Anatomy of a Message in the Alewife Multiprocessor*

John Kubiatowicz and Anant Agarwal
Laboratory for Computer Science, NE43–633
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

Shared-memory provides a uniform and attractive mechanism for communication. For efficiency, it is often implemented with a layer of interpretive hardware on top of a message-passing communications network. This interpretive layer is responsible for data location, data movement, and cache coherence. It uses patterns of communication that benefit common programming styles, but which are only heuristics. This suggests that certain styles of communication may benefit from direct access to the underlying communications substrate. The Alewife machine, a shared-memory multiprocessor being built at MIT, provides such an interface. The interface is an integral part of the shared memory implementation and affords direct, user-level access to the network queues, supports an efficient DMA mechanism, and includes fast trap handling for message reception. This paper discusses the design and implementation of the Alewife message-passing interface and addresses the issues and advantages of using such an interface to complement hardware-synthesized shared memory.

1 Introduction

Given current trends in network and systems design, it should come as no surprise that most distributed shared-memory machines are built on top of an underlying message-passing substrate [1, 2, 3]. Architects of shared-memory machines often obscure this topology with a layer of hardware that implements their favorite memory coherence protocol and that insulates the processor entirely from the interconnection network. In such machines, communication between processing elements can occur *only* through the shared-memory abstraction. It seems natural, however, to expose the network directly to the processor, as shown in Figure 1, thereby gaining performance in situations for which the shared-memory paradigm is either unnecessary or inappropriate.

For example, consider a thread dispatch operation to a remote node. This operation requires a pointer to the thread's code and any arguments to be placed atomically on the task queue of another processor. The task queue resides in the portion of distributed memory associated with the remote processor. To do so via shared-memory,

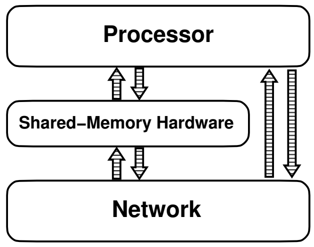

Figure 1: Alewife's integrated interface.

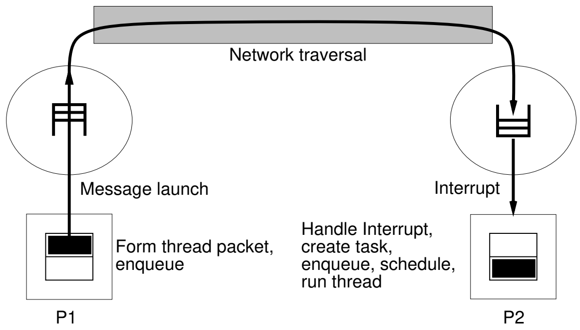

Figure 2: A thread dispatch to a remote node.

the invoking processor must first acquire the remote task queue lock, and then modify and unlock the queue using shared-memory reads and writes, each of which can require multiple network messages. As depicted in Figure 2, a message-based implementation is substantially simpler: all the information necessary to invoke the thread is marshalled into a single message which is unmarshalled and queued atomically by the receiving processor. In this manner, synchronization and data transfer are combined in a single message.

The message-passing implementation yields substantial performance gains over a pure shared-memory implementation. We characterize the performance of these two implementation schemes by measuring two intervals: $T_{invoker}$, the time from when the invoking processor begins the operation until it is free to proceed with other work, and $T_{invokee}$, the time from when the invoking processor begins the operation until the invoked thread begins running. With our best shared-memory implementation, these times are 10.7 and 24.4 μsec, respectively. With the message-based implementation, both times are reduced drastically, to 0.5 and 7.4 μsec, respectively. These numbers were derived from a cycle-by-cycle simulation of the Alewife machine, assuming a 33 MHz clock.

While supporting an efficient message interface is advantageous, we believe it is important to provide support for the shared-

* Appeared in the 7th ACM International Conference on Supercomputing. Tokyo, Japan. July 20-22, 1993.

memory abstraction because of the simplicity it affords those programs where communication patterns cannot be determined statically. Furthermore, shared-memory mechanisms are superior to message-passing mechanisms for operations that require fine-grain communication and computation.

Accordingly, Alewife provides support for both styles of communication. It implements support for coherent shared memory, which implies mechanisms for automatic renaming of memory locations on loads and stores (whether they are local in the cache, in local memory, or in remote memory), automatic local copying of data blocks and coherence checks.

Alewife integrates direct network access with the shared-memory framework. The message-passing mechanisms include direct processor access via loads and stores to the input and output queues of the network, a DMA mechanism, and a fast trap mechanism for message handling. With the integrated interface in Alewife, a message can be sent with just a few user-level instructions. A processor receiving such a message will trap and respond either by rapidly executing a message handler or by queuing the message for later consideration. Scheduling and queuing decisions are made entirely in software.

The challenge in implementing such a streamlined interface to the interconnection network is to achieve a performance that rivals that of purely message passing machines, while at the same time coexisting with the shared memory hardware.

The integration of shared memory and message passing in Alewife is kept simple through a design discipline that provides a *single*, *uniform interface* to the interconnection network. This means that the shared memory protocol packets and the packets produced by the message passing facilities use the same format and the same network queues and hardware.

The message interface itself follows a similar design discipline: provide a single, uniform communications interface between the processor and the interconnection network. This uniformity is achieved by using a single packet format and by treating all message packets destined to the processor in a uniform way. Specifically, all (non-protocol) messages interrupt the processor. The processor looks at the packet header and initiates an action based on the header. Possible actions include consuming the data into its registers directly, or issuing a command to storeback data via DMA.

The major contributions of this work include: (1) the design of the streamlined and uniform packet interface into the interconnection network, and (2) the mechanisms used to support its integration with the shared-memory hardware. The mechanisms required to integrate the message passing interface with the shared memory hardware include support for coherence on message data, high-availability interrupts, and restrictions placed on message handlers.

This paper describes the design and implementation of the message-passing interface, focusing on the issues related to its integration with the shared-memory layer. Section 3 provides an overview of the message interface. Section 4 outlines the mechanisms provided and presents the rationale behind our design decisions. Section 5 focuses on the mechanisms needed to support the integration with shared memory, and Section 6 describes the opportunities afforded by an integrated interface. Section 7 highlights issues encountered during the implementation of the Alewife machine and discusses the status of this implementation. Section 8 presents empirical evidence of the benefits of implementing an integrated interface. Section 9 discusses related work and Section 10 presents the status of the design and summarizes the major points in this paper.

