# Integrated Shared-Memory and Message-Passing Communication in the Alewife Multiprocessor

by

John David Kubiatowicz

S.B., Massachusetts Institute of Technology (1987)
S.M., Massachusetts Institute of Technology (1993)

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1998



Signature of Author________________________________________

Department of Electrical Engineering and Computer Science
December 15, 1997

Certified by________________________________________

Anant Agarwal
Associate Professor of Computer Science and Electrical Engineering
Thesis Supervisor

Accepted by________________________________________

Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

# Integrated Shared-Memory and Message-Passing Communication in the Alewife Multiprocessor

by

John David Kubiatowicz

Submitted to the Department of Electrical Engineering and Computer Science
on December 15, 1997, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

## Abstract

To date, MIMD multiprocessors have been divided into two classes based on *hardware communication models*: those supporting *shared memory* and those supporting *message passing*. Breaking with tradition, this thesis argues that multiprocessors should integrate *both* communication mechanisms in a single hardware framework. Such integrated multiprocessors must address several architectural challenges that arise from integration. These challenges include the *User-Level Access problem*, the *Service-Interleaving problem*, and the *Protocol Deadlock problem*. The first involves *which* communication models are used for communication and *how* these models are accessed; the second involves avoiding *livelocks* and *deadlocks* introduced by multiple simultaneous streams of communication; and the third involves removing multi-node cycles in communication graphs. This thesis introduces these challenges and develops solutions in the context of Alewife, a large-scale multiprocessor. Solutions involve careful definition of communication semantics and interfaces to permit tradeoffs across the hardware/software boundary. Among other things, we will introduce the *User-Direct Messaging* model for message passing, the *transaction buffer* framework for preventing cache-line thrashing, and *two-case delivery* for avoiding protocol deadlock.

The Alewife prototype implements cache-coherent shared memory and user-level message passing in a single-chip *Communications and Memory Management Unit* (CMMU). The hardware mechanisms of the CMMU are coupled with a thin veneer of runtime software to support a uniform high-level communications interface. The CMMU employs a scalable cache-coherence scheme, functions with single-channel, bidirectional network, and directly supports up to 512 nodes. This thesis describes the design and implementation of the CMMU, associated processor-level interfaces, and runtime software. Included in our discussion is an implementation framework called *service coupling*, which permits efficient scheduling of highly contended resources (such as DRAM). This framework is well suited to integrated architectures.

To evaluate the efficacy of the Alewife design, this thesis presents results from an operating 32-node Alewife machine. These results include microbenchmarks, to focus on individual mechanisms, and macrobenchmarks, in the form of applications and kernels from SPLASH and NAS benchmark suits. The large suite of working programs and resulting performance numbers lead us to one of our primary conclusions, namely that *the integration of shared-memory and message-passing communication models is possible at a reasonable cost, and can be done with a level of efficiency that does not compromise either model.* We conclude by discussing the extent to which the lessons of Alewife can be applied to future multiprocessors.

**Keywords:** multiprocessor, shared memory, message passing, cache-coherence, Alewife machine

Thesis Supervisor: Anant Agarwal
Title: Associate Professor of Computer Science and Electrical Engineering

# Acknowledgments

As I sit here in Harvard Square at my favorite cafe (Au Bon Pain), I think back to the long and arduous path that I have taken. Today is unseasonably warm, even as trees are caught between the riotous colors of autumn and the stark lines of winter. The last 10 years have seen tremendous growth in my understanding as a general "systems" person, and I can only attribute this to the number of people who have given me "enough rope", as it were, to jump forward. A few stand out in my mind, however . . .

* * *

First and foremost, my advisor, Anant Agarwal has given me more space and opportunity than any student could ever hope to acquire during the course of many graduate degrees. Over the years, we have clashed over philosophy, agreed vehemently over architecture, and enjoyed a healthy mutual respect that began from the day I accidentally wandered into his office. I have yet to encounter someone with a stronger mystical connection with the universe, *i.e.* the ability to make a counterintuitive conclusion based on no data which, upon further investigation, proved to be true beyond a measure of doubt. I can only hope to develop some small fraction of his intuition in the years to come. To a mentor, friend, colleague, and (oh yeah) advisor, I offer my heartfelt thanks.

* * *

In the late eighties, Dan Geer offered me a position at Project Athena, based only on an interview. At the time, I was a brash new graduate of MIT with absolutely no experience writing applications under UNIX, much less delving into its depths. The resulting two years there gave me a strong understanding of operating systems that I would never have gained otherwise.

* * *

To my colleagues in the Alewife group, I offer my heartfelt thanks. Without qualification, I can say that I was honored to work with each and every member of the Alewife group. The original core group, consisting of David Chaiken, Kirk Johnson, David Kranz, Beng-Hong Lim, Gino Maa, Ken Mackenzie, Dan Nussbaum, and Donald Yeung, contributed to the Alewife machine in vast and varied ways. They wrote and rewrote software systems, simulators, and compilers until we possessed a first-class system; they designed everything from compiler primitives to hardware test jigs and power systems. It is a testament to the competence of the Alewife group as a whole that we booted the operating system and began executing code within days of receiving the first copies of the CMMU.

I will miss all of you and can only hope that I find a similarly dynamic group of students as I begin my career in academia.

The attribution of ideas in a project as large as Alewife is difficult at best. However, I would like to specifically thank David Chaiken, who was a major contributer to the architecture of Alewife.

I can only say that I miss our interactions — the two of us argued the CMMU into existence (sometimes rather loudly), and I would not have done it any other way. In a similar vein, Dan Nussbaum, my office-mate of many years, served as an priceless sounding board to weed out new and occasionally misdirected concepts for the Alewife system. David Kranz, our "master of all trades", provided continuity and backbone to our software efforts — from compilation, to operating systems, to device drivers, and back again. Finally, Ken Mackenzie was an important collaborator in the development of the UDM model of Chapter 2.

---

Anne McCarthy provided the wonderful cartoon illustrations that grace the major section headings. She also provided a level of support over the years that is without measure.

---

Finally, to my family who remained supportive during my long years in school: Thank you!

---

To the reader: The process of academic research is an exercise in revisionist-history — the act of cutting, pasting, and reframing the goals and motivations of a research project until it appears that participants possess supernatural levels of knowledge and insight. This document is, perhaps, no less misleading in that regard. In the following pages, I will speak about portions of the Alewife machine that were some of my central contributions; although I like to think that these contributions arose in orderly fashion, this belief is probably optimistic on my part. I leave it to the reader to make his or her own judgment.

