5 Traditional Communication Architectures

Existing multiprocessor systems do not provide a communication architecture in the sense that they do not offer a well-defined layer of abstraction to which a variety of parallel programming models can be compiled efficiently. Each parallel system has a well-defined low-level communication "layer" which is considered to be "close to the hardware" (or implemented directly in hardware) and which is proposed as compilation target for HLLs, but closer inspection reveals that the major three such low-level communication layers, e.g., message passing, message driven, and shared memory, do not satisfy the three requirements for good communication architectures: versatility, efficiency, and incrementality.

These traditional low-level communication layers are not well suited as communication architectures: they are really higher-level layers and best implemented as communication models on top of a good general-purpose communication architecture. This view is attractive both to the hardware designer and to the high-level language implementor. To the hardware designer, it means that hardware tailored to the general-purpose communication architecture can support all three (and hopefully more) traditional communication models, not just one. To the high-level language implementor it means more optimization opportunities in compiling to simpler primitives and the possibility to define new communication models which are better suited to high-level languages.

This chapter serves two roles: first it supports the claim that the three traditional low-level communication layers (message passing, message driven, and shared memory) are best viewed as communication models and not as communication architectures and, second, it proposes that Active Messages is a good communication architecture for the implementation of these communication models. The chapter is divided into three sections, each one discussing a traditional communication layer. The sections proceed in roughly three phases and begin by viewing the traditional communication layer as a communication architecture, analyzing whether it is versatile, efficient, and incremental, and then illustrate how viewing it as a communication model and implementing it on top of Active Messages does not compromise efficiency and gains in overall flexibility.

5.1 Send&receive

The de-facto standard communication layer in use on distributed-memory multiprocessors today is send&receive. Users typically view send&receive as the communication architecture of these machines: it is often considered to be "native" to the machine and the belief that "the hardware implements send&receive" is wide-spread. However, under closer scrutiny, it turns out that send&receive is neither close to current micro-architectures nor particularly efficient: it requires elaborate message protocols, matching of message tags, and potentially unbounded storage resources. As a result, send&receive is best characterized as a communication model and implemented as a run-time substrate on top of a good communication architecture.

The purpose of this section is three-fold. First, it discusses implementations of send&receive to reveal that they consist of a sophisticated software layer and cannot be mapped efficiently onto micro-architectures. Second, it shows how viewing send&receive as a communication model leads to a natural (but still sophisticated) implementation on top of an Active Messages communication architecture: the Active Message handlers in this case serve to implement the handshake protocols, memory allocation, and tag matching required by send&receive. Third, this implementation of send&receive on top of Active Messages demonstrates the versatility as well as the efficiency of Active Messages on existing micro-architectures.

Unfortunately it is not possible to simply discuss *the* send&receive communication model because, there are many variations in use and there is no official standard. Intel's implementation in the NX operating system has been a de-facto standard for a number of years and the ongoing MPI effort intends to come to agreement soon, but in the meantime PVM [GBD+93] has emerged as the most widely distributed message passing library. In spite of all these implementations which provide a number of send&receive variants each, Subsection 5.1.1 attempts to identify *a* send&receive model by defining the aspects common to all variants. The discussion itself is then split into three subsections, each focussing on a particular variant of send&receive: blocking send with blocking receive, non-blocking send with blocking receive, and non-blocking send with non-blocking receive.

Before delving into the details it is important to remember the historical context in which the numerous variants of send&receive in existence today were developed. Most of this development took place at Caltech and JPL and today's successful commercial ventures started by picking up the hardware and software developed there. At Caltech and JPL, send&receive was developed little by little following the evolution of the early multicomputer hardware; the major milestones are recounted excellently in an article by Jon Flower and Adam Kolawa [FK91] and the reader is referred to Geoffrey Fox's book [Fox88] for description of the mind-set underlying the send&receive message passing programming style.

The successive versions of send&receive which were developed directly reflected the new capabilities of each multicomputer hardware generation. This made sense at the time when hardware was the limiting factor and drove software, the shape of which was rather fuzzy. Today the situation is almost reversed, parallel hardware is (almost) ubiquitous and further developments must be driven by the needs of software. The fact that send&receive inherits the deficiencies of early hardware, that it was never designed to support high-level parallel programming languages or to serve as a compilation target make it no surprise that it does not meet today's standards anymore.

5.1.1 Definition

With send&receive the storage resources on communicating nodes are completely decoupled and communication among nodes occurs in units of messages of arbitrary length. Conceptually, *send* takes a memory buffer and sends it to the destination node and *receive* accepts a specific message from the network and stores it into a memory buffer.

The many variants of send&receive differ in the allocation of the memory buffers, the naming of messages to be received, and the synchronization between the sender and the receiver. The characteristics common to all are the fact that messages originate in memory and end up in memory, the memory addresses on each end are specified by the respective processor, and the receiver must decide on its own when to receive (i.e., must poll the network).

The following definition of a prototypical send&receive communication architecture follows the schema developed in Subsection 2.1.3 and discusses each of the seven aspects a communication architecture must define.

Message format

Each message contains: the destination node number, the source node number, a message tag (typically a 16-bit or 32-bit integer), the message length and the data itself. The message length is usually bounded only by memory limitations.

Message data placement

The data originates in a memory buffer at the source node and is transferred into a memory buffer at the destination node. The buffers are either contiguous arrays or strided vectors. However, non-contiguous buffers may require additional copying or may not benefit from hardware support, such as DMA, available on certain platforms.

Send operation

The arguments passed to the send operation are: the destination node, the message tag, the message length and the buffer address. Send "injects" the message into the network; when it returns the message buffer can be re-used. Depending on the send&receive variant, send blocks until the receiver is ready or returns immediately, regardless of whether the message was actually injected or not (i.e., it is non-blocking).

Receive operation

Receive takes as arguments a source node, a message tag, a buffer address and length. It waits for a message with matching source and tag to arrive. Typically, receive accepts wildcards for the source and the tag such that it is possible to receive any message. As soon as a matching message is received, receive transfers the data into the specified buffer and returns. Variants return a pointer to a dynamically allocated buffer in order to avoid a message copy, or provide non-blocking receive which registers the reception parameters in advance of message arrival.

Reception events

Message reception events are handled entirely within send&receive and not passed on to higher software layers. Most implementations provide a test operation which checks whether a message matching a source and tag has arrived.

Send completion

Send always succeeds. Blocking send does not return until the message has been injected into the network and non-blocking send transparently queues outgoing messages if they cannot be injected into the network.

Synchronization

Send&receive does not offer asynchronous reception events, i.e., communication is treated as being synchronous to the computation. Each receive operation corresponds exactly to the reception of a single message[1].

[1]. This is not fully true for non-blocking send with non-blocking receive, discussed below.

5.1.2 Blocking send&receive

The first versions of send&receive were designed for the first multicomputers which had primitive network hardware: back-to-back parallel ports (64-bit FIFO) and just as rudimentary software. Pushing a word into one of the FIFOs was called *send* and pulling a word out was called *receive*. There was no notion of a destination address or of a message. The inherent features are that the address spaces of the sender and the receiver are distinct and that the scheduling of computation on the two nodes is independent.

Further software developments introduced a more complete notion of messages. Given the bounded size of the FIFOs the receiver had to be pulling the head of the message out of the FIFO before the sender could complete injecting it. This led to the notion of blocking send&receive which is still in use today, although routing in the network hardware allows distant and not just neighbor nodes to communicate.

In blocking send&receive the sender blocks until the corresponding receive is executed and only then is data transferred. The main advantages of blocking send&receive are simplicity and determinism. Since data is only transferred after both its source and destination memory addresses are known, no buffering is required at the source or destination nodes. The main disadvantage is that a round-trip time through the network is necessary to synchronize the sender and the receiver before data can be transmitted. In the early multicomputers with only neighbor communication this was not a handicap: this synchronization handshake was performed in hardware by the FIFOs linking the two processors. The rigid synchronization has the advantage that programs communicating exclusively using blocking send&receive can be shown to be deterministic [Kah74, Hoa78].

5.1.2.1 Critique

Implementing send&receive on today's multiprocessors requires a three-way protocol illustrated in Figure 5-1 which sends three messages for every "send&receive message": the sender issues a request-to-send message to the receiver which stores the request and sends a reply message back when the corresponding receive is executed, the sender receives the reply and finally transfers the data. This three-way handshake necessary for every transmission implies that it is not possible to overlap communication and computation. The sender and the receiver are both blocked for at least a full round-trip time dur-

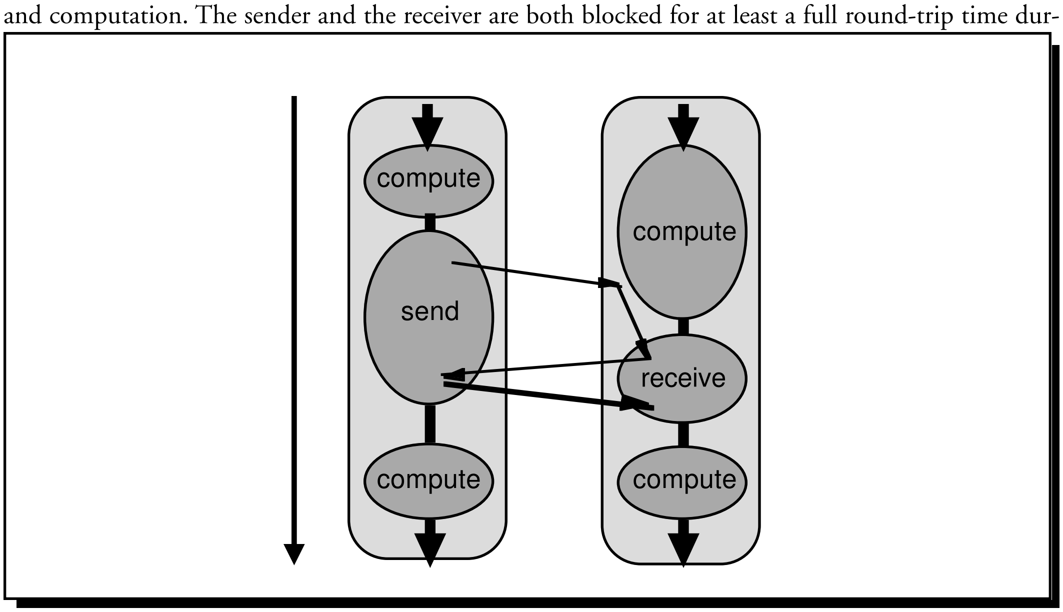

Figure 5-1: Blocking send&receive handshake protocol.

ing which the processors are idle. In addition, in most cases the sender and the receiver do not arrive at exactly the same time so one spends additional time waiting for the other.

As a programming model, blocking send&receive is prone to deadlock. This problem is best illustrated with a simple ring communication pattern in which all processors send a message to their right neighbor. If all processors perform the obvious "send to right, receive from left" communication sequence the system will deadlock with all processors trying to send and waiting for an acknowledgment. Instead, every other processor must perform a "receive from left, send to right" sequence. Then the roles are switched, so half the performance is obtained.

5.1.2.2 Sample implementation using CM-5 Active Messages

The sample implementation of send&receive uses simple data structures and Active Message handlers to implement the three-way handshake protocol as shown in Figure 5-2. Each node maintains a matching table to record the arrival of send requests from other processors so they can be matched with the corresponding receive executed locally and the appropriate reply message can be sent back.

The pseudo-code for the CM-5 implementation is shown in Figures 5-3 and 5-4. Note that even though a three-way handshake protocol is implemented, only request and reply Active Messages are used. This is made possible by the fact that the sender is waiting for the reply, meaning that the reply handler does not need to send the data itself (which would violate the request-reply limitation of Active Messages) and can simply signal to the computation that the data can be sent.

