted in Figure 4-1. The three design criteria and the communication layering model represent the abstract constraints on Active Messages as a class of communication architectures while the four key issues represent the concrete issues presented by communication micro-architectures in general. A particular instance of Active Messages must take the details of the communication and processor micro-architectures into account and is defined in terms of seven aspects: message format, message placement, send and receive operations, message reception and send completion events, and synchronization events.

The first section of this chapter defines the Active Messages class of communication architectures. Sections 4.2 and 4.3 describe two implementations of Active Messages on the nCUBE/2 and the CM-5, respectively. The two implementations take different positions on several communication architecture aspects to achieve high efficiency and versatility. Besides describing the implementation, the two sections examine the performance of several micro-benchmarks closely to demonstrate the efficiency of the mapping onto the micro-architectures and to suggest further incremental improvements to support Active Messages better.

4.1 Active Messages

The key to both the versatility and efficiency of Active Messages lies in the ability to associate a small amount of computation in the form of a handler with the reception of each message. The handler is named in the message, typically by a pointer in the first word of the message, and is executed in the user process context immediately on message arrival. The underlying model is that at all times a computation is "going on" on each node and that this computation is interrupted by message arrival. The role of the message handler, illustrated in Figure 4-2, is then to transfer data into the computation's data structures and interact with the scheduling of computation or, alternatively, it may provide a remote service and send a reply message back.

4.1.1 Addressing the four key issues

Active Messages takes a strong position on the four key issues formulated in Section 3.3 (data transfer, synchronization, send failure, and network virtualization) with the goal of maximizing communication versatility and efficiency. This position manifests itself on the one hand through a set of micro-architectural characteristics that Active Messages generally exposes to the handlers (and the run-time substrate in general) to enable high-level optimizations. On the other hand, Active Messages imposes a set of restrictions on handlers in order to allow efficient implementation.

The general strategy employed to maximize efficiency is to take a compiled code approach in which message handling code is *directly executed*. This is in contrast to most systems in which messages are generally *interpreted* within the message layer[1]. While it is quite obvious that this direct execution approach provides the ultimate in versatility and efficiency during handler execution, care must be taken to keep the dispatch of handlers efficient as well. In particular, if handlers were allowed to perform arbi-

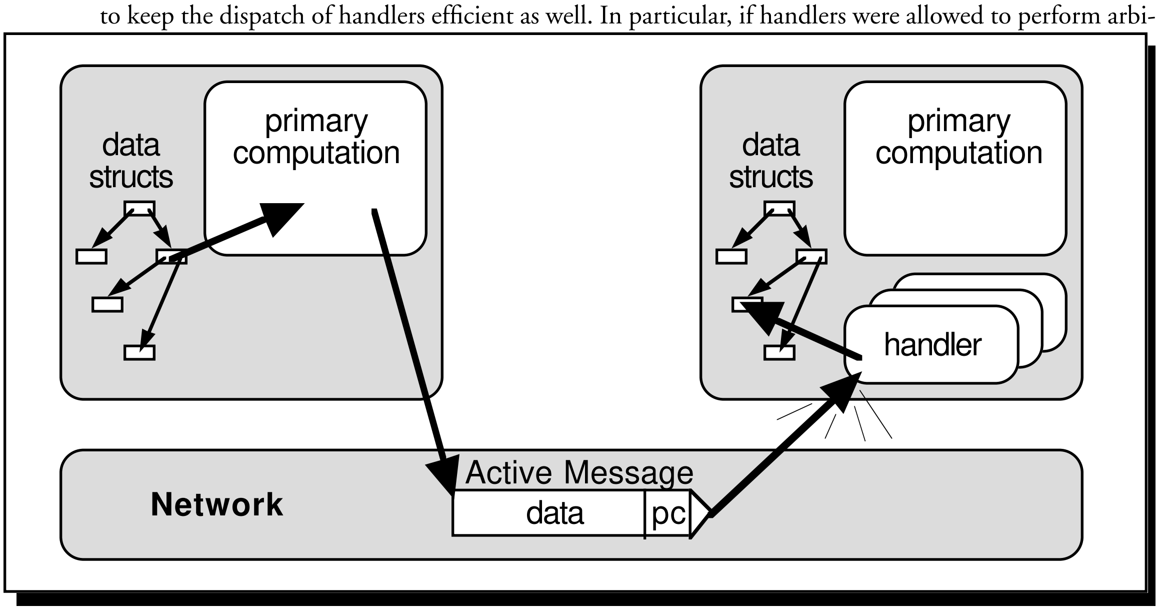


Figure 4-2: Active Messages model

The core idea in Active Messages is to use custom user-level handlers to dispose of messages quickly and optimally Each message carries at its head the address of the user-level handler which is executed on message arrival. The role of the handler is to get the message out of the network and transfer the data directly into the application data structures. In addition, the handler typically interacts with the scheduling on the node to enable the computation consuming the message data. Handlers may alternatively provide a small remote service and send a reply message back

[1] Active Messages, in some sense, interprets the head of the message as pointing to a handler and allows the handling of the rest of the message to be directly executed.

trary computation the handler dispatch would be equivalent to a full context switch to a new thread of control. To avoid this cost, a crucial aspect of Active Messages is to restrict handlers by limiting the resources available, requiring handlers to execute quickly and to completion, and restricting communication patterns to one-way and request-reply communication to prevent network deadlock (i.e., handlers receiving a reply message are not allowed to send messages themselves).

The remainder of this subsection discusses in detail how Active Messages addresses the four key issues, explaining the advocated general strategy as well as some typical implementation-specific choices. A few related additional restrictions that are particular to Active Messages are also discussed.

Data transfer

Active Messages promotes direct access to the network interface from the run-time substrate whenever this is compatible with virtualizing the network. A typical Active Messages implementation exposes the micro-architecture and may even let the compiler manage data held in the network interface itself. This is in contrast with most message layers which prescribe a standard data transfer mechanism. Current micro-architectures differ considerably in their network interface access facilities (user-level vs. kernel-only access, memory mapped FIFOs vs. registers vs. DMA) such that an implementation-specific approach is more appealing. The rationale is that the run-time substrate can transfer the data in and out of the network interface just as fast as the message layer can, but that it can often move the data directly into the application data structures without intermediate buffering.

Active Messages strives to eliminate the buffering common in other message layers. While buffering is useful to absorb rate variations, this particular function is best done within the micro-architecture where efficient VLSI structures can be used. Buffering in software does not yield good performance and can easily lead to unbounded storage resources hidden within the message layer. In cases where the run-time substrate requires queueing of messages, the message handlers can allocate and manage the required buffers more efficiently than the message layer. They possess more information on the purpose of the queueing and can use a customized allocator. In addition, the allocation automatically occurs within the application's address space.

Synchronization

The Active Message handlers provide a very flexible means of synchronizing communication and computation in that a handler encodes an arbitrary interaction between the message and the ongoing computation. Beyond this versatility, the custom Active Message handlers also offer high efficiency. The synchronization mechanism can be tailored as part of the implementation of a parallel execution model and the compiler can special-case each handler to include only the synchronization operations required for the particular type of message received. These possibilities are used extensively in the compilation of TAM, discussed in Section 6.2.

Support for efficient atomic updates to scheduling data structures are key to enabling a close interaction between handlers and the computation. Active Messages ensures that updates within handlers are atomic by keeping handler execution atomic relative to the computation and relative to other handlers, e.g., if additional messages arrive during the execution of a handler their respective handlers are not started until the current one terminates. This implies that there is no notion of a handler "suspending" and "resuming" later. Scheduling events in the computation are made atomic relative to handlers either by using appropriate atomic instructions (if provided in the processor's instruction set) or by efficiently disabling handler execution during such updates. The details are implementation specific, but every Active Messages implementation must support such atomic primitives.

Send failure

When attempting to send a message, the Active Messages implementation must be prepared to dispatch handlers for incoming messages to avoid deadlock. While this does not pose a problem for sends within the computation, it does imply that handlers attempting to send a reply message may have to be interrupted to allow the execution of nested handlers. This is the only exception to the atomicity of handler execution: an attempt to reply breaks the atomicity. Typically this problem is dealt with by moving the reply to the end of the handler such that all critical operations occur before the reply. The alternate approach of buffering the received messages and executing the handlers later may require an unbounded amount of memory.

In order to prevent uncontrolled handler nesting, Active Messages limits communication patterns to requests and replies and distinguishes the two types of messages. Messages sent from the computation are *requests* and handlers receiving them may send *replies* back. Handlers receiving replies, however, may not send any messages. If a nested handler were allowed to reply, an arbitrary number of handlers could have to be nested. This situation can lead to a stack overflow or, from a different point of view, to a livelock of the lower handlers.

The solution adopted in Active Messages of limiting communication patterns relies on having (at least) two levels of priority in the network hardware, either using virtual channels to implement different message priorities or simply using two disjoint networks.This allows the deadlock/livelock problems to be solved by using the lower priority for request messages and the higher priority for replies and by dictating that a request handler can only send a reply message and a reply handler cannot send any message. This restriction ensures that:

- the communication graph is acyclic (computation $\rightarrow$ request handler $\rightarrow$ reply handler),
- when a request handler sends a reply it is required to accept incoming reply (i.e., high-priority) messages, but can ignore request (i.e., low-priority) messages, and
- a sequence of reply handlers can nest within a request handler one after another but request handlers are never nested within other handlers.

While Active Messages requires at least two priority levels to ensure deadlock/livelock free request-reply communication patterns, it is perfectly possible to provide additional priority levels to allow more flexibility.

Network virtualization

Active messages lead to a relatively simple protection model: the destination handler address can be viewed as a global address which must be checked at the originating node for a valid destination and which is used at the receiving node to dispatch the correct handler within the appropriate address space. Typically this global address consists of a node address, a process id, and the handler start address.

Active Messages is targeted at closely coupled parallel systems and therefore assumes that the scheduling of communicating processors is coupled. In such systems, the communication latencies are several orders of magnitude smaller than the scheduling quantum and coordinating context switches among the processors ensures that the computation corresponding to arriving messages is currently running. On platforms where the micro-architecture can compare the process id of the message with that of the currently running process the scheduling coordination can be relaxed such that message arrival for a suspended process may occur. It is assumed that this remains infrequent enough such that it can be dealtwith using a bounded amount of queueing in the message layer.

Processor and storage resource restrictions

Given that handlers are executed atomically one after another it is important to keep their execution short enough to keep up with the network. For this reason the maximum execution time of a handler should be less than the reception time of its message. Note, however, that this is not an absolute requirement: in certain situations a handler may have to take special actions which take more time. This is acceptable as long as it remains an infrequent event and does not represent a communication bottleneck.

An important issue is what storage resources should be made available to handlers. In particular, which processor registers can be used in handlers. The general Active Messages approach is to keep handler dispatch simple and to avoid unnecessary saving of processor state. Thus, handlers typically have access

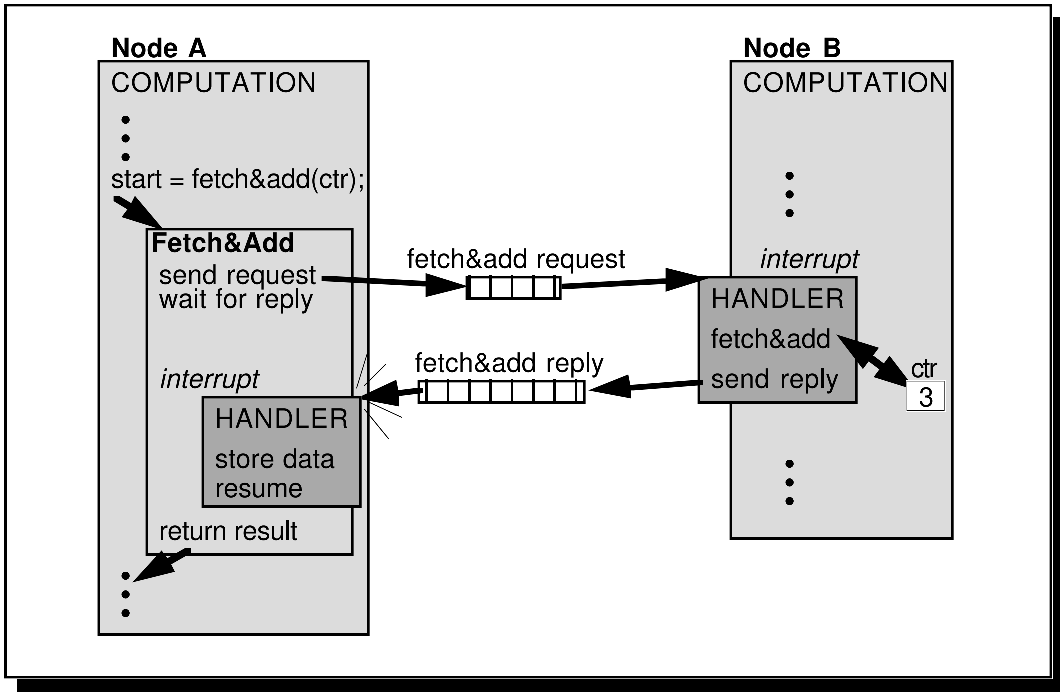


Figure 4-3: Components of a fetch&add implementation.

The computation on the initiating node formats a request Active Message and sends it to the fetch&add handler on the remote node. The request message consists of the remote address, the addend, and the information necessary to send the reply message. The execution of the handler interrupts the ongoing (and in this case typically unrelated) computation, performs the fetch&add, and sends a reply message with the result value back to the fetch&add reply handler. That handler saves the result value and signals the completion of the operation which causes the computation to pick-up the value and continue.

only to a few registers which are saved and restored as part of the dispatch. Handlers requiring additional resources may save and restore registers explicitly or, alternatively, the run-time substrate may permanently reserve a set of registers for handlers by not allocating these registers in the compilation process.

Regarding storage in memory, handlers can, in principle, dynamically allocate memory, but it is generally unwise to do so. Not only does this require the memory allocator to be made atomic relative to handlers, but also, if the memory allocation fails it is difficult to take corrective action within the handler. In particular, saving the message data is impossible (memory being full) such that the only remaining recourse is to send the message back with an error. The preferred approach is to preallocate the storage required by the handler and to pass a pointer to it in the message. The designs of Split-C and of TAM (described in Section 6.1 and Section 6.2, respectively) demonstrate how this preallocation can occur with very little overhead in the context of appropriately designed run-time substrates.

4.1.2 Active Messages example: Fetch&add

In this subsection, a simple fetch&add example shows the typical use of Active Messages in implementing simple higher-level communication operations.

The components involved in an implementation of a remote fetch&add primitive are depicted in Figure 4-3. The computation on the initiating node formats a request message and sends it to the fetch&add handler on the remote node. The request message consists of the remote address, the addend, and the information necessary to send the reply message. The execution of the handler interrupts the ongoing (and in this case typically unrelated) computation, performs the fetch&add, and sends a reply message with the result value back to the fetch&add reply handler. That handler saves the result value and signals the completion of the operation which causes the computation to pick-up the value and continue.

In this simple example the computation on the source node busy-waits until the fetch&add completes. A more advanced implementation could suspend the current thread and switch to some unrelated computation while the fetch&add is in progress. In this case the reply handler would reenable the suspended computation by adding it to an appropriate scheduling queue.

An implementation of this simple fetch&add example would consist of the request and reply handlers, and of the code sending the request message. The latter can be a library function or it could be directly generated by the compiler. Concrete implementations on the nCUBE/2 and the CM-5 are presented in § 4.2.3.3 and § 4.3.3.3, respectively.

4.2 Active Messages Architecture on the nCUBE/2

The send&receive architecture implemented in nCUBE/s Vertex system takes about 160µs end-to-end to transmit an 8-byte message among neighbor nodes. Yet, the message injection time and the routing accounts for only 14µs of these. Assuming an execution rate of about 5 MIPS the remaining 146µs represent over 700 instructions of overhead which, contrasted to the hardware description in Section 3.1, seems outrageously high: sending a message should involve entering the kernel, setting up the outgoing DMA and returning to the user, and receiving should involve a similar path through the kernel with possibly an additional interrupt routine. All in all, it seems that at most a few dozen instructions might be involved in transmitting a message!

The goal of the Active Messages implementation is develop a message layer with the minimal possible communication overhead. The desire to integrate this implementation into the existing nCUBE/2 software environment presents a number of constraints summarized in Subsection 4.2.1. The highlights of the nCUBE/2 implementation, in particular its handling of the four key issues, is presented in Subsection 4.2.2. Subsection 4.2.3 describes the nCUBE/2 Active Messages communication architecture interface, and illustrates its use with an implementation of fetch&add. Subsection 4.2.4 follows with several micro-benchmarks measuring the performance of individual Active Message operations and demonstrating a five-fold reduction in communication overhead over Vertex's send&receive. Subsection 4.2.5 describes the implementation and shows that the performance of Active Messages is very close to the absolute limit of the hardware: a few dozen instructions indeed suffice and the resulting Active Messages layer succeeds in achieving low overhead and low latency communication with small messages. Measurements of the implementation reveal the bottlenecks in the current hardware and suggest avenues for further improvements. Subsection 4.2.6 measures the performance of two larger benchmarks based on matrix operations and Subsection 4.2.7 concludes the nCUBE/2 section.

While this section focuses exclusively on Active Messages, Section 5.1 contrasts Active Messages with send&receive such as implemented in Vertex and explains why send&receive is inherently a more expensive communication architecture.

4.2.1 Design constraints

On the nCUBE/2, the network can only be accessed via the on-chip DMA channels and the instructions to control DMA operation are privileged (refer to Section 3.1 for details). This means that all communication must involve the kernel which is also responsible for multiplexing network access between itself and all user processes. nCUBE's Vertex operating system allows multi-user operation using a space-sharing paradigm in which each parallel program is assigned to a power-of-two sub-cube and, in essence, owns that sub-cube for the duration of the program execution. While each node thus executes a single program at a time, the program can fork multiple processes on each node. Consequently, the responsibilities of the kernel are to enforce protection boundaries around each sub-cube and to potentially multiplex the network among multiple processes on each node.

To facilitate debugging, the Active Messages implementation is designed to coexist with, rather than replace, Vertex's send&receive communication architecture. At start-up, a user process has only access to the Vertex send&receive message passing primitives and must initialize Active Messages explicitly. From then on both sets of communication primitives are available.

Achieving this coexistence required careful integration of Active Messages into the existing kernel architecture and ultimately required certain compromises to be made. The most severe is the elimination of multitasking on nodes using Active Messages. This restriction was chosen for practical reasons: the Active Messages implementation as such already stresses the existing kernel architecture beyond its limits and nodal multitasking as implemented in Vertex is not of much use in parallel programs anyway (it was mainly designed to support transaction processing systems in which each node handles independent transactions and uses multiple processes per node to cover I/O latencies). A kernel redesign could allow multitasking with Active Messages at little overhead, but such a redesign was beyond the scope of the prototype effort discussed here.

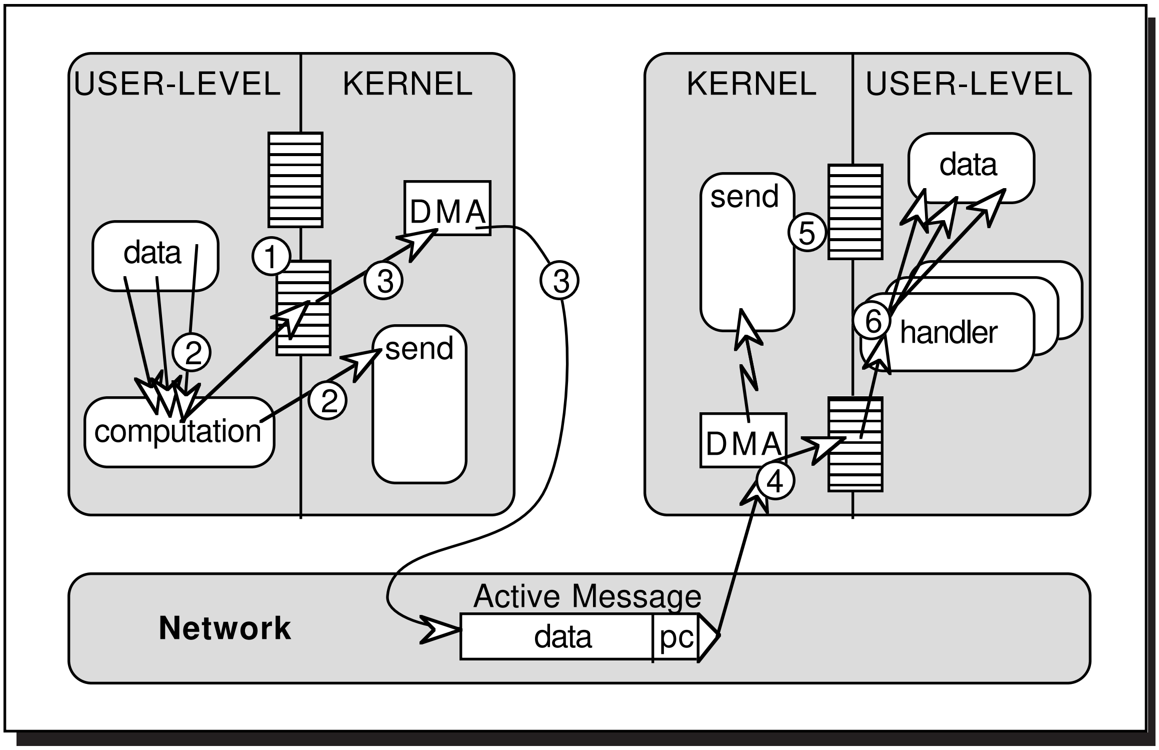


Figure 4-4: Steps involved in sending and handling an Active Message.

Implementation overview

Sending and handling an Active Message on the nCUBE/2 involves the following steps, illustrated in Figure 4-4:

1. the kernel provides a message buffer to the user process in user address space,
2. the user process composes the message in this buffer and calls `AMsend` which traps to the kernel,
3. the kernel verifies message destination and length, starts the appropriate DMA, sets-up a new message buffer, and returns to the user process,
4. the message arrives at the destination node into a DMA buffer and causes an interrupt,
5. the kernel re-initializes the input DMA to a new buffer, creates a stack frame for the Active Message handler in the user process' stack and returns—thereby starting the Active Message handler, and
6. the handler consumes the message, possibly sends a reply, and returns to the interrupted computation.