Since large systems such as Alewife are the result of complex collaborations between many people, I will do my best to give credit where it is due. If I should fail to do so at someplace in this document, it is entirely unintentional and I offer my sincerest apologies in advance.

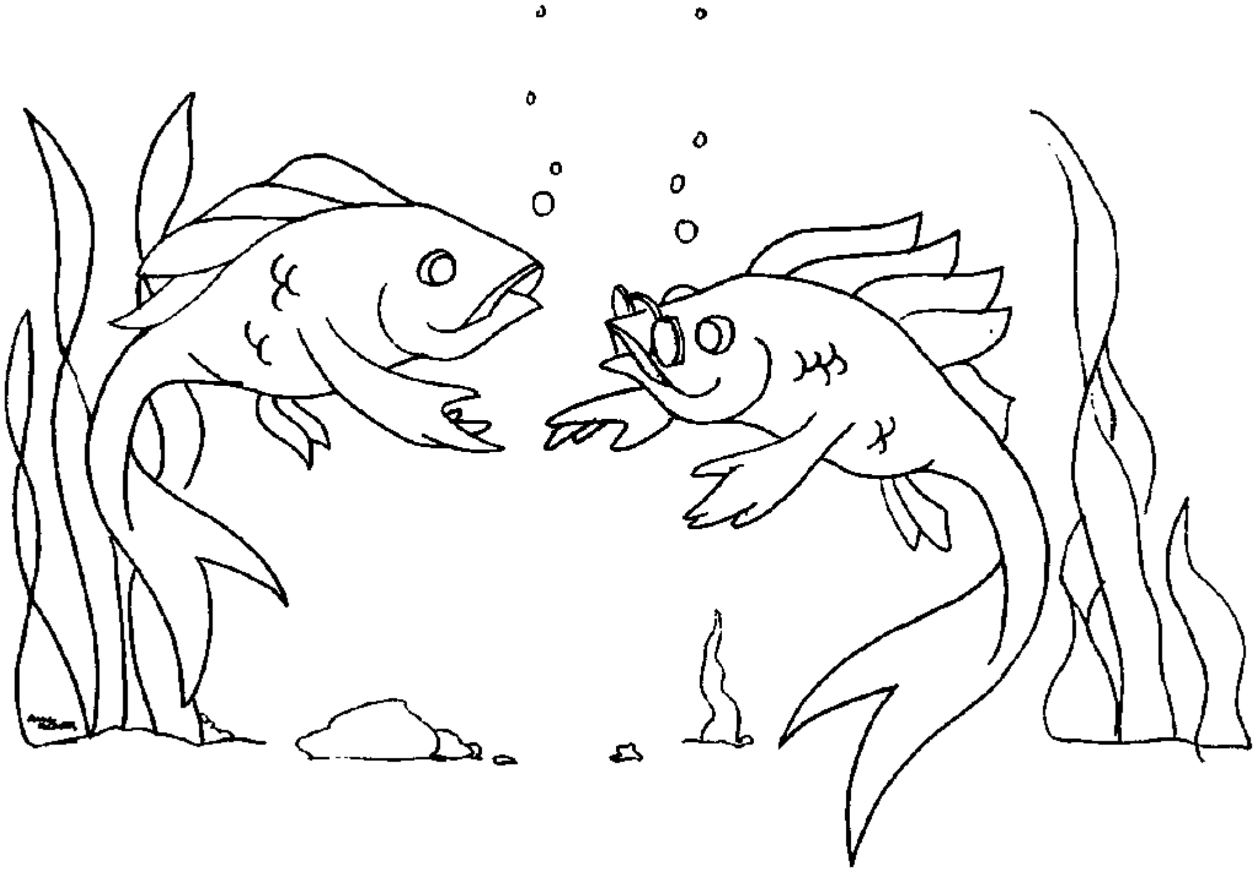
In the years that I worked on Alewife, I became somewhat attached to the project as a whole and the resulting prototype, warts and all. For good or bad, I touched upon all aspects of the project with my opinions and suggestions — always in search of improving the final result. Somewhat like a parent, I guess. Perhaps you can smile and nod as I tell you about Alewife.

# Contents

# List of Figures

# List of Tables

# Introduction



Early multiprocessor research was divided into two separate camps based on *hardware communication primitives*: those advocating *shared memory* and those advocating *message passing*. Supporters of shared memory pointed to the fact that a shared-address space programming model is easy to reason about; its location-independent communication semantics frees programmers from the burden of explicitly locating shared data and from orchestrating interprocessor communication. For performance, they argued, shared-memory should be implemented directly in hardware. Supporters of message passing, on the other hand, maintained that shared-memory hardware was unscalable and difficult to build; message-passing is much simpler to implement, they argued, and provides a "minimum" communication primitive upon which other communication models (including shared memory) can be constructed. In this early state of affairs, researchers adhered religiously to one or the other of the research camps, often ignoring the fact that each approach has advantages and disadvantages.

Breaking with this tradition, the following thesis argues that an *integration* of shared memory and message passing is desirable from both architectural and implementation standpoints. Since shared-memory and message-passing *communication models* have different applications domains in which they are ideally suited, a system that integrates them can exploit the advantages of each. Further, such an integrated system can be efficiently *implemented* by combining primitive shared-memory and message-passing hardware mechanisms with a thin veneer of runtime software.

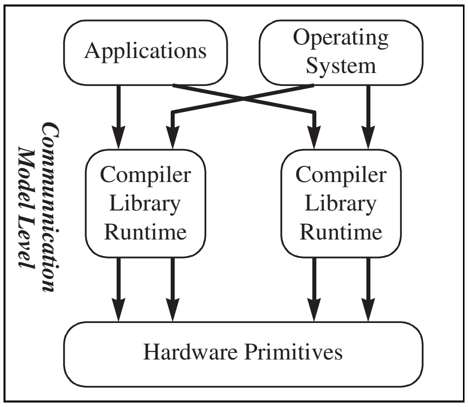
At another level, the architecture community has long struggled with the degree to which multiprocessor mechanisms should be integrated with memory and processor components. Proponents of least-effort "commodity parts" design methodologies attempt to bootstrap their multiprocessor design efforts by combining off-the-shell processor and memory components with a minimum of network coupling hardware. However, this does not necessarily represent the best implementation strategy. Low-latency, high-throughput access to data storage is of fundamental importance in uniprocessor systems, as demonstrated by the degree of engineering that is typically expended on uniprocessor memory systems design. *The memory system is no less important within a multiprocessor.* Further, well-designed uniprocessor memory systems share many features with multiprocessor memory systems: they are split phase and highly pipelined; they effectively process streams of requests from other levels of the memory system and respond with data. Thus, to exploit the similarity between uniprocessor and multiprocessor memory systems, this thesis argues for a tight integration of network and memory control, where integration here refers to single-chip integration.