To measure the performance of the prototype send&receive implementation a simple ping-pong benchmark program sends a small message back and forth between two nodes. The result is shown in Figure 5-5: the end-to-end latency of one message is about 25µs. This number can be explained simply by summing the components used: three Active Messages with a message overhead of 3.1µs each and a network latency of 2.5µs each, the overhead of allocating a communication segment for the xfer protocol, and a small processing overhead in the actual handlers.

In summary, the sample implementation shows that: Active Messages allows a straightforward implementation, the resulting performance matches the expectations, and the discussion of the handshake protocol inherent in blocking send&receive demonstrated that Active Messages is inherently closer to

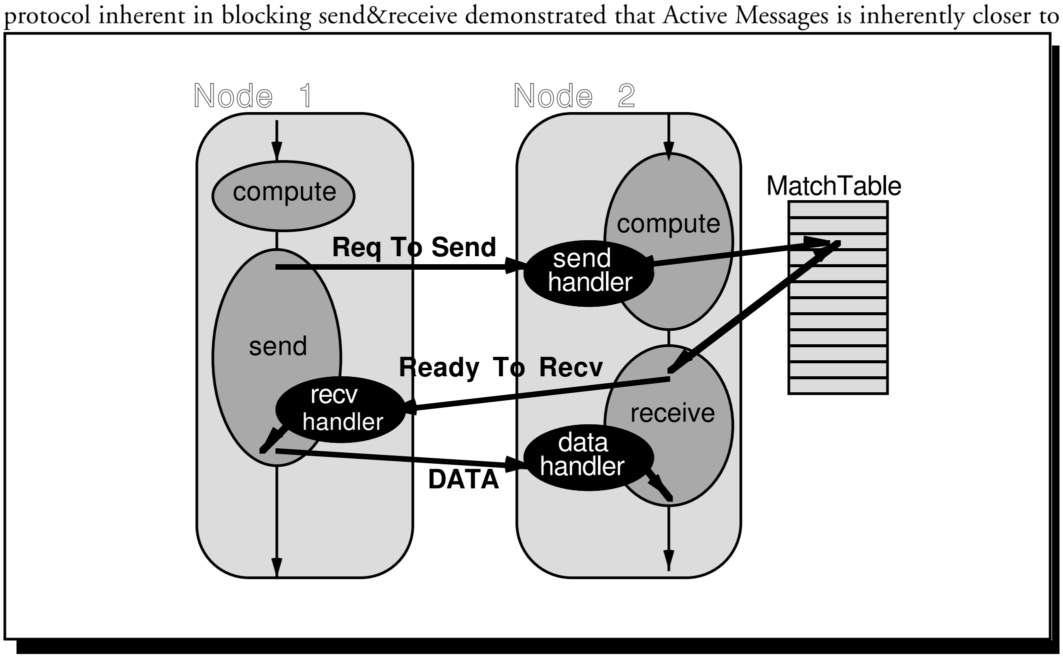

Figure 5-2: Implementation of blocking send&receive on Active Messages

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

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

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

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

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

Figure 5-3: Pseudo-code for blocking send&receive using CM-5 Active Messages.

the hardware. As "icing on the cake" the sample implementation of send&receive using Active Messages outperforms the vendor's send&receive library (25µs vs. 80µs) even though the functionality is the same and the vendor made significant optimization efforts. The most likely cause for the improvement seen with Active Messages is a greatly simplified code structure due to the versatility and efficiency of the handlers.

```
# Handle send acknowledgment
static void send_set_seg(int seg, int count)
{
        send_count = count; send_seg = seg;
}
```

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

Figure 5-4: Send&receive part 2.

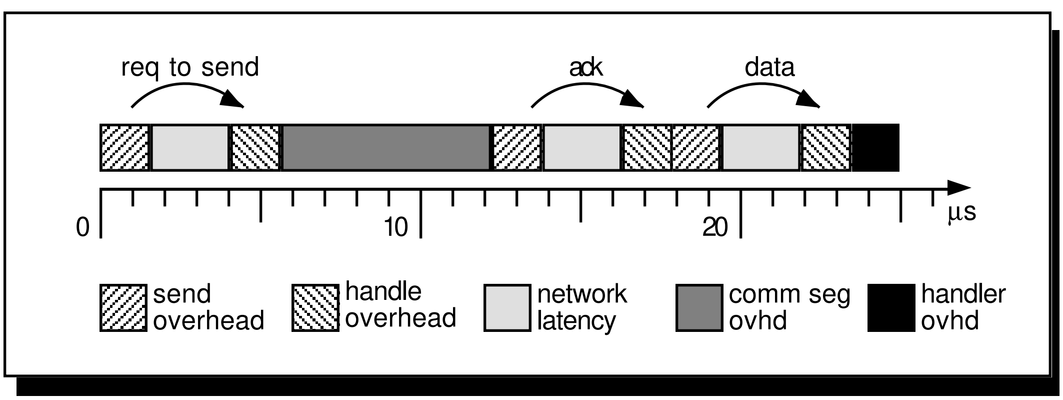

Figure 5-5: Timing for blocking send&receive implementation on CM-5 Active Messages.

5.1.3 Non-blocking send and blocking receive

Historically, non-blocking send was introduced in order to break out of the neighbor-only communication imposed by blocking send&receive at the time. The hardware did not provide message routing, thus forwarding messages had to be performed in software. This led to an interrupt-driven message layer which buffered entire messages at each intermediate hop. Given this buffering required for the routing, supporting non-blocking send was for free: it requires message buffering at the source if the outgoing channel is blocked and at the destination if the corresponding receive has not yet been executed.

The first attempt (called “rdsort” [Joh85]) was rather similar to Active Messages: a message could be sent from any node to any other node, at any time, and the receiving node would have its program interrupted whenever a message arrived. At this point the user was expected to provide a routine called “rdsort” which, as its name implies, needed to read, sort and process the data in the message. Apparently the reason rdsort never caught on was that the application programmers, physicists to a large part, had to deal with the concept of interrupts in their programs which proved to be difficult and error prone. It became clear that exposing the asynchrony of communication to the applications programmer was not appropriate. Because no high-level parallel languages were used, there was no concept of a communication model distinct from the communication architecture and the asynchrony had to be hidden within the latter. This lead to non-blocking send and blocking receive which hides the asynchrony behind a queueing mechanism implemented within the communication architecture.

In non-blocking send with blocking receive, a processor can send a message to any other processor at any time without blocking. The message layer takes care of buffering and queueing the message wherever necessary until a receive accepts the data. The receive operation itself is still blocking: if no message matching the requested source node and tag is available the processor blocks. Note that most systems provide an operation to test whether a matching message is present.

5.1.3.1 Critique

On current multiprocessors where the hardware routes messages the communication architecture still has to queue messages at the sending and receiving ends. This creates major difficulties because a potentially unbounded amount of buffer space must be provided. For example, while the ring communication introduced in § 5.1.2.1 can be written straight-forwardly as “send to right, receive from left”, a processor which slows down (e.g., because of IEEE denorm exceptions) may have to buffer up to P-2 messages, and not just one as one might expect at first.

The buffer management in non-blocking send and blocking receive is particularly troublesome because messages are of variable length and may be consumed in any order by receives (due to the matching) which means that a fully general-purpose dynamic memory allocator must be used. Each message reception (as well as each message send if the outgoing channel is blocked) therefore includes the equivalents of UNIX `malloc` and `free`. In addition, on machines where access to the network is privileged, the message buffers are typically in system space and must either be copied into user space or the (arbitrary size) message buffer must be mapped into user space when receive is executed.

Implementations of non-blocking send and blocking receive pay varying degrees of attention to this potentially unbounded storage resource in the communication architecture. This is reflected in the various protocols used and is explained in more detail below.

Optimistic one-way protocol

The simplest approach is just to send messages off and not to worry about buffer space. This approach is used in nCUBE’s Vertex system. The send system call allocates a local buffer in system space and copies the user data into it. The buffer is then enqueued on the appropriate outgoing DMA engine. When that becomes available the message is sent in two segments: the header indicating message tag and length, followed by the body of the message with the data. On the receiving side the kernel inspects the header, allocates the appropriate amount of memory for the body, and starts the second DMA. The receive operation finally copies the data from the system buffer into user space.

A variant available in Vertex avoids the copies of the data: before composing the message the user process traps to the message layer to acquire an appropriate buffer and composes the message in-place. Similarly, a variant of receive passes the address of the system buffer (which is mapped into user space) to the user process which, after consuming the message, must trap back to the kernel in order to free the buffer. Mapping the system buffer into user space is made possible by the fact that the message header is received into a protected fixed-size system buffer and is used to allocate the actual message buffer directly in the process' address space. Given the simple segmented memory model there is no problem with page boundary crossings when performing DMA directly into user space. The cost however is the additional interrupts and DMA set-up operations required to send the message header and the additional traps into the kernel to allocate and free message buffers. The result is that this variant is only beneficial if messages are very large and saving the copying compensates for the additional message-size independent overheads.

With respect to the potentially unbounded storage requirements, the Vertex implementation simply aborts the program if memory for a message cannot be allocated. The only recourse to the user is to restart the program with an option increasing the amount of memory allocated to message buffers at process start-up!

Safe three-way protocol

The three-way protocol uses a handshake similar to the one used in blocking send&receive to ascertain that the destination node has enough memory to accept the message before sending the data. This is the default approach used in Intel's NX system. If the destination node does not have a sufficient amount of memory, it simply delays sending an acknowledgment until other messages are consumed. Of course this may lead to deadlock given that the user program may execute a receive which only matches the message being held-up. The frustrating aspect of this deadlock is that the program itself appears perfectly correct and deadlock is only introduced due to message buffer limitations combined with a particular message arrival order. It is possible to avoid this deadlock if the message layer checks the list of delayed messages for a match and uses the user-supplied message buffer to transfer the appropriate message.

Two-way protocol with pre-allocation

To avoid the cost of the safe protocol with its three-way handshake and associated communication latency, the Intel NX system supports a two-way protocol for short messages. Each processor pre-allocates a fixed-size buffer for every other processor in the system. A message shorter than one of these pre-allocated buffers can be sent immediately with the guarantee that buffer space is available at the destination. The destination node replies with an acknowledgment when the buffer is freed again. The source node thus keeps track of the state of its associated buffers in all other nodes and only sends short messages if it has received an acknowledgment indicating the availability of the appropriate buffer.

The costs of this protocol lie in the acknowledgments and the amount of memory reserved for messages which grows with the square of the number of processors in the system. As a result the two-way protocol is only appropriate for small messages.

5.1.3.2 Implementation using CM-5 Active Messages

Implementing non-blocking send and blocking receive with Active Messages is relatively straight-forward following the same scheme as for blocking send with blocking receive. A difficulty only arises if the send operation is supposed to queue outgoing messages when the network is temporarily backed-up. The Active Messages implementations to date block in this situation. The problem is that in order to transparently queue the outgoing message a copy of it must be made. This hidden memory allocation and copying with the associated costs is at odds with the goals of Active Messages.

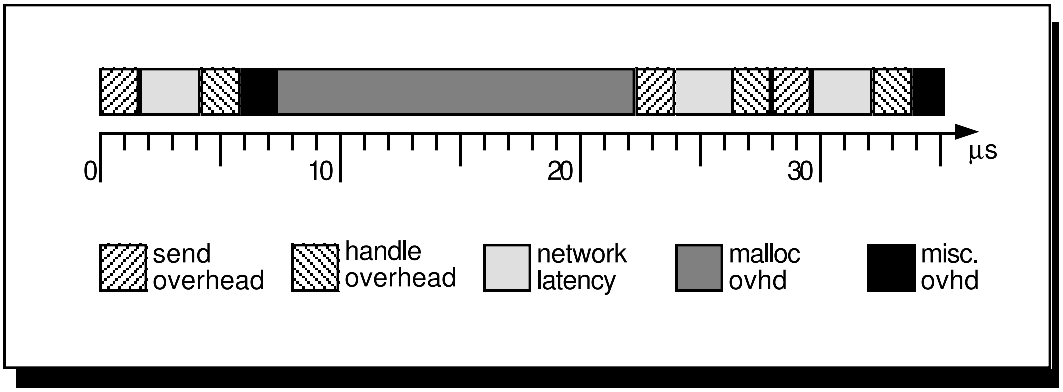

Figure 5-6: Timing estimate for send&receive implementation.