The implementation consists of the following parts:

- new kernel traps to send request and reply Active Messages,
- new message reception interrupt handlers to launch Active Message handlers,
- a user-level library interface between the kernel and Active Message handlers, and
- various modifications to the kernel, in particular to the scheduler, to accommodate Active Messages.

4.2.2 Design highlights

The major design issues in the Active Messages communication layer revolve around addressing the four key issues of Section 3.3. In addition, due to the fact that all communication has to involve the kernel, streamlining the user-kernel interface is of prime importance.

4.2.2.1 Streamlining the user-kernel interface

To keep the cost of Active Messages low the number of interrupts and traps must be kept to a minimum. This raises the question whether the nCUBE/2's facility of sending the message header and the data in separate message segments (cf. Subsection 3.1.3, page 29) is worth it. The benefit would be that on the receiving end the information in the header can be used to set-up the DMA for the data. If the headers are kept small, preallocating their buffer space is simple and the data itself can potentially be transferred directly into the application memory, e.g., buffer allocation costs can be kept low or eliminated altogether. The problem, however, is that the arrival of the header causes an interrupt (to set-up the DMA for the data). Unless the kernel busy-waits for that DMA to complete—which can take a long time for large messages—another interrupt must be signaled at the end of the data DMA. On the sending side the problem is similar and typically requires one trap to send the header and one interrupt to initiate the data DMA. (Note, however, that if the outgoing channel is idle the kernel can conceivably busy-wait for the header to be sent.) The bottom line is that using two message segments requires twice as many traps and interrupts than using a single segment.

The Active Messages implementation avoids interrupts completely on the sending side by sending only single-segment messages and by not queueing outgoing messages if the network is backed-up. The send kernel trap simply busy-waits for the previous message to go out and then sets-up the new DMA. The result of the efforts to minimize the number of privilege level switches is that, in the normal case:

- a message send never involves more than one kernel trap,
- a message reply involves one trap, and
- a message reception involves one interrupt or trap.

If the network is backed-up a reply from within a handler can involve several traps (as explained in more detail below). In addition, the first of a series of messages arriving during handler execution involves both, an interrupt and a kernel trap.

4.2.2.2 Data transfer

The Active Messages implementation reduces buffering to a minimum. On the nCUBE/2, it is not possible to forego intermediate buffering because all DMA data transfer has to be initiated by the kernel. Exposing the network interface to user-level is therefore impossible. The necessary DMA buffers are pre-allocated and buffer allocation and de-allocation is reduced to interchanging two pointers.

The DMA buffers are directly mapped into the user process address space in order to avoid copying of the message data from a system buffer into user space. With the simplistic memory management scheme supported by the nCUBE/2—only four contiguous data segments per user process—it is not practical to map individual buffers into the user address space. Instead a shared buffer area is pre-allocated in the data segment of the user process and used for all buffers.

The Active Messages implementation limits the message size and uses fixed-size buffers. The size is chosen such that the overhead of splitting-up larger messages is small compared to the message injection time, i.e., such that the message size limit does not affect peak communication bandwidth significantly.

To determine the minimal number of buffers required it is important to remember that Active Messages guarantees, by definition, that buffers of arriving messages can be re-used immediately after handler execution, and that the message layer blocks the sender until the outgoing network port is available, i.e., no queueing of outgoing messages occurs. Thus the minimal number of buffers is:

- each input DMA requires a buffer set-up for reception,

- each outgoing DMA has one buffer being sent[2],
- one output buffer is "being filled" by the computation, and potentially
- one input buffer is "being consumed" by an Active Message handler,.

Figure 4-5 shows the resulting buffer system with it three possible state transitions:

- on message reception the incoming DMA input buffer is switched with the user input buffer,
- on message send the user output buffer is switched with the outgoing DMA output buffer, and
- on message reply the user input buffer is switched with the outgoing DMA output buffer.

The last point implies that a handler must compose a reply message in its input buffer as the output buffer may already contain a partial message. Using the input buffer to compose the reply message has the benefit that information from the request message that is returned to the sender (e.g., to identify the reply) does not need to be copied if the message formats are designed appropriately.

4.2.2.3 Virtualizing the network

Given the strict space-sharing used by the Vertex operating system, virtualizing the network simply consists of ensuring that user messages are confined to the process' subcube. This requires a simple destination node check. In addition, the message length must be limited to the size of the buffers to avoid overrun at the destination.

Unlike traditional message passing systems where the user process controls message buffers and indicates to the kernel which buffer to send or receive next, the Active Message buffering scheme lets the kernel control the buffers and dictate which one the user process must consume and in which one it must compose the next message. This role reversal has the benefit that the user process never passes a pointer to the kernel which the latter would have to validate. The kernel can pre-translate all buffer addresses to user virtual addresses when the buffers are initially allocated such that it can pass these pre-translated addresses to the user process whenever necessary.

The disadvantage of this scheme is that the user process cannot send from its data structures without copying the data. Given that it is only rarely practical to add the Active Message headers in-place in front of the data to be sent this does not seem to be a severe restriction. Moreover, adding the capability of sending an arbitrary buffer for the cases where the advantage of eliminating the data copy outweighs the costs would be straightforward. Note that the overhead not only consists of the required buffer address validation, but also of the send completion interrupt required to notify the user process that the data has been sent and can be modified again. Such a direct send would validate the buffer addresses and set-up the outgoing DMA engine without changing the normal buffer associated with that DMA engine and without otherwise interfering with normal operation.

4.2.2.4 Send failure

On the nCUBE/2, the kernel and the user process must cooperate to deal with send failure within reply handlers. This need arises from the fact that the nCUBE/2 does not provide multiple priority levels for messages which would allow a straightforward implementation of deadlock/livelock-free request-reply communication patterns. The Active Messages implementation distinguishes between request and reply messages and uses separate kernel traps for each category. When sending a reply, the kernel attempts to inject the message into the network. If it is unsuccessful after a short period of time it simply returns to the handler with an error indication. At that point the handler is expected to free the reply message buffer and trap back to the kernel to accept incoming messages and nest their respective handlers onto the user stack. Eventually these nested handlers terminate and the original request handler can attempt to send the reply message again. Unfortunately the nesting depth cannot be limited, thus a stack over-

[2] The strict minimum would be one buffer for all outgoing DMA channels, but that does not bear any significant advantage.

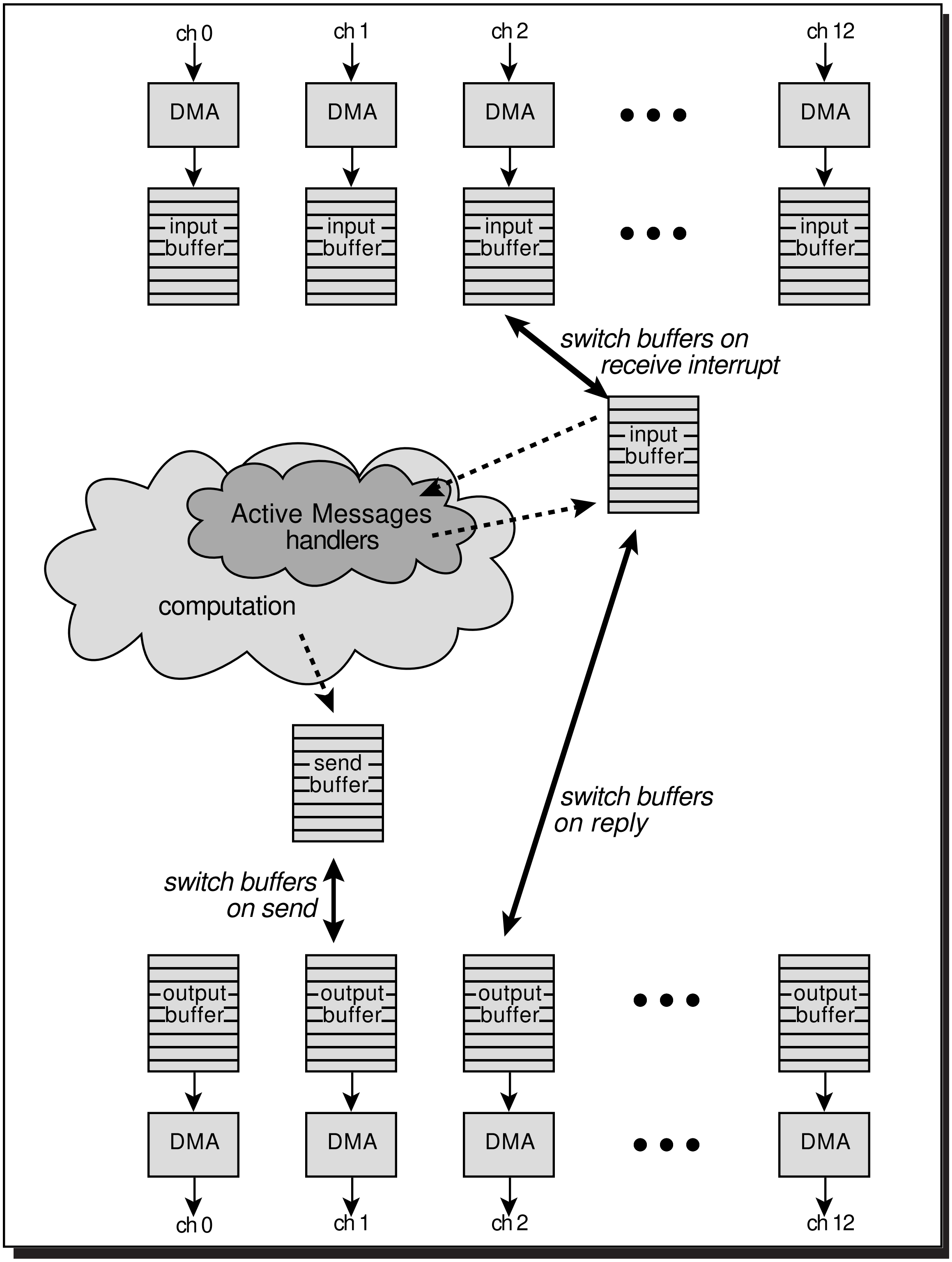


Figure 4-5: nCUBE/2 Active Messages buffer management scheme.

The buffering scheme minimizes buffer management. All buffers are pre-allocated and mapped into user-space. On send, buffer management consists of exchanging the buffer holding the message with the DMA engine buffer. On receive, the DMA buffer is switched with the buffer of the previously received message.

flow could cause termination of the program. It is recommended that handlers issuing reply messages use as small a stack frame as possible[3].

The solution of returning an error to the handler and have *it* free the reply message buffer was chosen to avoid user memory allocation within the kernel. In addition, the handler can decide whether it is best to save the reply message itself, or whether it is better to save a descriptor of the request. For example, while it is preferable to save the acknowledgment reply of a block-write it is probably better to save a descriptor for a block-read request.

4.2.2.5 Synchronization

In order to allow atomic sections in the computation, the kernel must somehow be notified when the program enters and leaves such a critical section so it can prevent Active Message handlers from interrupting. In addition, while the kernel must manage the DMA buffers it does not get notified of all the buffers state transitions. In particular, the kernel does not automatically know when an output DMA completes and thus the corresponding buffer becomes available again. It also does not know when an Active Message handler completes at which point its buffer becomes available and the next handler can be dispatched. While all these events could simply be signalled via kernel traps and interrupts, the Active Messages implementation uses a sophisticated user-kernel handshake in order to avoid the high cost of the straightforward solution.

The user-kernel handshake is based on a set of flags in a memory region shared by the user process and the kernel (the same region as for the DMA buffers is used). The user process controls two atomicity flags indicating the execution of an Active Message handler or of a critical section. If messages arrive while one of these flags is set the kernel ignores the message, sets a message pending flag and lets the user process proceed. At the end of the handler or critical section the user process checks the message pending flag and traps into the kernel if it is set. This scheme eliminates kernel traps to enable or disable interrupts which would be necessary in a more traditional approach.

Another difficulty arises from the fact that Active Message handlers are executed “at interrupt time” and that to guarantee the atomicity of handlers further interrupts must be prevented. In order to avoid deadlock in the case of a non-terminating Active Message handler the kernel starts a short timer whenever it disables message interrupts. Should the timer be exhausted without the user process having made any progress (e.g., completing the handler or leaving the critical section) the user process is considered to violate the protocol and is aborted. Given that all user-level communication is restricted to within the sub-cube “owned” by the parallel program slow handlers cannot hurt the performance of other programs. Setting the timer to a generous value (e.g., several hundred microseconds) is therefore acceptable as it really only needs to prevent a node from becoming unreachable to the point where it is impossible to force termination of the program.

3. In hindsight, a different implementation strategy which deals with the reply failure problem using flow-control should have been used.

User-kernel handshake details

The user-kernel handshake uses three variables in memory shared between the user process and the kernel:

RetPC	PC of interrupted computation, set by kernel
RetPSW	PSW of interrupted computation, set by kernel
Flags	Handshake flags: `inHandler`: handler currently executing, set by kernel, reset by user `inCritical`: critical section currently executing, set/reset by user `msgPending`: additional message(s) pending, set/reset by kernel

On message reception, the interrupt handler starts the Active Message handler only if both `inHandler` and `inCritical` flags are clear. If either one is set the interrupt handler delays the message by turning off the DMA to clear the interrupt cause, starting the time-out, setting the `msgPending` flag, and returning to the user. When an Active Message handler or a critical section completes it checks the `msgPending` flag and traps back into the kernel to pick-up the pending message.

4.2.3 Communication architecture interface

From the description of the design highlights it may seem that the Active Messages communication architecture is very complicated and requires its users to thoroughly understand the kernel-level implementation. Fortunately this is not the case: part of the implementation is a user-level library that hides the details of the user-kernel handshakes and presents a simple interface. This Subsection describes this interface to the nCUBE/2 Active Messages communication architecture. It is intended as a self-contained unit and therefore repeats some information presented above.

4.2.3.1 Overview

The nCUBE/2 Active Message layer implements an Active Messages communication architecture optimized for moderate sized messages with low-latency. On message arrival the handler referenced by the message is executed at interrupt time.

Message format

The maximum message size is fixed at boot time and on the order of one Kbyte. The first word of the message holds the address of the handler and the remainder is available for data.

Message placement

Messages are composed in memory in the outgoing buffer whose address is indicated by a global variable set by the message layer. Similarly, during handler execution, a global variable points to the memory buffer holding the received message. Replies are composed in the same buffer that the request message was received in. The rationale is that a request typically contains information to be returned in the reply. Using the same buffer can avoid copying this return information.

Message send and receive operations

The `AMsend` call sends the request message held in the outgoing buffer. `AMreply` similarly send the reply message held in the incoming buffer. No particular receive operation is provided as the handler is invoked automatically and is passed the address of the received message.

Message reception events

Handlers interrupt the ongoing computation and are nested on the normal execution stack, i.e., appear as if a spontaneous function call had been executed. The message buffer address is passed as an argument and is also available in a global variable. The message length is unknown; it may be placed in the message if the handler needs to know. The handler must "consume" the message quickly and return; as soon as the handler terminates, the message buffer is reused by the message layer.

Synchronization

Handler execution is atomic in that a message arriving during the execution of a handler is delayed until the handler terminates. To delay a message the message layer disables network interrupts in hardware. To safeguard from network deadlock, a timer is started when interrupts are disabled and the user process is aborted if the Active Message handler does not complete within a millisecond. Critical sections in the computation can be formed by setting a flag in memory to disable message reception.

Send completion

In principle Active Messages restricts communication patterns to requests and replies and expects the micro-architecture to provide some form of message priorities so that such patterns can be implemented deadlock and livelock free. While the nCUBE/2 does not support priorities of any form in hardware and there is little advantage to this restriction, the implementation still distinguishes between request and reply messages to help avoid stack overflow due to excessive handler nesting. This allows applications to guarantee that no overflow can occur by limiting the number of requests outstanding to any one processor at the same time and allocating enough stack space.

When sending a request message the message layer accepts incoming messages and dispatches the appropriate handlers. When sending a reply message, i.e., from within a handler, the message layer does not accept incoming messages. If the reply message cannot be injected into the network because of congestion the message layer returns an error to the handler. At that point the handler is expected to free the reply buffer and call the message layer to accept incoming messages, nesting the respective handlers.

4.2.3.2 Interface definition

Send request and reply Active Messages

`void *AMsendBuf;`	*pointer to current send buffer*
`void AMsend(int node, int length);`	*send request message*
`void *AMreplyBuf;`	*pointer to receive/reply message buffer*
`int  AMreply(int node, int length);`	*send reply message*
`void AMrecv(void);`	*handle incoming message after reply failure*

`AMsend` sends the message pointed-to by `AMsendBuf` and sets `AMsendBuf` to a new free buffer for the next message. At the destination the first word of the message is interpreted as the Active Message handler.

`AMreply` attempts to send the message pointed-to by `AMreplyBuf` and sets the latter to a new free buffer for another message. At the destination the first word of the message is interpreted as the Active Message handler. `AMreply` returns 0 if the message injection succeeds and a non-zero result otherwise. In case of failure, the handler must save the message buffer contents and call `AMrecv` to initiate handling of pending messages. When `AMrecv` returns the handler may attempt another `AMreply`. Note that `AMrecv` should only be called when a reply fails.

Handler execution

```
typedef void AMhandler(void *buffer);
inline void AMdisable(void) { AMflags |= inCritical; }
void AMenable(void);
```

The handler is called as if a spontaneous function call had been executed with the first argument pointing to the received message (`AMreplyBuf` points to the message as well). The message length is unknown. After "consuming" the message the handler simply returnsat which point the message buffer is reused by the message layer. Critical sections in the computation can be formed by using `AMdisable` and `AMenable` to control handler execution.

4.2.3.3 Example of use: Fetch&add

The fetch&add example introduced in Subsection 4.1.2 is easily implemented with Active Messages on the nCUBE/2. Figure 4-6 shows the complete implementation consisting of an internal synchronization variable, a stub to format and send a fetch&add request message, the request handler performing the fetch&add and sending a reply message back, and the reply handler completing the operation. In this simple implementation the processor busy-waits after launching the request message awaiting the reply. A more sophisticated version could continue computing, possibly launching additional fetch&adds and only checking the flag when the result is actually needed. In such a case the message would have to contain an synchronization identifier so that each reply can be matched with the correct request.

4.2.4 Micro-benchmarks

The objective of the micro-benchmarks is to obtain detailed best-case timings of the various Active Messages operations. The benchmarks are organized around a simple *signal* primitive which sends an Active Message to a handler incrementing a counter whose address is passed in the message. The message length can be varied from 1 to 256 words although no data is moved in and out of the buffer. The benchmark operates in four phases:

1. Node 0 sends N 4-word signal messages back-to-back to node 1. The total time divided by N gives the message handling overhead[4].
2. Node 0 sends N 4-word signal messages back-to-back to node 1 and node 2, alternating between the two. The total time divided by N gives the sending overhead.
3. Node 0 sends one 4-word message to node 1 which replies immediately, this is repeated N times. The total time divided by N gives the round-trip delay. This phase is repeated for nodes 2^i-1, i=1..10 and serves to verify the parameters of the network hardware.
4. Node 0 sends N 256-word signal messages back-to-back to node 1. The total time divided by N allows to verify the network injection time.

The results of running these benchmarks on a 1024-node nCUBE/2 system are shown in Table 4-1 and demonstrate that the Active Messages implementation achieves almost a 7-fold reduction in overhead over send&receive from 194µs to 28µs! Phase 3 shows an average increase in the round-trip delay of 4.5µs per extra hop which (divided by 2 to account for the round-trip) corresponds to the 2.2µs routing time claimed by the manufacturer. The per-byte injection time is also as expected.

The minimal round-trip time of 76.2µs is unfortunately not easily explained. The overhead of one message (28µs) plus the injection time (5.9µs for 3 words with an end-of-message byte) yields a round-trip time of 67.8µs. The missing 8.4µs are probably due to a combination of delays in the DMA start-up and the memory system. The memory cycles stolen by the DMA are relatively expensive because they cannot use the single cycle DRAM page mode access.

4. Really: the maximum of the sending and handling overheads. It turns out that handling is more expensive.

```
# Synchronization variable
typedef struct { volatile int flag, value; } fetch_add_sync;

# Message format
typedef union {
  struct {    request message
    void      handler;              request handler
    fetch_add_sync  *sync;          synchronization variable
    int       *addr, incr;          address to increment
    int       ret_node;             node to send reply to
  } req;
  struct {    reply message
    void      handler;              reply handler
    fetch_add_sync  *sync;          sync. variable (same location as in request message)
    int       value;                fetch&add result
  } repl;
} fetch_add_msg;

# Initiate fetch&add request, wait for completion and return result
int fetch_add(int node, int *addr, int incr) {
  fetch_add_sync sync;                    synchronization variable
  fetch_add_msg *msg = AMsendBuf;         compose request message
  msg->req.handler = fetch_add_h;
  msg->req.sync = &sync;
  msg->req.addr = addr;  msg->req.incr = incr;
  msg->req.ret_node = MYPROC;
  sync.flag = 0;                          init synchronization flag
  AMsend(node, sizeof(fetch_add_msg.req)); send request
  while(sync.flag != 1) ;                 busy-wait for completion
  return sync.value;                      return result
}

# Handler for fetch&add request
void fetch_add_h(fetch_add_msg *msg) {
  fetch_add_msg temp;                     ... in case of reply failure
  int ret_node = *msg->req.ret_node;
  int value = *msg->req.addr + msg->req.incr; perform fetch&add
  *msg->req.addr = value;
  msg->repl.handler = fetch_add_rh;       compose reply message
  msg->repl.value = value;
  while(AMreply(ret_node, sizeof(fetch_add_msg.repl))){
    temp.repl = msg->repl;  msg = &temp;  reply failure: save message
    AMrecv();                             ... handle incoming messages
  }                                       ... and try again
}

# Handler for fetch&add reply
void fetch_add_rh(fetch_add_msg *msg) {
  msg->repl.sync->value = msg->repl.value;
  msg->repl.sync->flag = 1;
}
```

Figure 4-6: Fetch&add implementation with nCUBE/2 Active Messages.

Phase	Metric	Active Messages time	Send&receive time
1	receive overhead	15.0µs	102µs
2	send overhead	13.1µs	92µs
3	round-trip node 0 to node 1	76.2µs	328µs
	round-trip node 0 to node 3	80.7µs	334µs
	round-trip node 0 to node 7	85.2µs	337µs
	round-trip node 0 to node 15	89.1µs	—
	round-trip node 0 to node 31	94.0µs	—
	round-trip node 0 to node 63	98.5µs	—
	round-trip node 0 to node 127	102.8µs	—
	round-trip node 0 to node 255	106.9µs	—
	round-trip node 0 to node 511	111.6µs	—
	round-trip node 0 to node 1023	116.3µs	—
4	per-byte injection time	0.45µs	0.45µs

Table 4-1. Micro-benchmark results for Active Messages on the nCUBE/2.