In short, this thesis is *about* integration at many levels of a multiprocessor system: communications models, operating-systems, hardware. Themes of integration will arise time and time again, but will be moderated by concerns of practicality – avoiding capricious or excessive implementation skirmishes in favor of carefully chosen battles. The ultimate test of a system design is, of course, a working system. In this thesis, the vehicle for exploring integration and the embodiment of the ideas contained herein is the Alewife multiprocessor. Alewife integrates cache-coherent shared-memory and user-level message passing, employs a scalable cache-coherence scheme, and functions with a single, bidirectional network channel. Its implementation exhibits a constant per-node hardware cost in the form of a handful of VLSI chips, including the Sparcle processor and the Alewife *Communications and Memory Management Unit*(CMMU). The CMMU provides fast, efficient communication mechanisms and can directly support machine-sizes up to 512 nodes. From the standpoint of the rest of this thesis, however, the most important feature of the Alewife machine is that it *exists* and is being used by a number of researchers. A 32-node machine has been operational since June of 1994.

## 1.1 What are "Message Passing" and "Shared Memory"?

In motivating the integration of message passing and shared memory, it is first important to understand what we mean by these terms. In fact, these terms may be interpreted at two different levels of abstraction, that of *communications model* and that of *hardware mechanism*.

**Communications Models:** At the highest level of abstraction, the terms "shared memory" and "message passing" refer to communications models or paradigms that are directly interpreted by applications or systems programmers. The semantics of a given communications model can be thought of as a contract between the programmer and the underlying system, describing the behavior and guarantees that a programmer can expect, assuming that they fulfill their side of the bargain. In this context, the collection of semantics for shared memory are often referred to as the *memory model*; the pros and cons of various memory models form a large body of literature[1, 10, 41, 38]. The semantics of message passing paradigms have also been explored to some extent, but many of these models have deficiencies of one sort or another. In fact, in searching for a set of message passing semantics that are compatible with shared memory, we will spend a fair amount of time defining the *User-Direct Messaging* model in later parts of this thesis. Note that support for a complete communications model is provided by the system as a whole: an amalgam of hardware, runtime systems, libraries, and compiler software that synthesizes the model as presented to the programmer. The figure to the left shows this concept and serves to highlight the fact that some layers of software, namely those that are used to *implement* communication models, must operate outside the guarantees and semantics of any one model. Instead, this software makes use of raw hardware mechanisms that may present incomplete guarantees or partial communication functionality.




For the purpose of the current discussion, we will content ourselves with a brief summary of the predominant features of shared memory and message passing. We will elaborate on these in later portions of the thesis.

SHARED MEMORY: One fundamental feature of shared memory that all communication is *implicit*, via loads and stores to a global address space. The physical location of data is completely unspecified by the model, except in so far as some data may be considered more expensive to access than other data. To communicate information, the processor issues load and store operations; the underlying system is responsible for deciding whether or not data is cached and for locating data in the event that it is not cached. A second important feature of shared memory is that synchronization is distinct from communication: special synchronization mechanisms must be employed *in addition* to load and store operations in order to detect when data has been produced or consumed.

MESSAGE PASSING: In contrast, message passing is an *explicit* communication model; all communication consists of explicit messages from processors that have data to those that need data. Any communication between processors must be accomplished by such messages. Further the arrival of a message at a destination processor constitutes an asynchronous event, since software must be invoked to handle messages. Consequently, synchronization and communication are unified with message passing; the generation of remote, asynchronous events is an integral part of any message-passing communication model.

These two characterizations of shared memory and message passing will be sufficient for the upcoming discussion of the advantages of integration.

**Hardware Mechanisms:** Having defined what the terms "shared memory" and "message passing" mean in the context of communication models, we would now like to turn to another usage of these terms, to refer to individual hardware mechanisms. In this context, "shared memory" and "message passing" refer to these individual hardware mechanisms that have the "look-and-feel" of message passing or shared memory, but that may not provide all of the facilities of a complete communications model. Thus, a shared-memory hardware mechanism would likely support a load-store interface that handles the common case aspect of global access to cached data directly in hardware. Other aspects of global access, such as the location of data during a cache-miss or the tracking of widely shared data items might require software intervention. Similarly, a message-passing hardware mechanism might provide hardware facilities for launching and consuming messages, but might not provide guarantees of deadlock freedom or atomicity without software support.

## 1.2 Why Integrate Communication Semantics?

Given the clarifications of the previous section, we can restate one of our primary goals: this thesis will argue that an integration of message passing and shared memory is important *both* from the standpoint of communications model *and* hardware implementation. Before continuing this discussion, it is important to note that shared memory and message passing communication models are universal: each can be employed to simulate the other[1]. Thus the advantages to be attained via integration are not of the variety typically extolled by theorists: they have to do with slippery concepts such as communication overhead, implementation efficiency, and ease of programming.

Although the integration of shared memory and message passing has recently become a hot topic in the research community, this research has not made a clear distinction between the integration of communication models and implementation mechanisms. In an attempt to correct

[1]However, it is much easier to simulated shared memory with message passing than the converse. The primary difficulty involves simulating the asynchronous event semantics of message passing with the polling semantics of shared memory.

this situation, we first talk about the intrinsic advantages and disadvantages of each of the communication models. In a subsequent section, we will consider reasons for integrating hardware communication mechanisms.

**Advantages of the Shared Memory Model:** Conventional wisdom in the multiprocessor research community is that it is important to support a *shared-address space* or *shared-memory* communication model. One of the primary reasons for this is that shared memory shelters programmers from the details of interprocessor communication: the location-independent semantics of shared memory allow programmers to focus on issues of parallelism and correctness while completely ignoring issues of where data is and how to access it. This can allow the "quick construction" of algorithms that communicate implicitly through data structures. In some sense, the shared-memory communication model offers one of the simplest extensions of the uniprocessor programming paradigm to multiprocessors.

Further, because shared-memory semantics are independent of physical location, they are amenable to dynamic optimization by an underlying runtime or operating system. In particular, techniques such as caching, data migration, and data replication may be employed transparently to enhance locality and thus reduce latency of access. This is extremely powerful, because it means that data layout can be dynamically adjusted toward optimal, even though the programmer ignores it. As with any online algorithm, of course, the dynamic reordering mechanisms of the underlying system are unlikely to achieve a global optimum in data placement given an arbitrarily naive initial placement. In this sense, the implicit nature of the shared-memory programming model is both its principal strength and weakness: since placement *is* implicit, the programmer is not required to help with the data placement process (advantageous), leaving the underlying system the onerous task of extracting an ideal data placement given only a memory reference stream (unfortunate).