Timing estimate for non-blocking send and blocking receive implementation using CM-5 Active Messages.

A more attractive alternative is to expose the queueing. A queueing Active Message send operation could return a completion code which indicates whether the message has been enqueued or not. If it has been enqueued then the caller is not allowed to change the buffer holding the message until the message layer indicates (via a flag or a call-back) that the message is sent. This alternative moves the responsibility for managing the buffer space out of the message layer which thus needs to signal the buffer state transitions to the user.

Adding such a queueing Active Message send operation to the nCUBE/2 implementation is straightforward. The user process allocates a message buffer and passes its address to the kernel. If the outgoing DMA engine is busy, the kernel enables the appropriate send-completion interrupt and enqueues the message. When the DMA becomes available, the message is injected into the network and another interrupt signals completion of the DMA. At that point the kernel indicates to the user process that the message buffer can be freed either by setting a flag in memory or by running a send-completion handler. No modification to the normal Active Message send is necessary as this is actually the technique used to send Vertex's messages with which the Active Messages implementation is compatible.

The CM-5 implementation can be modified in a similar manner. A buffer pointer can be passed into a queueing Active Message send function which attempts to send the message once or twice and saves the data in the provided buffer if the outgoing FIFO is full. The problem on the CM-5 is that the hardware does not support any form of FIFO non-full interrupt. Thus either all other message sending and receiving operations have to be modified to check the queue and retry pending sends or a periodic interrupt must be used. Neither solution is very satisfactory, but this is purely due to limitations of the micro-architecture and is independent of Active Messages.

Another problem with implementing the three-way handshake is that it violates the limitation to request-reply communication imposed by Active Messages: the data is sent as reply to a reply. Given that a non-Active Messages implementation does not make any livelock-freeness guarantees one can take the standpoint that the Active Messages version does not need to guarantee anything either. However, it is probably better to rely on some form of mechanism which allows the acknowledgment handler to post a signal to send the data.

Apart from these two caveats the implementation follows the same model as the blocking send and blocking receive implementation. In order to sum up the components of the implementation to get an estimate for the overhead an estimate for the buffer management is needed. A small test program which calls malloc and free repetitively to allocate a 100 byte block of memory shows that each malloc-free pair costs from 15µs to 17µs depending on the allocation and deallocation order. Figure 5-6 shows the resulting timing estimate.

5.1.4 Non-blocking send and non-blocking receive

In addition to non-blocking send Intel's NX system supports non-blocking receives (called asynchronous receives by Intel). A non-blocking receive specifies a message source, tag, and buffer and returns a message id. When a matching message arrives, it is transferred into the specified buffer. The user process can detect the arrival by querying the message status using the id.

The potential advantage of non-blocking receive is that if the user process posts a receive before the arrival of the corresponding message the message layer can move the data directly into user memory without having to allocate and free a buffer on its own.

The difficulty in implementing non-blocking receives is that it allows a process to have many receives pending. On message arrival, the message source and tag have to be compared to that of all posted receives. If a receive matches, the message can be transferred straight into user memory, otherwise the message layer must allocate a buffer for temporary storage. In order to keep track of multiple pending receives, each one requires a small message descriptor with the buffer address, length and the receive status used to service the queries. An Active Message implementation is straight-forward as the handler can perform all these operations easily.

5.1.5 Critique

In examining the variants on send&receive it seems that each one has a severe source of inefficiency which the next one corrects at the cost of introducing yet another inefficiency. The problems associated with each variant are illustrated below in a discussion of how the four key issues (data transfer, send failure, synchronization, and network virtualization) are addressed.

Data transfer

All variants deal with data transfer identically and provide a standard interface which hides the peculiarities of the underlying micro-architecture. The basic model is memory-to-memory communication, e.g., send transfers data from memory into the network and receive from the network into memory. The advantage of this standard is that it allows the use of DMA devices to transfer messages into and out of the network.

The major problem with such a standard interface is that it is not versatile. Typically only contiguous buffer and simple strided vectors are supported. The result is that often the application must copy data from its data structures into a contiguous buffer before sending a message and vice-versa after receiving one.

Send failure

Send failure poses no problems in blocking send&receive: all storage for arriving messages is preallocated and thus accepting incoming messages while attempting to send poses no problems.

The situation is rather different for non-blocking sends where a send failure requires the outgoing message to be queued as failing sends cannot simply block because that could introduce deadlocks. The missing feedback when sending means that the storage for queued messages cannot be bounded. In addition, queueing messages is expensive due to the dynamic buffer allocation and message copying involved. On machines with a single outgoing channel the buffer allocation and deallocation is in FIFO order while machines with multiple outgoing channels (such as the nCUBE/2) require fully general memory allocation.

Synchronization

Blocking send&receive forces the communication to be synchronous to the computation by synchronizing the sender and the receiver. In addition to the network round-trip latency required for the synchronization, this costs in idle time whenever one of the two communicating processors waits on the other.

Non-blocking send synchronizes communication and computation at the receiving end using message queueing and associative matching. Messages arriving before the corresponding receive is executed are queued; this requires dynamic buffer allocation on message arrival as well as copying from the buffer to the final destination when receive is executed. Non-blocking receive can avoid the queueing if the receive is posted before the message arrives. Messages and receive operations are paired using associative matching on the source node and the message tag. In the case of blocking receive, a receive operation requires a match against all queued messages while an arriving message needs to be matched against at most one receive. In the case of non-blocking receive, however, an arriving message may have to be matched against a large number of posted receives.

Network virtualization

In order to virtualize the network, message sends must check the destination node address and tag the outgoing message with the source process id. At the receiving end, the process id must be included into the matching along with the source node and message tag.

5.1.5.1 Versatility

The above discussion of the key issues provides insight into the difficulties posed by the implementation of send&receive. The remainder of this section uses a simple example taken from the compilation of Fortran-D [Koe91] to illustrate that the send&receive model is not versatile enough as a compilation target for parallel languages, leading to inherently inefficient implementations.

Figure 5-7 shows a simple loop together with the typical code generated by the Fortran-D compiler for non-blocking send and blocking receive. The objective in generating the code shown is to overlap communication and computation. Each processor starts by sending data it holds to all processors needing it for the loop execution. While all the messages are "in transit", each processor proceeds with the loop iterations requiring only local data. If the layout of the arrays was well chosen, then at the end of the local loop iterations all messages have arrived such that all the receives execute without blocking. After that, each processor executes the remaining loop iterations.

This example illustrates the three major problems with non-blocking send and blocking receive: (i) if the array layout was well chosen then all messages arrive before any receive is executed implying that all messages are queued and incur the associated buffer allocation and copying costs, (ii) the buffer storage required to hold the queued messages depends solely on the array sizes and layout which is only bounded by the total memory available, and (iii) each receive must be matched against many messages (the matching is required to transfer each message into the right memory location).

The queuing costs could be avoided through the use of non-blocking receives. The idea is to post all receives before performing the local loop iterations (or even before starting to send) such that arriving messages can be stored directly into the appropriate memory location. While such a variant eliminates the queueing, the send&receive layer must now match each arriving message against all pending receives. In addition, the program must still check for the completion of each send individually.

A variant using blocking send and receive is even less attractive than the two previous ones: it is impossible to overlap communication and computation and avoiding deadlock requires a complicated communication schedule.

This simple example nicely illustrates the problems inherent in send&receive. The overall operation is very simple: all processors cooperate in a communication phase where the storage for all messages can be preallocated by the compiler, where the source and destination addresses for all messages are known, and where only simple bulk synchronization is required (after performing the local loop iterations each processor must wait for all its data to arrive). Yet, send&receive prevents the compiler from exploiting the fact that the high-level language defines a global address space with the result that unnecessary dynamic buffer allocation, message copying, and per-message synchronization is performed.

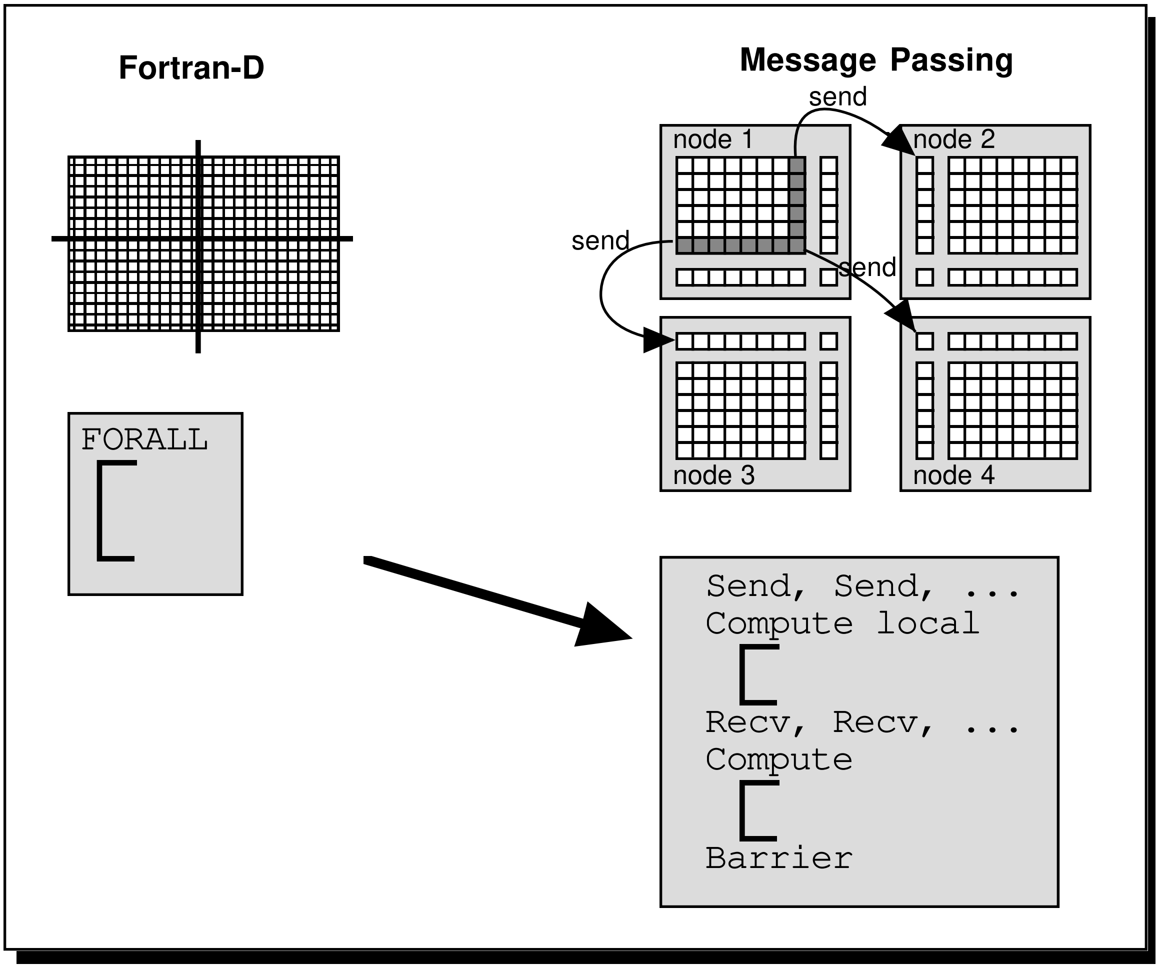

Figure 5-7: Fortran-D compiled to non-blocking send and blocking receive

5.1.6 Summary

While send&receive is often touted as being "close to the hardware" this section has shown that this is not the case and that send&receive is neither efficient nor versatile. Therefore send&receive is not suited as general-purpose communication architecture. All send&receive variants involve a combination of round-trip latencies, multiple message copies, dynamic buffer allocation, unbounded storage, and costly synchronization in form of tag matching. While each successive send&receive variant eliminates the worst problem of the previous one, it increases the cost of another operation in the process: eliminating round-trips costs queueing and copying, eliminating the copies worsens buffer allocation, eliminating queueing altogether increases the tag matching cost.