4.2.4.1 Performance modeling

The timings gathered with the micro-benchmarks can be expressed more conveniently using the performance models introduced in Section 2.3. Converting µs to clock cycles yields around 700 CPM (clocks per message) for small messages. The message start-up cost α is 28µs and the per byte cost β is 0.45µs. The maximum communication throughput R_∞ is 2.2Mb/s and half this throughput is achieved with a message length $N_{1/2}$ of 62 bytes. Table 4-2 expresses the performance in terms of the LogP model. Given that LogP assumes fixed size messages, the table shows two sets of values, one for short 8-word messages and one for long 1Kbyte messages.

4.2.5 Active Messages implementation

While the micro-benchmarks show an impressive performance improvement of Active Messages over send&receive, it is not immediately obvious why Active Messages are not even faster: the entire path to send a signal message, as used in the micro-benchmark above, is only 24 instructions long. How can these take 260 clock cycles (13µs at 20Mhz) for a CPI of almost 11 when the average CPI is close to 4? The situation on message reception is similar: 37 instructions take 300 clock cycles (15µs at 20Mhz) for a CPI of over 8. The paragraphs below discuss the interesting phenomena in the micro-architecture that explain the observed performance and that point out simple hardware fixes that would increase the performance substantially. Given that the relevant parts of the implementation are short they can be shown here.

Short 8-word (32-byte) messages		
L	= 25µs	Injection for 8 word message + 5 hops = 14.4µs+10µs (5 hops is avg. distance for *P*=1024)
o	= 14µs	(Send overhead + Recv overhead) / 2 = (15µs+13µs)/2
g	= 14µs	*g*=*o*, possibly a bit larger On avg. the network volume is 2 messages per node, given that $L \approx 2o$ there should be little contention
P	= 2..1024	
Long 256-word (1K-byte) messages		
L	= 471µs	Injection for 256 word message + 5 hops = 461µs+10µs (5 hops is avg. distance for *P*=1024)
o	= 14µs	(Send overhead + Recv overhead) / 2 = (15µs+13µs)/2
g	≥ 236µs	$g \geq L/2$, probably significantly larger On avg. the network volume is 2 messages per node, but *g* should be even larger to account for routing contention
P	= 2..1024	

Table 4-2. LogP parameters for nCUBE/2 Active Messages

4.2.5.1 Message send

The 15-instruction path taken through the kernel when sending an Active Message is shown in Figure 4-7 with Table 4-3 defining the kernel variables used. Noteworthy details are:

- The first 4 instructions adjust the destination node address for the sub-cube origin and store it at the front of the message buffer. The message length is limited in instruction 6 through a simple mask.
- A difficulty with the nCUBE/2 routing is that the message must be initially sent out on the correct network port. Furthermore, while the DMA instructions require the register number corresponding to the chosen port, the information in status registers has one bit per port and thus must be masked appropriately. In order to determine all the required values quickly, the kernel maintains a table mapping the node number to a port index into several tables, one for each type of value needed (lines 5, 8, 10, 11, 12, and 14).
- The entire instruction sequence contains only a single conditional branch (instruction 9). This branch is nominally taken if the output DMA engine is busy. However, the node mapping tables are set-up such that the local node address and the addresses of nodes beyond the user processes' sub-cube are mapped into bits of the status register which are always zero. Thus the branch is also taken for messages which should be looped-back on the local node and for messages causing an error.
- Buffer management is reduced to flipping two pointers in instructions 12 and 14.
- The physical-to-virtual address adjustment for the message buffer addresses is precomputed and can be applied in a single instruction (line 13).

From the code it becomes clear that one of the factors contributing to the observed CPI for this instruction sequence is the high frequency of complex addressing modes and of operands in memory. To help the analysis, Table 4-4 breaks the instruction count down into categories representing the functions performed[5] and shows the number of memory accesses performed. The 18 memory accesses shown, however, explain only part of the cost. As it turns out (and only a run on the hardware simulator at nCUBE could confirm the details) a large fraction of the cycles is lost in pipeline bubbles in the instruc-

```
# AMsend: Active Message send system call interface
inline void AM_send(int dest_node, int length)
#          -> R0=dest_node, R1=length
{
        asm("trap #SEND");
}
```

```
# Active Message send system call
SEND:   addw3  cubase,r0,r2            adjust destination for sub-cube origin
        andw   #cumask,r2              limit destinations to machine size
        movw   send_buf,r3             buffer to be sent
        movw   r2,(r3)                 store dest in message
        mvub   out_port(r2),r2         xlate node to output port offset
        andw   #lenmask,r1             limit message length
        stpr   #outtrr,r4              get output DMA status register
        bitw   bitmsk(r2),r4           test for channel ready
        be     s1                         not ready: busy-wait
s0:     lptr   r3,out_reg(r2)          load dma pointer
        lcnt   r1,out_reg(r2)          load dma count
        movw   out_buf(r2),send_buf  set new user output buff
        addw3  segoff,send_buf,send_usrbuf  xlate to virtual user addr
        movw   r3,out_buf(r2)          set new channel buffer
s4:     reti                           return to user
# channel not ready, first test if send to self or send beyond sub-cube
s1:     cmph   nodeid,(r3)             compare node to local node id
        be     snd_self                send to self (not shown)
        cmph   cusize,(r3)             check for node beyond sub-sube
        ble    s4                      error, drop message
# wait for previous message to go out
        ldpr   #mskin,#psw             enable vertex's output interrupts
s2:     stpr   #inrdy,r4               get input DMA status reg.
        bne    snd_recv                just recv'd a msg? service it (not shown)
        stpr   #outtrr,r4              get output DMA status register
        bitw   bitmsk(r2),r4           wait for prev msg to go out
        bne    s0                      ready, send message
```

Figure 4-7: `Send` system call implementation for nCUBE/2 Active Messages.

The Active Message send system call interface uses the calling convention to load the destination node and the message length into registers R0 and R1 and traps to the kernel. The typical path through the kernel is 15 instructions long and includes protection checks, DMA set-up, and buffer management. If the outgoing DMA channel is busy the kernel busy-waits and allows handlers for incoming messages to be dispatched.

prefetch pipeline. The reason is that these instructions are implemented in the prefetch unit. This prevents any prefetching from occurring during their execution, which takes almost a "dozen cycles". (The exact details remain company confidential, but do not provide further insights anyhow.) The interesting observation is that the hardware designers did not feel compelled to particularly optimize the network access instructions given that normally (i.e., in the send&receive implementation) they are

[5] It obviously would be more precise to break the costs down into cycles instead of instructions. Unfortunately the CISC nature of the instruction set and the lack of precise information on the operations of the micro-engines prevents precise accounting of clock cycles.

Kernel variables and constants	
cubase	origin of sub-cube current program runs in
cumask	cube mask: $2^{(\text{machine size})}-1$
culimit	number of nodes in the subcube
nodeid	number of local node
lenmask	mask to limit message length
segoff	user data segment physical to virtual offset
usrstk	user stack base (where the PC/PSW is saved on kernel entry)
usrint	PC and PSW of handler stub
tstamp	time stamp when interrupts disabled
send_buf	current user send buffer address
recv_buf	current user receive buffer address
out_port	table of output port number indexed by destination node
bitmsk	table of channel bit mask indexed by port number
in_reg	table of input DMA register offsets indexed by port number
in_buf	table of input DMA buffers indexed by port number
out_reg	table of output DMA register offsets indexed by port number
out_buf	table of output DMA buffers indexed by port number
User-kernel shared handshake variables	
send_usrbuf	current user send buffer user virtual address
recv_usrbuf	current user receive buffer user virtual address
flags	handshake flags
usrret	PC and PSW of computation interrupted by handler

Table 4-3. Kernel variables for nCUBE/2 Active Messages implementation.

Instruction category	instructions	memory accesses
kernel trap	2	6
check destination+length	4	2
determine output port	1	4
check port ready	3	0
set-up DMA	2	0
buffer management	4	6
total	16	18

Table 4-4. Instruction breakdown by category for nCUBE/2 Active Message send.

```
 # Input DMA interrupt handler (for channel 0)
 1: intr0: movd   r15,stk                    push user reg onto stack
 2:        movw   in_buf+0*16,r15            load DMA buffer pointer
 3:        xorb   flags,3(r15)               fizzzz...
 4:        cmpb   #0x40,3(r15)               test msg type and handshake flags
 5:        bne    bsy0                       user busy or vertex message
 6:        lptr   recv_buf,#0                set new DMA input buffer
 7:        lcnt   #0,#0                      start DMA (count is ignored)
 8:        orb    #inHandler,flags           set inHandler bit
 9:        movw   recv_buf,in_buf+0*16  swap buffers
10:        movw   r15,recv_buf
11:        addw3  segoff,r15,recv_usrbuf  set user's pointer
12:        movw   usrstk,r15                 where to find user's PC
13:        movd   (r15),usrret               save user's interrupted PC and PSW
14:        movd   usrint,(r15)               set user's int. handler PC and PSW
15:        movd   stk,r15                    restore user reg
16:        reti                              return to user to handler
17: bsy0:  user busy or vertex message
18:        bg     vtx0                       vertex message (code not shown)
19:        xorb   flags,3(r15)               unfizzzz
20:        ldpr   #0x80000000,#inen          disable input ints
21:        stpr   #timlo,tstamp              take time-stamp
22:        orb    #msgPending,flags          set msgPending flag
23:        movd   stk,r15                    restore user reg
24:        reti                              return to user
```

Figure 4-8: Input DMA interrupt handler for nCUBE/2 Active Messages.

hidden among 100's of instructions and do not contribute a significant fraction of the communication overhead.

Besides highlighting the need to optimize the DMA instructions, the instruction category breakdown in Table 4-4 shows that a few simple additional hardware features could cut down the communication overhead dramatically. The checks on the destination node address and on the message length could be performed in hardware easily. The selection of the correct output port could be simplified or eliminated altogether by multiplexing the DMA engines onto a single set of interface registers. Given the small volume of the network, it is actually not clear whether multiple DMA engines are of great benefit. Reducing the buffer management further is somewhat difficult without eliminating the DMA altogether. For small messages, such as the ones used in the fetch&add example, this may well be worthwhile.

Message reception

The reception of Active Messages occurs in two phases: first the kernel interrupt handler switches buffers, re-initializes the DMA for the next message, and starts a user-level stub which, in a second phase, dispatches the Active Message handler proper, checks for additional messages, and finally returns to the computation.

Figure 4-7 shows the path through the kernel interrupt handler. The code is straightforward with two exceptions:

- An otherwise uninterpreted bit of the destination node address word is used to flag Vertex messages allowing Active Messages to coexist with Vertex's send&receive. The only conditional (lines 3-5) combines this "vertex bit" with the user-kernel handshake flags (the bit encod-

instruction category	instructions		memory accesses	
	kernel	user stub	kernel	user stub
interrupt	2	—	7	—
check for Vertex message	1	—	1	—
buffer management	4	—	6	—
restart DMA	2	—	1	—
start user-level interrupt handler	3	—	5	—
handler atomicity handshake	3	2	4	2
dispatch Active Message handler	—	3	—	3
return to computation	—	7	—	6
save registers	2	6	2	6
total	17	18	26	17

Table 4-5. Instruction breakdown for nCUBE/2 Active Message reception

ings were chosen to allow this) and simultaneously checks both with a single compare and branch.

- In order to start the user-level stub the interrupt handler substitutes the PC and PSW of the stub for those of the interrupted computation. A precomputed pointer points to the base of the kernel stack to allow the quick location of the saved PC and PSW.

The user-level stub is shown in Figure 4-9. Besides saving a few registers and dispatching to the actual Active Message handler, the stub implements the user-kernel handshake and simulates a `reti` (return from interrupt) instruction to return to the interrupted computation. The difficulty in returning to the computation lies in restoring the PSW. The condition codes in the PSW must be shifted before being set with a load condition codes instruction (`ldflg`), there is no direct way to restore the IEEE rounding mode, and, worst of all, the repeat mode (used to emulate vector instructions) cannot be restored. The solution adopted in the implementation is to check the PSW for the default setting of the rounding and repeat mode bits and to break out to a special restore sequence which traps to the kernel to execute a real `reti` if necessary.

Note that in order to perform the user-kernel handshake the stub must atomically clear the `inHandler` flag and check the `msgPending` flag. Thanks to the powerful CISC instruction set and to an appropriate bit encoding this is possible with a single `and` instruction (line 13) which clears the flag and compares the resulting byte against zero.

Table 4-5 shows the breakdown of instructions into categories by the function performed. Again, the interrupt is expensive in terms of memory accesses and the DMA instructions take many cycles each. However, the numbers also show that the Active Messages dispatch does not come for free. Passing the interrupt through to user-level is not cheap and the second dispatch to the Active Message handler adds to it. The difficulty of returning to the interrupted computation is also out of proportion. If restoring the PSW were simple it could be incorporated directly into the Active Message handler together with the user-kernel handshake, eliminating the stub as such altogether. The kernel could then dispatch the Active Message handler directly saving a significant number of cycles.

Summary

The instruction sequences implementing Active Messages are very short. A major part of the execution cost stems from the privilege level crossings and from the DMA instructions which any message layer must include. Only a few instructions could hypothetically be eliminated by another message layer and would not affect performance dramatically. The precise cost breakdown possible due to the simplicity

```
# User-Kernel handshake variables, in user space
outbuf .equ   0x80000000        output buffer pointer
inbuf  .equ   uoutbuf+8         input buffer pointer
iret   .equ   uoutbuf+12        save area for interrupted PC
ipsw   .equ   uoutbuf+16        save area for interrupted PSW
flags  .equ   uoutbuf+20        user-kernel handshake flags

# Bits in uflags
inHandler  .equ 1               in handler
msgPending .equ 2               additional message(s) pending
up_crit.equ   4                 in critical section
```

```
# Active Messages handler stub
_AM_hstub:
       movw   iret,stk              push the interrupted PC
       movd   r0,stk                save caller-saves registers
       movd   r1,stk
       movd   r2,stk
       movd   r3,stk
       movd   r4,stk
       movd   r5,stk
       rotw3  #2,ipsw,stk           push int. PSW, ready for restore
       movw   inbuf,r0              get input buffer address
       movw   (r0),r1               get Active Messages handler addr
       call   (r1)                  dispatch to handler
       andb   #~inHandler,flags     clr up_hand&up_crit, check up_pend
       bne    stub_more             oops, another message is pending
       movw   stk,r0                pop interrupted PSW
       bitw   #0x0c000003,r0        test if rounding control or repeat mode bits set
       bne    stub_special          yes, play tricks (not shown)
       ldflg  r0                    restore PSW: load CC and RC=0
       movd   stk,r5                restore saved user registers
       movd   stk,r4
       movd   stk,r3
       movd   stk,r2
       movd   stk,r1
       movd   stk,r0
       ret                          return to interrupted computation

# More messages arrived while servicing this one
stub_more:
       orb    #inHandler,flags      set up_hand flag again
       trap   #RECV                 trap to get next message
       bitb   #msgPending,flags     got one?
       be     stub_m2               nope
#      got one: ... handler dispatch code similar to above elided...
stub_m2:
#      none: return to user code similar to above elided...
stub_special:
#      play special tricks to deal with repeat-mode and rounding-mode
```

Figure 4-9: User-level interface to interrupt handler in nCUBE/2 Active Messages.

of the implementation suggests a number of evolutionary hardware changes that could further reduce the overhead dramatically.

4.2.6 Macro-benchmarks

The micro-benchmarks have shown that Active Messages are capable of high peak performance. But does this performance level carry through to real programs? The following two simple benchmarks suggest that meaningful higher-level communication abstractions can be layered on top of Active Messages without compromising performance.

4.2.6.1 Matrix multiply

The first benchmark implements a simple version of matrix multiply using split-phase remote memory fetches. The benchmark shows that it is possible to obtain close to peak performance using the small Active Messages. The design decision to concentrate on small messages in order to provide the lowest possible overhead is shown to pay off.

The benchmark program multiplies two 64-bit floating-point matrices stored as blocks of columns per processor as shown in Figure 4-10. For *N*x*R* and *R*x*M* input matrices *A* and *B* spread across *P* processors, each processor starts with *R*/*P* and *M*/*P* columns and ends up with *M*/*P* columns of *C* (the benchmark code assumes that *M* and *R* are multiples of *P*).

Algorithm

In the inner loop shown in Figure 4-11, each processor fetches one, typically remote, column of A and multiplies it (outer product) by a local sub-row of B, summing the result into the local part of C. The program is derived from a simple uniprocessor version using an outer product formulation:

```
for k = 0 to R-1 do
  for j = 0 to M-1 do
    for i = 0 to N-1 do
      C[i,j] = C[i,j] + A[i,k] + B[k,j]
```

An efficient method to access the remote values of A is to fetch a column of A in one block before operating on it. The following matrix multiply sample code uses a `get` to fetch remote memory. The se-

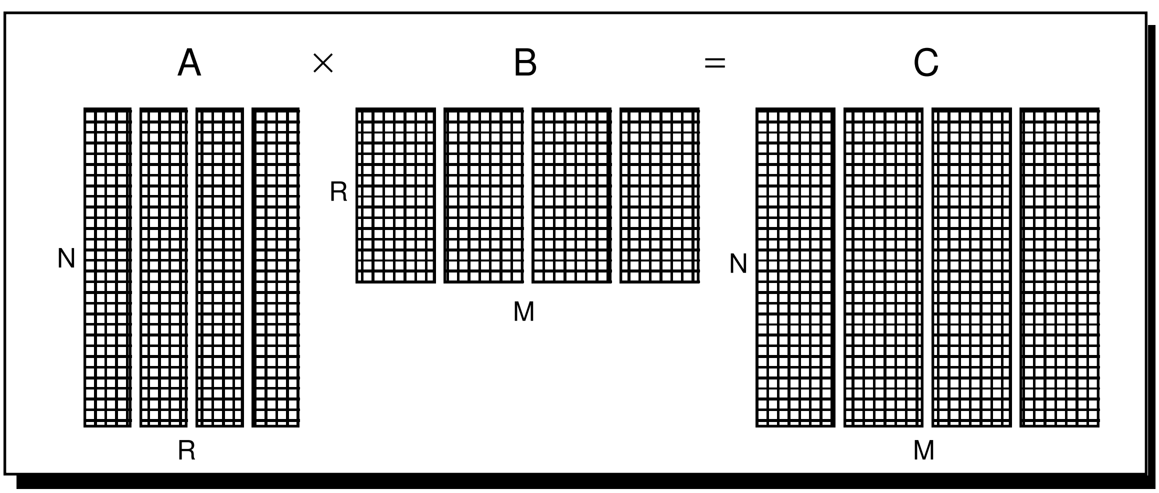


Figure 4-10: Matrices layout in blocks of columns for matrix multiply.

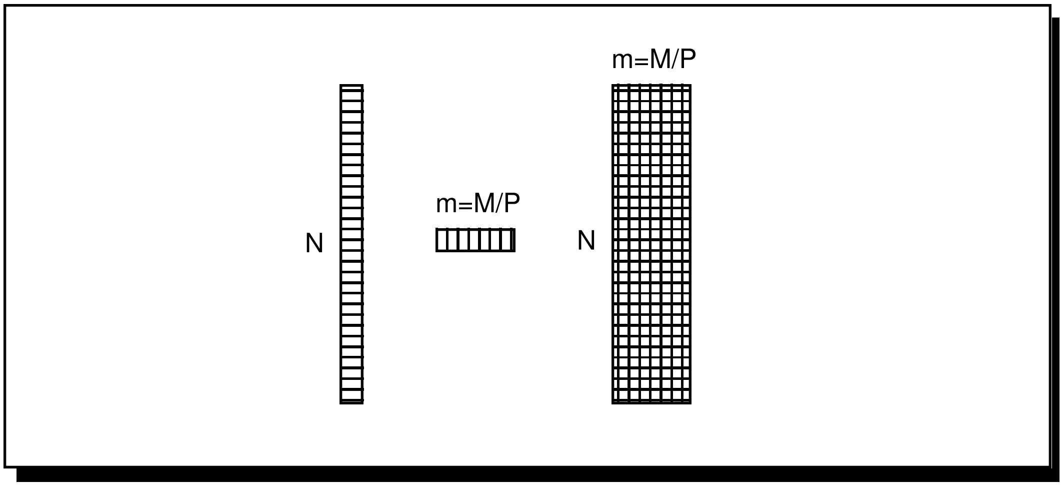


Figure 4-11: Matrix multiply inner loop.

mantics of `get` are described in detail as part of the Split-C language in Section 6.1. In summary, `get` fetches a remote memory block described by memory address and length. It is split-phase in that it only initiates a request Active Message and immediately returns. The handler for the reply stores the data in memory and increments a flag which can be used by the computation to determine that the `get` has completed.

The straight-forward multiprocessor version shown below is adapted from the uniprocessor code. It initiates the fetch for a column of *A* and immediately busy-waits for it to complete. In addition, the iteration order of the outer loop is shifted such that processor *i* starts with its columns of *A*, the fetches from processor *i*+1, and so forth. This staggering of the communication pattern avoids that processors 1 through *P*-1 start by requesting a column from processor 0 which would create a hot-spot.

```
for dk = 0 to R-1 do
  k = (k0+dk) % R                    stagger communication pattern
  get(k/P, A[*,k%P], V, flag);       get column into temp vector V
  wait(flag, 1);                     wait for column to arrive
  for j = 0 to M/P-1 do
    for i = 0 to N-1 do
      C[i,j] = C[i,j] + V[i] + B[k,j]

```

This simple version can be optimized by taking advantage of the split-phase nature of `get` to pipeline the remote fetches. The final version uses two temporary vectors and prefetches the column used in the next iteration. This prefetching permits the processor to continue computing while communication occurs. Note that the final matrix multiply code unrolls the two inner loops in order to achieve peak flop rate:

```
get(k0, A[*,k0%P], V1, flag);                get first column (prime the pump)
nk = k0;
for dk = 0 to R-2 by 2 do  unroll 2 times, assume dk mod 2 = 0 for simplicity
  k = nk; nk = (k+1) % R                     advance column index
  wait(flag, 1)                              wait for this column to arrive
  get(nk/P, A[*,nk%P], V2, flag);            prefetch next column
  for j = 0 to M/P-1 do
    for i = 0 to N-1 do
      C[i,j] = C[i,j] + V1[i] + B[k,j];
  k = nk; nk = (k+1) % R                     advance column index
  wait(flag, 1)                              wait for this column to arrive
  get(nk/P, A[*,nk%P], V1, flag);            prefetch next column
  for j = 0 to M/P-1 do
    for i = 0 to N-1 do
      C[i,j] = C[i,j] + V2[i] + B[k,j]
wait(flag, 1)                                wait for last column to arrive
for j = 0 to M/P-1 do
  for i = 0 to N-1 do
    C[i,j] = C[i,j] + V2[i] + B[k,j]
```

Performance

The performance of the matrix multiply can be predicted using a detailed analysis of the outer loop timing. In each iteration of the outer loop computation and communication are performed simultaneously. The computation time consists mainly of the outer product whose execution time can be estimated using the uniprocessor Mflops rate and the communication time can be estimated from the individual overheads and latencies involved. Figure 4-12 shows the timing diagrams for the outer loop for the cases where either the communication or the computation forms the bottleneck.