Note that this complete absence of information flow from application to underlying system is not inherent in the shared-memory model. For instance, when sufficient parallelism is identified by the programmer or compiler, then rapid context-switching can be employed to overlap latency. Similarly, latency tolerance can also be achieved when the compiler identifies access patterns in advance and inserts non-binding prefetch operations into the instruction stream. Both of these latency tolerance techniques can dynamically adjust for poor or non-ideal data placement. Further, as will be discussed in a moment, there is a lot of potential for the compiler to assist in choosing a good initial data placement as well as in choosing timely points for data migration. In fact, static analysis can occasionally eliminate the need for general cache-coherence, permitting the use of more efficient communication mechanisms such as message-passing. All of these techniques aid in the *implementation* of the shared-memory communication model, maintaining the same location-independent semantics for the programmer.

**Disadvantages of Shared Memory:** One of the deficiencies of the shared-memory communication model is the fact that it is, by nature, a polling interface. While this can make for extremely efficient communication under some circumstances, it can have a negative impact on synchronization operations. This has been enough of a concern that many multiprocessor architects have augmented the basic shared-memory communication model with additional synchronization mechanisms. Another disadvantage of shared memory is that every "real" communication operation (*i.e.*

one that causes data to cross the network) requires a complete network round-trip; no one-way communication of data is possible.

**Advantages of the Message Passing Model:** One of the biggest semantic advantages of the message-passing communication model is the fact that it is, by nature, interrupt driven. Messages combine both data and synchronization in a single unit. Furthermore, the efficiency of this combination can easily be maintained by simple implementations – we shall see this in later chapters of this thesis. In this section, we want to distinguish those applications of message passing that are natural from a programmers perspective from those that represent the result of automatic code generation in support of other communication models, *e.g.* shared memory.

Message passing provides an "interrupt-with-data", that is desirable for a number of *operating systems* activities in which communication patterns are explicitly known in advance: I/O, scheduling, task and data migration, and interprocessor interrupts. Further, manipulation of large data structures such as memory pages is ideally mediated with a messaging communications model, both because bulk transfer (DMA) is a natural adjunct to message passing (send a well-defined block of data from here to there) and because messaging operates "outside" of the shared-memory mechanisms that are being handled. This latter property is important when manipulating the data structures that shared memory depends on, such as global page mappings. Note that, although message passing requires explicit management of data locality and communication, this is not a disadvantage in an operating system, since most operating systems explicitly manage their own data structures anyway.

In addition to operating systems functions, certain applications are amenable to a message-passing communication model. Such applications have a large synchronization component and are typically referred to as "data-driven". Examples include event-driven simulation and solution of systems of sparse matrices by substitution. Also included in this category are programs written with *Active Messages*[113, 81], to the extent that Active Messages represents a desirable source-level communication model rather than a compilation target. In addition, message passing communication models are natural for client-server style decomposition, especially when communication must occur across protection domains. Note that the last decade has seen the emergence of a number of message-passing interfaces that support various polling semantics, such as *CMMD*, *p4*, and *MPI*. It is this author's belief, however, that these interfaces have arisen largely in reaction to the lack of good compilation and runtime-systems technology; they represent systems that grant users the ability (and corresponding need) for explicit management and tuning of communication[2].

**Disadvantages of Message Passing:** In addition to requiring the explicit management of data and communication, the message passing paradigm has one intrinsic disadvantage that can be mitigated but not eliminated by good interface design: the presence of higher endpoint costs in message passing models as compared to shared memory. The cost of assembling and disassembling messages, often called *marshaling cost*, is intrinsic to message passing because messages are, by

[2] One advantage to message passing that is being explicitly rejected here is the statement that it permits the easy porting of "dusty-deck" applications that have been written with message-passing interfaces.

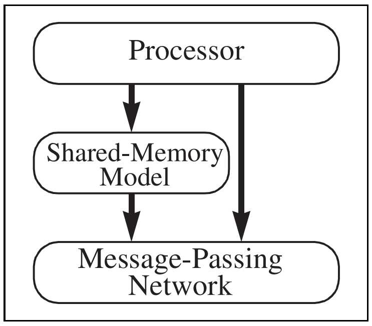
nature, transient and unassociated with computational data structures. At the source of a message, data is gathered together from memory based data structures and copied into the network. At the destination, the reverse must occur, namely data must be copied from the network into memory based data structures. This is an intrinsic cost of message passing and may be contrasted with shared memory, for which data remains associated with memory-based data structures at all times, even during communication.

It is important to note, however, that the marshaling cost of message passing can be less of an issue in comparison with shared memory when *implementation* and *communication patterns* are taken into consideration. For instance, communication patterns that exhibit poor cache locality and multiple round-trip network latencies can strongly degrade the performance of hardware-based shared-memory mechanisms. Further, certain classes of communication require copying anyway; page migration is a good example of this. In such cases, the marshaling cost in message passing is directly matched by copying costs in shared memory.

**Combined Semantics:** One obvious conclusion from the above discussion is that the shared-memory and message-passing communication models each have applications domains to which they are ideally suited. In fact, a mixture of the two may even be appropriate within different phases of a *single* application. A corollary to this conclusion is that *it is desirable for general purpose multiprocessors to support* both *communication models, and to do so efficiently*. Interestingly enough, this conclusion has not been uniformly embraced by multiprocessor architects or the research community. However, even skeptics should agree that this conclusion seems inevitable given the fact that shared memory is universally viewed as advantageous for applications writers, while the advantages of message passing to operating systems design is manifest and continues to be noted by systems architects. This author, however, maintains something stronger: integration is important at the applications level. This conclusion will become even clearer when we look at the way in which communication models at one level (say applications source) are implemented by mechanisms at lower levels (compilers, runtime system, and hardware).

## 1.3 Why Integrate Communication Mechanisms?

In the previous section we concluded that a multiprocessor system should provide both shared-memory and message-passing communication models. With this goal in mind, the question then becomes one of how to best implement such a multiprocessor. An extended answer to this question is, in fact, one of the primary topics of this thesis. A more simplistic answer to this question, however, is that we should provide a mixture of both shared-memory and message-passing communication mechanisms in hardware[58]. There are two reasons for this integration. The first, illustrated to the left, is one of opportunity: in striving for scalable performance, multiprocessor architects have abandoned buses in favor of generalized packet-switched networks to carry information between processors. As a consequence, the lowest level of communication (*i.e.* the physical transport layer)



carries messages, *even in a shared-memory multiprocessor*. By itself, this observation would seem to indicate that the most prudent course of implementation would be to provide message passing support directly in hardware and to emulate shared memory in software. However, this proposal leads us to the second reason for employing an integration of mechanisms: the fact that the shared-memory and message-passing communication models have sufficiently different properties that it is expensive to implement one entirely with the other. Let us expand on this further.