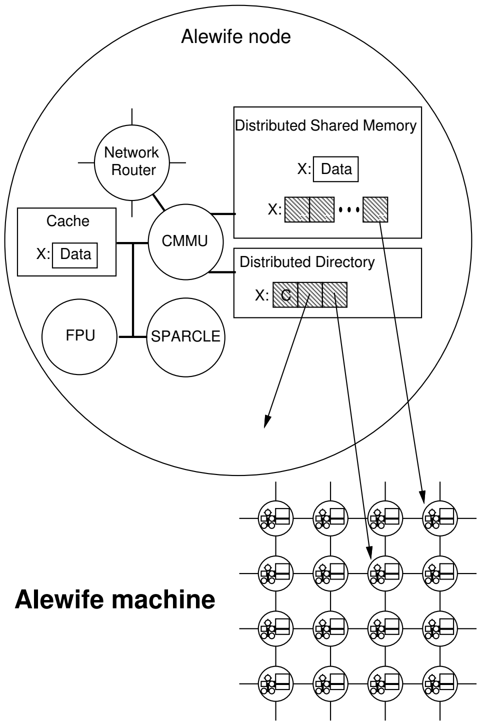

Figure 3: An Alewife processing node.

2 The Alewife Machine

Alewife is a large-scale multiprocessor with distributed shared memory. The machine, organized as shown in Figure 3, uses a cost-effective mesh network for communication. This type of architecture scales in terms of hardware cost and allows the exploitation of locality.

An Alewife node consists of a 33 MHz Sparcle processor, 64K bytes of direct-mapped cache, 4M bytes of globally-shared main memory, and a floating-point coprocessor. Both the cache and floating-point units are SPARC compatible [4]. The nodes communicate via messages through a direct network [5] with a mesh topology using wormhole routing [6]. A single-chip Communications and Memory Management Unit (CMMU) on each node holds the cache tags and implements the cache coherence protocol by synthesizing messages to other nodes. This chip also implements the message interface.

A unique feature of Alewife is its *LimitLESS* directory coherence protocol [7]. This scheme implements a full-map directory protocol [8] by trapping into software for widely shared data items. As discussed in Section 6.1, the Alewife message interface is necessary for implementing LimitLESS.

3 Overview of the Message Interface

Alewife's communications interface is unique for two reasons: first, it integrates message passing with a shared-memory interface, and second, the interface is highly efficient and uses a uniform packet structure. This section provides an overview of this interface and discusses the rationale behind the design.

3.1 A Uniform Message Interface

The message-passing interface in the Alewife machine is designed around four primary observations:

1. Header information for messages is often derived directly from processor registers at the source and, ideally, delivered directly to processor registers at the destination. Thus, efficient messaging facilities should permit direct transfer of information from registers to the network interface. Direct register-to-register transmission has been suggested by a number of architects [9, 10, 11, 12].

2. Blocks of data that reside in memory often accompany such header information. Consequently, efficient messaging facilities should allow direct memory access (DMA) mechanisms to be invoked inexpensively, possibly on multiple blocks of data. This is important for a number of reasons, including rapid task dispatch (where a task-frame or portion of the calling stack may be transmitted along with the continuation) [13] and distributed block I/O (where both a buffer-header structure and data may reside in memory).

3. Some modern processors, such as the Alewife's Sparcle processor [14], MOSAIC [15], and the MDP [10], can respond rapidly to interrupts. In particular, vectored interrupts permit dispatch directly to appropriate code segments, and reserved hardware contexts can remove the need for saving and restoring registers in interrupt handlers. This couples with efficient DMA to provide another advantage: virtual queuing. Here, a thin layer of interrupt-driven operating system software can synthesize an arbitrary network queueing structure in software, although possibly at the cost of extra memory-to-memory copies.

4. Permitting compilers (or users) to generate network communications code can have a number of advantages [11, 9, 16], but requires user-level access to the message interface.

Accordingly, the Alewife machine provides a uniform network interface with the following features:

- Sending a message is an *atomic*, *user-level*, two-phase action: describe the message, then launch it. The sending processor describes a message by writing into coprocessor registers over the cache bus. The resulting descriptor contains either explicit data from registers, or address-length pairs for DMA-style transfers. Because multiple address-length pairs can be specified, the send can *gather* data from multiple memory regions.

- Message receipt is signalled with an interrupt to the receiving processor. Alternatively, the processor can mask interrupts and poll for message arrival. On entering a message reception handler, the processor examines the packet header and can take one of several actions depending on the header. Actions include discarding the message, transferring the message contents into processor registers, or instructing the CMMU to initiate a storeback of the data into one or more regions of memory (*scatter*).

- Mechanisms for *atomicity* and *protection* are provided to permit user and operating system functions to use the same network interface.

3.2 Integration of Shared Memory and Message Passing

Integration of message passing with shared memory is challenging because of their different semantics. The shared-memory interface (as depicted in Figure 1) accepts read and write requests from the processor and converts them into messages to other nodes if the desired data is neither present in the cache nor in the local memory of the requesting node. The Alewife memory system is sequentially consistent.

However, allowing the processor direct access to the interconnection network bypasses the shared-memory hardware, and permits the processor to transmit the contents of its registers or regions of memory to other processors. The direct transmission of memory data through messages interacts with the implicit transmission of (potentially the same) data through loads and stores. The interactions must be designed and specified in such a way that compilers and runtime systems can make use of the two classes of mechanism within the same application, while minimizing implementation complexity.

The following are the important issues that arise when integrating shared memory with message passing:

- The interaction between messages using DMA transfer and cache-coherence. Our solution (as discussed in detail in Section 5.3) guarantees local and remote coherence. This means that data at the source and destination are coherent with respect to local processors. If global coherence is desired, it can be achieved through a two-phase software process. Guaranteeing only local and remote coherence significantly reduces implementation complexity, while still optimizing for common-case operations.

- The need for *high-availability interrupts*. Consider a processor that has issued multiple shared-memory requests and is currently blocked waiting for the return of data. If non-shared-memory messages precede the arrival of the requested data and occupy the head of the message queue, then the processor must trap and dispose of these messages before it can make forward progress.

  To address the above issues, Alewife supports high-availability interrupts. This support allows the Alewife processor to service external messages in the middle of a pending load or store operation.

- Special restrictions on global accesses by message handlers. As discussed in Section 5.3, global shared-memory accesses

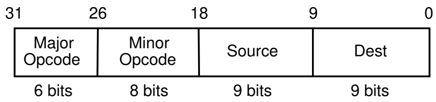

Figure 4: A uniform packet header format. The major opcode distinguishes between protocol, system, and user messages.

made by critical message handlers can lead to harmful interactions.

4 Mechanisms

This section describes the architectural mechanisms supported by Alewife's communications interface. A detailed programmer's interface is given in [17]. We first present the basic messaging facilities, then follow with a discussion of features necessary for user-level messaging.