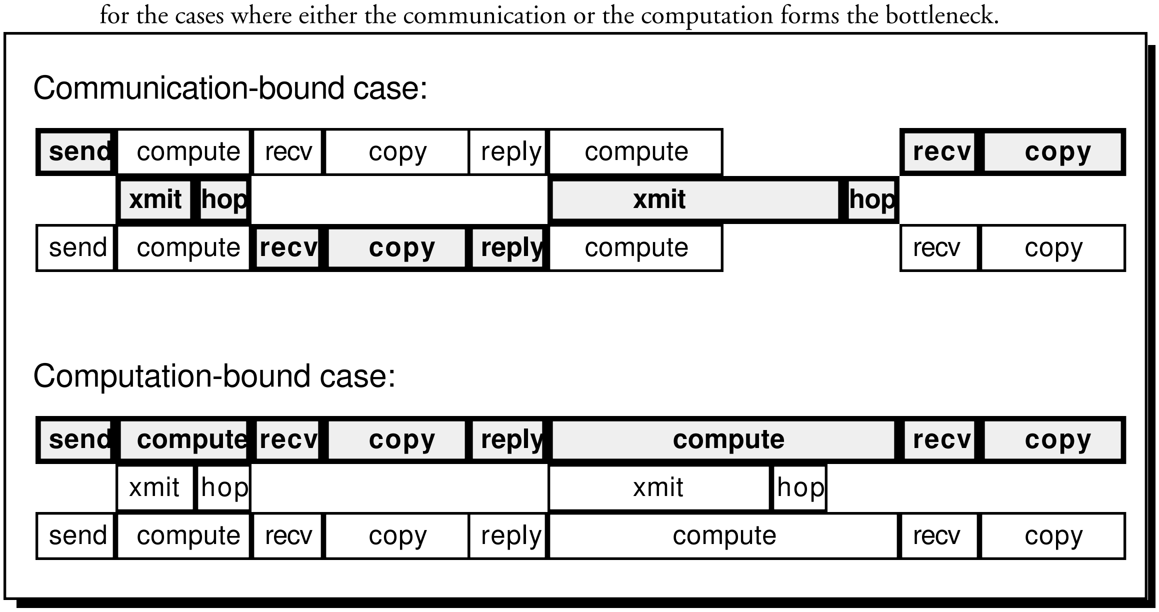


Figure 4-12: Timing diagram for outer matrix multiply loop.

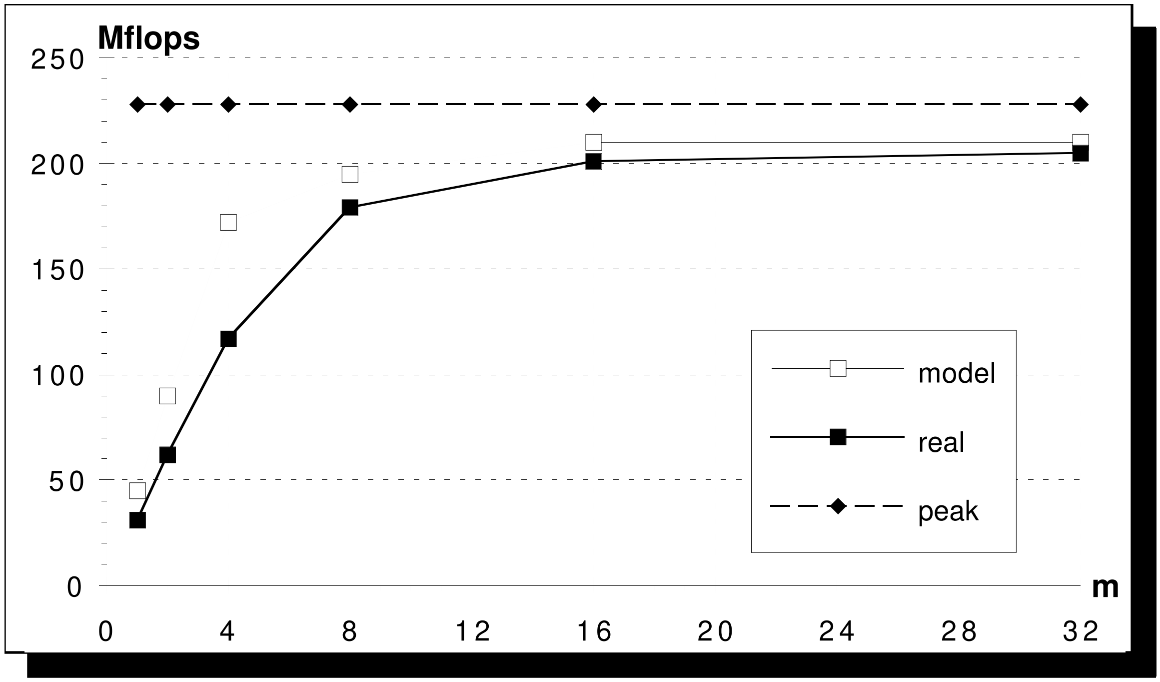


Figure 4-13: Matrix multiply performance in Mflops.

In order to validate the performance model the benchmark was run on a series of matrix sizes chosen to vary the computation/communication ratio. Figure 4-13 shows the results of 6 runs on 128 processors where the message size (determined by *N*) is kept constant, the communication to computation ratio (determined by *M/P*) is varied from 256 flops per message to 8192 flops, and the total number of flops is kept constant (by adjusting *R*). The figure shows that the model predicts the performance characteristics of the actual program rather well. The absolute numbers predicted are a little higher than the achieved performance due to the fact that the model does not take any network congestion into account.

While matrix multiply performance is generally measured by the processor performance, it is also interesting to look at the network performance. Figure 4-14 superimposes the processor utilization (in % of peak Mflops) and the network utilization (in % of peak Mb/s) for the same six runs. At low computation to communication ratios (e.g., small values of *m*) the processor is not very heavily utilized but the network is. Here the discrepancy between the model and reality is largest because the model does not take congestion into account.

Overall the performance of the matrix multiply benchmarks shows that due to the low overhead of Active Messages peak performance can be achieved with moderate size messages (here 1Kb) and small compute/communicate ratios (1/4 flop per byte at *m*=1 to 8 flops per byte at *m*=32). In addition, by building the right communication abstraction on top of Active Messages this peak performance can be achieved by a simple program.

4.2.6.2 Matrix transpose

The matrix transpose benchmark is intended to demonstrate that optimizing for small messages pays off doubly: not only is the communication overhead of Active Messages low, but the small messages create less network congestion. In matrix transpose the maximum message size is inherently small, as it scales at best with the square root of the matrix size.

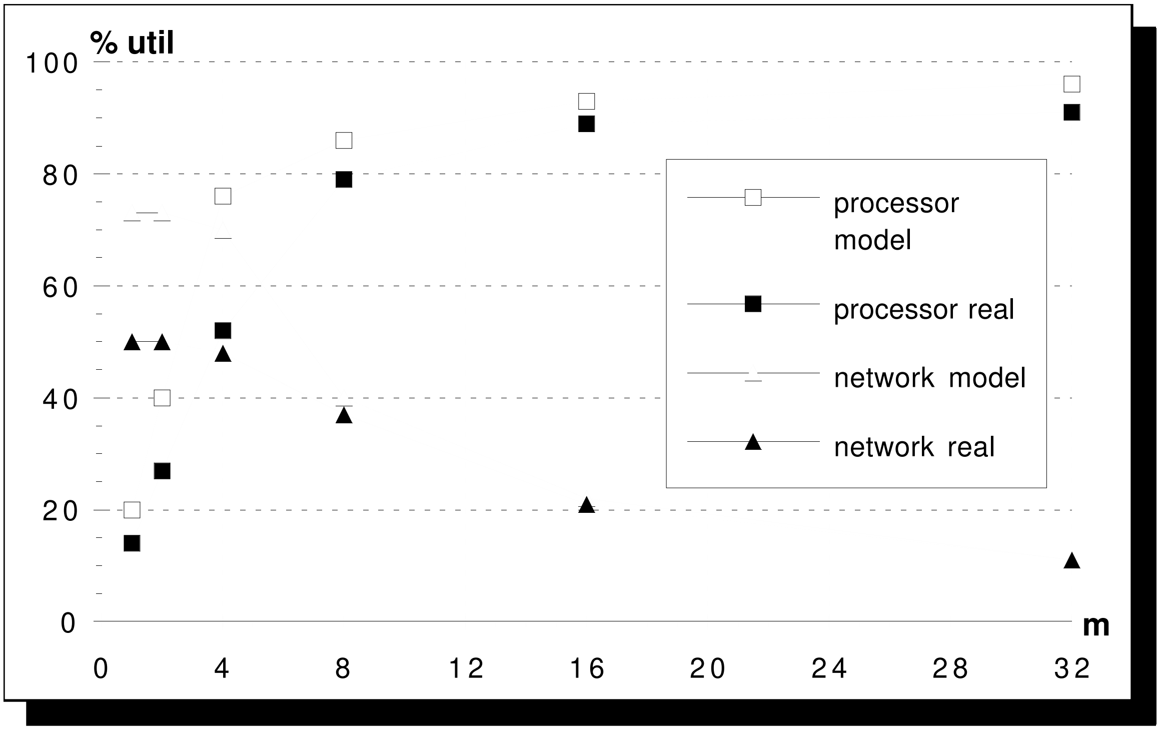


Figure 4-14: Matrix multiply processor and network utilization.

The matrix layout used in the transpose is identical to that used in the matrix multiply. For an *N*x*M* input matrix spread across *P* processors, each processor starts with *M*/*P* columns and ends up with *N*/*P* columns (the benchmark code assumes that both *M* and *N* are multiples of *P*).

The basic algorithm is simple: each processor walks through its sub-matrix, gathers data into a message, and sends a Split-C `store` message[6] to a remote processor. Each `store` message carries a destination address at which the data is placed. In addition, the handler increments a counter by the number of bytes received. After sending all messages, each processor waits until the counter indicates that it has received all its data from the other processors. For timing purposes all processors start with a barrier and enter another barrier at the end.

The actual benchmark consists of three variations on the basic algorithm differing in the amount of data sent in each message. Processors send the their data in a staggered order in which processor *i* starts sending data to itself, then processor *i*+1, and so forth. The three variants measured are:

1. *word*—a single value is sent per message,
2. *row*—a sub-matrix row is gathered and sent in one message, and
3. *block*—all sub-matrix rows destined to a processor are gathered in one message.

The performance of these variants is measured in two series of runs described in Table 4-6. The first one varies the message size (by adjusting the matrix dimension) and keeps the number of processors constant, and the second one varies the number of processors and keeps the message size constant.

6. The semantics of store are explained in detail in the discussion of Split-C in Section 6.1. While this benchmark is not written in Split-C (in fact there is no Split-C compiler for the nCUBE/2) it uses some of the Split-C primitives.

		constant machine size				constant message size				
matrix size		256	1024	2048	4096	256	1024	2048	4096	8192
message payload size	word	8	8	8	8	8	8	8	8	8
	row	16	64	128	256	64	64	64	64	64
	block	32	512	2048	8192	512	512	512	512	512
total message size	word	29	29	29	29	29	29	29	29	29
	row	37	85	149	277	85	85	85	85	85
	block	65	545	2081	8225	545	545	545	545	545
LogP params	L	45.8	45.8	45.8	45.8	36.8	45.8	50.3	54.8	59.3
	o	63	63	63	63	63	63	63	63	63
	g word	6.5	6.5	6.5	6.5	6.5	6.5	6.5	6.5	6.5
	g row	8.3	19.1	33.5	62.3	19.1	19.1	19.1	19.1	19.1
	g block	14.6	122.6	468.3	1851	122.6	122.6	122.6	122.6	122.6
	P	128	128	128	128	32	128	256	512	1024

Table 4-6. Message sizes and LogP parameters for the various matrix transpose runs.

The message and matrix sizes for two series of runs are shown. The first series keeps the machine size constant (at 128 processors) and varies the message size. The second series keeps message size constant and varies the machine size. For each machine/message size combination three versions of matrix transpose are measured: the first (*word*) sends a single 64-bit value per message, the second (*row*) sends a sub-row in each message, and the third (*block*) sends a sub-matrix per message. In each case, the total message size (e.g., including headers) as well as the size of the payload is indicated.

To evaluate the performance of the benchmark the algorithms are modeled using LogP. Because the model leaves the message size implicit, it is necessary to calibrate the parameters for the various message sizes used (the resulting LogP parameter values are shown in Table 4-6):

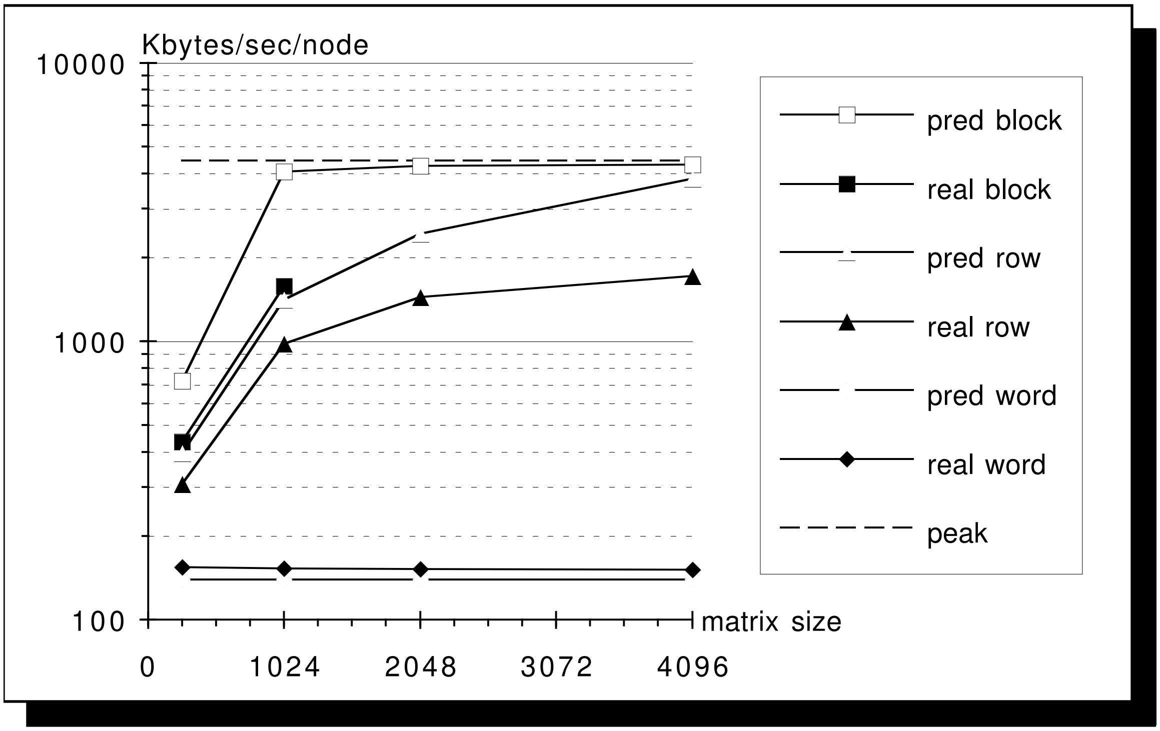
- o is the overhead of a `store` which is measured to be 36µs,
- g is limited by bisection bandwidth and is assumed to be $n/4.44$Mb/s, where n is the gross (i.e., incl. headers) message length in bytes,
- L is 14.3µs+$\log_2 P$*4.5µs for the average distance, and
- C, the per byte copy time, which is derived from an additional preliminary run on a single node.

Figures 4-15a and 4-16a compare the communication (payload) bandwidth measured in the various runs with the execution time predicted by the LogP model using the formula:

The results show that the version sending a single value per message performs poorly and the LogP parameters indicate that the communication is purely overhead limited. As a result the average network wire utilization during these runs, shown in Figures 4-15b and 4-16b, is around 12%.

Sending a row per message amortizes the overhead and brings the wire utilization above 30%, and that with messages as small as 85 bytes (64 bytes of data payload). Larger messages in the fully blocked version brings the utilization up to 40%. The comparison of the LogP predictions with the real runs is accurate for the runs which do not push the network to the limits, but the fact that the model does not take routing contention within the network into account yields optimistic predictions in the other cases.

(a) network bandwidth



(b) wire utilization

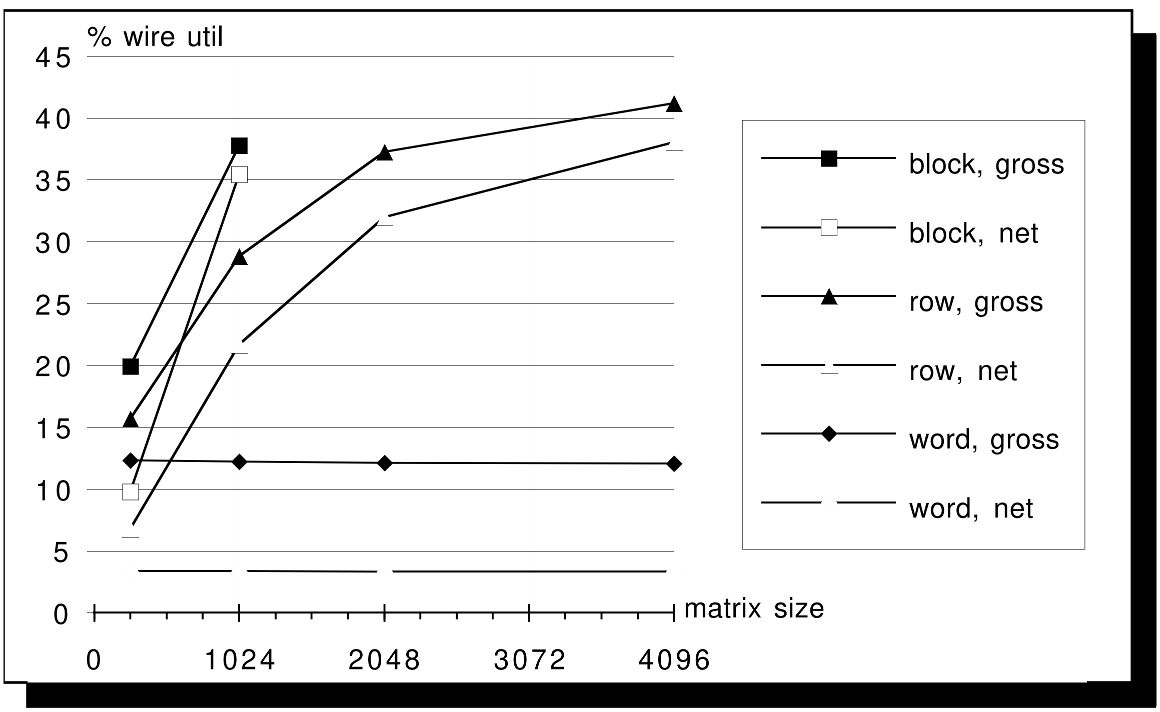


Figure 4-15: Matrix transpose on 128 processors.

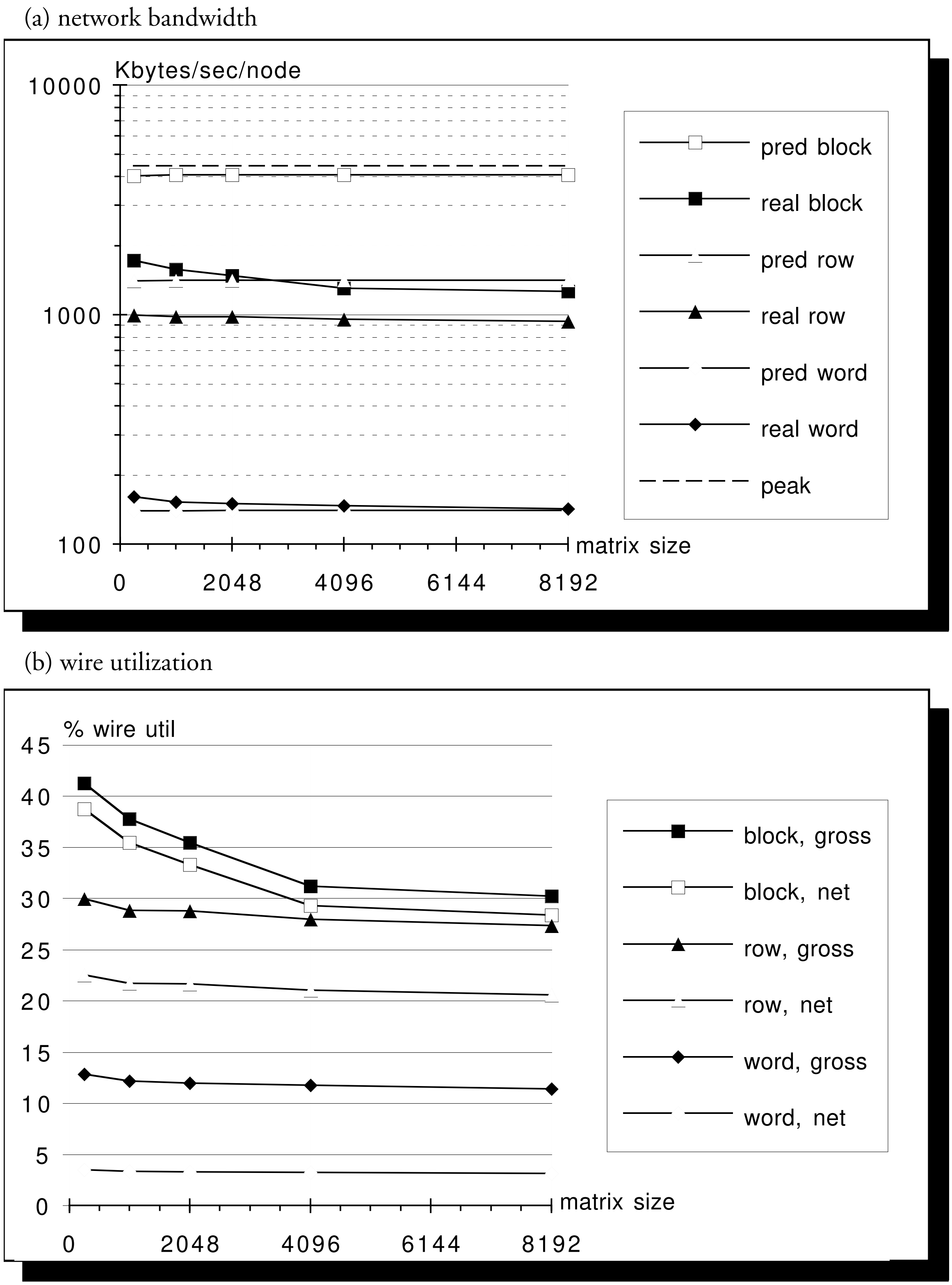


Figure 4-16: Matrix transpose with constant message size.