**Anatomy of a Shared-Memory Reference:** One of the primary virtues of the shared-memory communication model is its location independence. Such independence comes at a cost, however, especially when coupled with caching. During the execution of an application written with a shared-memory communication model, the actions indicated by pseudo-code in Figure 1-1 must be taken for *every* reference to shared data. This pseudo-code represents the process of discovering *where* data actually resides (local cache, local node, or remote node) and *retrieving* the data once it has been located. We will call this process *locality resolution*. The first test in this figure is only necessary if local caching is allowed; however, the second test is required for any implementation of a shared-memory communication model. Although locality resolution must occur for every shared reference, it may be performed at a variety of levels in the compiler, runtime system, or hardware[3]. Lets explore this a bit further.

```
shared-address-space-reference(location)
    if currently-cached?(location) then
        // satisfy request from cache
        load-from-cache(location)
    elsif is-local-address?(location) then
        // satisfy request from local memory
        load-from-local-memory(location)
    else
        // must load from remote memory; send remote
        // read request message and invoke any actions
        // required to maintain cache coherency
        load-from-remote-memory(location)
```

Figure 1-1: Anatomy of a Memory Reference.

For simplicity, assume that there are two types of shared-memory applications: *static* and *dynamic*. In static applications, the control flow of the program is essentially independent of the values of the data being manipulated. Many scientific programs fit into this category. Static applications have the virtue that some or all of the locality resolution code of Figure 1-1 can be performed by the compiler; no runtime overhead is incurred as a result. There has been a great deal of work in this area for scientific programs written in various dialects of FORTRAN and targeted at message-passing hardware [7, 19, 57, 74, 94, 121]. In dynamic applications, on the other hand, control flow is strongly dependent on the data being manipulated. Many symbolic applications fit into this category. Dynamic applications are not amenable to the same degree of compiler analysis as static applications; it is usually not possible to know *a priori* whether or not a particular reference will be to local or remote memory. Consequently, locality resolution must occur at runtime. Although most real programs will lie somewhere between these two extremes (by having some parts that are dynamic and others that are static), state-of-the-art compiler technology cannot yet reliably disambiguate communication styles[4]. As a result, most shared-memory applications end up performing locality resolution at runtime.

[3] Although these actions could also be coded explicitly by the programmer, we exclude that as an option here because such coding would represent use of a message-passing communication model.

[4] This is a situation that the author believes will continue to improve, however.

Hardware support for locality resolution is the essence of the distinction between shared-memory and message-passing hardware architectures. In the former, the instruction to reference memory is the same whether the object referenced happens to be in local or remote memory; the local/remote checks are facilitated by hardware support to determine whether a location has been cached (cache tags and comparison logic) or, if not, whether it resides in local or remote memory (local/remote address resolution logic). If the data is remote and not cached, a message will be sent to a remote node to access the data. Because shared-memory hardware provides direct support for detecting non-local requests and for sending a message to fetch remote data, a single instruction can be used to access *any* shared-address space location, regardless of whether it is already cached, resident in local memory, or resident in remote memory. Assuming that local shared data access is handled at hardware speeds, this has the important consequence that access to shared data can be extremely rapid (*i.e.* of the same speed as an access to unshared data) when data is cached or resident on the local node. Thus, shared-memory hardware enables dynamic locality optimization through caching and data migration.

As argued above, the handling of local cache-misses to shared data in hardware is desirable from the standpoint of permitting transparent locality optimization through data migration. There is another way to consider this. To the extent that "real" communication must cross node boundaries *and* traverse the network, access of local shared data does not involve communication. As a result, the locality resolution that accompanies such access is an artifact of the shared-memory communication model. By placing locality resolution in hardware (including the fetching of local shared data during a cache miss), shared-memory architectures avoid penalizing accesses that *would* be handled entirely in hardware for hand-optimized message-passing programs. Further, as will be shown later in this thesis, the difference between handling local and remote shared data access in hardware is insignificant, especially for a restricted class of sharing patterns[5]. This, in turn, means that real communication through shared-memory can be extremely efficient, easily rivaling or surpassing that of hand-tuned message-passing programs on fine-grained message-passing architectures. The hardware handling of remote shared-memory accesses has a hidden advantage: no need for asynchronous interrupt handling or polling to service requests from remote nodes (at least not in the "common" case). This can be extremely important on machines that have expensive interrupt handling.

In contrast to shared-memory architectures, message-passing hardware architectures do not provide hardware support for either local/remote checks or caching[6]. Hence, they cannot use a single instruction to access locations in the shared-address space. In the most general case, they must implement the locality resolution code of Figure 1-1 entirely in software. This increases the cost of accessing shared-data, even if it is present on the local node. In addition, implementation of the general shared-memory communication model on a message-passing architecture incurs an additional cost, that of handling asynchronous requests for data from remote nodes. This, in turn, forces the use of frequent polling or interrupts in order to ensure that nodes make reasonable forward progress.

A simple conclusion here is that message-passing architectures are insufficient platforms for

[5] *e.g.* handling limited read-sharing directly in hardware[23].

[6] The J-machine[87] provides a hardware associative memory that served as a cache on object locations. This approach merely accelerates the local/remote check of Figure 1-1; it does not eliminate it.

the high-performance implementation of shared-memory communication models. In the past, it was deemed that these deficiencies were unavoidable, given the difficulty of building large-scale shared-memory multiprocessors. However, machines such as DASH [69, 70], Alewife[3] (the topic of this thesis), and others, have shown that the mechanisms required to build scalable shared-memory multiprocessors are not of unreasonable complexity.

**Anatomy of a Message Transaction.** One advantage of the message-passing communication model is its integration of data and synchronization. Another is the direct nature of communication: data flows directly from source processor to destination processor, without unnecessary network crossings. One would hope that both of these properties would be directly retained by any implementation. Figure 1-2 illustrates the anatomy of a message communication in the context of the User-Direct Messaging model. This figure illustrates the four different phases of a message communication. The first of these, *message description*, involves describing the contents of the outgoing message to the communication layer; this is terminated by a message `launch` operation. The second, *network transit*, represents the time after the message is committed to the transport layer but before has been received. On reception, an interrupt is generated and a message handler is invoked as a result. The third phase of communication, *the atomic section*, is the period during which the message handler executes atomically with respect to other message handlers, *i.e.* with message interrupts disabled. Among other things, a handler must examine and free its message from the communication layer during the atomic section. The fourth and optional phase of message communication, termed the *global section*, allows the message handler to continue to execute as a first-class thread with interrupts enabled.