4.1 Basic Messaging Facilities

Uniform Packet Header Format As a starting point for the integration of message-passing and shared memory, the Alewife machine employs a uniform packet structure for all classes of messages. All packets in the network must have the single, uniform header format shown in Figure 4 as their first word. The three packet classes - *coherence-protocol packets*, *system-level messages*, and *user-level messages* – are distinguished by ranges of the *major opcode*, also shown in this figure. The *minor opcode* contains unrestricted supplementary information, while the *source* and *destination* fields contain, respectively, the node-numbers for the source and destination processors. Only coherence-protocol packets are directly interpreted by hardware; such packets provide coherent shared-memory.

Output Interface Messages in the Alewife machine are sent through a two phase process: first *describe*, then *launch*. A message is described by writing directly to an array of registers in the CMMU, called the *output descriptor array*. Although this array is memory-mapped, the addresses fit into the offset field of a special store instruction called `stio`, as described in Table 1[1]. Consequently, the compiler can generate instructions which perform direct register-to-register moves from the processor into this array. These moves proceed at the speed of the *cache bus*.

Alewife packet descriptors for register-to-register or memory-to-memory transfers have the common structure shown in Figure 5 and consist of one or more 64-bit double-words. The descriptor consists of zero or more pairs of explicit *operands*, followed by zero or more *address-length pairs*. The address-length pairs describe blocks of data which will be fetched from memory via DMA; thus, requests for DMA are an inexpensive and integral aspect of packet description. Packet transmission begins with the operands and finishes with data from each of the requested blocks. The CMMU interprets the first operand (or first word in memory if no operands are present) as a packet header. The packet descriptor can be up to eight (8) double-words long.

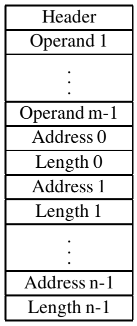

Figure 5: Packet descriptor format. The Header is Operand 0.

Once a packet has been described, it can be launched via an *atomic*, single-cycle, launch instruction, called `ipilaunch`. (IPI stands for interprocessor-interrupt). As shown in Table 1, the opcode fields of an `ipilaunch` specify the number of explicit operands (in double-words) and the total descriptor length (also in double-words). Consequently, the format of a packet must be known at compile-time. The execution of a launch instruction atomically commits the message to the network. Until the time of the launch, the description process can be aborted, or aborted and restarted without leaving partial packets in the network. After a launch, the descriptor array may be modified without affecting previous messages[2]. The `ipilaunch` and other instructions in Sparcle provide a tight coupling between the processor and the network.

Since requested DMA operations occur in parallel with processor execution, data blocks which are part of outgoing messages should not be modified until after the DMA mechanism has finished with them. Consequently, we provide a second flavor of launch instruction, `ipilaunchi`, which requests the generation of an interrupt as soon as all data has been retrieved from memory and committed to network queues. This *transmission completion* interrupt can be used to free outgoing data blocks or perform other post-transmission actions.

If the output network is blocked due to congestion, then it is possible that the CMMU has insufficient resources to launch the next message. This information is handed to the processor in one of two ways. First, the *space-avail* register in the CMMU indicates the maximum packet descriptor which can be generated at the time it is read. Second, if the processor attempts to store beyond this point in the descriptor array, then the offending `stio` instruction is blocked until resources are available[3]. Since the availability of resources is verified during the description process, launch instructions always complete.

Rather than blocking on insufficient descriptor resources, the software can optionally request that the CMMU generate a *space-request* interrupt when a specified number of double-words of descriptor space are available.

[1]Note that this store instruction differs from normal store instructions only in the value that it produces for the SPARC *alternate space indicator* (ASI) field [4].

[2]In fact, descriptor contents are not preserved across a launch. See Section 7.

[3]An `stio` that is blocked under such circumstances may be faulted during network overflow, as described later.

Instruction		Description
`ldio`	`Ra+AddrOffset, Rd`	Load from CMMU register space.
`stio`	`Rs, Ra+AddrOffset`	Store to CMMU register space.
`ipilaunch(i)`	`Numops, Length`	Launch packet from descriptor. Optionally, generate an interrupt when finished.
`ipicst(i)`	`Skip, Length`	Discard/storeback input packet. Optionally, generate an interrupt when finished.

Table 1: Network instructions

Interrupt	Event that it signals
Reception	Arrival of data. Separate vectors are provided for user and system packets.
Storeback	Latest `ipicsti` has completed.
Transmission	Latest `ipilaunchi` has completed.
Space Request	Requested descriptor space exists.
Network Overflow	Network output queue has been clogged for an extended period.

Table 2: Network Interrupts

Register	Description
`space-avail`	Output descriptor space available
`space-request`	Output descriptor space requested
`desc-length`	Current descriptor length
`storeback-addr`	Address for next DMA storeback
`window-length`	Number of double-words in input window
`net-ovf-cnt`	Network overflow count in cycles

Table 3: Network registers. Sizes are in double-words.

Table 2 lists the interrupts associated with the message interface. These interrupts are maskable and may be individually enabled or disabled. The processor can poll for disabled interrupts by examining a CMMU status register. Table 3 lists the CMMU registers that participate in message sending and receiving.

Input Interface Efficient receipt of messages and dispatching to appropriate handlers is facilitated by an efficient interrupt interface. Upon reception of the first double-word of a packet, the CMMU generates one of two *reception* interrupts, depending upon whether the message is a user message or a system/coherence message[4]. The processor can begin flushing its pipeline and vectoring to the interrupt handler in parallel with reception of the remainder of the message. Upon entering the interrupt handler, the processor can examine the first 8 double-words of the packet through the *packet input window*. As with the descriptor array, the packet input window is memory mapped with short addresses and accessed through a special load instruction, `ldio`. Consequently, the compiler can generate register-to-register moves from the input window to the processor registers that proceed at the speed of the cache bus. If the processor attempts to access data that is not yet present, then the CMMU will block the processor until this data arrives. Portions of the message that are outside the packet input window are invisible to the processor. If a packet is longer than eight (8) double-words, then only the first eight double-words appear in the window. The remainder of the packet is invisible to the processor, possibly stretching into the network.

Once the processor has examined the head of the packet, it invokes a single-cycle storeback instruction, called `ipicst` (for IPI coherent storeback). As shown in Table 1, this instruction has two opcode fields, *skip* and *length*. The skip field specifies the number of double-words that are discarded from the head of the packet, while the length field specifies the number of double-words (following those discarded) that should be stored to memory via DMA. Either of these fields can contain a reserved "infinity" value that denotes "until the end of the packet". When invoking DMA, the processor must write the starting address for DMA to the *storeback-address* register before issuing the storeback instruction. If the sum of the skip and length fields is shorter than the length of the packet, then the remainder of the packet will appear at the head of the packet input window and another reception interrupt will be generated. Multiple storeback instructions can be issued for a single input message to scatter its data to memory (the Alewife CMMU can permit two `ipicst` instructions to issue without blocking).