Overall the gross wire utilization achieved with the better algorithm variant is impressive in absolute terms: all wires in the entire hypercube are kept busy 40% of the time for a quite irregular communication pattern. Achieving this with a very simple algorithm which does not take the network topology into account and which uses short messages (well under 1Kbytes in all cases) demonstrates that Active Messages provides access to the full hardware performance.

4.2.7 Conclusions

The design of the Active Message layer focuses on delivering low-overhead communication with small messages. This is achieved through a minimalist approach in which every feature in the communication layer is carefully weighted against its execution cost. In particular, the Active Messages layer does not manage storage, does not queue messages, and performs only the simplest scheduling. These tasks are pushed up to the next layer of software where they can often be solved in a simpler more specific manner. For example, the two macro-benchmarks (matrix multiply and matrix transpose) did not require any special memory allocation because all matrices are allocated before communication occurs, they do not need queueing of messages because simple staggering of the message order generates a well balanced communication pattern, and the handlers maintain a few completion counters which allow simple synchronization of computation and communication.

The micro-benchmarks and the study of the implementation show that the efficiency of mapping Active Messages onto the micro-architecture is very close to the limits of the hardware. Implementing Active Messages instead of some ad-hoc access to the DMA has a measurable cost, but it is small. The cost is mainly in the dispatch of hardware interrupts to user level and could be reduced if minor instruction set problems were fixed, such as the difficulty in returning to the computation from the Active Message handler.

The macro-benchmarks demonstrate the efficiency of the mapping onto Active Messages. The performance achieved in the benchmarks is high and predictable (e.g., using the LogP model) and the limitation on message size is no obstacle to peak performance. In fact, while small messages do not always achieve quite as high performance as larger ones, the performance is more predictable because network congestion is less severe.

Beyond demonstrating performance, the macro-benchmarks also show that the versatility of the Active Messages communication architecture is conducive to the development of higher-level communication abstractions such as `store` and `get` without compromising efficiency.

Developing these higher-level abstractions is straightforward, as the fetch&add example show. Even though the details of the kernel implementation are subtle and the user-kernel handshake is intricate, the clean interface of the Active Messages communication architecture isolates the run-time substrate developer from these concerns. In the application itself these higher-level abstractions completely hide the details of sending and handling messages and of respecting the restrictions imposed by Active Messages such as request-reply communication.

The main problem in the Active Message interface is the semantics of `AMreply` which can return with an error without sending the message. The source of the problem lies in the lack of priorities in the network which does not allow for a clean solution. While the chance of a stack overflow is statistically very small if communication is random, very regular communication patterns can cause worst case behavior. In the matrix multiply benchmark, if one processor is delayed due to network congestion it will quickly receive additional `get` requests which then are likely to cause reply failures. Unfortunately in this case the failures have a positive feed-back effect as they delay the processor further. To solve this problem without hardware support requires either a more rigid global synchronization, for example a barrier every few `get`s, or some form of flow-control.

The Active Messages implementation uncovered a number of problems in the nCUBE/2 instruction set architecture and in the present implementation. The small number of instructions involved in an Active Message transmission allows a fairly careful accounting of where the time is spent and reveals a number of inadequacies in the hardware which are not really due to particular design difficulties but simply to the fact that, buried in the 100's of instructions the send&receive message layer takes to transmit a message, they were not deemed worthwhile worrying about. The most serious inadequacies are:

- the high cycle cost of the `trap`, `reti` and DMA instructions which are executed in the prefetch unit and cause prefetching to stall,
- the inability to inspect the message header before setting-up the DMA transfer of the data,
- the requirement of selecting the correct output DMA channel in software instead of having the router take care or instead of having only a single output DMA, and
- the fact that it is difficult to restore the PSW in user-mode in order to return from a user-level interrupt handler.

A number of interesting hardware features were found to be tempting but in the end inefficient to use:

- The support for messages composed of multiple message segments, each requiring a separate DMA transfer, is intended to allow the separation of the message header from the data such that the latter can be moved directly into user space. However, the cost of the additional DMA (including the additional interrupt) is too high to make this option useful. It would have been better to allow programmed access to the network FIFO to send and receive the header.
- The multiple sending and receiving DMA engines are intended to provide ample bandwidth between the memory system and the network. On the receiving side the many channels generate separately vectored interrupts and thereby do not cause additional software complexity over a single DMA channel. On the sending side however, the cost of choosing the correct output DMA channel is significant and does not seem worthwhile. Investing the real-estate used for the 14 output DMA controllers into deeper channel output buffers might have absorbed bursts just as well without software cost.
- The message multicast and message forwarding features built into the network routing are virtually useless during normal operation because they pose significant livelock and deadlock problems. However, note that message forwarding is required to send messages to/from I/O subsystems.

4.3 Active Messages Architecture on the CM-5

The send&receive message passing library provided by Thinking Machines Corp. (TMC) takes over 90µs end-to-end to transmit a small message between neighbor processors [LTD+92][7]. Starting from the hardware performance numbers presented in Section 3.2 it is difficult to explain such a high end-to-end latency: a few accesses to the network FIFOs plus the network routing time can account for at most 10µs. Clearly there must be a way to obtain better communication performance on the CM-5!

This Section demonstrates that an Active Messages communication architecture reduces the communication overhead on the CM-5 by over an order of magnitude. The important features of the design of the CM-5 Active Message layer (CMAM[8]) revolve around addressing the four key issues described in Section 3.3. The highlights of the CMAM design are presented in Subsection 4.3.2 which is followed by a description of the programmer's interface in Subsection 4.3.3 and includes an implementation of the fetch&add example sketched in Figure 4-3. Detailed performance data from micro-benchmarks in Subsection 4.3.4 demonstrates that the overhead of communication is less than 3µs. Subsection 4.3.5 uses the Active Messages instruction sequences to highlight that this level of performance represents the limit of the hardware capabilities. The performance of a larger benchmark program is shown in Subsection 4.3.6 and analyzed using the LogP model of parallel computation. The trade-offs between synchronous and asynchronous message reception are discussed in Subsection 4.3.7. Subsection 4.3.8 concludes the CM-5 Section with a review of the important implementation features.

While this Section focuses exclusively on the CMAM communication architecture Section 5.1 discusses the reasons for the low performance observed using send&receive.

4.3.1 Design constraints

One the major innovations in the CM-5 architecture is that the network interface is accessible from user-level. This considerably simplifies the implementation of the Active Messages communication macro-architecture, in particular as compared to the nCUBE/2 implementation. The CM-5 version of Active Messages does not require any modifications to the kernel: the CMAM message layer is a conventional user-level library.

While the ability to implement Active Messages without any kernel modifications had important benefits in terms of ease of development and compatibility with all CM-5 installations there is a serious drawback. Although the kernel is not involved in normal message send and reception, message arrival interrupts are dispatched to the kernel by the hardware. Without kernel modifications it was not possible to optimize the dispatch of these interrupts to user-level which, in the current kernel version, is prohibitively high. As a consequence, CMAM does not use interrupts and relies on regular network polling. This means that handlers are executed synchronously to the computation and only when the network is polled. A newer version of Active Messages implemented as part of CMMD 3.0 uses interrupts and is briefly discussed in Subsection 4.3.7.

Implementation overview

The CMAM implementation is heavily influenced by the fact that messages in the CM-5 are limited to five words in hardware. CMAM exposes this limitation, i.e., Active Messages on the CM-5 are limited to four words of data, the fifth being used to name the handler.

The combination of this short message length and the Sparc calling convention allows for a very efficient interface between C (or other similar HLL) and CMAM in that the message data can simply be

---

7. These measurements are for versions 1.3 and 2.0 of the message passing library CMMD. The performance of CMMD 3.0is discussed in Ch.6.

8. Pronounced \'se-mam\.

passed in registers as arguments to or from the CMAM functions. Sending and handling an Active Message on the CM-5 involves the following steps:

- the CMAM send function `CMAM_4` is called with all message information passed in registers,
- the message is pushed into the outgoing FIFO, if the FIFO drops the message `CMAM_4` retries,
- at the destination node the network interface status register is polled and indicates message arrival,
- the message is extracted from the FIFO and the Active Message handler is called with the message data passed as arguments in registers,
- the handler consumes the message, possibly sends a reply, and returns to the dispatch which polls for further messages and eventually returns to the interrupted computation.

The implementation of CMAM is contained in a normal user-level library and consists of the following parts:

- functions to send request and reply Active Messages and automatically poll the network,
- functions to explicitly poll the network,
- code to receive messages and dispatch to the appropriate handlers, and
- special handlers to receive bulk data requiring streams of messages.

4.3.2 Design highlights

This subsection discusses in detail the approach taken in CMAM with respect to the four key issues: handling send failure, supporting efficient synchronization, transferring data in and out of the network, and virtualizing the network. In addition, the short message size supported in hardware on the CM-5 poses additional challenges in defining a unique message format and in transferring bulk data which are also discussed.

4.3.2.1 Send failure

The two separate data networks in the CM-5 are perfect to support the request-reply communication patterns advocated by Active Messages. CMAM distinguishes request from reply messages and uses the “left” network for requests and the “right” network for replies (henceforth referred-to as the “request” and “reply” networks). When sending a request, incoming messages on either network may have to be handled before the FIFO accepts the outgoing message. However, when sending a reply, only messages arriving on the reply network need to be accepted and the request network can be safely ignored. Thus, only reply handlers will nest within request handlers and no further nesting is possible given that a reply handler is not allowed to send a message.

4.3.2.2 Synchronization

Without kernel modifications the cost of interrupts on message arrival is rather steep on the CM-5: while the hardware interrupt itself costs only a few tens of cycles, the path through the kernel takes a few hundred. In addition to this per-interrupt cost, forming critical sections in the computation is very expensive. The atomic Sparc instructions (`ldstub` and `swap`) which can be used to perform atomic memory updates issue an uncacheable read-modify-write memory cycle and thus operate at DRAM speed, taking 30 cycles each! Bracketing multi-instruction critical sections with interrupt disable and enable requires expensive kernel traps. Changes to the kernel could bring all these costs down, but were not considered in this version of CMAM.

Fortunately, the computation can periodically poll the network interface status to detect the arrival of messages instead of using message arrival interrupts. In many cases this turns out to be far simpler than one might expect: because sending messages requires checking the status register anyway (to make sure the message got sent), it is simple to include a poll in every message send. This means that as long as a node keeps sending messages, it will automatically poll the network. Only when the program enters a compute-only section of the code it is necessary to insert explicit polls. Depending on the compilation path these explicit polls can be inserted by the compiler or they remain the responsibility of the application programmer. Note that in all cases the polls must respect the use of the two networks, thus: when sending a request message both networks are polled, when sending a reply message, only the reply network is polled, and both types of explicit polls are available and must be used appropriately. When a poll detects the presence of a message, it inspects the first word and dispatches to the appropriate handler. When the handler returns, the network is automatically polled again such that successive messages are handled quickly.

Forming critical sections in the computation is trivial: it just entails not polling, thus, any instruction sequence not containing any message send or any explicit polls is atomic relative to Active Message handlers. The combination of having such "free" critical sections and of saving the cost of interrupts on message arrival compensates for the cost of explicit polling.

4.3.2.3 Message format

A difficult design decision was not to directly support Active Messages larger than the native CM-5 message size, which means that CMAM Active Messages are limited to four words of arguments. (Support for large transfers is available in the form of an extension to CMAM which provides a set of hard-coded Active Message handlers described in § 4.3.2.5.) This decision was taken because larger messages sent in multiple fragments cannot be reassembled at the destination transparently, i.e., without significant performance or storage impact. The origin of this problem lies in the fact that the network can deliver messages out of order; therefore each message must carry the equivalent of a sequence number and the receiver must keep track of the number of fragments received in order to detect the arrival of the entire message.

A hypothetical implementation of long Active Messages could operate as follows:

- Before sending the first fragment, the sender allocates a unique counter at the destination node to keep track of the fragments.
- When the first fragment arrives at the destination, the receiver allocates memory for the message.
- When all fragments have arrived, the Active Message handler proper is called and the memory can be deallocated after the handler terminates.
- Before the counter can be reused, the sender must be notified that it is available again.

Instead of hiding synchronization counters, memory allocation, and acknowledgment messages within the message layer it is more efficient to let higher software layers implement equivalent protocols because they can be tailored to each particular use. For example, a node requesting a large amount of data can allocate the counter and the memory before sending the request message. The replying node can then send the data in many small messages to a handler which knows the counter and memory addresses. Similarly, if the sending processor knows the destination address for the Active message data on the remote node (e.g., in a remote block-write) a dynamically allocated buffer is not necessary and if the high-level protocol itself requires an acknowledgment the availability of a synchronization counter can be signalled in the same message.

The bottom line is that in most cases the run-time substrate can transfer bulk data more efficiently than a universal low-level message fragmentation and reassembly protocol in CMAM could.

4.3.2.4 Data transfer

In order to achieve peak communication it is essential to streamline the data transfer into and out of the network interface. The position taken in Active Messages allows each implementation a large degree of flexibility such that mechanisms allowing direct data transfer between the network and application data structures can be devised.

The CM-5 communication micro-architecture allows the user-level code to directly access the network interface and store data into the outgoing FIFO or load data from the incoming FIFO. This raises the temptation to simply expose the FIFOs in the communication architecture in the expectation that the higher software layers can inline the instructions to move data directly from the application data structures into the FIFO and vice-versa. Upon closer inspection of the instruction sequences, however, two facts become apparent: the scheduling of the load and store instructions accessing the FIFOs is very critical to prevent processor stalls due to the slow network interface access across the MBUS, and inlining the instructions to access the network interface inflates the total code size and increases the register pressure[9] such that the benefits are substantially smaller than expected.

Using a conventional function call interface to CMAM proves to have many advantages. The Sparc calling conventions allow the first six words of arguments to a function to be passed in registers (the calling convention details are summarized in Subsection 3.2.2). Given that all the information necessary to send a message fits into six words (destination node, handler address, and four words of data) the entire message can be passed to CMAM in registers. Thus for sending messages, CMAM exposes the responsibility of loading the message data into the processor registers and hides the details of optimally pushing it into the outgoing FIFO. For handling messages, the situation is reversed. The message dispatch code loads the message from the incoming FIFO into processor registers and calls the handler, again passing the entire message in registers as supported by the calling convention. It is up to the handler to move the data into the application data structures as appropriate.

Temporary registers

The Sparc register windows prove convenient for allocating temporary registers in both the CMAM functions and the Active Message handlers themselves. With two instructions the CMAM functions can acquire a new register window with enough registers for all the temporary variables and, similarly, Active Message handlers can use a fresh register window if the eight input registers are not sufficient.

Unfortunately the floating-point register set is not windowed and the calling convention designates all FP registers as caller-saves. This means that any call to CMAM saves all FP registers in use. While this means that handlers can perform floating-point calculations freely, it would have been better if only the few handlers which actually do save the FP registers explicitly.

The use of registers in a typical scenario is illustrated in Figure 4-17. The communication substrate calls `CMAM_4` to send a request Active Message, passing the message itself in the `i0`/`o0` through `i5`/`o5` registers. `CMAM_4` bumps the window pointer and uses the local registers to hold pointers, masks, etc. In the example shown, the poll included in `CMAM_4` (as in all other message send functions) indicates that a message has arrived. The dispatch code loads the message into its `i0` through `o5` registers and calls the Active Message handler as a normal function which receives the message as arguments. The handler can acquire another register window for local variables but most handlers operate as leaf functions and only use the eight argument registers.

Due to the Sparc calling convention and the register windows the overhead of calling CMAM functions, rather than inlining the equivalent code, is rather small. The call linkage costs 3 cycles and the register window allocation costs 2 cycles (assuming no window overflow occurs). Compared to inlining, spilling a single register due to the increased register pressure costs 5 cycles as well.

Variable message length

As a consequence of the data transfer mechanism CMAM always sends full (5-word) messages even though the communication micro-architecture supports variable message sizes ranging from zero to five words. While sending shorter messages has the benefits of reducing the number of FIFO accesses—an attractive perspective given the cost of accessing the NI—and of using less network bandwidth, the ma-

[9] A substantial number of registers re required to hold pointers to the NI, status words, bit masks, and, last but not least, hold the message itself in order to push it into the FIFO.

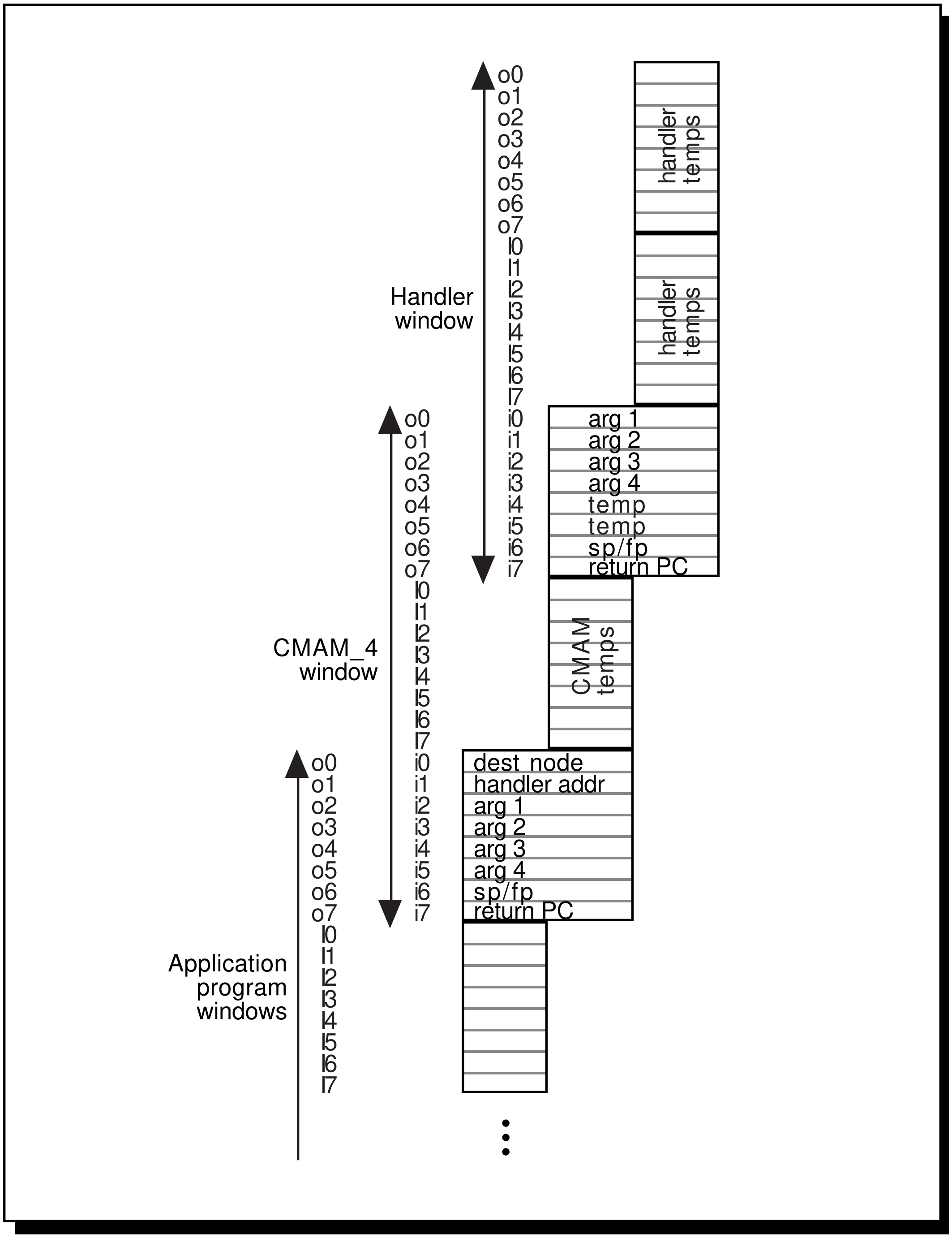


Figure 4-17: Usage of the Sparc register windows in CMAM.

In a typical scenario the application program calls `CMAM_4` to send a message, passing the message itself in registers. `CMAM_4` acquires a new register window for its temporary variables which include pointers to the NI and status register values and masks. In the case shown, `CMAM_4` polls the network, detects a message arrival and dispatches to the appropriate handler. The dispatch loads the message from the network interface its output registers such that the handler is called as a normal subroutine with the message as arguments. Small handlers typically work within the eight output registers (like any small leaf subroutine on the Sparc) but large handlers can acquire a full register window, as shown here.

jority of messages uses the full payload and supporting variable-size messages requires an additional dispatch at the receiving end.