A different approach to message passing is represented by Hoare's [Hoa78] Concurrent Sequential Processes which formalizes message passing and is implemented (more or less directly) in the OCCAM programming language and the Transputer VLSI processor. The communication in OCCAM is equivalent to blocking send&receive but the basic execution model includes multithreading on each processor. Thus, communication and computation can be overlapped by switching to another process that is ready to run. This model still suffers from long communication latencies as a three-way protocol is still required although, in the case of the Transputer, the handshake is implemented in hardware. Multithreading the processor leads to weak instruction sets, typically with very small registers sets: the Transputer incidentally uses only three registers organized as an operand stack.

Perhaps the most significant fallacy in send&receive is to keep the address spaces of all the nodes disjoint, i.e., not to form a global address space. As sketched in the Fortran-D example (and as demonstrated in detail in Section 6.1) using global addresses in the message layer can eliminate message buffer allocation and data copying in most cases. Given that it is generally agreed that a global address space is beneficial from the programming point of view, virtually all new parallel programming languages use global addresses and, as consequence, implementations on top of send&receive are not very attractive.

5.2 Dataflow and Message Driven

Message driven execution models have been developed specifically to support languages with fine-grain dynamic parallelism such as CST [HCD89, Hor91], Multilisp [Hal85] and Id90 [Nik91]. The basic argument for such execution models is that long, unpredictable communication latencies are unavoidable in large-scale multiprocessors and that multithreading at the instruction level provides an attractive avenue towards tolerating them. Supporting the resulting frequent asynchronous transfers of control (context switches) efficiently requires integrating communication deeply into the architecture which, in effect, is what is done in dataflow and message driven architectures.

This section examines these two execution models[2] in light of their versatility, efficiency and incrementality, and, after a brief description of the models in Subsection 5.2.1, argues that while these models are very versatile they map the high-level language execution model directly onto the micro-architecture which leads to inefficient implementations reminiscent of high-level language instruction set architectures. It is further argued that message driven architectures do not represent an incremental evolution from state of the art uniprocessors; they represent a step back to primitive instruction sets with few registers. Subsection 5.2.3 briefly compares message driven models with Active Messages, which uses a similar message reception mechanism, but imposes stringent restrictions on communication precisely in order to tame the inefficiencies inherent in message driven models. The comparison of message driven models with Active Messages communication architectures is completed in Section 6.2, which describes how the dynamic parallel languages for which message driven models were designed can be implemented more efficiently using Active Messages. This completes the proof that Active Messages is versatile enough to achieve the goals of message driven architectures, and that it offers higher efficiency and incrementality.

5.2.1 Description

The central idea in message driven execution models is that computation is driven by messages which contain the name of a *task* and some data. On message arrival, storage for the message is allocated in a scheduling queue and when the message reaches the head of the queue, the task is executed with the data as arguments.

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

In the fine-grain parallel languages for which message driven architectures were primarily developed, the frequency of message arrival is high and the corresponding tasks are small. As a consequence, the machines implement message reception, task creation and dispatch in hardware to the point where these phases form the basic execution step. The prototypical architecture of a message driven processor can be described as follows (adapted from [DW89]):

- The machine consists of a set of nodes.
- Each node contains a set of memory segments. Each segment consists of a set of words and a *ready* flag. If the ready flag is set, the segment contains a task that is ready to run.
- Each word of a segment may contain a value, a pointer to a segment, or a distinguished *not present* value that indicates that the location is empty and awaiting data.
- A ready task may: operate on the contents of its state, read or write the contents of another segment in the same node, create or destroy a segment on the same node, and send a message

[2] In this section the term message driven model will be used collectively to include dataflow models as well.

to another node. When a task segment is destroyed, the corresponding task is terminated. Writing a task segment's ready flag has the effect of suspending or resuming the task.

- If a task attempts to read a *not present* value, it traps to an exception handler. In most cases the exception handler clears the task's ready flag.
- On message arrival a segment is allocated to hold the message and the ready flag of this segment is set.

5.2.2 Critique

The above description of a prototypical message driven processor is quite different from the functioning of a conventional processor, yet, Dally [Dal89] stresses that message driven processing is not really that radical a departure from conventional processing. He presents the following analogy: "a [message driven] processor receives a message from the network and dispatches a control sequence to execute a task specified by the message" while, in contrast, "a conventional processor is instruction-driven in that it fetches an instruction from memory and dispatches a control sequence to execute it".

The troublesome aspect in Dally's analogy is that tasks are dynamic entities which must be *created*, requiring the dynamic allocation of storage, and they must be *scheduled* as they suspend and resume. This really raises the level of the architecture to include responsibilities that are conventionally part of the operating system or the language run-time system. In fact, the phrasing of the analogy reveals the flaw in the argument. The correct analogy would be that messages name tasks which are executed and instructions name micro-sequences that are executed (whether the micro-sequences are represented as micro-code or as hard-wired pipeline-stage logic is an orthogonal issue). The important difference here is that while the micro-sequences are hidden within the architecture, the tasks in message driven architectures are exposed at the architectural level and can consist of arbitrary code sequences. As a consequence, they can have arbitrary lifetimes and require arbitrary storage allocation patterns. Worse, the number of tasks in a system is only bounded by the available parallelism in the program [CA88] while the number of simultaneous micro-sequences is typically limited to the number of pipeline stages in each processor. All this is not to say that the notion of tasks is bad, but that requiring the architecture to manage tasks without controlling their use seems dangerous given that the designer cannot ensure that tasks are used in such a manner that their management in hardware is simple and efficient.

Pushing the analogy between message driven and conventional processors a bit further illustrates the problem nicely. Consider the internals of a conventional superscalar processor; one could argue that after fetching an instruction from memory it *creates* a micro-*task* to execute it. The task allocates storage (registers) and other resources (function units, issue slots) and may suspend and resume multiple times (while other micro-tasks continue executing) if resources are not available. The key point, however, it that the set of tasks, their resource demands, and their scheduling patterns are hidden below the architectural level and are carefully crafted by the processor designer towards a precise cost/performance trade-off goal. A superscalar processor has a fixed amount of resources for parallelism to unfold and, in some sense, determining the amount of resources to provide is the major high-level design challenge in superscalar processors: less restrictions on micro-tasks can result in higher performance, but requires more resources and possibly a more complicated design. For example, if micro-tasks are not restricted to FIFO creation and termination, instructions can be executed out of order which is more complicated to manage (as witnessed by the small number of processors supporting it [Cas92]). The allocation of resources is usually kept very simple, for example, most superscalar processors execute instructions in-order which corresponds to allocating all resources at task creation time (many superscalars allow out-of-order instruction *issue* which is compatible with allocation at task creation time).

In summary, by exposing the notion of tasks, message driven architectures loose control over the complexity of their management and are therefore doomed to support very general forms of memory allocation and task scheduling. Brief descriptions of two message driven machines in the following two paragraphs confirm these problems. The J-Machine features a fairly simple design at the cost of not implementing the message driven model fully. Monsoon does implement the model and achieves higher performance than the J-machine, but pays with a complex and expensive design.

5.2.2.1 The J-Machine

The J-Machine hardware [Dal89] is designed to support the message driven model directly. The J-machine consists of a 3-D mesh of single-chip Message-Driven Processors (MDP) each with a 256K by 36-bit DRAM bank. The CPU has a 32-bit integer unit, a 4K by 36-bit static memory, a closely integrated network unit, a 3-D mesh router, and an ECC DRAM controller (but no floating-point unit). The 36-bit memory words include 4-bit tags to indicate data types such as booleans, integers, and user-defined types. Two special tag values *future* and *cfuture* cause a trap when accessed. The MDP has three separate execution priority levels: background, level 0, and 1, each of which has a complete context consisting of an instruction pointer, four address registers, four general-purpose registers, and other special-purpose registers.

The MDP implements a prioritized scheduler in hardware. When a message arrives at its destination node, it is automatically written into a message queue consisting of a fixed-size ring buffer in on-chip memory. The scheduler interprets the first word of each message as an instruction pointer to the task to be executed when the message reaches the head of the scheduling queue. Background execution is interrupted by priority 0 message reception, which in turn may be interrupted by priority 1 message reception.

The J-Machine fails to fully support the message driven model in several aspects. (Interestingly this is exactly in the points where the designers optimized for a simple and efficient VLSI implementation rather than full support of the model.) The implementation of the message queues as ring buffers restricts task scheduling to FIFO order. This means that tasks cannot suspend and resume as the model allows. Instead, to suspend, a task must save its state in a separately allocated memory segment. To resume a task requires sending a local message to insert the task into the scheduling queue. This additional copying is reportedly [Hor91] necessary for roughly 1/3 of all messages in CST programs.

A second problem is that the size of the hardware message queue is limited to a small fraction of all memory, namely to on-chip memory. Yet, in a dynamic parallel program the size of the scheduling queue (which is mapped onto the message queue in the J-Machine) depends on the amount of excess parallelism in the program. Given that the excess parallelism can grow arbitrarily (as can the conventional call stack) it is impractical to set aside a fraction of memory for the message queue. Rather it must be able to grow to the size of available memory. In other words: the limited size of the scheduling queues places an arbitrary limit on the total number of tasks in the system.[3]

Another problem in the J-machine design is the indirect cost of context switches. While the processor provides three sets of registers to allow fast switching between tasks of different priority levels, this does not help switches between tasks at the same level. In order to keep these context switches cheap the processor minimizes the per-task state to be saved by providing only eight registers. Thus, context switches at the same level have a significant indirect cost in slowing down execution in general by requiring more memory accesses than if more registers had been provided [SGS+93].

While the J-Machine does not fully support the message driven execution model it turns out to support Active Messages quite well: the background execution level can be used for computation and the two higher levels for request and for reply handlers, respectively. The message ring-buffers are then used as true buffers to absorb fluctuations in communication rates and not as scheduling queues. The message send instruction and the hardware dispatch provide excellent support for low-overhead communication supporting Active Messages directly.

Besides the insufficient number of registers available for the computation another problem becomes apparent in the use of Active Messages on the J-machine. The close interaction of Active Message handlers with the computation requires atomic operations on data structures which the MPD does not support

3. Strictly speaking, the J-Machine scheduling queue can grow to the size of memory: when the network is backed-up the send instruction causes a trap which can store the outgoing message in off-chip memory until the network flows again. However, executing these traps is rather time-consuming and causes additional network congestion [NWD93].

efficiently [SGS+93]. The MDP does not provide any instructions to atomically update memory, instead interrupts must be temporarily disabled at a cost of 4 cycles. Similarly, the separation of the register set for each priority level prevents handlers from sharing registers with the computation for variables such as the head of the scheduling queue or the pointer to the current activation frame.

5.2.2.2 Monsoon

Monsoon implements an Explicit Token Store (ETS) dataflow architecture [PC90]. The machine includes a collection of pipelined processors, connected to each other and to a set of interleaved I-structure memory modules via a multistage packet switch as shown in Figure 5-8a. Messages in the interprocessor network are tokens identical to those entering at the top of the processor pipeline. Thus, the hardware makes no distinction between computation ongoing on each processor and computation launched by arriving messages.