A second version of the storeback instruction, called `ipicsti` requests a *storeback completion* interrupt upon completion of the storeback operation. This signals the completion of input DMA, and can be used to export blocks of data to higher levels of software.

4.2 User-Level Messaging

There are numerous advantages to exporting a fast message interface to user code. The Alewife messaging interface has many aspects that can be directly exploited by the compiler, including direct construction of the descriptor and the format of packets themselves. This suggests that unique send and receive code might be generated for each type of communication, much in the flavor of active messages [11]. Further, when a message can be launched in less than ten cycles, the time to cross a protection barrier can easily double or triple the cost of sending that message.

In the Alewife machine, support for user-level messaging includes *atomicity* and *protection* mechanisms, which are described below:

Atomicity User code executes with interrupts enabled. This is necessary since interrupts represent time-critical events that may adversely affect forward-progress or correctness if not handled

[4] The LimitLESS coherence protocol invokes software in certain "rare" circumstances (such as wide spread sharing of read-only data). In these cases, coherence protocol messages are passed directly to the software.

promptly. Unfortunately, message construction represents a multi-cycle operation that can yield incorrect results if interrupted at the wrong time. Thus, some method of achieving atomicity is necessary.

One solution would be to allow the user to enable and disable interrupts. This is undesirable, however, since user code should not, in general, be allowed to perform actions that may crash or compromise the integrity of the machine. Alternately, we could provide separate output interfaces for the user and supervisor. This solution is also undesirable: on the one hand, it is overkill, since the chance that both the user and supervisor will attempt to send messages simultaneously is very low. On the other hand, the division between user and supervisor is somewhat arbitrary; we may have multiple levels of interrupts.

Consequently, the Alewife machine adopts a more general mechanism. It is designed with the assumption that collisions are rare, but that the highest priority interrupt should always have access to the network. To accomplish this, we start with an *atomic message-send*, as described above. Then, since message launching is atomic, interrupts are free to use the network providing that they restore any partially-constructed message descriptors before returning. Thus, the mechanism is the familiar "callee-saves" mechanism applied to interrupts.

Since the implementation described in Section 7 does not guarantee the contents of the descriptor array after launch, one additional mechanism is provided. This is the *desc-length* register of the CMMU. Whenever the output descriptor array is written, desc-length is set to the maximum of its current value and the array index that is being written. It is zeroed whenever a packet is launched. Consequently, this register indicates the number of entries in the descriptor array that must be preserved. It is non-zero only during periods in which packets are being described.

Protection Historically, there has been tension between protection mechanisms and rapid access to hardware facilities. The Alewife network interface is no different. Protection in the Alewife machine is not intended to hide information, but rather to protect the machine from errant user-code. Such protection is as follows:

- The user is not allowed to send system or coherence messages. To enforce this restriction, we require the user to construct messages with explicit headers (i.e. one or more operands). In this fashion, the opcode can be checked at the time of launch. If a violation occurs, then the `ipilaunch` instruction is faulted.
- The user is not allowed to issue storeback instructions if the message at the head of the queue is a system or coherence message.
- The user is not allowed to store data into kernel space. This rule is enforced by checking the storeback address register at the time that an `ipicst` is issued.

These protection mechanisms are transparent to both user and operating system under normal circumstances[5].

[5] Along the same lines, message hardware in a multiuser system could automatically append the current process identifier (PID) to outgoing packets. At the destination, either hardware or interrupt software could then check this PID and deliver messages to an appropriate user message handler. This is beyond the scope of this paper, however.

5 Interactions Between Shared Memory and Message-Passing

In this section we present three issues that arise when integrating message-passing with cache-coherent shared memory. These issues are the need for high-availability interrupts, special restrictions on message handlers, and data coherence for the DMA mechanism. To some extent, these interactions arise from the fact that the network provides a single logical input and output port to the memory controller. While networks with multiple channels are possible to implement, they are invariably more expensive.

5.1 High-Availability Interrupts

The need for *high-availability interrupts* [18] arises because shared-memory introduces a dependence between instruction execution and and the interconnection network. "Normal" asynchronous interrupts, which occur only at instruction boundaries, are effectively disabled when the processor pipeline is frozen for a remote read or write request. Unfortunately, as shown in Figure 6, the requested data may never arrive if it is blocked behind other messages. This figure illustrates a situation in which the processor has issued multiple shared-memory requests and is currently blocked waiting for the return of data. Unfortunately, several non-shared-memory messages have entered the network input queue ahead of the desired response. Unless the processor traps and disposes of these messages, it will never receive its desired data. Thus, the successful completion of a spinning load or store to memory may require faulting the access in progress so that a network interrupt handler can dispose of the offending messages. The term *high-availability interrupt* is applied to such externally initiated pipeline interruptions.

High-availability interrupts introduce an associated problem: when a load or store is interrupted by a high-availability interrupt, it is possible for its data to arrive *and* to be invalidated while the interrupt handler is still executing. The original request must then be reissued when the interrupt finishes. In unfortunate situations, systematic thrashing can occur. This is part of a larger issue, namely the *window of vulnerability*, discussed in [18]. For a single-threaded processor, the simplest solution is to defer the invalidation until after the original load or store commits.

5.2 Restrictions on Message Handlers

A second issue is the interaction between message handlers and shared memory. When an interrupt handler is called in response to an incoming message, the interrupt code must be careful to ensure the following before accessing global-shared memory:

- The network overflow interrupt must be enabled. (The network overflow handling mechanism is discussed in the next section.)
- The input packet must be *completely* freed and network interrupts must be reenabled.
- There must be no active low-level hardware locks that will defer invalidations in the interrupted code.

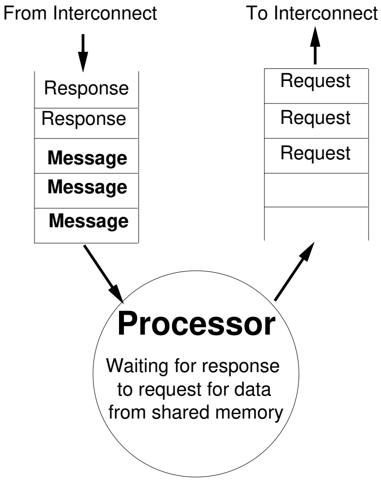

Figure 6: The need for high-availability interrupts.

The first of these conditions arises because all global accesses potentially require use of the network. Consequently, they can be blocked indefinitely if the network should overflow. The second arises for the same reason that high-availability interrupts were introduced into the picture: any global data that is accessed may be stuck behind other messages in the input queue. The last condition prevents deadlocks in the thrash elimination mechanism. See [18] for details.