As shown in Figure 1-2, the message-passing communication model differs from shared memory in two important ways. First, it consumes minimal resources from the physical transport layer, requiring no more than a single network traversal from source to destination. In contrast, typical shared-memory cache-coherence protocols can require up to *four* network traversals for the simple transaction illustrated by Figure 1-2. This may seem surprising until one considers the fact that these protocols make use of communication *heuristics* which are optimized for good average behavior over a range of shared-memory communication patterns; consequently, they are unlikely to produce an optimum pattern of physical messages for all applications. Message-passing, on the other hand, provides access to the lowest-level primitives of the physical transport layer, thus permitting arbitrarily simple (or complex) communication patterns to be constructed, although at the cost of additional software overhead at the endpoints[7].

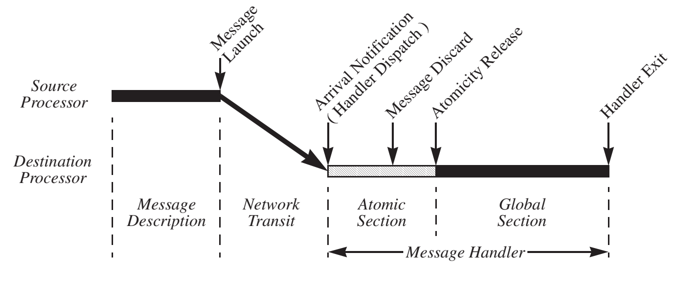


Figure 1-2: Anatomy of a User-Direct Message.

The second important way in which message passing differs from shared memory is that mes-

[7]Some researchers have attempted to acquire this same network efficiency by augmenting their cache-coherence protocols with *remote-write* capability, but this is not general and does not convey the synchronization advantages presented by a message-passing communication model.

sage passing involves an explicit interrupt model that permits users to receive message notification via interrupts. Such a user-level interrupt model requires an associated atomicity mechanism that permits users to disable interrupts without violating the integrity of the operating system; in Alewife, atomicity is provided by a *revocable interrupt disable* mechanism. Although we will discuss this mechanism later when we present User-Direct Messaging, the important point here is that atomicity is unnecessary for shared-memory communication models.

A third point that we advance here is the fact that typical shared-memory architectures do not provide support for large block transfers. Such transfers are desirable for a range of operations, from block I/O to data migration. Although support for block transfer can be included in a shared-memory architecture (in the form of hardware-accelerated memory-to-memory copy) this operation can be more efficiently accomplished via message-passing: the large quantity of data argues for efficiency at the transport level, *i.e.* direct transfer of data from source to destination (no extra network round-trips) via large network packets (reducing network routing overhead). Consequently, message-passing communication models ideally export DMA mechanisms which are not required for shared-memory.

Thus we have provided at least three reasons that a machine which provides only shared-memory is not entirely suitable for message-passing communication models: transport efficiency, atomicity, and block transfer.

**An Integration of Mechanisms.** As a result, one answer to the question of why to integrate shared-memory and message-passing *mechanisms* is that no one set of mechanisms is sufficient to implement both share-memory and message-passing *communication models*. This answer we have examined in the previous pages.

The integration of mechanisms can have additional benefits, however, in that communication models can be implemented more efficiently than might otherwise be possible. For instance, the presence of integrated mechanisms enables compiler transformations which analyze source code written in one communication model to emit instructions in a more appropriate communication model. The earlier discussion of compiler-based locality resolution is one example of this – static applications written with a shared-memory communication model can be compiled to produce binaries which utilize the message-passing communication model at runtime; for certain classes of applications, this may actually result in the shortest possible runtime.

Another, more concrete example of the benefits of integration involves Alewife's shared-memory communication model, called LimitLESS. The presence of hardware message-passing mechanisms relaxes the functionality required from the hardware shared-memory: these mechanisms can implement a restricted subset of the cache-coherence protocol[23] and can ignore the issues of protocol deadlock entirely[60]. We will explore these benefits in later portions of the thesis.

Figure 1-3 previews the various levels of hardware and software mechanisms present in the Alewife machine and forms a road map to the implementation portions of the thesis, since most of these mechanisms (as well as justification for this particular partitioning) will be described in detail later. For now, note that this figure shows four distinct levels of implementation, from the application level through to the hardware. Although the machine as a whole integrates both shared-memory and message-passing communication models, this diagram also highlights several