The processor pipeline, depicted in Figure 5-8b, consists of eight stages. Tokens carrying an instruction pointer, a frame pointer and a 64-bit data value enter the top of the pipeline. After the instruction fetch

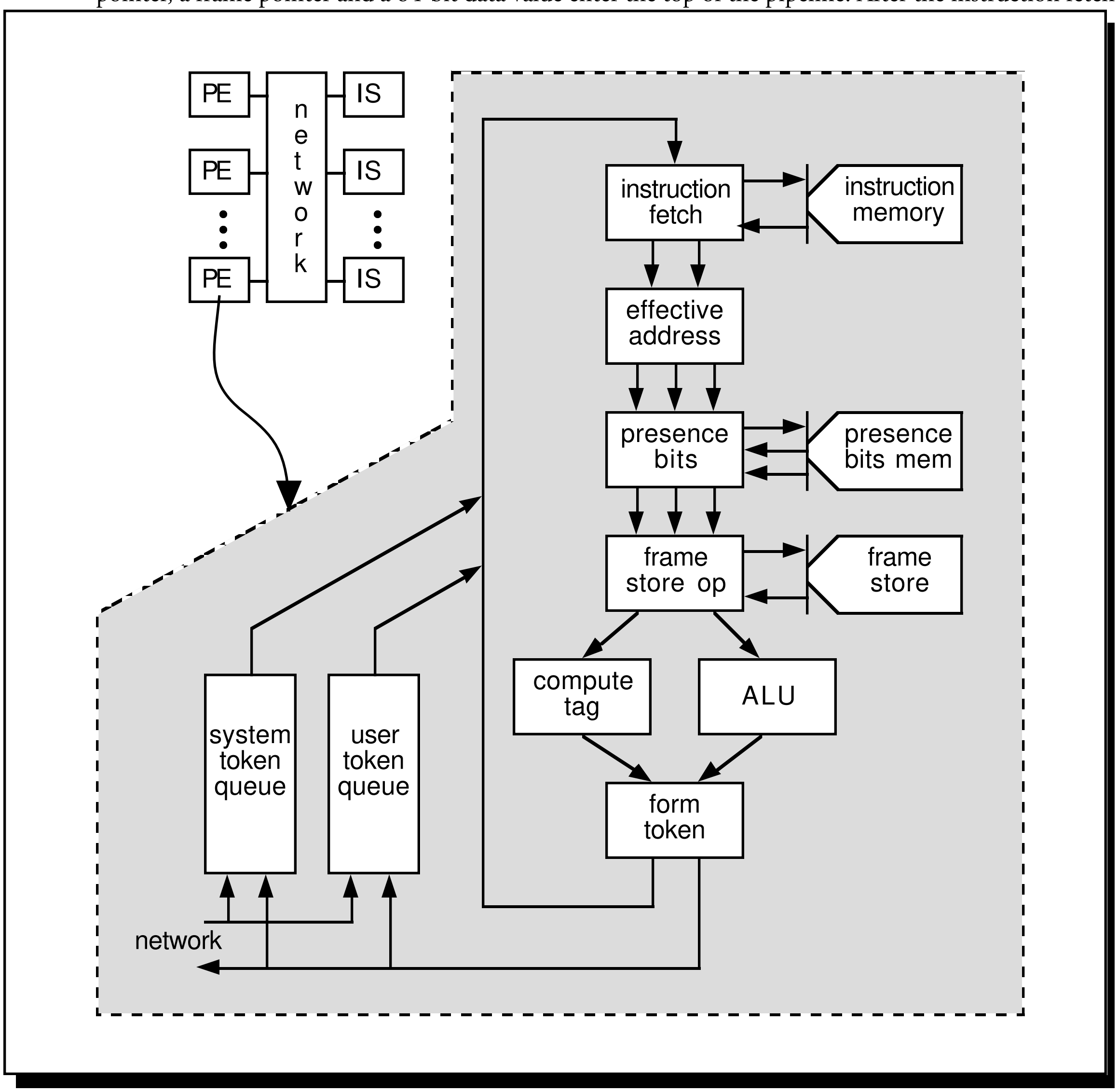

Figure 5-8: Monsoon processing element pipeline.

and the effective address calculation stages, the presence bit stage performs the synchronization, which forms the heart of the execution model. At that point, instructions specify a frame location where the two operands to a diadic instruction are synchronized. The presence bits associated with the frame location are checked: if the location is empty the next pipeline stage stores the data value in the fame and the instruction is annulled, otherwise the location is fetched to yield the second operand and the instruction proceeds through the pipeline.

Problems arise in a number of components of the Monsoon processor. The token queue, which buffers messages arriving from the interconnect as well as instructions forked locally, holds the excess parallelism available in the program. For this reason it is kept in a separate memory of 64K tokens proportional in size to the 256Kword activation frame store. If one assumes an average frame size of 64 words, the token store provides for roughly 16 tokens per frame on average. Unlike the J-Machine, the Monsoon hardware does allows the scheduling queue to grow to the limits of memory, but at the cost of a special-purpose large and fast (one token is written and one is read every cycle) memory.

The processor pipeline is 8-way interleaved, so eight threads of control can be active simultaneously. As soon as a thread terminates or suspends by blocking on a synchronization event, a token is popped from the queue and a new thread starts executing in the vacated pipeline interleave. The fact that tokens are popped from the queue means that the storage allocated for an arriving message is deallocated upon thread (i.e., message handler) execution. If a thread suspends, all relevant data is stored in pre-allocated storage in the activation frame, thus, unlike the J-machine, Monsoon does implement the message driven model fully.

A hidden cost of the ETS execution model lies in the relative weakness of the instructions. To understand this point, it is best to view the instruction set as implementing a multithreaded machine [PT91]. Each token (or message) represents a virtual processor consisting of an instruction pointer, a frame pointer and a single accumulator holding a 64-bit value. (The latest version of Monsoon recognizes the lack of fast registers as a problem and extends the instruction set to allow three accumulators to be used by a thread of execution as long as it remains active.) The processor simulates an arbitrary number of virtual processors by switching among them on a cycle-by-cycle basis and the token queue holds the collection of virtual processors that are ready to execute. The instructions executed by the virtual processors basically form a one-address instruction set [PT91] and the frame store represents the level of the storage hierarchy where local variables are kept. This means that the frame store must be implemented in high-speed memory. "Enhancing" the instruction set to the equivalent of the three-address forms typical of RISC architectures would require building a three-ported register file the size of the activation tree and supporting single-cycle read-modify-write access to the presence bits associated with each location.

5.2.3 Relationship to Active Messages

The handler dispatch mechanism used in Active Messages is very similar to message driven execution. The fundamental difference is that Active Message handlers do not compute: the computation proper occurs in the "background" and handlers only remove messages from the network and integrate them into the computation. Because of this, Active Message handlers, unlike message driven tasks, can be restricted sufficiently to ensure that their management remains simple and efficient. In contrast to message driven tasks, Active Message handlers:

- execute immediately on reception,
- cannot suspend and resume,
- are allocated only an instruction pointer (and possibly a PSW), and
- are scheduled atomically and in FIFO order (at two priority levels—request and reply).

This means that the micro-architectural support for Active Message handlers is limited to inserting a spontaneous "`jsr`" into the instruction stream on message reception and preventing further reception until execution of the corresponding return [vECG+92].

The impact of the differences reach beyond message reception per-se. In message driven execution models message reception is frequent and tasks are small. Thus processors must switch contexts quickly and accumulate little state while executing each task. In contrast, conventional processors (which ignore communication) achieve high performance by bringing a considerable amount of state close to the function units and, because moving this state is time consuming (providing several sets is expensive), switch contexts infrequently. With Active Messages, message reception can be frequent which requires switching between computation and message handling to be fast, but the computation itself remains decoupled from the communication such that it can build up enough state to yield high performance. If dynamic scheduling of computation is required by the language model it should be implemented in the run-time substrate and carefully trade-off context switch frequency against computational efficiency. Stated differently, Active Messages allows scheduling *events* (e.g., changes in the scheduling data structure) to occur frequently in handlers without requiring special hardware support, but expects scheduling *actions* (e.g., context switches) to be infrequent as they are expensive regardless of hardware support.

5.2.4 Summary

This section has shown that message driven and dataflow execution models which incorporate the communication architecture deep into the processor micro-architecture are neither efficient nor incremental. These models are reminiscent of high-level language architectures and, in effect, incorporate the language run-time system directly into the hardware. This not only precludes compiler optimizations possible when targeting simpler primitives, but requires overly complex hardware implementations.

These claims are illustrated by two existing machine implementations. The J-Machine implementation is rather simple, but this simplicity is achieved by not fully implementing the message driven model: the number of tasks is arbitrarily limited by on-chip queues, tasks cannot suspend and resume, and context switches are expensive because the register set is too small. Monsoon, on the other hand, does implement the model fully, but at high hardware cost: very wide data paths and several large high-speed special-purpose static memories, yet weak instructions and few registers.

The comparison of message driven models with Active Messages shows that while both use the same message handler dispatch mechanism, the restrictions imposed on Active Message handlers completely change the implementation. Active Messages are fundamentally simpler and hence more efficient than message driven models. Interestingly, the J-Machine not only implements a subset of the message driven model to keep the implementation simple but it turns out that this subset is very close to Active Messages and can be used to implement Active Messages on the J-Machine efficiently. The J-Machine compute performance, however, remains crippled by the frequent context switch assumptions underlying the message driven design.

This subsection did not address the question whether message driven and dataflow models offer greater versatility than Active Messages or not. This aspect is discussed in Section 6.2 which shows that the high-level languages that message driven and dataflow models were originally designed for are supported efficiently by Active messages.

5.3 Shared-memory

In shared memory architectures communication is hidden within the memory system. Fundamentally, the view is that all processors have equal access to all memory locations and communication is expressed using reads and writes of memory locations accessed by multiple processors. This section examines shared memory as a communication architecture and discusses whether shared memory can be used efficiently as the foundation on which a variety of communication models can be built. It argues that while shared memory offers substantially lower communication overheads than traditional message passing, overall it is not as versatile or efficient as Active Messages. The two main advantages of shared memory are that a local/remote check (or memory renaming) is incorporated in the address translation performed for every memory access, and that the cache controller/network interface formats messages automatically.

Shared memory machines are not all alike. In sufficiently small systems it is possible to realize the fundamental model: all processors can be connected to the memory (or memories) via a broadcast bus and each processor's cache can observe bus traffic to keep its data consistent with all other caches. For small scale systems bus based shared memory multiprocessors have many advantages. The backplane-style bus design is very similar to that of a minicomputer, the cache coherency protocols are well understood, and they add relatively little to the normal memory latencies. Larger systems, however, require more bandwidth than a bus can provide. The emerging structure is similar to large scale message passing machines in that each node contains a fraction of all memory and nodes are interconnected by a message passing network. Unlike in message passing machines the network remains hidden within the memory system: the memory or cache controller transforms memory accesses into messages if the location accessed by the processor is remote.

Due to the variety of shared memory models used it is not possible to discuss *the* shared memory model. Instead, after a first subsection defining the common elements, the discussion splits into two parts according to the degree to which the illusion of a shared uniform access memory is maintained.

The first approach, discussed in Subsection 5.3.2, does not attempt to hide the distributed memory nature of the machine and only provides the convenience of mapping loads and stores of remote memory locations into the appropriate messages required for communication. Programming these machines with non-uniform memory access (NUMA) requires similar data layout and placement strategies as message passing machines do, except that the access methods to remote data are different and have the benefit that pointers can be followed irrespective of their location. This allows global data structures to be represented in a straight-forward manner.

The second approach, discussed in Subsection 5.3.3, does attempt to provide the illusion of a true shared memory by caching recently used memory locations on each node (this scheme is called cache-coherent shared memory with non-uniform memory access, or CC-NUMA for short). If the caching performs well then the shared memory appears to have the same access time as the local cache. Another way of viewing it is that these machines extend the bus-based cache coherency protocols to work on a network. To allow the coherency protocols to scale, directories are introduced to keep track of all copies of a given memory location and to limit communication to the cache controllers affected by a memory access.[4]

5.3.1 Definition

This discussion assumes a fairly simple model of a distributed shared-memory machine supporting atomic updates to shared locations in addition to simple reads and writes. The machine consists of P nodes each with a processor, a memory management unit, a cache, local memory, and a network interface. For the purpose of this discussion, the cache, the memory, and the network interface are intercon-

4. While cache-only memory architectures (COMA) are not explicitly mentioned here, the discussion of CC-NUMA shared memory applies with only minor exceptions to COMA shared memory.