5.3 Local Coherence for DMA

Since Alewife is a cache-coherent, shared-memory multiprocessor, it is natural to ask which form of data coherence should be supported by the DMA mechanism. Three possibilities present themselves:

1. Non-Coherent DMA: Data is taken directly from memory at the source and deposited directly to memory at the destination, regardless of the state of local or remote caches.
2. Locally-Coherent DMA: Data at the source and destination are coherent with respect to local processors. This means that source data is retrieved from the cache at the source, if necessary. It also means that destination memory lines are invalidated or updated in the cache at the destination, if necessary.
3. Globally-Coherent DMA: Data at the source and destination are coherent with respect to all nodes. Source memory lines that are dirty in the caches of remote nodes are fetched during transmission. Destination memory lines are invalidated or updated in caches which have copies, ensuring that all nodes have a coherent view of incoming data.

For reasons described below, the Alewife machine supports locally-coherent DMA.

Both locally-coherent DMA and non-coherent DMA have appeared on numerous uniprocessors to date. Local coherence gives the programmer more flexibility to send and receive data structures that are in active use, since it removes the need to explicitly flush data to memory. In addition, it is relatively straightforward to implement in a system that already supports a cache invalidation mechanism. Section 7.3 discusses this point. Non-coherent DMA can give better performance for messages that are not cached (such as certain types of I/O), since it does not produce cache invalidation traffic. However, it is less flexible.

Globally-coherent DMA, on the other hand, is an option only in cache-coherent multiprocessors. Globally-coherent DMA can assist in the improvement of locality by allowing globally-shared data items to be easily migrated between phases of a computation or during garbage-collection. This has a particular appeal for machines, such as Alewife, that physically distribute their global address space.

However, while attractive as a "universal mechanism," globally-coherent DMA is not a good mechanism to support directly in hardware. There are a number of reasons for this. First, it provides far more mechanism than is actually needed in many cases. As Section 8 demonstrates, message passing is useful as a way of *bypassing* the coherence protocol. In fact, many of the applications of message-passing discussed in [13] do not require a complete mechanism.

Second, a machine with a single network port cannot fetch dirty source data while in the middle of transmitting a larger packet since this requires the sending of messages. Even in a machine with multiple logical network ports, it is undesirable to retrieve dirty data in the middle of message transmission because the network resources associated with the message can be idle for multiple network round-trip times. Thus, a monolithic DMA mechanism would have to scan through the packet descriptor twice; once to collect data, and once to send data. This adds unnecessary complexity.

Third, globally-coherent DMA complicates network overflow recovery. While hardware can be designed to invalidate or to update remote caches during data arrival (using both input and output ports of the network simultaneously), this introduces a dependence between input and output queues that may prevent the simple "divert and relaunch" scheme described in Section 6.2 for network overflow recovery: input packets that are in the middle of a globally-coherent storeback block the input queue when the output queue is clogged.

In the light of these discussions, the Alewife machine supports a locally-coherent DMA mechanism.

Synthesizing Global Coherence We have argued above against a monolithic, globally-coherent DMA mechanism. However, globally-coherent DMA can be accomplished in other ways. The key is to note that software desiring such semantics can employ a two-phase "collect" and "send" operation at the source and a "clean" and "receive" operation at the destination.

Thus, a globally-coherent send can be accomplished by first scanning through the source data to collect values of outstanding dirty copies. Then a subsequent DMA send operation only needs to access local copies. With the send mechanism broken into these two pieces, we see that the the collection operation can potentially occur in parallel: by quickly scanning through the data and sending invalidations to all caches which have dirty copies.

At the destination, the cleaning operation is similar in flavor to collection. Here the goal of scanning through destination memory blocks is to invalidate all outstanding copies of memory lines before using them for DMA storeback. To this end, some method of marking blocks as "busy" until invalidation acknowledgments have returned is advantageous (and provided by Alewife); then, data can be stored to memory in parallel with invalidations.

It is an open question whether the collection and cleaning operations should be assisted by hardware, accomplished by performing multiple non-binding prefetch operations, or accomplished by scanning the coherence directories and manually sending invalidations[6]. If globally-coherent DMA operations are frequent, then a hardware assist is probably desirable. At this time, however, the Alewife machine provides no hardware assistance for these operations.

6 Opportunities From Integration

In this section, we touch on two unique opportunities, over and above the software advantages mentioned earlier, which arise from the inclusion of a fast message interface in a shared-memory multiprocessor. These are the LimitLESS cache-coherence protocol, and network overflow recovery.

6.1 The LimitLESS Cache Coherence Mechanism

One opportunity that arises from integrating message-passing and shared memory, is the ability to extend the hardware cache-coherence protocol in software. Permitting software to send and receive coherence-protocol packets requires no additional mechanism over and above the basic messaging facilities of Section 4. In Alewife, the memory system implements a set of pointers, called *directories*. Each directory keeps track of the cached copies of a corresponding memory line. In our current implementation, the size of the directory can be varied from zero to five pointers. The novel feature of the LimitLESS scheme [7] is that when there are more cached copies than there are pointers, the system traps the processor for software extension of the directory into main memory[7]. The processor can then implement an algorithm of its choice in software to handle this situation.

The LimitLESS scheme leaves ample opportunity for designing custom protocols which are invoked on a per-memory-line basis. Individual directories can be set to interrupt on all references. Then, all protocol messages which arrive for these memory-lines are automatically forwarded to the message input queue for software handling. In fact, our runtime system makes use of several extended applications of the LimitLESS interface, such as FIFO queue locks, fetch-and-op style synchronizations, and fast barriers.

6.2 Recovery from Network Overflow

Cache-coherence protocols introduce a dependence between the input and output queues of a memory controller, since they process read and write requests by returning data. This leads to a possibility for *protocol deadlock*, since it introduces a circular dependence between the network queues of two or more nodes. Architectures such as DASH [19] have avoided this problem by introducing independent networks for request and response messages. However, the existence of separate request and response networks is not sufficient in a general messaging environment since it can only prevent deadlock for a restricted class of request-response communication. Since protocol deadlock results from insufficient network queue space, we can correct this problem in a different fashion by *augmenting the hardware queues in software*, when a potential for deadlock is detected. We refer to this as *network overflow recovery*. Note that this technique does not require multiple logical networks.

The heuristic that we use to detect protocol deadlock is to initialize a hardware timer with a preset value. Then, whenever the network output queue is full and blocked, the timer begins counting down from the preset value, generating a *network-overflow* trap if it ever reaches zero. This counter affords some hysteresis for overflow detection, since protocol deadlock is a rare event and some queue blockage is expected.

The network overflow handler places the network in "divert mode," diverting all packets from the network *input* queue to the IPI input queue. It then uses DMA to store all incoming packets into a special queue-overflow region of local memory. This process continues until the network output queue has drained sufficiently (a controller status bit indicates that the output queue is half full). As a final phase of recovery, the diverted packets are relaunched with the IPI output interface. A low interrupt priority is used by the relaunch code, to permit normal message processing and network interrupts on relaunched packets.