The following table describes the trade-off by comparing the cost of fixed-size 5-word messages with variable length messages. The values take the cost of accessing the network interface in to account as well as the fact that checking the message length after reading the status register requires an additional 4 cycles:

message length (in words)	change in cost (in cycles relative to fixed-size 5-word message)			
	dispatch	stores	loads	total
5	+4	0	0	+4
4	+5	-1	-7	-3
3	+7	-4	-8	-5
2	+8	-5	-15	-12
1	+8	-8	-16	-16

Early experience with Split-C indicated that more than 75% of all messages use the full payload and consequently lead to the decision to only support full-size messages in CMAM. Note that by passing the messages in registers neither the sender nor the handler need to be aware of this fact: as long as the sender and the handler agree on the number of arguments the extra data in the message is of no importance.

4.3.2.5 Bulk data-transfer

The fact that the CM-5 hardware only supports small messages may lead to the conclusion that there is no particular benefit to sending large blocks of data in a single (logical) transfer; this is in contrast to the nCUBE/2 where the DMA hardware naturally seems to favor long messages. Just as short messages were demonstrated to perform well on the nCUBE/2, long transfers have advantages on the CM-5 because the payload of each (hardware) message can be used more efficiently. For this reason, using CMAM directly for bulk data transfer is not optimal. Instead of placing a handler address into the first word of each message it is more efficient to use it to encode transfer and message identifiers such that the four remaining words remain available for two double-words of data. With CMAM only three words of data could be transmitted per message.

To cater for bulk transfer CMAM introduces the concept of a communication segment. A communication segment is a memory region on the receiving node into which other nodes can transfer data. Each segment is set-up by specifying a base address, a byte count and an end-of-transfer handler function. The receiver can hand the segment identifier to any number of senders which then use a special protocol, called *xfer*, to transfer data into the segment. Each xfer message holds up to two double-words, stores the data into the segment at an offset indicated by the sender, and decrements the segment's byte count. When the count expires and end-of-transfer handler is called at the receiver.

While the xfer protocol appears as being distinct from the normal CMAM Active Messages, it really extends CMAM with a small set of hard-wired Active Message handlers to transfer data at the peak hardware bandwidth. Messages can be sent to these hard-wired handlers using a special set of message send functions. Thus, even though the bulk transfer extension is, strictly speaking, not part of CMAM it fits into the same Active Messages framework and could be integrated into CMAM if the hardware were improved to support longer messages (thereby reducing the overhead of dedicating the first word to the handler address).

The xfer protocol can be seen as being part of Active Messages at two levels. Each individual xfer message is an Active Message which happens to be sent to a hard-coded handler. In addition, an entire bulk transfer acts as a long Active Message sent to the end-of-transfer handler.

The xfer protocol achieves a peak data transfer rate of 10Mb/s while keeping the overhead low. It achieves that by decoupling the data transfer itself from the coordination and memory allocation. The former has to do with efficient communication which is appropriately addressed in the communication architecture, while the coordination and allocation are better left to higher software layers which can deal with these issues more efficiently.

4.3.2.6 Virtualizing the network

On the CM-5 virtualizing the network is handled by the hardware and the kernel. The CMAM communication architecture therefore does not deal with this issue.

4.3.3 Communication Architecture Interface

The combination of the small CM-5 hardware message size and the Sparc calling conventions allow a very simple interface to the CM-5 Active Messages (CMAM) layer: the message send functions and the handlers receive the message as arguments such that communication libraries based on Active Messages can written in C without loss of efficiency. This Subsection defines the details of the CMAM library as well as a sample implementation of the fetch&add example. It is intended to be a self-contained unit and therefore repeats some information presented above.

4.3.3.1 Overview

The CMAM layer constitutes a very thin veneer over the communication micro-architecture, providing the functionality of Active Messages while retaining most of the hardware characteristics. The core Active Message functionality is provided in the form of two message send functions (one for request messages, the other for reply messages) and a small number of explicit network polling functions. In addition to these, a set of pre-defined Active Message handlers use a specialized message format to implement the notion of a communication segment which supports bulk data transfers more efficiently than the core Active Message primitives.

Message format

The CMAM layer exposes the hardware message length limit of five 32-bit words and does not attempt to fragment larger messages. In the assumption that more than 75% of all messages use all four data words, no attempt is made at sending messages of less than five words. Thus, all CMAM messages carry exactly four words of data, the first word of each message being used for the handler address.

Message placement

Messages are composed and received in registers. The sender passes the outgoing message as arguments to the CMAM send functions which, due to the calling conventions, has the effect of composing the message in the register window output registers. On reception, the CMAM dispatcher passes the incoming message as arguments to the handler, again in registers.

Message send operations

Messages are sent using `CMAM_4` or `CMAM_reply_4` for request and reply messages, respectively.

Message receive operations

No particular receive operation is provided as the handlers are invoked automatically.

Message reception events

The current implementation does not use message arrival interrupts and instead polls the network. To simplify the usage, all message send functions poll automatically such that code sections which send messages do not need to poll explicitly. For computation-only sections a number of explicit polling functions are provided.

Send completion

To provide deadlock and livelock-free operation, CMAM uses the two CM-5 networks and limits the communication patterns to one-way and request-reply communication. When sending a request message, the CMAM layer repeatedly attempts to inject the message into the "request network"[10], handing incoming requests and replies until the outgoing message is sent. When sending a reply message, the message is injected into the reply network and only that network is serviced.

Synchronization

Due to the absence of message arrival interrupts critical sections can be formed trivially by not polling. Thus any code sequence without message sends and without explicit polls is atomic relative to handler execution. Handlers are atomic relative to other handlers, with the exception that reply handlers may be executed when a request handler attempts to send a reply message.

4.3.3.2 Interface definition

Send request and reply Active Messages

```
void CMAM_4(int node, void (*fun)(), ...);
void CMAM_reply_4(int node, void (*fun)(), ...);
```

`CMAM_4` sends an Active Message to the remote `node` where the message is handled by `fun`. While the message always carries 4 32-bit words of data, `CMAM_4` is declared as a *varargs* function and can be called with fewer arguments to transfer less data. Due to SPARC calling conventions, the arguments can be any combination of integer and floating-point values but not C structs.

`CMAM_4` sends a request Active Message and services both, the request and reply networks. `CMAM_reply_4` is identical to `CMAM_4`, but sends a reply message and services only the reply network.

Handler execution

```
typedef void CMAM_handler(int w1, int w2, int w3, int w4);

void CMAM_poll(void);
void CMAM_request_poll(void);
void CMAM_reply_poll(void);
void CMAM_wait(volatile int *flag, int value);
```

Handlers are executed during a send attempt or during an explicit poll. The handler named in the first message word is called with the four data words as arguments. Again, due to the calling conventions, the handler can be defined with any combination of integer and floating-point arguments.

A number of explicit poll functions allow servicing of the network during compute-only sections. 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., until `*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.) Other wait mechanisms may be implemented using `CMAM_poll` which polls both data networks. `CMAM_request_poll` and `CMAM_reply_poll` are similar, but poll only one network.

[10] The current implementation uses the left network for requests and the right network for replies.

Communication segments

```
typedef int CMAM_end_xfer (void *info, void *base);
int CMAM_open_segment (void *base_addr, int byte_count,
    CMAM_end_xfer *end_xfer_fun, void *info);
void CMAM_shorten_segment(int segment_id,
    unsigned int delta_count);
```

In addition to the above core Active Message primitives, CMAM provides a number of pre-defined handlers which support efficient bulk data transfer using an abstraction called communication segments. A communication segment represents a memory area on a given processor into which other processors can transfer data. When opening a segment (using `CMAM_open_segment`) the recipient specifies the segment base address, the number of data bytes to be received, and an end-of-transfer function. `CMAM_open_segment` returns a segment identifier or -1 if no segment is available. The end-of-transfer function is called when all the data has arrived and is passed the base address and an arbitrary `info` argument specified when opening the segment. The end-of-transfer function must return a new byte count or 0 if the segment is to be closed. Note that the segment identifiers returned by `CMAM_open_segment` encode the segment number and the alignment of the base address. Currently, there are 256 segments available and `CMAM_open_segment` always returns the highest free segment.

`CMAM_shorten_segment` reduces the remaining byte count of the specified segment. This is useful, for example, when the sender cannot provide as much data as the recipient asked for.

Transfer data into a remote segment

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

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

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

A set of special `CMAM_xfer` functions must be used to send data to the handlers implementing communication segments. `CMAM_xfer_4i` transfers 16 bytes of data to a segment on the remote node. 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 a pre-defined handler stores the data in memory and decrements the byte count associated with the segment. If the count becomes zero, the end-of-transfer function is called as described in `CMAM_open_segment`.

`CMAM_xfer` is similar but transfers the block of memory at address `buff` and of length `byte_count` to a segment on the remote `node`.

`CMAM_xfer_4i` and `CMAM_xfer` send packets on the request data network and poll both, the request and reply networks. `CMAM_reply_xfer_4i` and `CMAM_reply_xfer` are similar but send packets on the reply data network and poll only that network.

4.3.3.3 Example of use: Fetch&add

As a simple example for the use of CMAM Figure 4-6 shows an implementation of the fetch&add introduced in Subsection 4.1.2. As an additional twist the implementation shown provides a split-phase

```
# Synchronization variable
typedef struct { volatile int flag, value; } fetch_add_sync;

# Initiate fetch&add request
void fetch_add_initiate(int node, int *addr, int incr,
                        fetch_add_sync *sync)
{
  sync->flag = 0;
  CMAM_4(node, fetch_add_h, MYPROC, sync, addr, incr);
}

# Wait for fetch&add reply and return result
int fetch_add_complete(fetch_add_sync *sync)
{
  CMAM_wait(&sync->flag, 1);
  return sync->value;
}

# All-in-one blocking fetch&add
int fetch_add(int node, int *addr, int incr)
{
  fetch_add_sync sync;
  fetch_add_initiate(node, addr, incr, &sync);
  return fetch_add_complete(&sync);
}

# Handler for fetch&add request
void fetch_add_h(int ret_node, void *sync, int *addr, int incr)
{
  int value = *addr + incr;
  *addr = value;
  CMAM_reply_4(ret_node, fetch_add_rh, sync, value);
}

# Handler for fetch&add reply
void fetch_add_rh(fetch_add_sync *sync, int value)
{
  sync->value = value;
  sync->flag = 1;
}
```

Figure 4-18: Fetch&add implementation with CM-5 Active Messages.

version of fetch&add as well, allowing the processor initiating the fetch&add to continue computing while communication is in progress. In order to use this split-phase version the caller of fetch&add must allocate a synchronization variable which is used in matching the reply with the fetch&add completion check.

The `fetch_add_initiate` function initializes the synchronization variable, starts the fetch&add by sending a request message to the remote node, and immediately returns. A call to `fetch_add_complete` then waits for the reply message to arrive and returns the result.

The `fetch_add` function combines the individual parts to implement the original blocking version. In this case the synchronization variable is simply allocated on the stack. To actually take advantage of the split-phase version the synchronization variables have to be allocated on a per-thread basis.

4.3.4 Micro-benchmarks

The micro-benchmarks measure the various CMAM operations in isolation to determine best-case timings. In order to be able to distinguish the sending overhead, the handling overhead, and the network latency each CMAM operation is measured in several phases:

1. one node sends a stream of messages alternating between two destination nodes to ensure that the sender is the bottleneck,
2. two nodes send a stream of messages to one node such that the receiver is the limiting factor, and
3. two nodes ping-pong a message back and forth to measure the round-trip latency.

As can be seen in Table 4-7 an Active Message costs well under 2µs on each end! The overhead of sending a request message is slightly higher than that of sending a reply due to the fact that both networks have to be polled. Handling a message is slightly more expensive than sending it—the handler used in this example is minimal and only increments a counter in memory.

The costs for the xfer protocol show more variation. Sending a block of memory using `CMAM_xfer_4` achieves 7Mb/s while using `CMAM_xfer_reply_4` peaks at over 9Mb/s. The difference is again due to the additional poll when sending request messages. Sending the same memory block using `CMAM_xfer` improves the performance due to a tighter loop. The handler for xfer messages operates in two regimes: for messages arriving back-to-back the segment descriptor is kept in registers allowing to handle the messages quickly if they are destined for the same segment. If successive messages are for different segments, the respective descriptors have to be fetched from memory for each message. Note that the xfer benchmarks use buffers smaller than the 64 Kb cache. Sending from out-of-cache buffers takes a 10 cycle hit per message, but receiving is not affected given that the cache is in write-through mode and the writes are absorbed by the write-buffer.

The table also shows the cost of explicit polls: a normal explicit poll takes 0.7µs while polling only the reply network (in the unlikely event that a handler requires an explicit poll) costs about half of that.

operation	overhead (per message)		bandwidth (16bytes/msg)
Send request	1.5µs	50.7cyc	10.6Mb/s
Send reply	1.3µs	42.8cyc	12.3Mb/s
Handle request or reply	1.6µs	52.2cyc	10.0Mb/s
Open/close segment	6.7µs	221.7cyc	n/a
Xfer_4 request	2.3µs	75.4cyc	7.0Mb/s
Xfer_4 reply	1.6µs	54.1cyc	9.8Mb/s
Xfer_N request	1.7µs	56.7cyc	9.3Mb/s
Xfer_N reply	1.6µs	52.9cyc	10.0Mb/s
Handle xfer for 1 segment	1.6µs	52.6cyc	10.0Mb/s
Handle xfer for 2 segments	2.2µs	71.1cyc	7.4Mb/s
Explicit poll request&reply	0.7µs	22.0cyc	n/a
Explicit poll reply	0.3µs	11.0cyc	n/a
Round-trip to neighbor	11.2µs	368.1cyc	n/a
Latency per hop	0.8µs	8.0cyc	n/a

Table 4-7. Micro-benchmark timings for CM-5 Active Messages.

The round-trip time to a neighbor node is measured by sending a request message to a handler which immediately sends a reply back. The total time can be estimated to be the sum of a request and a reply Active Message overhead (6µs) and two message injection and routing times (2.8µs). The actual time measured is 2.4µs higher which is probably due to a combination of additional latencies in the NI itself and of not hitting the poll loop optimally.

The performance of the fetch&add example is shown in Table 4-8 and presents no surprises: all overheads are slightly higher than in the previous benchmark due to the extra memory accesses required to perform the fetch&adds.

4.3.4.1 Performance modeling

The timings gathered with the micro-benchmarks can be expressed more conveniently using the performance models introduced in Section 2.3. Converting µs to clock cycles yields around 100 CPM (clocks per message) for CMAM messages. It is difficult to express the communication performance for long messages in terms of start-up cost and per-byte cost. Using the xfer bulk data transfer protocol, the minimal start-up cost is opening a segment and passing the segment id to the sender. This results in an α of 15.5µs and a per byte cost varying from a β of 0.1µs to 0.14µs depending on which xfer function is used. Expressed as throughput, R_∞ is 10Mb/s and half this throughput is achieved with a message length $N_{1/2}$ of 155 bytes. Table 4-2 expresses the performance of core Active Messages in terms of the LogP model.

4.3.5 CMAM Implementation

The results demonstrated in the micro-benchmarks are impressive: for short messages CMAM improves the overhead of approximately 80µs of send&receive by more than an order of magnitude. Instead of taking several thousands of cycles to send and handle a message CMAM takes around a hundred. But even so, why does it still take fifty cycles to push a six-word message into the network interface FIFO and check a status register? And a similar time to check the receive status, pull five words out of the FIFO, and perform an indirect call through the first word? Part of the answer is that each load and store accessing the NI takes 7 to 8 cycles. This subsection examines the CMAM send and handle code sequences carefully to determine where all the cycles are spent and how the CM-5 communication micro-architecture could be modified to better support communication architectures such as Active Messages.

Fetch&add	cost	
Requesting node overhead	4.3µs	143cyc
Service node overhead	3.5µs	115cyc
Round-trip time (neighbor)	12.2µs	404cyc

Table 4-8. Fetch&add benchmark results for CM-5 Active Messages.

Short 4-word (16-byte) messages		
L	= 6µs	Injection + 8.3 hops = 1.7µs+6.6µs (8.3 hops is avg. distance for P=1024)
o	= 1.6µs	(Send overhead + Recv overhead) / 2 = (1.5µs+1.6µs)/2
g	= 4µs	The network bisection bandwidth is 5Mb/s at 20 bytes of payload per message.
P	= 2..1024	

Table 4-9. LogP parameters for CM-5 Active Messages

4.3.5.1 Message send

The instruction sequence for sending a request Active Message is shown in Figure 4-19. The first four instructions acquire a register window and set-up pointers to the network interface. The address for pushing the first message word into the FIFO encodes the message tag and length and is loaded from a variable. This allows the tag used by CMAM to be chosen at run-time in order not to conflict with other message libraries (such as CMMD). Instructions 6 and 7 correct for a chip bug related to the high-order address bits. After this set-up which takes 7 cycles comes the loop attempting to inject the message and receive incoming messages.

Figure 4-20 shows the cycle-by-cycle timing of a message send (assuming the injection succeeds and no messages arrive). The three double-word stores and the status register loads are ordered to avoid stalls (refer to Figure 3-8 for the timings of individual loads and stores). The three branches jump to instruction sequences (not shown) which receive messages and/or retry the send, as appropriate.

The timing diagram shows that most of the cycles are spent in the load and store instructions. While the cost of checking the request network receiver is low (2 cycles), the 9 cycles it takes to check the reply network are noticeable.

The total number of 44 cycles for a send matches well with the 50 cycles measured in the micro-benchmark given that the latter includes the loop calling `CMAM_4` repetitively.

4.3.5.2 Network poll

Most of the time message arrival is detected by the automatic polls in the message send functions. During compute-only sections or when a processor busy-waits for a reply explicit polls must be inserted.

```
 # CMAM_4 — Send request Active Message, service both networks
 # void CMAM_4(int node, void (*fun)(), ...);
 1: _CMAM_4: Set-up registers
 2:        save   %sp,-SA(MINFRAME),%sp            get register window
 3:        sethi  %hi(_CMAM_NI_first),Rni1st       load addr for first msg word
 4:        ld     [Rni1st+%lo(_CMAM_NI_first)],Rni1st
 5:        set    NI_BASE,Rnibase                  get base address of NI chip
 6:        sethi  %hi(~0x8007ffff),Rtmp            mask destinations (NI bug)
 7:        andn   Rnode,Rtmp,Rnode
 #         Loop: send, check right, check left
 8: s1:    ld     [Rnibase+NI_RDR_STATUS_O],Rstat2 load reply net status
 9:        std    Rnode,[Rni1st]                   push message
10:        std    Rout1,[Rnibase+NI_LDR_SEND_O]
11:        std    Rout3,[Rnibase+NI_LDR_SEND_O]
12:        andcc  Rstat2,1<<NI_REC_OK_P,%g0        reply message arrived?
13:        bne    s5                               ...yes (code not shown)
14:        ld     [Rnibase+NI_LDR_STATUS_O],Rstat  load request net status
15:        andcc  Rstat,1<<NI_SEND_OK_P,%g0        message sent?
16:        be     s4                               ...nope (code not shown)
17:        andcc  Rstat,1<<NI_REC_OK_P,%g0         request message arrived?
18:        bne,a  s3                               ...yup (code not shown)
19:        sethi  %hi(_CMAM_left_htab),Rjtab       start of dispatch code
20: s2:    ret                                     done
21:        restore                                 pop register window
```

Figure 4-19: Send CM-5 Active Message (`CMAM_4`) implementation.

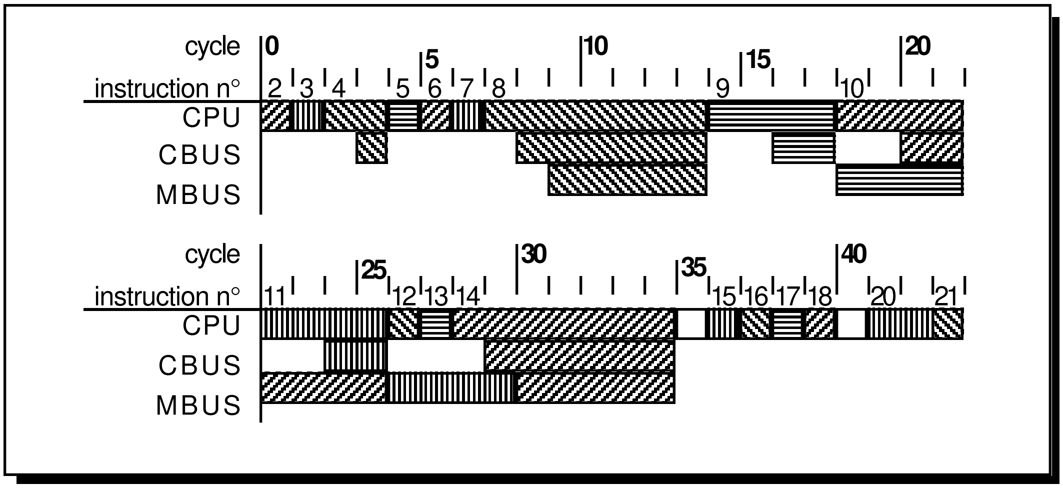


Figure 4-20: Send CM-5 Active Message (CMAM_4) timing.

Sending an Active Message on the request network takes a total of 44 clock cycles. The timing diagram shows during which clock cycles each instruction occupies the Sparc execute pipeline stage (CPU), the cache bus (CBUS), and the memory bus (MBUS). The instruction numbers refer to the lines in Figure 4-19. Note that during cycle 35 a load interlock stalls the processor and an annulled instruction executes during cycle 40.

```
# CMAM_poll - poll network once and handle all waiting messages
# Note: the original is written as a GCC inline function, shown here is the code generated by GCC.
_CMAM_poll:
        set    NI_BASE,%o0                         get base address of NI chip
        ld     [%o0+NI_LDR_STATUS_O],%o1           load request net status
        andcc  %o1,1<<NI_REC_OK_P,%g0              request message arrived?
        be,a   p1                                  ...nope
        ld     [%o0+NI_RDR_STATUS_O],%o1           load request net status
        call   _CMAM_got_left                      handle message
        nop
        ld     [%o0+NI_RDR_STATUS_O],%o1           load request net status
p1:     andcc  %o1,1<<NI_REC_OK_P,%g0              request message arrived?
        be,a   p2                                  ...nope
        nop
        call   _CMAM_got_right                     handle message
        nop
p2:     ret
        nop
```

Figure 4-21: Poll CM-5 network for Active Messages.

The code for an explicit poll is written in C as a GCC in-line function. To simplify the accounting of clock cycles Figure 4-21 shows the code generated by the compiler.