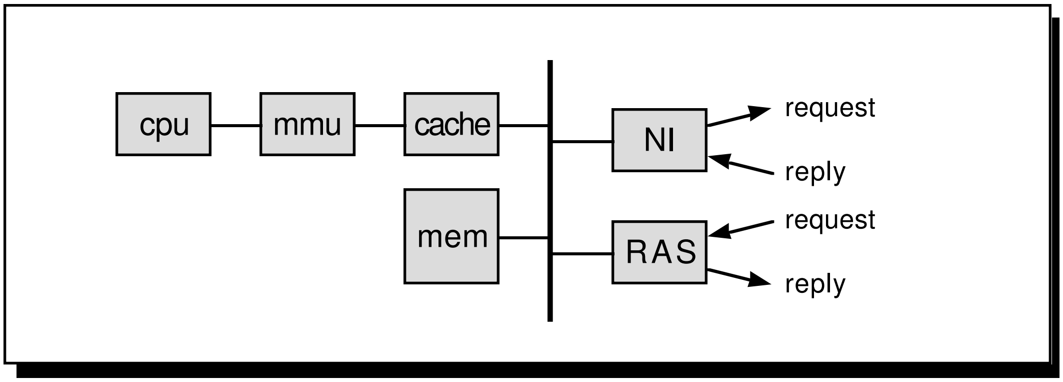

Figure 5-9: Generic shared memory multiprocessor node structure

The network interface (NI) responds to memory accesses on the bus that cannot be serviced locally and sends a request to the appropriate remote node. At the remote end, the Remote Access Server (RAS) receives the request, performs a local memory access on behalf of the remote processor and sends a reply back. The reply is received by the NI which completes the local bus access.

nected by a bus as depicted in Figure 5-9. Whether a virtual or a physical cache is used is orthogonal to this discussion. A physical cache, as suggested by Figure 5-9, is typically easier to reason about.

The memory management unit divides the virtual address space into a number of segments to provide the programmer with some control over the communication required for each memory access. In typical systems, a segment is either private and maps directly to local memory or it is shared and maps to a collection of memories on the local and remote nodes. The network interface responds to addresses located on remote nodes. It forwards the access across the network to the remote access server (RAS) which performs the access as if it were a second processor on that node.

In most systems, the processor can access memory using one of three operations: read, write, and atomic update. Atomic update performs an atomic read-modify-write cycle of a memory location. The set of atomic operations supported varies with each machine, but typically includes test-and-set.

5.3.2 Shared memory with non-uniform memory access

Shared memory multiprocessors with non-uniform memory access are best thought of as distributed memory multiprocessors with a global address space implemented in hardware. The distinction between local and remote memory locations is well defined and the programmer is responsible for allocating data structures such that communication is minimized.

The programmer manages the data placement through a small number of memory segments which map virtual addresses to local and remote memories in different ways. Usually the address space of each process is divided into 3 segments:

- a *local* segment mapping local memory into the virtual address space,
- a *global* segment mapping all memories into the virtual address space with the high-order address bits selecting the node, and
- a *shared* segment interleaving all memories using the low-order address bits to select the node.

The local segment is used for the code, stack and private data regions of each processor and can be cached by each processor's private cache. The global segment is used for data owned by a particular processor, but accessed by others as well. Allocating global data structures in the global segment allows individual data elements (whether integers or aggregate structures) to be placed on specific nodes. The low-order interleaving of the shared segment is used to allocate global data structures such that the accesses are hashed across all memory modules.

5.3.2.1 Anatomy of a shared memory access

Memory management plays a very important role in shared memory systems and is often overlooked. The three-part memory mapping described above illustrates how memory management enables various forms of data placement. In addition, memory management performs an automatic local/remote check on every memory access to determine whether communication across the network is required or whether the location accessed resides in the local memory. Because of this intricate coupling of memory management along with communication, this discussion includes aspects of memory management into the communication architecture. The steps involved in the case of a remote access are:

1. *Memory access initiation*: the processor issues a read, write or atomic-update access with a *global virtual address*.
2. *Virtual to physical address translation*: the memory management unit translates the global virtual address into a *global physical address*; the global physical address is composed of two components: a node address and a local physical address.
3. *Local/remote check*: the node address component causes the network interface to respond to the memory access.
4. *Request message*: the network interface transforms the memory access into a request message which it sends to the remote node's remote access server.
5. *Service and reply message*: the remote access server uses the local address included in the message to access local memory and it sends a reply message back.
6. *Memory access completion*: the network interface receives the reply and completes the processor's access.

5.3.2.2 Definition

Defining a NUMA shared memory communication architecture in terms of the seven communication architecture aspects defined in Subsection 2.1.3 is a little awkward but it does illustrate some of the fundamental differences between NUMA shared memory and other communication architectures.

Message format

Each request message contains a physical memory address, an op-code, a return node address, and, in the case of a write request, a data value. The data value is represented in one of the data types supported by the architecture. Reply messages contain an op-code and, in the case of a read reply, a data value.

Message placement

Communication is between the originating processor's register set and the remote memory.

Send and receive operations

All communication is round-trip and is initiated by normal memory access instructions (loads and stores in a RISC architecture). Each load/store involves a send to launch the request message and it involves a receive to incorporate the reply message.

Send completion, message reception events and synchronization

The point of the shared memory model is to make communication transparent, thus it does not make sense to signal send completion or message reception events to the processor. To the processor, communication appears as a single long memory access during which it is stalled. Thus, with the exception of extensions to the base model such as prefetch and weak-stores, communication is strictly synchronous to communication.

5.3.2.3 Addressing the four key issues

Examining how NUMA shared memory designs address the four key issues is quite illuminating: the processor itself is involved in none of them! In some sense this is simultaneously the big advantage and the major drawback of shared memory architectures. The advantage is that the key issues can be dealt-with in an implementation specific manner while using an off-the-shelf uniprocessor, the drawback is that the processor, and thus the program, cannot be involved in resolving thorny situations.

Data transfer

On the node initiating a remote access the data is transferred directly between the processor's registers and the network using ordinary load and store instructions. The actual messages are formatted by the network interface. This means that the data for the message is encoded and transferred quite efficiently on the address and data buses, but it is limited to the size of the largest native data type.

At the remote end, the remote access server transfers data directly between the network and memory (typically DRAM), the remote processor is not involved.

Send failure

The processor initiating a remote memory access is simply stalled until the access completes: communication appears as a very long memory access. The network interface handles all network transactions and must be designed to avoid deadlock and livelock in the presence of network congestion. Most shared memory multiprocessors use two networks to support the deadlock and livelock-free request-reply communication patterns required for NUMA shared memory.

Synchronization of communication and computation

The communication issued by a processor is synchronous to its computation given that the processor is stalled until the communication completes. Accesses from other nodes are not synchronized with local operations, except that each individual memory access is atomic relative to all others. In addition, most multiprocessors support simple atomic read-modify-write updates (e.g., test-and-set, compare-and-swap, fetch-and-add) on remote memory and use a simple ALU within the remote access server to perform the operation. Thus, all synchronization issues are internal to the memory system.

Network virtualization

The network is virtualized by controlling the virtual-to-physical address translation: no kernel intervention is necessary for a user-level process to access the network. In essence, the network operates at a "physical" level and does not distinguish among multiple user processes at all. It is the operating system's responsibility to maintain the translation tables on all processors such that the process running on each processor can only access global physical memory locations that are part of its address space.

5.3.2.4 Critique

Programming NUMA multiprocessors is not all that different from programming message passing machines in that in both cases the programmer is responsible for distributing data structures across the machine and for making local copies of frequently accessed data. The difference is that in NUMA machines the virtual memory management is used judiciously to check on every data access whether communication is required and the network interface formats and handles messages automatically in hardware.

The following paragraphs examine NUMA shared memory more closely to compare its versatility, efficiency, and incrementality to Active Messages and to the other traditional communication architectures.

Versatility

Building complex global data structures (especially involving pointers) is simplified by the global address space. Data structure elements can be allocated on any node and pointers can be traversed by any processor. For performance, however, care must be taken to minimize the number of remote references. This means that for memory references that are never remote shared memory offers little advantage; for references that are always remote the advantage is that the communication is handled in specialized hardware which potentially reduce overheads, but the programmer remains responsible for making a local copy if the same remote location is accessed repetitively. The elements of shared memory communication architectures come fully into play in the memory accesses which are sometimes local and sometimes remote, i.e., in the cases where it is not easy to determine in advance whether communication is necessary or not. Here the memory mapping provides the local/remote check for free and the network interface formats messages efficiently.

The major limitation of shared memory models in general is that communicating an event (as opposed to data) is not directly supported and is expensive to emulate. Using polling to implement processor-to-processor communication may be acceptable if the destination processor is busy-waiting for the arrival of one particular event, but it quickly becomes impractical if computation is event driven and one of several events can occur at any point in time. In such a case, either multiple flags must be polled or a single flag must be updated atomically to implement an event queue. Using such atomic updates requires a number of remote memory accesses to enqueue a single event which makes the implementation of dynamic scheduling, such as required by message driven models of execution, expensive. Note that while many machines provide a "back-door" network to interrupt a remote processor, this facility is outside of the memory model (e.g., the synchronization between memory accesses and back-door interrupts is ill-defined) and typically not very efficient.

In addition, with shared memory it is difficult to move the computation to the data instead of moving the data to the computation. First of all, the remoteness of an access is detected outside of the processor, beyond the control of the program which should take the decision to move the computation. And second, the fact that communicating an event is inefficient makes it difficult to move computation which requires interaction with the remote processor itself.

Efficiency

Shared memory allows communication to be initiated with a single load or store instruction. All the information to compose request and reply messages is exchanged between the processor and the memory controller using the address and data buses which are highly optimized. However, this ability to initiate communication with a single-instruction also has drawbacks. In particular, the largest transfer size is that of the largest data type which typically limits the per-message payload to 64 or 128 bits. This is inefficient for the transfer of large blocks of data because a large fraction of each message (up to 50%) is wasted by address and other header information. As a result the transfer rate is low unless special forms of memory accesses (e.g., prefetches and weak writes) are used to have multiple outstanding requests.

Due to the sequentially consistent memory model supported by most shared memory multiprocessors, each processor has at most one outstanding memory access. This means that either the processor is computing locally or it is waiting for communication to complete. In the base model there is no overlap between communication and computation and the processor utilization drops dramatically as the remote reference rate increases: let R be the average run length in processor cycles before a remote reference occurs and let L be the average latency of remote references, then the processor utilization is $u = r/(r+L)$ (useful work over total work).

All communication is round-trip and it is always the initiating processor which detects completion of a communication operation. The paradigm underlying most of message passing and data parallel in which processors producing data *push* it to those needing the values is not well supported in that it is not possible to achieve the efficiency of one-way communication and there is no way for the recipient to detect the arrival of the data other than by the sender setting an additional flag explicitly (i.e., for each datum transferred an additional write is required to set the flag).

Handling the first-level (on-chip) cache found in all high-performance processors in the presence of NUMA shared memory is not obvious. The easy observation is that local segments can be cached normally and that remote data cannot be cached unless the application manages the cache in software and flushes stale data explicitly[5]. Caching local data that can be accessed by other processors, however, is not easy: it requires maintaining coherency between the memory accesses performed by the remote access server on behalf of other processors and data cached in the processor. One solution is to use a write-back cache and pay careful attention to ordering effects due to the write buffer[6]. Alternatively, a cache-coherent bus can be use to keep the processor cache and the remote access server coherent. Whether such a scheme works is not obvious. For example, consider the situation where the local processor is stalled waiting for a remote access to complete and the remote access server receives a request from another node. In this case, while the cache is accessing the network interface via the bus for the processor's remote access, the remote access server must gain access to the bus to service the request. Furthermore, if the memory location accessed by the arriving request is present in the cache, then the cache controller must respond to the remote request before it can complete the processor's access.