Consequently, to permit network overflow recovery, we supplement the mechanisms of Section 4 with four additional mechanisms:

1. A countdown timer which can be used detect that the network output queue has been clogged for a "long" time.
2. The ability to force all incoming packets to be diverted to the IPI input queue, rather than being processed by the shared-memory controller. Note that we have only added the ability to force this switch. The data path must already be present to permit both shared memory and message passing to coexist. See Section 7.
3. A flag which indicates that the hardware output queue is empty or half full.
4. An internal loopback path from the IPI output mechanism back to the controller input, which permits packets to be relaunched to the hardware during recovery without routing through the network hardware.

Note that the fourth mechanism is not strictly necessary, but desirable since the network is backed up during network overflow processing. Section 7 shows a diagram of the network queue structure. A final requirement is more a design philosophy than anything else:

- All controller state machines must be designed such that they never attempt to start operations which have insufficient queue resources to complete. In this context, DMA requests are broken into a series of short atomic memory operations.

Adherence to this philosophy is simpler than attempting to abort operations during network overflow.

[6] An option that is uniquely available with Alewife.

[7] A trap bit in main memory associated with each block of data indicates whether accesses of this location must interrupt the processor for software extension.

Component	Size
Coprocessor Pipeline	1589 gates
Asynchronous Network Input	1126 gates
Asynchronous Network Output	1304 gates
Network Control and Datapath	2632 gates
Network Input Queue (1R/1W)	32×65 bits
Network Output Queue (1R/1W)	32×65 bits
IPI Input Queue (2R/1W)	8×64 bits
IPI Output (descriptor) Queue (2R/1W)	8×64 bits
IPI Output Local Queue (1R/1W)	8×65 bits
IPI Input Invalidation Queue (2 entries)	1610 gates
IPI Output Invalidation Queue (2 entries)	1306 gates
IPI Input Interface	4181 gates
IPI Output Interface	3393 gates

Table 4: Module sizes in gates or bits.

7 Implementation and Status

The Alewife message interface is implemented in two components, the Sparcle processor [20] and the A-1000 Communications and Memory Management Unit (CMMU). The Sparcle processor includes the special `ldio` and `stio` instructions for direct access to the IPI output descriptor queue and the IPI input window. In addition, a SPARC-like coprocessor interface enables pipelining of message sends and storeback requests. This processor has been operational since March 1992.

The A-1000 CMMU attaches to the cache bus of the Sparcle processor and integrates both coherent shared-memory and message-passing. It consumes roughly 80,000 gates and 100K-bits of SRAM (for cache tags and network queues) in the LEA 300K hybrid gate-array process from LSI Logic. Most of the logic is implemented in a high-level synthesis language provided by LSI and optimized with the Berkeley SIS tool set. By the time of publication, this chip will have gone to LSI for fabrication.

Table 4 gives a rough breakdown of the sizes of major components of the network interface. All data paths are 64 bits. Note that edge-triggered flip-flops are relatively expensive in this logic, at nine (9) gates per bit (this includes scan).

The Coprocessor Pipeline supplements the basic Sparcle instruction set with a number of instructions, including `ipilaunch` and `ipicst`. The IPI Input Interface and IPI Output Interface include all the launching, storeback, and DMA logic. The Invalidation queues permit locally coherent DMA and include the double-queue structures described below. Together, the logical components of the IPI interface (invalidation queues and interface logic) consume roughly 12% of the controller area. What these numbers do not indicate is the additional logic within the cache and memory controllers to support requests for invalidations and data from the DMA control mechanisms. The size of this logic is difficult to estimate, but appears to be a small fraction of the total logic for cache and memory control.

7.1 Network Queue Structure

Figure 7 shows the network queue structure of the Alewife CMMU. All datapaths are 64 bits. The *IPI output descriptor queue* and *IPI input queue* are visible to the processor and are mapped directly to the output descriptor array and packet input window of section 4.1. The remaining queues are internal queues and not visible to the programmer.

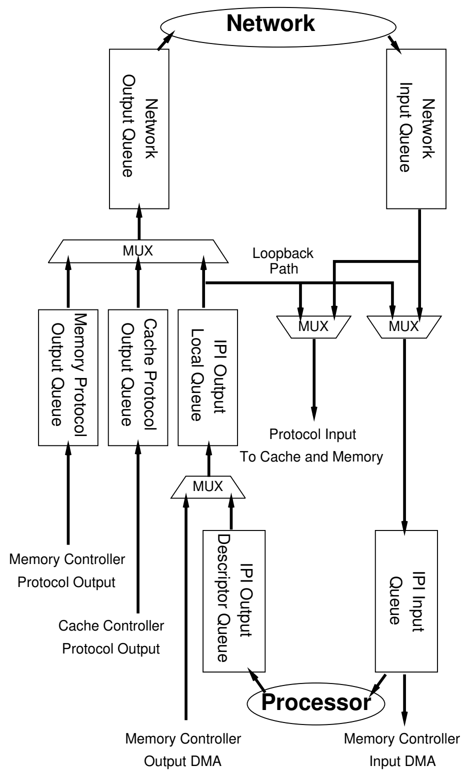

Figure 7: Network queue structure for the Alewife CMMU

7.2 Programmer Visible Queues

`Stio` operations to the descriptor array write *directly* into the circular output descriptor queue, using a hardware tail pointer as the base. The atomic action of an `ipilaunch` instruction moves this tail pointer and records information about the size and composition of the descriptor. Consequently, description of subsequent packets is allowed to begin immediately after an `ipilaunch`, provided that data from pending launches is not overwritten. Queue space becomes available again as descriptors are consumed by the IPI output mechanism. This implementation, while reasonably simple, does not preserve the contents of message descriptors after a launch has occurred.

As shown in Figure 8, messages are committed atomically in the writeback stage of the *ipilaunch* instruction. Here, message throughput is limited by the two-cycle latency of Sparcle stores and the lack of an instruction cache. More aggressive processor implementations would not suffer from these limitations. In addition, the use of DMA on message output adds additional cycles (not shown here) to the network pipeline.

`Ldio` operations from the packet input window receive data directly from the circular IPI input queue, using the current processor head pointer as a base. The effect of an `ipicst` instruction is

`stio`	`Header, N0`	I	D	E	M	M	W						
`stio`	`Data, N1`		I	—	—	D	E	M	M	W			
`ipilaunch`	`1, 1`			I	—	—	—	—	D	E	W	Q1	Q2
							I	—	—	D	E	W	
										I	D	E	W
	Cache Bus:	I	I	I	N0	N0	I	N1	N1	I	· · ·		

Figure 8: Pipelining for transmission of a message with a single data word. Sparcle pipeline stages are **I**nstruction fetch, **D**ecode, **E**xecute, **M**emory, and **W**riteback. Network messages are committed in the Writeback stage. Stages **Q1** and **Q2** are network queuing cycles. The message data begins to appear in the network after stage **Q2**.

to move this head pointer and to initiate DMA actions on the data which has been passed. A separate queue (not shown) holds issued *ipicst* instructions until they can be processed.