The code for polling both networks is straightforward: read request status register, check and call tag dispatch if a message has arrived, and repeat with the right network. Due to the nested conditionals most of the delay slots can unfortunately not be filled. The total cost for an unsuccessful poll is 22 cycles of which the two loads alone take 16 (note that there is an interlock pipeline stall after each load). Polling only the reply network requires about half the code and half the time.

```
 # CMAM_got_left — poll indicates message arrival, now dispatch on message tag
 # void CMAM_got_left(int *ni, int status);
 # Note: CMAM_got_right is similar, but with different names
 1: _CMAM_got_left: dispatch to tag-handler
 2:        save   %sp,-SA(MINFRAME),%sp              acquire register window
 3:        set    _CMAM_left_htab,Rjtab              base of handler table
 4:        srl    %i1,NI_DR_REC_TAG_P-TAB_SHIFT,%i1; extract tag bit-field
 5: g1:    and    %i1,0xf<<TAB_SHIFT,%i1;
 6:        jmpl   Rjtab+%i1,%o7;                     dispatch to tag-handler
 7:        mov    %i0,Rnibase;                       pass pointer to NI
 8:        ld     [%i0+NI_LDR_STATUS_O],%i1;         get status again
 9:        andcc  %i1,1<<NI_REC_OK_P,%g0;            check recv bit
10:        bne,a  g1;                                dispatch again
11:        srl    %i1,NI_DR_REC_TAG_P-TAB_SHIFT,%i1
12:        ret                                       done
13:        restore
```

Figure 4-22: Dispatch on message tag for CM-5 Active Messages.

4.3.5.3 Tag dispatch

Once a poll has detected the arrival of a message the message tag must be checked. The message tag consists of four bits which can be used to differentiate among several communication protocols. In addition, the kernel reserves message tags for its use and controls which message tags generate interrupts and which do not. The current kernel reserves eight tags for its use. CMAM uses two tags to differentiate Active Messages and xfer messages.

The use of the tags might seem peculiar given that the Active Message mechanism is exactly designed to avoid the use of a limited idiosyncratic message type selector. In fact, the alternative of not using tags and saving the tag-check and dispatch cycles is very appealing. This is unfortunately not feasible. Due to the latency of interrupts from the NI, it is possible that a user-level poll detects the arrival of an interrupting kernel message before the Sparc processor recognizes the interrupt input. By the time the user process attempts to read the message, the kernel has already retrieved it and the reads of the empty FIFO cause an error. Thus the user process *must* check the tag and ignore messages with kernel tags and using the tags productively incurs no additional overhead.

The tag dispatch code shown in Figure 4-22 is called from the poll and is passed the status register. The dispatch is done by jumping into a table which contains the tag-handlers (the tag-specific code). Each entry in this table has space for 64 instructions and each tag-handler is copied into the table at initialization time. The first instruction in Figure 4-22 acquires a new register window and is not executed for polls embedded in the message send functions. Instructions 3 to 7 shift the tag into the appropriate position and jump into the tag-handler table. The remaining instructions are executed after the message has been handled and perform another poll such that all pending messages are handled one after another.

The cost of the tag dispatch itself is 10 cycles plus 11 cycles for the integrated poll.

4.3.5.4 Active Message handler dispatch

In the case of an Active Message, the tag-dispatch mechanism jumps to the tag-handler shown in Figure 4-23 which loads the message into the in/out registers and dispatches to the actual Active Message handler. Note that this dispatch is a tail-call which causes the handler to return directly into the tag-dispatch code.

```
# CMAM_handle_req — Dispatch to Active Message request handler
_CMAM_handle_req:
        ld      [Rnibase+NI_LDR_RECV_O],Rtmp       load function ptr
        ldd     [Rnibase+NI_LDR_RECV_O],%o0        load message data
        jmp     Rtmp                               tail-call to handler
        ldd     [Rnibase+NI_LDR_RECV_O],%o2        load message data
```

Figure 4-23: Dispatch to CM-5 Active Message handler.

The Active Message dispatch is expensive due to the cost of the loads from the network interface. The four instructions take a total of 25 cycles!

An estimate for the total overhead of handling an Active Message can be obtained by summing up the costs of a poll, a tag-dispatch, and a handler dispatch. However, the result of 69 cycles is higher than the value measured by the micro-benchmark (52 cycles) which can be explained by the fact that the poll integrated in the tag-dispatch picks-up the back-to-back messages that were used in the benchmark. Taking this into account reduces the estimate to 47 cycles which leaves a few cycles for the handler used in the benchmark to increment a memory location. Thus, the handling of sporadic messages costs a little over 2µs whereas the handling of back-to-back messages is cheaper.

4.3.5.5 Data transfers using xfer

The xfer protocol uses a separate tag from Active Messages and the messages are consumed by the tag-handler itself. While at first sight this seems to save only the two cycles of the `jmp` instruction in the Active Message tag-handler, the real savings come from the fact that the first word of the message can be used to encode the segment id and the offset for the data. In addition, the xfer tag-handler “cheats” by polling the network to handle additional xfer messages immediately. For back-to-back messages into the same segment the segment descriptor can be kept in registers to arrive at the data transfer rate of 10Mb/s (counting only the 16 bytes of payload per message) shown in Table 4-7. Without these tricks the peak transfer rate would be 8.7Mb/s.

4.3.6 FFT macro-benchmark

The fast Fourier transform benchmark demonstrates that the performance characteristics determined by the micro-benchmarks are applicable in an (albeit small) application program. The benchmark is taken from the work on the LogP parallel model of execution [CKP+93] in which the LogP model is used to estimate and analyze the program behavior.

The benchmark uses the “butterfly”' algorithm [CT65] for the discrete FFT problem, most easily described in terms of its computation graph. The n-input (n being a power of 2) butterfly is a directed acyclic graph with $n\log(n+1)$ nodes viewed as n rows of $\log(n+1)$ columns each. For $0 \le r < n$ and $0 \le c < \log n$, the node (r, c) has directed edges to nodes $(r, c+1)$ and $(\bar{r}_c, c+1)$ where $\bar{r}_c$ is obtained by complementing the $(c+1)$-th most significant bit in the binary representation of r. Figure 4-24 shows a 16-input butterfly.

The nodes in column 0 are the problem inputs and those in column $\log n$ represent the outputs of the computation. (The outputs are in bit-reverse order, so for some applications an additional rearrangement step is required.) Each non-input node represents a complex operation.

In order to achieve good performance the FFT algorithm must take particular care in mapping the butterfly structure onto the CM-5. While one might be lead to investigate mappings of butterfly networks onto fat trees the approach advocated by the LogP model and followed here is to ignore the structure of the network. One simple justification is that n data inputs and the $n\log n$ computation nodes must be

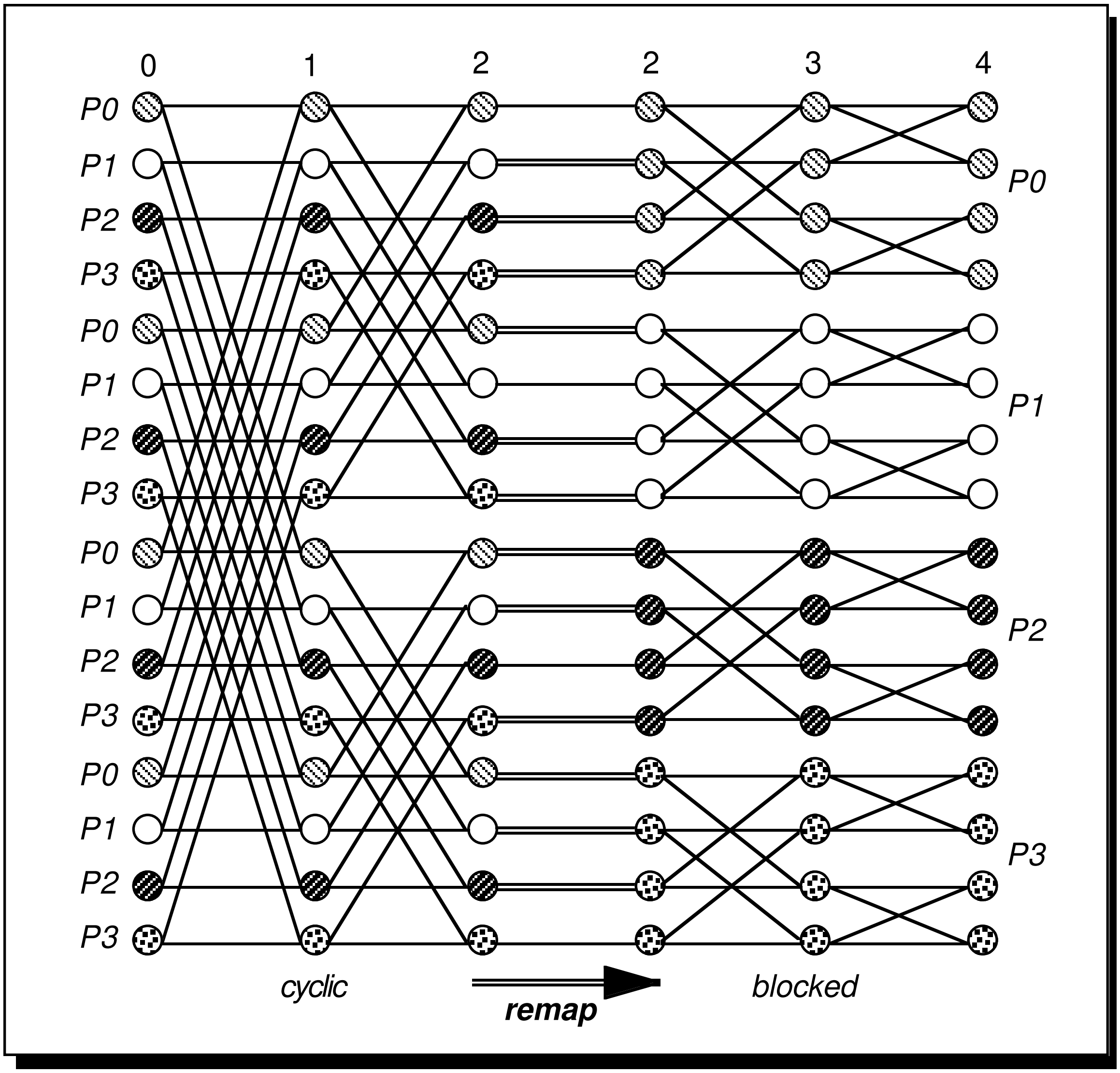


Figure 4-24: An 16-input butterfly laid-out on four processors

results in mostly all-to-all communication patterns instead of butterflies as the algorithm might at first suggest.

The layout chosen in this benchmark is a combination of a *blocked* layout and a *cyclic* layout. The cyclic layout assigns the first row of the butterfly to the first processor, the second row to the second processor and so on. Under this layout, the first $\log(n/p)$ columns of computation require only local data, whereas the last $\log p$ columns require a remote reference for each node. An alternative blocked layout places the first n/P rows on the first processor, the next n/P rows on the second processor, and so on. This means that each of the nodes in the first $\log p$ columns requires a remote datum for its computation, while the last $\log(n/p)$ columns require only local data.

The *hybrid* layout chosen for the benchmark combines the best of the two above layouts: the initial computation of the first $\log P$ columns uses a cyclic layout and the final computation of the last $\log P$ columns uses the blocked layout. The only communication required is remapping from cyclic layout to blocked layout which can occur at any column between the $\log P$-th and the $\log(n/P)$-th (assuming that $n > P^2$). Thus the algorithm has a single "all-to-all" communication step between two entirely local computation phases.

Timing estimate

The model predicts that the computational time for the hybrid layout, assuming that a butterfly computation takes unit time, is $(n/p)\log n$ and that the communication phase in which each processor sends n/P^2 messages to every other requires $g(n/P - n/P^2) + L$ time. Achieving this communication time, however, requires a well-crafted balanced communication schedule. A naive schedule would have each processor send data starting with its first row and ending with its last row. Notice, that all processors first send data to processor 0, then all to processor 1, and so on. All but L/g processors will stall on the first send and then one will send to processor 0 every g cycles. A better schedule is obtained by staggering the starting rows such that no contention occurs: processor i starts with its $(in)/P^2$ -th row, proceeds to the last row, and wraps around.

Measurements

The benchmark implementation[11] measures the performance of the three phases of the algorithm: (I) computation with cyclic layout, (II) data remapping, and (III) computation with blocked layout. Figure 4-25 demonstrates the importance of the communication schedule: the three curves show the computation time and the communication times for the two communication schedules. With the naive schedule, the remap takes more than 1.5 times as long as the computation, whereas with staggering it takes only $1/7$ th as long.

The two computation phases involve purely local operations and are standard FFTs. Figure 4-26 shows the computation rate over a range of FFT sizes expressed in Mflops/processor. For comparison, a CM-

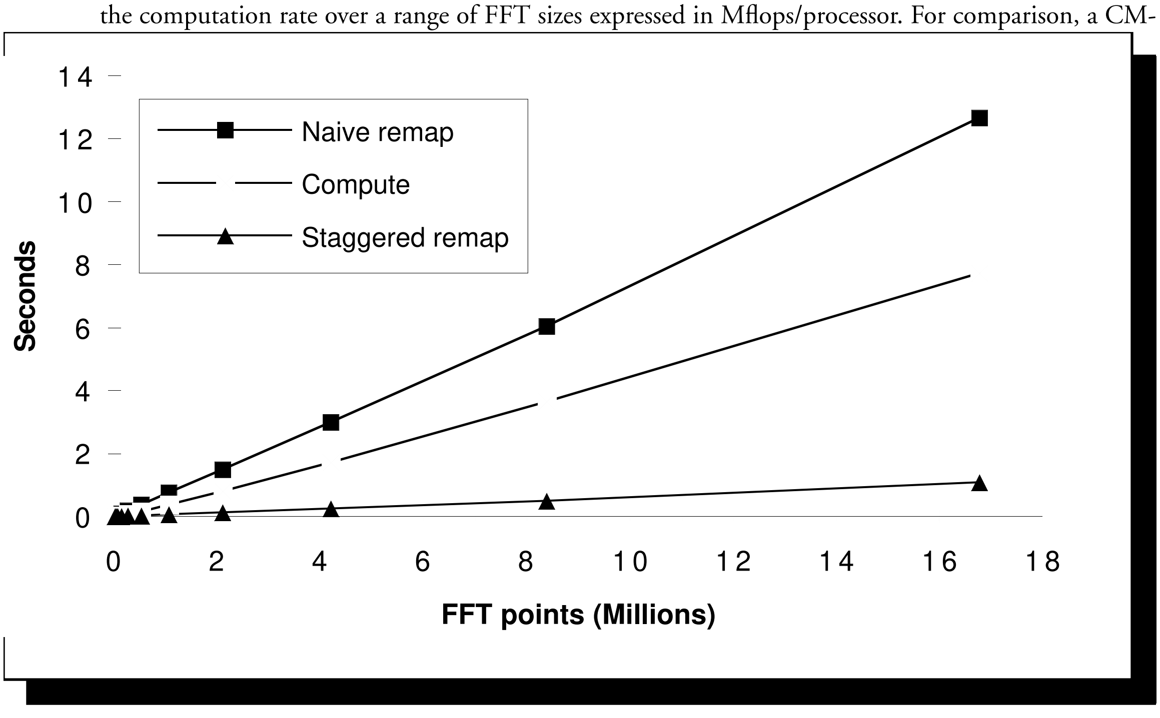


Figure 4-25: Execution times for FFTs of various sizes on a 128 processor CM-5.

The *compute* curve represents the time spent computing locally. The *bad remap* curve shows the time spent remapping the data from a cyclic layout to a blocked layout if a naive communication schedule is used. The*good remap* curve shows the time for the same remapping, but with a contention-free communication schedule, which is an order of magnitude faster. The X axis scale refers to the entire FFT size.

[11] Note that the implementation does not use the vector accelerators.

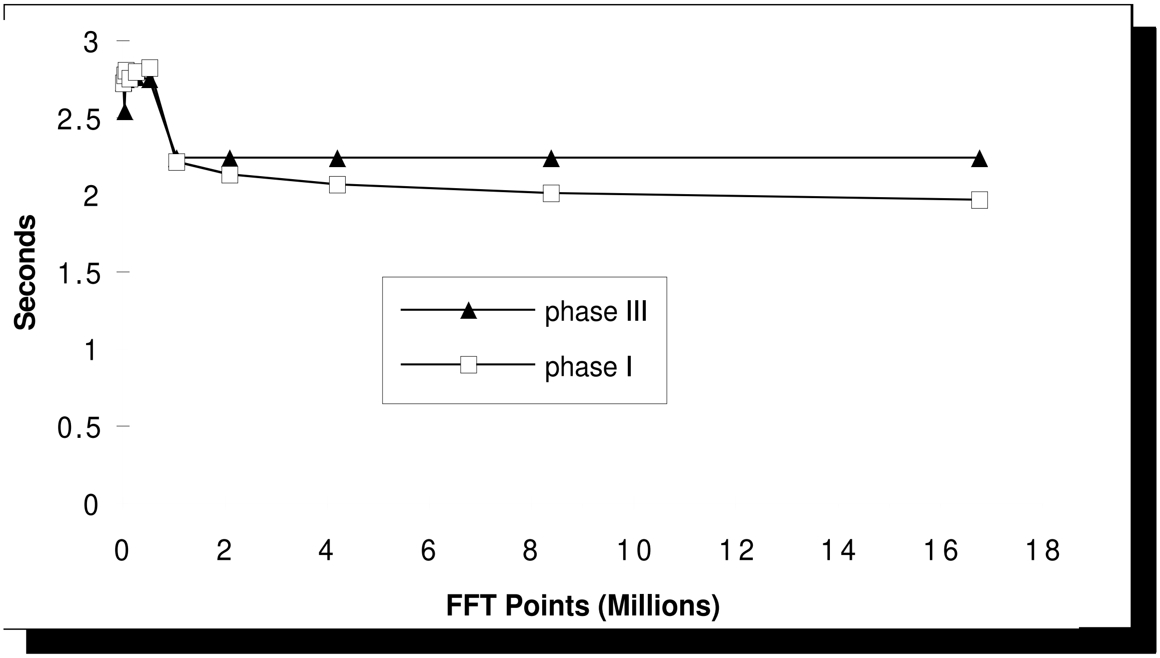


Figure 4-26: FFT computation rates.

Per processor computation rates for the two computation phases of the FFT in Mflops (millions of floating-point operations per second).

5's Sparc node achieves roughly 3.2 Mflops on the Linpack benchmark. This example provides a convenient comparison of the relative importance of cache effects and communication performance. The drop in performance for the local FFT from 2.8 Mflops to 2.2 Mflops occurs when the size of the local FFTs exceeds cache capacity. (For large FFTs, the cyclic phase involving one large FFT suffers more cache interference than the blocked phase which solves many small FFTs.) The implementation could be refined to reduce the cache effects, but the improvement would be small compared to the speedup associated with improving the communication schedule.

Quantitative analysis

Using Figure 4-26 the computational performance can be calibrated with respect to the LogP model: at an average of 2.2 Mflops and 10 floating-point operations per butterfly each butterfly operation corresponds to 4.5µs and § 4.3.4.1 determined LogP parameters of L=6µs, o=1.6µs, and g=4µs. In addition to these costs, there is roughly 1µs of local computation per data point to load/store values to/from memory. Analysis of the staggered remap phase predicts the communication time is $(n/P)\,\max(1\,\mu s + 2o, g) + L$. For these parameter values, the transmission rate is limited by processing time and communication overhead, rather than bandwidth. The remap phase is predicted to increase rapidly to an asymptotic rate of 3.2 MB/s. The observed performance is roughly 2 MB/s for this phase, nearly half of the available network bandwidth.

The analysis does not predict the gradual performance drop for large FFTs. In reality, processors execute asynchronously due to cache effects, network collisions, etc. It appears that they gradually drift out of sync during the remap phase, disturbing the communication schedule. This effect can be reduced by adding a barrier synchronizing all processors after every n/P^2 messages. Figure 4-27 shows that this eliminates the performance drop.

The effect of reducing g can be tested by improving the implementation to use both fat-tree networks, i.e., using both request and reply xfers to send the data, thereby doubling the available network bandwidth. The result shown in Figure 4-27 is that the performance increases by only 15% because the network interface overhead (o) and the loop processing dominate.

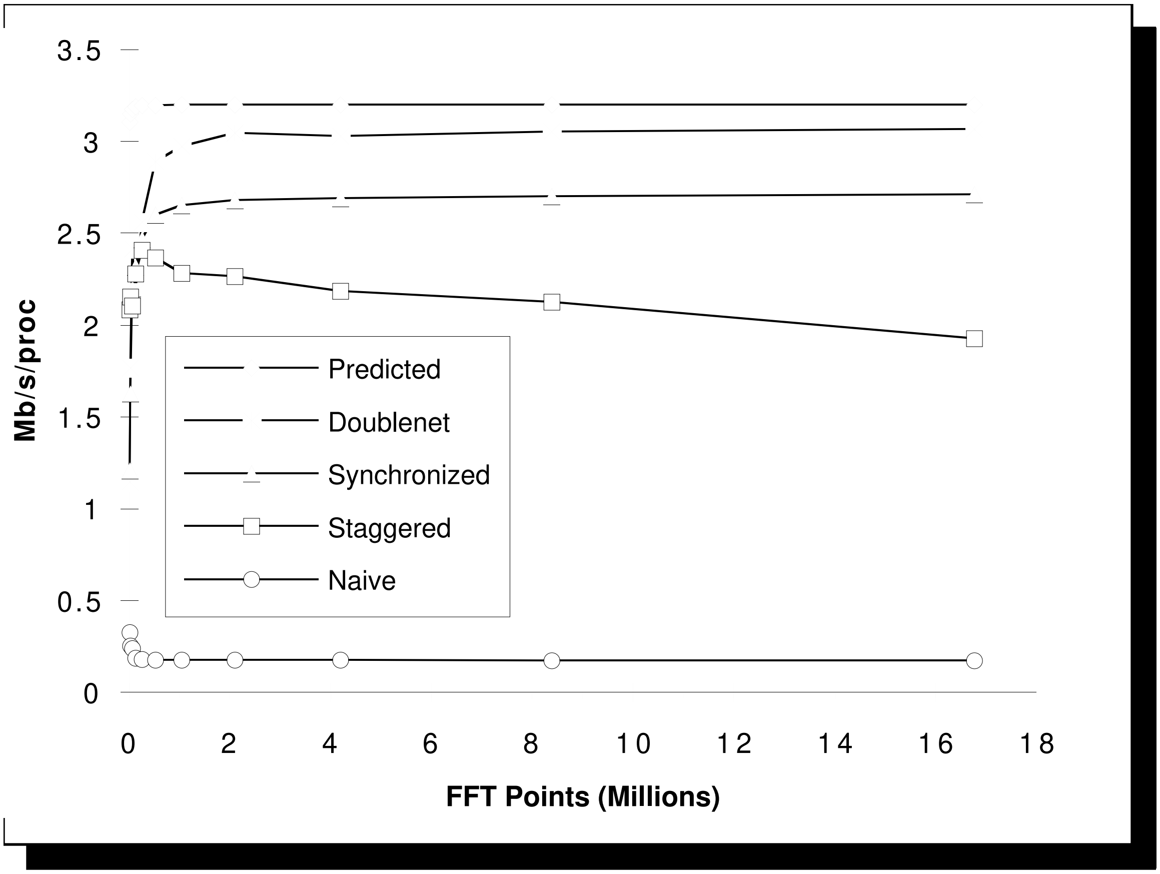


Figure 4-27: FFT predicted and measured communication rates.