Incrementality

An important advantage of shared memory over other communication architectures is that, to the first order, it does not affect the processor design, yet couples the network tightly to the processor by using the fast memory interface. Shared memory multiprocessors can typically use off-the-shelf processors because all logic specific to communication is hidden in the memory system. This, of course, comes at the cost of adding, essentially, a simple second processor on each node to service remote accesses.

On closer inspection, however, the processor is not completely unaffected by the changes in the memory system. The considerations above illustrate that the on-chip cache must be designed specially to work efficiently with a NUMA shared memory communication architecture. It must not only support a cache-coherency protocol, but it must also be able to relinquish the bus to another bus master (the remote access server in this case), continue snooping and possibly even intervene, and finally retry the bus cycle it initiated.

Overlapping communication and computation in order to improve the efficiency of the system also requires support in the processor. A small amount of overlap can be achieved by exploiting instruction-level parallelism and allowing arithmetic instructions to proceed while a remote memory access is stalled. Prefetches and weak stores generalize this type of out-of-order execution using compiler inserted instruction to allow overlap across statically determined code sequences. Further overlap requires the processor to abort the memory access and to switch to another thread of control, i.e., multithreading. Such support is non-trivial and can only cover a limited amount of communication latency [GHG+91, SBCvE90].

5.3.2.5 Sample implementation using Active Messages

This sample implementation of NUMA shared memory using Active Messages shows that Active Messages is in some sense more primitive than shared memory and that the latter can be viewed as a communication model rather than a communication architecture. Building shared memory on top of Active Messages does not preclude the use of special-purpose hardware to perform a local/remote check or to format messages. Hardware similar to that used in shared memory multiprocessors can be used to accelerate the specific Active Messages required for remote memory accesses while other Active Messages remain available normally.

The basic idea is to transform load and store instructions into short code sequences to perform the address translation, the local/remote check, and potentially to communicate. A straight-forward scheme would transform every heap access, e.g., excluding memory accesses identifiable as stack or private data

5. Not many architectures provide user-level instructions to do this efficiently.

6. Another simple solution is to service remote requests from the processor's side of the cache. This is unattractive, however, because it effectively steals a cache cycle from the processor for every access from a remote processor and it affects the cache hit rate adversely.

```
# synchronization
volatile int count = 0;                number of outstanding requests
volatile int buf;                      temporary to hold remote value
inline void wait() { while(count != 0) CMAM_poll(); }
# perform shared read
inline int shared_read(int addr)
{
  int node = addr>>24;                 assume 128 nodes, max 16Mb/node
  wait();
  if(node == MYPROC) return *addr; assumes correct mapping
  if(node < 0) {
    node = (addr>>3) & 0x7f;           assume 128 nodes, double-word interleaving
    addr = (addr>>7) & ~0x7) | (addr & 0x7); assume correct mapping
  }
  count++;
  CMAM_4(node, read_handler, MYPROC, addr);
  wait();
  return buf;
}
# service read request
void read_handler(int ret_node, int addr)
{ CMAM_reply_4(ret_node, read_reply_handler, *addr); }
# service read reply
void read_reply_handler(int value)
{ buf = value; count--; }
# perform shared write
inline void shared_write(int addr, int value)
{
  int node = addr>>24;                 assume 128 nodes, max 16Mb/node
  wait();
  if(node == MYPROC) { *addr = value; return; } assumes correct mapping
  if(node < 0) {
    node = (addr>>3) & 0x7f;           assume 128 nodes, double-word interleaving
    addr = (addr>>7) & ~0x7) | (addr & 0x7); assume correct mapping
  }
  count++;
  CMAM_4(node, write_handler, MYPROC, addr, value);
}
# service write request
void write_handler(int ret_node, int addr, int value)
{
  *addr = value;
  CMAM_reply_4(ret_node, write_reply_handler);
}
# service write reply
void write_reply_handler(void) { count--; }
```

Figure 5-10: NUMA read and write using CM-5 Active Messages.

references. More aggressive implementations can use language extensions allowing the programmer to classify variable accesses, compiler analysis to identify local heap accesses, or to lift address translation and local/remote checks out of loops, potentially generating separate local and remote code sequences.

The sample implementation of read and write in Figure 5-10 assumes that memory accesses are transformed early in the compilation process such that the functions used in the implementation can be in-

```
  # initiate prefetch
inline void shared_prefetch(int addr)
{
  int node = addr>>24;              assume 128 nodes, max 16Mb/node
  wait();
  if(node == MYPROC) { buf = *addr; return; } assumes correct mapping
  if(node < 0) {
    node = (addr>>3) & 0x7f;        assume 128 nodes, double-word interleaving
    addr = (addr>>7) & ~0x7) | (addr & 0x7); assume correct mapping
  }
  count++;
  CMAM_4(node, read_handler, MYPROC, addr);
}
  # complete prefetch
inline int accept(void)
{
  wait();
  return buf;                       at most one outstanding prefetch!
}
  # unordered write allowing overlap
inline void shared_weak_write(int addr, int value)
{
  int node = addr>>24;              assume 128 nodes, max 16Mb/node
  if(node == MYPROC) { *addr = value; return; } assumes correct mapping
  if(node < 0) {
    node = (addr>>3) & 0x7f;        assume 128 nodes, double-word interleaving
    addr = (addr>>7) & ~0x7) | (addr & 0x7); assume correct mapping
  }
  count++;
  CMAM_4(node, write_handler, MYPROC, addr, value);
}
```

Figure 5-11: Weak NUMA shared memory operations.

lined by the optimization passes of the compiler. Virtual addresses are represented in 32 bits using the top eight bits to identify the various memory segments. For global segments the top bit is clear and the next seven bits indicate the node number (for up to 128 nodes); for the low-order interleaved segment the top bit is set and the 7 low-order bits of the double-word address (i.e., ignoring the lowest 3 bits) indicate the node. It is not necessary to distinguish between a local segment and the global segment mapped to the local node. In order to avoid having to mask the high order bits the code assumes that the local memory mapping has been set-up appropriately.

In the example the sequential memory consistency model is enforced by using a counter of outstanding requests and checking that no requests are in progress before issuing a new one. Implementing weaker consistency models is a matter of changing these simple completion checks. As an example, Figure 5-11 shows the implementation of a split-phase prefetch and of a weak write.

Implementing atomic operations on shared variables is straight-forward using the handler to perform the atomic update. Note that this allows the set of atomic operations to be easily extended.

Performance

As shown in Table 5-1 the CM-5 the Active Message implementation of shared memory achieves one third the communication performance (relative to the computational performance) of hardware implementations such as in the BBN TC-2000. By adding some simple hardware support on the CM-5 to format read and write messages quickly in the network interface itself (e.g., a 64-bit store of the remote

	CM-5	BBN Butterfly	BBN TC-2000
floating point operation	0.2µs	3µs	0.1µs
remote access latency	13µs	6µs	2µs
local/remote check overhead	1µs†	0µs	0µs

†. Most of the cost is due to the poll included in every local/remote check to ensure that processors keep servicing the network even if all memory references are local. A more sophisticated scheme would associate polls with basic blocks instead of with individual local/remote checks.

Table 5-1. Comparison of shared-memory in hardware with Active Messages.

node and memory addresses into the network interface could launch a read request) it is conceivable to narrow the gap. However, the real benefit of the hardware implementation lies in the "free" local/remote check on every memory access (free in terms of cycles, not in terms of hardware).

5.3.3 Cache-coherent shared memory with non-uniform memory access

Multiprocessors with cache-coherent shared memory with non-uniform memory access (CC-NUMA for short) attempt to support the illusion of a true shared memory with uniform access cost. The machines are built using the same physically distributed memory architecture as NUMA shared memory multiprocessors, but a cache on each node is used to keep a copy of recently used remote data. As long as memory references hit in this cache most of the time, communication is avoided and all of memory appears to be accessible in the time of a cache access. The overall structure of each node is similar to NUMA shared memory and to Figure 5-9 with the main difference being that a directory entry is associated with each memory block[7] to record which caches hold a copy of the memory block. The challenge in these designs is to keep data that is replicated in several caches consistent which is achieved by exchanging coherency messages among the caches.

The CC-NUMA hardware relieves the programmer from managing the data placement explicitly, except that in order for the illusion of true shared memory to work the programmer must optimize for high spatial and temporal locality in the data references. For simplicity, the address space is still split into a local segment and a global segment, mainly to allow the program code, constant data, and runtime stack (which are all strictly local) to start at the same address on all processors without interfering.

5.3.3.1 Anatomy of a shared memory access

The intricate coupling of memory management with communication and the overall sequence of events occurring for a CC-NUMA shared memory access are similar to NUMA shared memory. A detailed discussion of cache coherency protocols can be found in [LLG⁺90, CKA91, KUB91, WCF⁺93]. The following sequence illustrates the events occurring when a processor writes to a cache line which has just been written by a processor other than the one at the home node:

1. *Memory access initiation*: the processor issues a write access with a *global virtual address*.
2. *Virtual to physical address translation*: the memory management unit translates the global virtual address into a *global physical address*.
3. *Local/remote check*: the cache tag lookup determines whether a local copy of the cache line accessed is available. If not, the global physical address is decomposed into a home node address and a local physical address.
4. *Request message*: the network interface transforms the memory access into a request message which it sends to the home node's remote access server.

7. A memory block is the set of bits that can be held in a single cache block.

5. *Service and forward message*: the remote access server uses the local address included in the message to check the directory and determine that a third cache holds the up-to-date copy, and then forwards the request to that node.
6. *Service and reply message*: the remote access server accesses the local cache and sends a reply to the requesting node.
7. *Memory access completion*: the network interface receives the reply and completes the processor's access.

Note that unlike in NUMA shared memory the initial request message does not contain the value to be written. Instead, the cache line is brought into the local cache and modified there.

5.3.3.2 Definition

The definition of CC-NUMA shared memory communication architectures according to the seven communication architecture aspects established in Subsection 2.1.3 is largely identical to NUMA shared memory. The three most important differences are that:

- CC-NUMA shared memory messages carry a cache block of data instead of a single data value,
- in CC-NUMA shared memory whether a load or store involves communication depends on the cache state and not solely on the address,
- CC-NUMA shared memory uses a larger number of message formats to implement the cache coherency protocols.

5.3.3.3 Addressing the four key issues

The most insight about CC-NUMA communication architectures can be gained by examining how the four key issues are addressed. As previously discussed in the case of NUMA shared memory the key issues are all dealt with in the memory system and the processor is not affected—at least to the first order.

Data transfer

Data transfer is optimized for moving cache blocks from one cache to another or between a cache and memory. Irrespective of the type and size of memory access issued by the processor, the messages sent across the network always carry a full cache block of data. The processor itself never interacts directly with the network. The cache always serves as intermediary and brings the appropriate cache block into the local cache and completes the processor's memory access locally.

Each cache, besides playing its traditional role of keeping a copy of recently used data, also serves as renamable buffer storage. In effect, when a processor requests a remote location, the cache automatically allocates a cache line (i.e., a fixed-size buffer) to hold the data. When that data is not used anymore, the cache reallocates the cache line (i.e., frees the buffer).

Send failure

The handling of send failure is very similar to NUMA shared memory, but with the important difference that the communication patterns are no longer simple requests and replies. As illustrated in § 5.3.3.1, if the home node does not have an up-to-date copy of the data it must forward the request to another node.[8] Providing deadlock and livelock free operation in these circumstances requires three message priority levels. Incidentally, the DASH network only provides two and the coherency protocol is not livelock-free: if a home node cannot forward a request it replies with a negative acknowledgment to the requestor which must retry.

---

8. A slightly worse pattern occurs in certain coherency protocols (such as the one used in DASH) if many copies of a cache line must be invalidated. In this case, the home node must send many invalidation messages in response to a request and each of the caches receiving an invalidation must send an acknowledgment back to the original requestor.

Synchronization of communication and computation

With CC-NUMA shared memory communication remains synchronous to computation and the processor is stalled if a memory access cannot be satisfied by the local cache. The cache coherency makes atomic memory updates easier, however. Once a cache block is exclusively owned by the local cache, any ordinary read-modify-write update supported by the processor can be performed simply by preventing requests from other nodes to steal the cache line away.