7.3 Local Coherence

As mentioned in the discussion on DMA coherence (Section 5.3), supporting locally-coherent DMA is straightforward in a machine with an invalidation-style cache-coherence protocol. In the CMMU, we coordinated the invalidation processes by using double-headed invalidation queues. The DMA controllers generate addresses and place *requests* on these queues as fast as possible (moving the tail of the queue). As soon as requests are written, the cache controller sees them, causes appropriate invalidations, and moves its head pointer. The memory has a second head pointer which lags behind the head pointer of the cache controller. Whenever the two pointers differ, the memory machine knows that it can satisfy the DMA request at its head, since the corresponding invalidation has already occurred. When the memory machine moves its head pointer beyond an entry, that entry is freed. Each of the invalidation queues have two, double-cache-line entries.

Care must be taken with the input interface so that the processor cannot re-request a memory-line after it has been invalidated but before the data has been written to memory. This situation can arise from the pipelining of DMA requests and would represent a violation of local coherence. The difference in area between the input and output invalidation queues (see Table 4) results from address-matching circuitry that serves as an interlock to prevent this "local coherence hole".

7.4 Network Overflow

The *memory protocol output queue* and the *cache protocol output queue* handle protocol traffic from the memory and cache, respectively. While such queues are important for performance reasons, they are also important for a more subtle reason: they simplify the checking of network resources. To allow network overflow recovery, controller state machines must check that all operations have sufficient resources to complete *before* they are initiated (Section 6.2). These two queues simplify this task, since resource checking can be done locally, without arbitrating for the output queue.

Note also that the style of network overflow recovery described in this paper requires the input and output DMA controllers to be independent of each other. This independence is necessary since we relaunch packets from memory (output DMA) which may require storeback operations during processing (input DMA).

8 Results

This section summarizes results comparing the performance of the shared-memory and integrated implementations of several library routines and applications; for a detailed discussion of our experience with this integrated interface see [13]. The results are obtained on a detailed, cycle-by-cycle machine simulator, using Alewife's run-time system based on *lazy task creation* [21] for load-balancing and dynamic partitioning of programs. Our results include a comparison of the performance of purely shared memory and integrated versions of several application kernels.

The applications include a thread scheduler with a synthetic fine-grain tree-structured application, barrier synchronization using combining trees, remote thread invocation, bulk data transfer, and Successive Over-Relaxation (SOR). Each application is implemented both using pure shared memory and using a hybrid system. The thread scheduler uses message sends when searching for work, migrating and invoking remote tasks, and performing certain types of synchronization. The applications run on 64 processors.

The hybrid version yielded substantial improvements over the shared-memory version for the thread scheduler. For very small grain sizes, the run time is almost twice as fast under the hybrid implementation. The barrier synchronization with message passing takes only 20 μsec, while the best shared memory version runs in 50 μsec. Comparable software implementations (e.g. Intel DELTA and iPSC/860, Kendall Square KSR1) take well over 400 μsec [2]. These numbers can also be compared with hardware-supported synchronization mechanisms, such as on the CM5, that take only 2 or 3 μsec but that require separate, log-structured (and potentially less-scalable) synchronization networks.

As described in Section 1, a remote thread invocation using messages reduces the invoker's overhead over a purely shared-memory implementation by a factor of 20 and that of the invokee by a factor of three. Memory-to-memory copy of data for 256-byte blocks is faster than shared-memory copy without prefetching by 2.4, and faster than shared-memory copy with prefetching by 1.5.

These results should not suggest that shared memory is unnecessary or expensive. For programs that have unpredictable, highly data-dependent access patterns, message passing implementations resort to implementing much of the shared-memory interpretive layer in software (for data location and data movement), with a corresponding loss in performance. In other cases, such as SOR, a simple block-partitioned Jacobi SOR solver, we observe little difference between well coded shared-memory and message-passing implementations.

9 Related Work

The CM5 provides a message passing interface and uses SPARCs as its processing nodes. The message interface is implemented using register reads and writes into the network interface (NI) chip. Because the reads and writes are implemented over the main memory bus, they are slower than network register reads and writes in Alewife, which are implemented over the processor cache bus. The CM5 interface does not provide support for DMA or shared memory and requires the processor to be involved in emptying out the message queue. The processor in the CM5 can be notified on message arrival either through an interrupt or by polling [22].

Our interface is different from that provided by the message passing J-machine [10] in that our processor is always interrupted on the arrival of a message, allowing the processor to examine the packet header, and to decide how to deal with the message. Messages in the J-machine are queued and handled in sequence. (The J-machine, however, allows a higher priority message to interrupt the processor.) The J-machine does not provide DMA transfer of data. Finally, message sends in Alewife are atomic in that correct execution is supported even if the processor is interrupted while writing into the network queue.

Somewhat in the flavor of the Alewife machine, the J-machine generates a *send fault* when the network output queue overflows. In addition, a *queue overflow fault* is generated when the input queue overflows. These faults can be used to trigger network overflow recovery similar to that of Section 6.2. Additionally, the J-machine network includes a second level of network priority which can be used to shuffle excess data to other nodes, should local memory for supplementary queue space be unavailable. Unfortunately, the J-machine mechanisms are extremely pessimistic, trapping as soon as local queue space is exhausted. In contrast, Alewife's network overflow mechanism provides hysteresis to ignore temporary network blockages. Further, the lack of message atomicity in the J-machine complicates the functionality of network overflow handlers.

Support for multiple models of computation has been identified as a promising direction for future research. For example, the iWarp [9] integrates systolic and message passing styles of communications. Their interface supports DMA-style communication for long packets typical in message passing systems, while at the same time supporting systolic processor-to-processor communication. In the latter style, a processor could be producing data and streaming it to another, while the receiving processor could be consuming the data using an interface that maps the network queue into a processor register.

To our knowledge, there are no existing machines that support both a shared-address space and a general fine-grain messaging interface in hardware. In some cases where we argue messages are better that shared-memory, such as the barrier in Section 8, a similar effect could be achieved by using shared-memory with a weaker consistency model. For example, the Dash multiprocessor [3, 23] has a mechanism to deposit a value from one processor's cache directly into the cache of another processor, avoiding cache coherence overhead. This mechanism might actually be faster than using a message because no interrupt occurs, but a message is much more general.