Expressed in Mbytes/second per processor for the staggered communication schedule.

The staggered schedule is theoretically contention-free, but the asynchronous execution of the processors causes some contention in practice. The *synchronized* schedule performs a barrier synchronization periodically (using a special hardware barrier). The *double net* schedule uses both data networks, doubling the available network bandwidth.}

Summary

The communication performance achieved with the FFT demonstrates the efficiency of Active Messages, and, in particular, that the performance parameters established through micro-benchmarks translate well into an application setting.

In addition to efficiency, the versatility of Active Messages is an essential assumption underlying the algorithm design. In effect, the algorithm design is only concerned with the actual information transfer and an not at all with the communication mechanisms. The fact that the final algorithm can be mapped efficiently onto Active Messages is taken for granted.

4.3.7 Pros and cons of asynchronous handler execution

One of the major debatable CMAM design decisions is not to use interrupts. While having to poll the network periodically seems rather cumbersome at first, the fact that all message sending functions include automatic polls makes them invisible in most applications. Problems do, however, occur during compute-only phases. One typical example is an all-to-all communication pattern in which each processor fetches the remote data it needs for the following compute phase. If processors enter the compute phase as soon as they have gathered all their data then they stop servicing the requests of other processors which lag behind. The typical solution is to slow down the fast processors either by synchronizing all processors at the end of the communication phase using a barrier or by having each processor wait until it has serviced the appropriate number of requests. An alternative would is to add explicit polls into the computation phase such that processors service requests at all times, but this is often not very compatible with software engineering as it requires two versions of computational kernels such as matrix multiply: one with polling and one without.

Besides these drawbacks, synchronous handler execution has many benefits. One is that a few polls are cheaper than an interrupt. The more significant advantage, however, is that atomic operations and critical sections in the computation are free. For example, Split-C's `get` operation (described in Section 6.1 and used in the nCUBE/2 matrix multiply example in § 4.2.6.1) becomes more complicated to implement. The difficulty is maintaining the counter of outstanding requests: when issuing a `get` the computation increments this counter and the reply handler decrements it. This means that the counter increment must be made atomic relative to handler execution. In this case, and in most others, bracketing the short load-add-store sequence by kernel traps to disable and enable interrupts is a steep price to pay! Fortunately, changing the representation of the variables to be updated atomically can sometimes help. In the case of `get`, using a counter of issued requests and a separate counter of received replies solves the problem. Testing for the completion of all outstanding operations requires subtracting the reply counter from the issue counter (using appropriate modulo arithmetic) and comparing the result against zero.

This simple example illustrates the trade-off: synchronous message reception is cheaper and simplifies writing handlers but asynchronous handler execution avoids having to insert explicit polls or to synchronize processors before entering a compute-only phase. Given that handlers are in general designed by the language implementor while compute phases are written by application programmers, software engineering issues alone suggest that the cost of asynchronous handler execution, at least as an option, is worthwhile.

Asynchronous handler execution in CMMD 3.0

Thinking Machines' message passing library, CMMD 3.0, uses Active Messages as the foundation and supports synchronous as well as asynchronous Active Messages handler execution. Interrupts are dispatched into the kernel which checks the message tag to determine whether it is a kernel I/O message or whether it is an Active Message. In the latter case, the kernel returns to a user-level stub which reads the first word of the message and dispatches the appropriate handler. When the handler returns, the stub checks for the arrival of further messages and traps back into the kernel if none are present.

Unfortunately on the Sparc it is not possible to return from the Active Message handler directly to the computation. The last instruction of the handler would have to simulate a return from trap instruction which is not possible. The problem on the Sparc is that the last two instruction must restore the PC, the nPC (next PC), and restore the previous register window. In addition, on the CM-5, interrupt would have to be re-enabled. As consequence, each message interrupt requires two traps into the kernel.

In order to form critical sections, CMMD 3.0 provides a flag in memory which can be set to disable interrupts. The flag is checked by the user-level interrupt stub and, if an interrupt occurs in a critical section, causes the stub to trap back into the kernel and disable interrupts instead of dispatching the handler. This mechanism could be improved substantially if the kernel cooperated with the user-process in a manner similar to the nCUBE/2 Active Messages implementation.

The cost of an interrupt is approx. 9.5μs, but the fact that a sequence of messages arriving back-to-back can be handled within a single interrupt amortizes the cost of interrupts. To help estimating the per-message cost of interrupts it is useful to classify program behavior into three categories:

1. infrequent communication: the cost of interrupts does not affect overall performance significantly and the software engineering value of asynchronous message reception is high,
2. frequent communication: most messages arrive back-to-back and are received without interrupts, the cost of an interrupt on the first message of a series can be amortized, and

3. moderate communication: each message causes an interrupt and communication is frequent enough to affect overall performance, switching to synchronous message reception during such phases may be a solution.

Fast path for user-level interrupts

The current version of the kernel running on the processing nodes does not particularly optimize the dispatch of message reception interrupts to the user process. The interrupt vector used for message arrival is shared with multiple other network interface interrupt sources. Thus, the interrupt handler must first read several NI registers to determine that a message has arrived on the data network. It then must examine the message tag to separate user-level messages from kernel-kernel messages and from kernel-I/O subsystem messages. The current interrupt dispatch is not optimized for user messages and, in addition, information such as the message tag is not passed to user-level where it must be read again from the network interface. Experiments with a modified version of the kernel have achieved a reduction of the interrupt overhead down to approximately 8μs.

The most interesting aspect of the kernel interrupt handler is hidden in the return-from-interrupt system call. After handling the interrupt, the user-level interrupt routine must trap back to the kernel in order to re-enable interrupts and resume the interrupted computation. The interesting detail here is that an apparently benign characteristic of the network interface costs a significant number of cycles: the network interface interrupt signal is edge triggered and not level sensitive (as in most I/O devices). This means that if an interrupting event (such as a message arrival) occurs while interrupts are disabled this event goes by unrecorded and when interrupts are re-enabled again no interrupt is signalled (if interrupts are level sensitive, such an event would cause the interrupt signal to be asserted but prevented to reach the processor until the interrupt mask is changed). As a result, the kernel must check explicitly for the presence of another message after re-enabling interrupts.

The bottom line is that careful attention to minute details in the kernel interrupt handler path and in the network interface interrupt logic could reduce the interrupt overhead significantly.

Cheap critical sections

Another possible kernel enhancement could provide cheap critical sections. The cost of critical sections is often overlooked when the cost of user-level interrupts is evaluated. Yet, the cost of critical sections is often more important that the overhead of interrupts! In cases where a data structure (such as a scheduling data structure) is accessed frequently by local computation and less frequently by interrupt handlers the cost of forming a critical section around each local access can quickly dominate the cost of the less frequent interrupts. (An example of this situation is described in Section 6.2.)

The idea underlying cheap critical sections is to move CMMD's flag indicating the execution of a critical section from memory into a global register. The Sparc ABI (Application Binary Interface) specifies that register `g7` (together with `g5` and `g6`) is reserved for the "execution environment". By using this register the user process can form critical sections with an overhead of two cycles as follows:

- the third least-significant bit of register `g7` (`inCritical`) indicates the execution of a critical section and the least significant bit (`msgPending`) indicates the arrival of a message during a critical section,
- the user process sets `inCritical` with an add instruction before entering a critical section,
- the kernel disables interrupts and sets `msgPending` should a message arrive while `inCritical` is set, and
- the user process clears `inCritical` using a tagged-subtract instruction which causes a kernel trap if the `msgPending` bit is set and allows the kernel to dispatch the delayed message.

Note that this solution requires all parts of a program to conform to the ABI and use register `g7` appropriately (or at least not clobber it).

4.3.8 Conclusions

The CMAM implementation of Active Messages on the CM-5 demonstrates the versatility, efficiency, and incrementality of the Active Messages communication architecture. The design of CMAM follows the principles described in Section 4.1 and pays careful attention to deadlock-free request-reply communication, cheap synchronization of communication and computation, and effective data transfer into and out of the network interface.

The micro-benchmarks demonstrate that Active Messages can achieve more than an order of magnitude reduction in communication overhead over send&receive for short messages. For long messages a set of "hard-wired" message handlers provide data transfer rates equal or superior to send&receive in the form of a more flexible protocol based on the notion of a communication segment.

The analysis of the instruction sequences sending and handling messages show that the performance of the CMAM implementation is at the hardware limit. The only possible way to achieve higher performance is to use the first word of the message to encode more specific information than a pointer to a handler can. This is, in fact, why "hard-wired" handlers are necessary to support long messages at peak bandwidth. If the message size were a little more generous, say 10 words, the advantage of using a custom encoding for the first word would not be worth the sacrifice in versatility.

The careful integration of the CMAM primitives with the Sparc calling conventions allows higher-level communication primitives such as fetch&add to be implemented in C without compromising the performance demonstrated in the micro-benchmarks. The communication library underlying Split-C, described in Section 6.1, is written entirely in C and without direct access to the data network hardware[12].

In order to reach the goals set forth, it was necessary to carefully trade-off which characteristics of communication on the CM-5 to expose to the higher software levels and which to hide. The most stringent restriction exposed is the limitation of CMAM messages to four words of arguments which corresponds to the maximal message size supported by the hardware. Exposing this restriction is necessary because the potential reordering of messages occurring in the network makes it impractical to transparently support larger messages. In most cases where the message size is too small the higher levels can use a protocol that is more efficient than the type of universal message fragmentation and reassembly that CMAM would have to use. The difficulty is in allocating memory in which the individual parts of a longer message can be reassembled and in initializing the synchronization variable necessary to detect the arrival of all fragments. Instead CMAM provides a communication segment abstraction which allows data to be transferred efficiently while leaving the allocation of storage and of synchronization variables up to the higher software levels.

The CMAM implementation does not use interrupts because of their cost. This means that polling of the network has to be exposed to a certain degree. By incorporating polling in all message send functions CMAM succeeds in hiding the polls to a surprisingly large degree, but the application programmer must be aware of the consequences. For example, during compute-only phases a processor does not service requests unless explicit polls are inserted. The CMMD 3.0 implementation of Active Messages supports both synchronous and asynchronous handler execution. Because a series of back-to-back messages generates only a single interrupt the cost of the latter can be amortized if communication is frequent. However, due to a lack of kernel support, critical sections are expensive and some of the savings from not having to worry about polls is offset by the difficulty in implementing atomic accesses to variables shared by handlers and the computation.

Possibilities for improvement

Of the 50 cycles it takes to send an Active Message 26 are spent in load and store instructions accessing the network interface. The situation on message reception is worse: 32 out of 52 are spent in loads.

---

12. A few primitives use the control network which is not available via Active Message and therefore access the network interface directly.

Simply bringing the network interface chip closer to the processor could speed-up communication considerably. The problem in doing so is that placing the NI closer to the processor is more difficult and less supported with every new generation of microprocessors. With the Cypress Sparc chipset used in the CM-5 it is conceivable to place the NI on the cache bus as a coprocessor with single-cycle access. However, the next generation processor (superSPARC) does not support the coprocessor interface anymore and the first-level cache bus is on-chip. Access to the second-level cache bus is conceivable but not supported by the architecture. Thus, while it is not impossible to bring the NI closer to the processor, recent microprocessor development trends clearly make it more and more difficult.

The remaining cost of sending and receiving messages could be cut down further by improving the NI status registers. For example, having to read separate registers to detect the arrival of messages on both networks costs over 15% of the `CMAM_4` execution time. If, in addition, variable length messages are to be supported the status register layout should be carefully optimized such that a single dispatch into a code table can decode all combination of events in a minimum of instructions.

The use of interrupts is painful in part because the network interface does not help dispatching message reception interrupts quickly to user-level. The kernel must spend considerable number of cycles detecting and resetting the interrupt cause before returning to the user process. After handling the message, the user-process must trap into the kernel which must again access the NI several times. In addition, forming critical sections in the computation by disabling interrupts in hardware must involve the kernel. If the NI provided a user-accessible interrupt enable register the cost could be cut down dramatically.

4.4 Summary

The goal of Active Messages is to provide an efficient and versatile communication architecture optimized for the use of high-level parallel languages on current and up-coming communication micro-architectures. The core of Active Messages consists of a strategy to address the four key issues: data transfer, synchronization of communication and computation, handling send failure, and network virtualization. This defines a class of communication architectures which present a uniform approach to communication. Individual implementations of Active Messages expose a number of micro-architecture features to fully exploit the performance of the hardware which means that the run-time substrate is expected to change from machine to machine.

By concentrating on resolving the key issues rather than on the specifics of the communication architecture interface, Active Messages leaves a large degree of flexibility in the actual definition of an implementation as represented by a concrete definition of the 7 aspects of a communication architecture. This means that there is no single standard Active Messages implementation and that the run-time substrate must be adapted for every implementation, just as it must be adapted to the variations in instruction set architectures from one processor to another.

The benefit of Active Messages lies in the consistent approach to communication. In that respect Active Messages is similar to the notion of load/store instruction set architectures. The latter form a class of instruction set architectures characterized by a common approach towards managing high-speed registers in the processor and towards transferring data in and out of these registers. By analogy, Active Messages form a class of communication architectures which employ the same data transfer, synchronization, send failure, and network virtualization concepts.

The core idea of Active Messages is to name a user-level message handler in every message. This handler is executed within the user-process' context upon message arrival and serves to integrate the message into the on-going computation or to provide a remote service and send a reply back. The fact that the handlers are part of the user process and are specific to each message enables a great deal of flexibility in synchronizing computation and communication. The main additional primitive that the communication architecture must ensure is atomic handler execution and facilities for forming atomic sections (relative to handler execution) within the computation. Send failure is dealt-with by continuing to service incoming messages while attempting to send. To prevent uncontrolled handler nesting, communication patterns are restricted to one-way and request-reply communication such that message priorities or flow control can be used. The second column of Table 4-10 summarizes the general Active Message approach.

The two sample implementations on the nCUBE/2 and the CM-5 serve to illustrate how the general Active Messages approach maps onto existing communication micro-architectures. The discussion of the development of the two implementations demonstrates how the very different characteristics of the two communication micro-architectures lead to quite different implementations within the common Active Messages framework. The approach towards the four key issues taken in the two implementations is summarized in Table 4-10 and the specifics of the interface presented by the two implementations is defined in Table 4-11 in terms of the 7 communication architecture aspects (defined in Subsection 2.1.3).

A first performance evaluation uses a set of micro-benchmarks to determine the LogP parameters shown in Table 4-12 for the two implementations. These performance characteristics represent roughly an order of magnitude reduction in the communication overhead as compared to the implementations of send&receive provided by the vendors.

A detailed analysis of the code sequences used to implement Active Messages on both machines indicate that the performance achieved is very close to the limits of the hardware. Figure 4-28 divides the communication overhead graphically into two parts: the minimum cost for sending and receiving a message and the added cost of Active Messages. On the nCUBE/2 most of the overhead is in the user-kernel interface and in the DMA device accesses. The overhead of sending Active Messages is negligible and the cost of implementing user-level interrupts to handle message arrival represents approx. 15% of

Key issue	Active Messages in general	nCUBE/2 Active Messages	CM-5 Active Messages
Data transfer	Expose micro-architecture to allow direct data transfer into and out of application data structures.	Via pre-allocated DMA buffers. Messages limited to DMA buffer size.	Via processor registers. Messages limited to hardware message size of 20 bytes.
Synchronization	Message handler is named in each message and executes upon message arrival. Handler execution is atomic and cheap interrupt enable/disable support critical sections in the computation.	Handler addressed by first message word and executed at interrupt time. Cheap critical sections using user-kernel handshake flag.	Handlers addressed by first message word and executed upon polling.† Polls integrated into all send operations. Explicit polls available. Critical sections are free by not polling.
Send failure	Communication patterns are limited to requests and replies. Deadlock-free request-reply supported using network priorities.	Reply attempts may fail and require explicit retry by handler. No control over handler stacking depth.‡	Requests and replies use two separate data networks. Handler nesting limited to a reply handler nesting within a request handler.
Network virtualization	A messages' dest. node, process id, and handler triplet is a global address which must be protected. Context switch the network by saving the state and coordinating process scheduling.	Protection enforced by message layer in software. Processes space-share the hypercube without time-sharing.	Protection enforced in network interface hardware. Kernel gang-schedules processes and context-swaps all network state.

†. The CMMD 3.0 implementation of Active Messages supports handler execution at interrupt time.
‡. A variant with flow control to avoid handler nesting is proposed but not implemented.

Table 4-10. Addressing the four key issues in Active Messages.

Comm. arch. aspect	nCUBE/2	CM-5
message format	Destination node, handler address, followed by data, limited to 1Kbytes +ε.	Destination node, handler address, followed by up to 16 bytes of data.
message placement	In-memory buffer pointed-to by global variable.	In procedure call argument registers.
send operations	Send request message and send reply message.	Send request message and send reply message.
receive operations	Receive messages, to be used after reply failure.	Explicit poll, to be used in compute-only sections.
reception events	Message reception interrupt causes execution of Active Message handler.	No message reception interrupts. All message send operations poll and cause execution of Active Message handler.
send completion events	Request messages are always sent. Replies may fail (time-out) and require saving the message and retrying.	Request messages and reply messages use distinct networks and guarantee deadlock-free operation.
synchronization events	Handler execution is atomic. Critical sections are formed by setting a flag in memory.	Handler execution is atomic. Critical sections are formed trivially by avoiding polls.

Table 4-11. Definition of two Active Messages implementations.

Parameter	nCUBE/2			CM-5
	Active Messages		Send/recv	CMAM
message length	8bytes	1Kbytes	1Kbytes	16bytes
L (latency)	25µs	471µs	471µs	6µs
o (overhead)	14µs	14µs	80µs	1.6µs
g (gap)	14µs	236µs	236µs	4µs
P (processors)	1024	1024	1024	1024

Table 4-12. LogP parameters for Active Messages and Send&receive

the total overhead. In the CM-5 implementation, 59% of the cost lies in the load and store instructions accessing the network interface. The additional complexity for Active Messages is only a few cycles for the jump instruction to the handler.

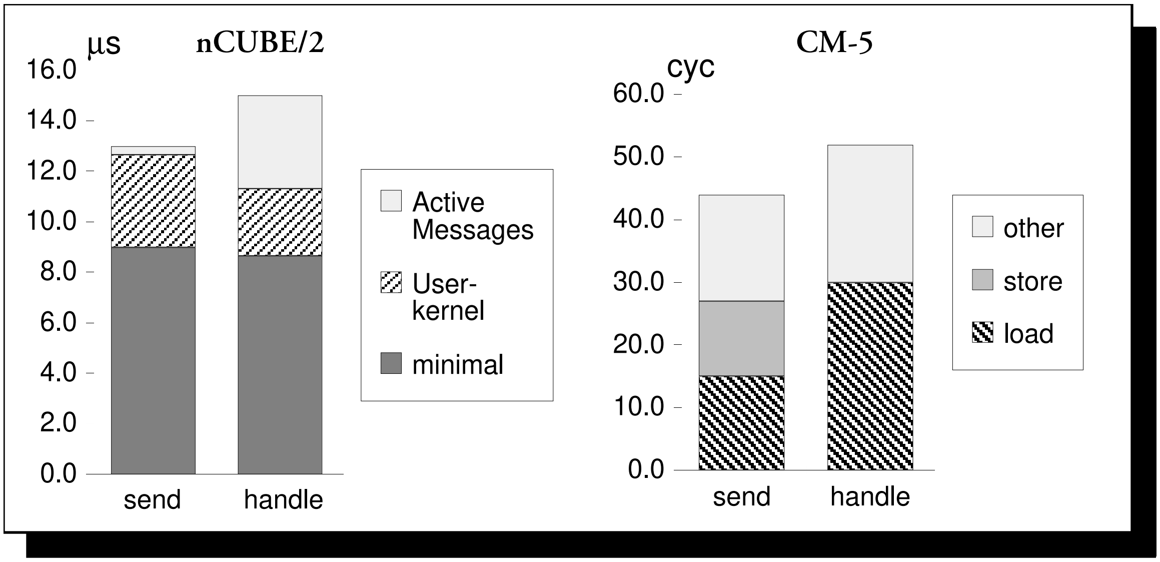


Figure 4-28: Communication overhead shown graphically

Small application benchmarks show that the overall program performance can be predicted from the LogP communication parameters derived from the micro-benchmarks. Detailed algorithmic development using the model yields well-behaved implementations and the measured performance can be explained using the parameters. This indicates that Active Messages succeeds at providing simple primitives which can be composed efficiently without uncovering hidden overheads.

In is interesting to compare the nCUBE/2 and the CM-5 Active Messages implementations. Both use the same core Active Messages idea and impose many similar restrictions to enable efficient implementation, yet, both expose most of the micro-architecture and, at first sight, may seem to have little in common. The examples presented here, however, in particular, the two implementations of fetch&add which function identically[13], show that even though the details are different the high-level concepts remain the same and provide an important leverage. In all examples, the linguistic and algorithmic design concentrates on organizing the transmission of information and is freed from the actual mechanics of message transfer.

Finally, a slightly different way to view Active Messages is that it allows “direct message execution” instead of requiring “message interpretation”. Active Messages interprets only the first word of a message, namely as a pointer to a handler. This is the practical minimum of interpretation possible. After the dispatch the message is executed in the sense that a custom compiled piece of code takes care of it.

13. Ignoring the fact that the CM-5 version supports split-phase fetch&add which the nCUBE/2 implementation could support just as well.