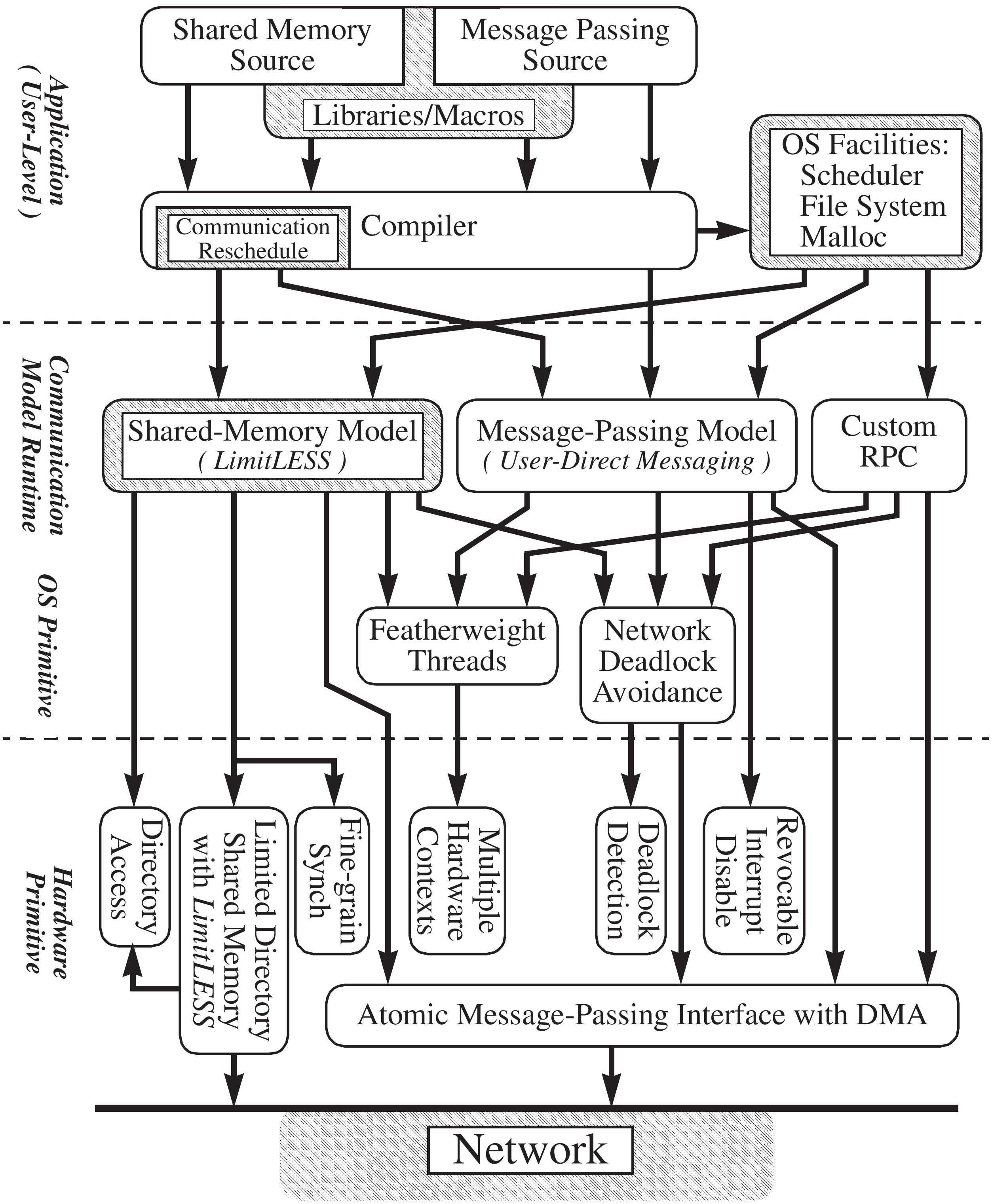



Figure 1-3: Integration of Communication Mechanisms in Alewife. Shaded boxes represent points at which Message Passing and Shared Memory are explicitly integrated.

point at which these models are entwined (shaded regions): special synchronization *libraries*[75] integrate both shared memory and message passing communication to exploit the best of each; the *compiler* attempts to extract an efficient mix of shared-memory and message-passing code from shared-memory source[8]; the LimitLESS cache coherence *model*[23, 24] explicitly integrates shared-memory and message-passing communication hardware mechanisms; finally, the *network* carries traffic for both shared-memory and message passing communication mechanisms.

## 1.4 Challenges To Integration

Having identified that integrated architectures are both interesting and desirable, we now consider some of the issues that complicate the design and implementation of a such architectures. At the level of design, we identify three major challenges to integration:

1. The User-Level Access Problem
2. The Service-Interleaving Problem
3. The Protocol Deadlock Problem

These are listed in decreasing order of architectural abstraction (or, alternatively, in decreasing order of impact on the user). These three issues will arise during much of the *Design* portion of this thesis. We want to take a few moments at this time to define these challenges and provide a brief outline of their solutions.

**The User-Level Access Problem:** Loosely stated, the *User-Level Access Problem* involves a desire to export hardware communication mechanisms directly to user-level while maintaining a clear distinction between system and user. This desire is motivated by the fact that the greatest combined performance and flexibility can often be achieved when user-code can interact directly with hardware mechanisms, without the intervention of system-level software. It reflects a strong belief in the integrated-systems approach that was the hallmark of the RISC revolution, *i.e.* a careful blending of primitive hardware mechanisms with a thin layer of runtime software and optimizing compiler technology. We intend to adopt this philosophy in a machine with integrated message passing and shared memory. Thus, to be more precise, our definition the User-Level Access Problem for an integrated architecture involves the search for a simultaneous, optimal solution to the following three questions:

1. What are the *semantics* of the communication models?
2. How do these models *interact* with one another?
3. What *interfaces* should the user employ to access communication mechanisms?

Our search for a simultaneous answer to these questions may be contrasted with less-optimal approaches that (1) choose a set of communication semantics without regard for implementation

[8]Note that this compiler is still in the early stages of exploration at this point.

complexity, or (2) which propose a set of hardware mechanisms without a well-defined user model, or (3) which construct hardware mechanisms without careful attention to interfaces.

One of the most desirable characteristics for a solution is that it achieves high performance without compromising protection or incurring unreasonable implementation complexity. Thus, we seek semantics that are just powerful enough to yield good and predictable performance – no more so. In guiding our search, we employ two rules of thumb:

1. View the set of hardware communication mechanisms as a "bag of tricks".
2. View communication interfaces as extending the instruction set architecture (ISA).

The first viewpoint states that communication operations that are used by the programmer are not directly reflected by hardware operations; rather, they are constructed by the compiler, library, or runtime system from a set of primitive hardware mechanisms ("bag of tricks"). This viewpoint is illustrated by Figure 1-3. It focuses attention on the construction of a minimal set of hardware mechanisms that satisfy our needs for correctness and performance. The second viewpoint advocates the direct export of hardware communication mechanisms to the user (via user-level instructions) while maintaining a clear distinction between the system and user. Philosophically, this approach reflects a view that communication is as fundamental as computation and should thus be reflected in the instruction set[9]. This viewpoint is not new (see, for instance, [116, 34, 14, 88, 89]); however, we seek an architecture which (1) integrates message passing and shared memory communication while (2) attempting to balance the performance of communication mechanisms against single thread performance and implementation complexity.

To briefly preview our result, the Alewife solution to the User-Level Access Problem embraces the following five hardware mechanisms:

1. *Sequentially consistent* shared memory with extensions for prefetching, fine-grained synchronization, and rapid context-switching.
2. A *single-user model* of the network which provides direct, user-level access to hardware queues while remaining virtualized for resource sharing and protection. Included as part of this mechanism is a user-level DMA mechanisms with *locally coherent* semantics.
3. *Multiple register sets* for fast thread creation, latency tolerance and rapid interrupt handling.
4. A formalized notion of *atomicity* which provides low-overhead, user-level control of message interrupts without violating protection.
5. A mechanism for detecting deadlock in the network.

Each of these mechanisms appears at the bottom of Figure 1-3.

[9]It should be noted that the process of adding user-level operations or instructions is often confused with the mechanism of interface/implementation. Consider, for instance, the progression of floating-point hardware from memory-mapped "coprocessor" interfaces to highly-integrated multi-issue floating-point pipelines.