Some shared-memory machines have implemented message-like primitives in hardware. For example, Beck, Kasten, and Thakkar [24] describe the implementation of SLIC—a system link and interrupt controller chip—for use with the Sequent Balance system. Each SLIC chip is coupled with a processing node and communicates with the other SLIC chips on a special SLIC bus that is separate from the memory system bus. The SLIC chips help distribute signals such as interrupts and synchronization information to all processors in the system. Although similar in flavor to this kind of interface, the Alewife messaging interface is built to allow direct access to the same scalable interconnection network used by the shared-memory operations.

Another example of a shared-memory machine that also supports a message-like primitive is the BBN Butterfly. This machine provides both hardware support for block transfers and the ability to send remote "interrupt requests." Nodes in the Butterfly are able to initiate DMA operations for blocks of data which reside in remote nodes. In an implementation of distributed shared memory on this machine, Cox and Fowler [25] conclude that an effective block transfer mechanism was critical to performance. They argue that a mechanism that allows more concurrency between processing and block transfer would make a bigger impact. It turns out that Alewife's messages are implemented in a way that allows such concurrency when transferring large blocks of data. Furthermore, the Butterfly's block transfer mechanism is not suited for more general uses of fine-grain messaging because there is no support in the processor for fast message handling.

10 Conclusion

This paper discussed the design of a streamlined message interface that is integrated within a shared-memory substrate. The integration of message passing with shared memory introduces many interesting issues including the need for high-availability interrupts, the need for special restrictions on message handlers, and data coherence requirements for the DMA mechanism. An interface that addresses these needs has been implemented in the Alewife machine's CMMU.

The integration of message passing mechanisms with shared memory affords higher applications performance than either a pure message passing interface or a shared memory interface. In addition, it provides unique opportunities, over and above the software advantages of multimodel support, including the LimitLESS cache-coherence protocol and network overflow recovery.

11 Acknowledgments

Many thanks to David Chaiken for the cycle-by-cycle Alewife simulator and LimitLESS protocol design. Discussions with David Kranz and Beng-Hong Lim provided invaluable feedback on the convenience of some of the more subtle aspects of the network interface and user-level messaging. Further thanks to Beng-Hong Lim, Kirk Johnson, and David Kranz for the applications results and runtime systems work. Finally, many thanks to those involved with testing the A-1000 CMMU, in particular Ken Mackenzie, John Piscitello, David Kranz, Arthur Altman, Beng-Hong Lim, David Chaiken, Jonathan Babb, and Donald Yeung. Many people read earlier drafts of this paper; their feedback and criticism was much appreciated.

The research reported in this paper is funded by NSF grant # MIP-9012773 and DARPA contract # N00014-87-K-0825.

References

[1] Anant Agarwal, David Chaiken, Godfrey D'Souza, Kirk Johnson, David Kranz, John Kubiatowicz, Kiyoshi Kurihara, Beng-Hong Lim, Gino Maa, Dan Nussbaum, Mike Parkin, and Donald Yeung. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In *Proceedings of Workshop on Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1991. An extended version of this paper has been submitted for publication, and appears as MIT/LCS Memo TM-454, 1991.

[2] Thomas H. Dunigan. Kendall Square Multiprocessor: Early Experiences and Performance. Technical Report ORNL/TM-12065, Oak Ridge National Laboratory, March 1992.

[3] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford Dash Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[4] SPARC Architecture Manual, 1988. SUN Microsystems, Mountain View, California.

[5] Charles L. Seitz. Concurrent VLSI Architectures. *IEEE Transactions on Computers*, C-33(12):1247–1265, December 1984.

[6] William J. Dally. *A VLSI Architecture for Concurrent Data Structures*. Kluwer Academic Publishers, 1987.

[7] David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 224–234. ACM, April 1991.

[8] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 15th International Symposium on Computer Architecture*, New York, June 1988. IEEE.

[9] Shekhar Borkar et al. iWarp: An Integrated Solution to High-Speed Parallel Computing. In *Proceedings of Supercomputing '88*, November 1988.

[10] William J. Dally et al. The J-Machine: A Fine-Grain Concurrent Computer. In *Proceedings of the IFIP (International Federation for Information Processing), 11th World Congress*, pages 1147–1153, New York, 1989. Elsevier Science Publishing.

[11] Thorsten von Eicken, David Culler, Seth Goldstein, and Klaus Schauser. Active messages: A mechanism for integrated communication and computation. In *19th International Symposium on Computer Architecture*, May 1992.

[12] Dana S. Henry and Christopher F. Joerg. A Tightly-Coupled Processor-Network Interface. In *Fifth Internataional Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, Boston, October 1992. ACM.

[13] David Kranz, Kirk Johnson, Anant Agarwal, John Kubiatowicz, and Beng-Hong Lim. Integrating Message-Passing and Shared-Memory; Early Experience. In *To appear in Proceedings of Practice and Principles of Parallel Programming (PPoPP) 1993*, New York, NY, May 1993. ACM. Also as MIT/LCS TM-478, January 1993.

[14] *MIT-SPARCLE Specification Version 1.1 (Preliminary)*. LSI Logic Corporation, Milpitas, CA 95035, 1990. Addendum to the 64811 specification.

[15] C.L. Seitz, N.J. Boden, J. Seizovic, and W.K. Su. The Design of the Caltech Mosaic C Multicomputer. In *Research on Integrated Systems Symposium Proceedings*, pages 1–22, Cambridge, MA, 1993. MIT Press.

[16] Mark D. Hill, James R. Larus, Steven K. Reinhardt, and David A. Wood. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, Boston, October 1992. ACM.

[17] John Kubiatowicz. User's Manual for the A-1000 Communications and Memory Management Unit. ALEWIFE Memo No. 19, Laboratory for Computer Science, Massachusetts Institute of Technology, January 1991.

[18] John Kubiatowicz, David Chaiken, and Anant Agarwal. Closing the Window of Vulnerability in Multiphase Memory Transactions. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pages 274–284, Boston, October 1992. ACM.

[19] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings 17th Annual International Symposium on Computer Architecture*, pages 148–159, New York, June 1990.

[20] Anant Agarwal, John Kubiatowicz, David Kranz, Beng-Hong Lim, Donald Yeung, Godfrey D'Souza, and Mike Parkin. Sparcle: An Evolutionary Processor Design for Multiprocessors. *To appear in IEEE Micro*, June 1993.

[21] E. Mohr, D. Kranz, and R. Halstead. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991.

[22] *The Connection Machine System: Programming the NI.* Thinking Machines Corporation, March 1992. Version 7.1.

[23] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings 17th Annual International Symposium on Computer Architecture*, New York, June 1990. IEEE.

[24] Bob Beck, Bob Kasten, and Shreekant Thakkar. VLSI Assist for a Multiprocessor. In *Proceedings Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, Washington, DC, October 1987. IEEE.

[25] A. Cox and R. Fowler. The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 32–44, December 1989. Also as a Univ. Rochester TR-263, May 1989.