Network virtualization

Virtualizing the network is the same with CC-NUMA as with NUMA shared memory. All communication occurs with physical memory addresses and it is the of the virtual memory address translation mechanism which provides the protection boundaries. Virtual memory accesses are mapped to physical memory at the issuing processor which requires all address translation tables on all nodes to be kept consistent.

5.3.3.4 Critique

The communication model presented by cache-coherent shared memory is very different from that of message passing or uncached shared memory. The programmer is essentially encouraged not to think about communication and to focus on locality, both spatial and temporal. However, due to engineering considerations in large scale machine design, the architecture underlying the cache-coherent shared memory must perform the same type of communication as all other communication architectures discussed in this dissertation. The argument in favor of cache-coherent shared memory over message passing is two-fold: the programming model (i.e., working with a global address space and reasoning about locality instead of communication) is claimed to be simpler and the communication that must occur is handled by specialized hardware which can be optimized for the task. The following paragraphs examine these arguments as well as some counter-arguments in more detail.

Versatility

Cache-coherent shared memory has similar advantages and disadvantage than uncached shared memory. Manipulating complex data structures is aided by the cache coherence because the programmer does not have to worry about making local copies of frequently accessed objects: the cache allocates a copy automatically. However, it becomes more difficult to move the computation to the data because the software, in general, cannot determine where the data is located.

Implementing processor-to-processor communication is complicated by the cache coherence and building Active Messages on top of shared memory does not seem practical. In the base model it is not possible to push data to a remote node, all memory accesses only have the effect of bringing cache lines close-by. In order to obtain the data to be sent, the other processor has to read it itself and incur the round-trip latency at that point. Recent cache-coherent multiprocessors (e.g., the KSR-1 [Ken92]) include special forms of write in an attempt to support "pushing data", but the fact that communication is cache-to-cache makes the implementation of such operations difficult. Essentially, to push data requires the source node to allocate cache lines at the destination.

The fact that all transfers are of cache block size has several disadvantages (advantages are discussed below). When the data transferred between two nodes is smaller than a cache block, bandwidth is wasted. In the case of transfers larger than a cache line, the fragmentation into fixed-size chunks introduces overheads in the form of redundant control information and, unless a processor can have multiple outstanding memory requests, the total cost of the transfer is the product of the round-trip latency and the overheads at each end instead of being the sum of the two.

The fixed-size cache block have another problem: if two processes access disjoint variables which happen to be allocated to the same cache block, the memory system will act as if they were accessing the same memory location and the coherency mechanism will enter into action and transfer the cache block back and forth. This phenomenon is called false-sharing as the two processor appear to share the memory location even though they are not.

Efficiency

The efficiency trade-offs for CC-NUMA shared memory revolve around the same issues as for NUMA shared memory: automatic local/remote check, communication initiated with a single instruction, and specialized hardware for message formatting and handling. With cache-coherence the local/remote check is no longer for free, however: the cache must include additional tag bits to indicate the coherency state of each cache line, and thus, even though the local remote check does not cost extra cycles, extra real-estate is required. In addition, an access in CC-NUMA may have to miss in all local caches (most architectures have multi-level caches) in order to determine that communication is required while the local/remote check could occur very early in a memory access in NUMA architectures (namely just after address translation).

Besides the use of the cache as renamable buffer space, as discussed in § 5.3.3.3, a second major efficiency advantages of cache coherence is the blocking into cache block-size transfers that occurs automatically. A simple example is if a processor reads consecutive memory locations the cache will always fetch remote locations a cache block at a time, irrespective of the processor's memory accesses.

A more interesting example where the blocking can help in situations in which the access pattern is not easy to determine is a large parallel sample sort [BLM91, CDMS93] (radix sort behaves similarly). Each processor permutes a chunk of a global array into a number of buckets, one for each processor participating in the sort and the order of the writes is arbitrary such that they cannot be blocked statically. Each write, however, adds a value to a bucket, so collecting a few writes to each bucket before transmitting them reduces communication. Barring cache conflicts or capacity problems, the cache essentially collects a cache block worth of writes to each bucket and eventually sends them all at once. This example also illustrates the limits of CC-NUMA: for the blocking to work, the buckets must be laid out carefully. Cache conflicts must be avoided and the cache must be able to hold more cache blocks than there are buckets. In addition, if the total size of all buckets is larger than the cache (very likely in a large parallel sort) then communication will be triggered because a block has to be evicted. In this case it is sent back to the home node and therefore, to avoid unnecessary communication, each bucket better be allocated on the node which will read it. Another problem with cache coherency is that in algorithms where the communication can be analyzed it is very difficult to schedule the communication explicitly to avoid contention at memory modules.

The protocols introduced to maintain coherency require rather complex communication patterns: the DASH protocol requires that a cache be able to multicast an invalidation to an arbitrary set of nodes in response to a single request message and, in addition, all nodes receiving this invalidation must send a reply back to the node which issued the original request. Simpler protocols which exclusively use request/reply communication have been proposed [DCF+93] but they either require the programmer to control the mapping more carefully or they perform worse.

Another problem with cache coherency protocols is that they are not optimal in the number of messages. All communication is round-trip, it is not possible to send one-way messages. Furthermore, communication always involves the home node of a cache block. This means that to avoid three-way communication (with the associated additional latency and bandwidth) data structures must be mapped judiciously such that one of the communicating nodes is the home node.

Incrementality

To the first order, CC-NUMA communication architectures do not affect the processor for the same reason NUMA architectures do not: all communication is hidden in the memory system. This really relies on the fact that today all high-performance processors have an on-chip cache which supports bus-based coherency protocols. The problem with this argument, however, is that the long latency of cache misses does require changes in the processor or the efficiency on problems which intrinsically require communication will be low.

One solution explored in many architectures is to use prefetches and weak forms of write to allow multiple outstanding requests per processor. While these are simple to introduce into the processor (adding a few control bits to each load/store is sufficient) the additional complexity in the memory system is substantial: allowing multiple prefetches to occur simultaneously requires the use of a non-blocking cache. In the case of the KSR-1 cache-only shared memory multiprocessor the use of an off-the-shelf processor was deemed unacceptable and a custom processor was designed. While the custom processor has numerous novel features, its development has, as one would expect, lagged behind the performance curve of mainstream microprocessors.

Tolerating memory latency by introducing multithreading has been proposed as well [ALKK90, WG89]. From a performance point of view, the most promising approach is to switch to another thread of control when a cache miss occurs. This reduces the frequency of context switches which cost a few cycles. From an engineering point of view, however, this is not a simple approach [SBCvE90] because the cache miss is detected very late in the pipeline as well as very late in the clock cycle[9]. This means that at context switch time multiple instructions past the load/store have entered the pipeline and must be handled appropriately.

5.3.3.5 Implementation using Active Messages

Given the complexity of CC-NUMA shared memory and especially given the fact that a shared memory primitive (i.e., a memory access) can cause a large number of communication operations to take place, it seems appropriate to consider CC-NUMA shared memory to be a communication model and to search for a communication architecture appropriate for its implementation. Such a consideration is actually not new: the Alewife project started with very similar ideas and built the coherency protocol on top of a communication architecture which transfers cache lines and which allows events relevant to coherency protocols to be signalled to the processor [ALKK90]. More recently, the Wisconsin Windtunnel project developed a framework for experimenting with cache coherency protocols on a CM-5. The Windtunnel uses the CM-5 micro-architecture to implement a substrate which transfers cache-lines and signals events such as cache misses to the processor. This substrate is, in essence, a communication architecture and the project has built a number of shared memory communication models on top of this communication architecture.

Compared to Active Messages, the communication architectures implemented in Alewife and in the Windtunnel are special-purpose, i.e., tailored for a restricted set of communication models, and they have not been abstracted beyond these two specific implementations. Unfortunately, at the time these two projects started Active Messages was not available and it is therefore unknown whether using Active Messages would have been feasible.

What appears to be clear is that the functionality and performance of Active Messages is adequate to implement the communication functions required for the shared memory models implemented in the Windtunnel and in Alewife. However, an additional set of operations is required, in particular to trap memory accesses depending on the state of cache tag bits.

5.3.4 Summary

Shared memory seen as a communication model has a number of appealing characteristics. It is versatile in that it supports complex global data structures and with cache-coherence it promises to relieve the programmer from data placement concerns. It is efficient in that communication can be launched in a single instruction and special-purpose hardware is used transparently to format and handle messages. Finally, it is incremental because communication and the associated key issues are hidden in the memory system without affecting the processor design.

Proposing shared memory as a general-purpose communication architecture, however, bears a number of problems. Foremost is the lack of support for the transmission of events: shared memory is only concerned with transmitting data and the mechanisms used reflect this limitation. Signalling an event to a remote processor, an operation required to implement message passing and driven models, is ineffi-

9. The hit/miss signal from the cache is latched as late as possible, often after the data is latched and sometimes even part-way into the next clock cycle.

cient. Further problems are that shared memory does not support the type of one-way communication that other models excel at, that cache-coherency makes it difficult to exploit high-level knowledge of the communication patters intrinsic to algorithms, and that shared memory does not allow communication and computation to overlap easily. This means that it is not attractive to use shared memory as a communication architecture on which other communication models such as message passing, message driven, or dataflow can be implemented.

Some of the above problems can be alleviated by augmenting the basic model with prefetches, weak forms of write, and other similar constructs. At that point, however, many of the advantages of shared memory are compromised. In particular, most of these extensions require modifications to the processor and they complicate the shared memory hardware to the point where the specialized network interface is as complex as the processor and offers little performance benefit over other systems relying on software.This increase in complexity also raises questions regarding the interpretation of performance measurements: in many current cache-coherent multiprocessors the complexity of the cache controller is comparable to that of the processor and therefore, when comparing these machines with simpler designs, one should charge for two processors per shared memory node.

The fact that shared memory, in particular cache-coherent shared memory, requires elaborate communication protocols leads to viewing shared memory as a communication model which can be implemented using a general-purpose communication architecture. Other research projects have in fact taken this approach, except that they have designed communication architectures tailored for a variety of shared memory protocols but not necessarily appropriate for other communication models. A sample implementation of shared memory of top of Active Messages demonstrates that Active Messages supports the communication involved in shared memory protocols efficiently. However, additional functionality is necessary to provide the memory management support required by shared memory. One possibility is to provide hardware support for the address translation. Another possibility is to reduce the number of potentially remote accesses through compiler optimizations or language features. Subsection 6.1 describes Split-C which explores the latter possibility.

5.4 Conclusions

This chapter has examined three communication layers: message passing, message driven, and shared memory. These layers are traditionally perceived as communication architectures in the sense that they are considered to be "close to the hardware" and that they serve as compilation target for the implementation of high-level parallel languages. The inspection of each of these layers, however, reveals that they are not nearly as low-level as claimed: communication in each of them involves some combination of elaborate message protocols, general memory allocation, unbounded storage resources, associative matching, and sub-optimal communication patterns. The inclusion of these rather high-level functions in the base communication layer creates the danger of semantic clashes with high-level programming languages and indeed neither of the three models is particularly suited to the implementation of the others. All of these layers are therefore best classified as communication models which should be implemented on an appropriate general-purpose communication architecture.

The discussion of each traditional communication layer also examines its implementation using Active Messages. In the case of message passing the situation is simple: send&receive can be implemented just as efficiently using Active Messages as using ad-hoc methods. In the case of message driven models, Active Messages represent an efficient subset of these models, as witnessed by the J-Machine which essentially implements Active Messages, and the discussion whether Active Messages is as versatile as message driven models must be relegated to Section 6.2 which shows how the high-level languages for which message driven models were designed can be implemented more efficiently on Active Messages. In the case of shared memory, Active Messages is well suited to the implementation of the communication protocols but it must be complemented with additional mechanisms to support the memory management required by shared memory.