Of the two communication models, message passing presents the greatest challenge for fast interfaces and protection. The shared-memory interface is well-established and easily protected with standard uniprocessor techniques. Further, shared-memory buries much of the details of communication at a level which the user is unable to influence: cache-coherence messages are of fixed size and well-defined composition, are typically restricted in quantity, and are consumed immediately at their destinations. In contrast, message passing can involve messages of arbitrary size and composition, may permit the network to be flooded, and may expose the system to deadlock since the user may refuse to consume messages.

**The Service Interleaving Problem:** The *Service Interleaving Problem* in an integrated architecture arises from the presence of uncontrolled simultaneity in an multiprocessing environment. This simultaneity arises from several sources: First, by their very nature, multiprocessors support multiple instruction streams (at least one per node), each of which can generate shared-memory communication traffic. Second, latency-tolerance techniques such as prefetching and rapid-context switching serve to overlap communication by increasing the number of outstanding requests from each node. Third, message-passing interfaces give rise to a potentially unbounded number of outstanding requests from each node. Hence, the memory system and network must correctly support many simultaneous requests with arbitrary ordering between them. Further, the boundaries between shared memory and message passing can be explicitly crossed when hardware events are handled by software (such as for the LimitLESS coherence protocol); this can introduce simultaneity in which cache-coherence structures are access by both hardware and software.

Hence, the Service-Interleaving Problem requires order amidst this chaos. It can be stated as requiring two guarantees:

1. All communication operations must complete eventually.
2. All communication operations must complete correctly.

The first statement asserts that the system is free of both livelock and deadlock. It requires the solution of three problems, the *refused-service deadlock*, the *window of vulnerability livelock*, and the *server-interlock problem*[10]. The refused-service deadlock refers to a form of priority inversion in which an asynchronous interrupt is deferred pending completion of a dependent operation or in which one class of communication (such as messages) prevents the completion of another class of communication (such as shared memory). The window of vulnerability livelock arises naturally in systems with split-phase memory transactions, when requested data can return to a node and be invalidated before it is accessed. Finally, the server-interlock problem is a memory-side dual to the window of vulnerability livelock and arises naturally in cache-coherence protocols. Ignoring this problem can lead to deadlock, while partial solutions can introduce livelocks, both of which can prevent the forward-progress of memory operations.

The second statement implies the existence of a cache-coherence protocol to maintain data consistency, and the presence of explicit mechanisms to deal with message and event reordering. Although all cache-coherence protocols deal explicitly with internode parallelism, such protocols

[10] Although the problem of protocol deadlock could also be construed as falling within the domain of the Service-Interleaving Problem, we have devoted split this off as a separate challenge – see Chapter 4.

may be sensitive to the reordering of messages in the network or communication subsystem[11]. Further, for systems such as Alewife that implement cache coherence via hybrid hardware/software schemes (*e.g.* LimitLESS), correct behavior requires mechanisms for the orderly hand-off of hardware events to software. We refer collectively to these issues as *protocol reordering sensitivities*.

Our solution to the Service Interleaving Problem restricts parallelism in problematic situations, as well as preventing pipeline interlocks from generating priority inversions. This solution can be summarized as follows:

1. Guarantee that high-priority asynchronous events can be delivered.
2. Track the status of outstanding memory transactions.
3. Forcing requesters to retry requests during interlock periods and guaranteeing that *at least one* writer eventually succeeds.
4. Provide locks on contended hardware data structures.

The first of these involves a centralized data structure called a *Transaction Buffer* which contains one entry for each outstanding memory transaction. This buffer supports sufficient associativity to defeat several different forms of livelock, including thrashing in the cache and inter-processor "ping-ponging" of shared data items. The second makes use of *high-availability interrupts* which are asynchronous interrupts that can be delivered synchronously in circumstances in which they would otherwise be ignored. The third avoids server-side queueing of requests in hardware while still permitting construction of software queueing locks (such as an MCS lock[84]) to guarantee forward progress on highly-contended memory locations. Finally, selective interlocking of hardware data structures (in particular cache-coherence directories) permits the atomicity normally exploited by hardware operations to be extended to software event handlers. Buried within this fourth solution is the presence of appropriate "channels" for delivery of hardware events to software, such as the notion of a *faultable flush queue* which can reflect hardware operations such as cache replacement back to software handlers[12].

**The Protocol Deadlock Problem:** Cache-coherence protocols introduce cycles into the physical topology of the network. The cycles arise because messages arriving in the input queue (e.g. shared-memory requests) cause the generation of messages on the output queue (e.g. data responses). This, combined with finite queue resources, leads to the possibility that two nodes could *deadlock* one another. While the communications patterns of shared memory are sufficiently constrained to permit deadlock to be avoided by removing cycles with virtual channels, the addition of general message passing hopelessly complicates the issue. The *Protocol Deadlock Problem* refers quite simply to the challenge of preventing protocol deadlock in an integrated architecture which supports both shared memory and message passing.

Our solution to the Protocol Deadlock Problem involves techniques that fall under the general heading of *two-case delivery*. Two-case deliver refers to on-demand software buffering of messages at their destination as a method for breaking deadlock cycles. Messages which are buffered

[11] In systems that implement two-case message delivery (see Chapter 4), messages may become reordered by the software recovery layers of software.

[12] In point of fact, the Transaction Buffer doubles as a faultable flush queue.

in this way are not dropped, but later delivered to the appropriate hardware or software consumer; hence the name "two-case delivery".

In the case of hardware-level deadlock (such as for the cache-coherence protocol), this solution is predicated on the position that actual protocol deadlock is rare. Rather than preventing deadlock, we employ heuristics for *detecting* and *recovering* from deadlock. This solution can be summarized in three parts as follows:

1. Detect potential deadlock by monitoring output queue for periods of congestion.
2. Divert *arriving* messages into local memory until the *output* queue is unclogged.
3. Relaunch diverted messages to the memory controller to permit processing.

Chapter 6 will present data to support the rarity of deadlock and the success of this methodology. For software-induced deadlock, a *user-level atomicity* mechanism detects that the network is not making forward progress and invokes buffering transparently.

## 1.5 The Alewife Prototype

To demonstrate the ideas presented earlier (and others), this author and his colleagues embarked on a six-year implementation effort, the result of which were several Alewife multiprocessor prototypes. An Alewife machine is organized as shown in Figure 1-4. Such a machine consists of multiple *nodes*, each of which has the equivalent computing power of a SPARCStation $1^{+}$. Memory is physically distributed over the processing nodes, and communication is via a packet-switched mesh network.

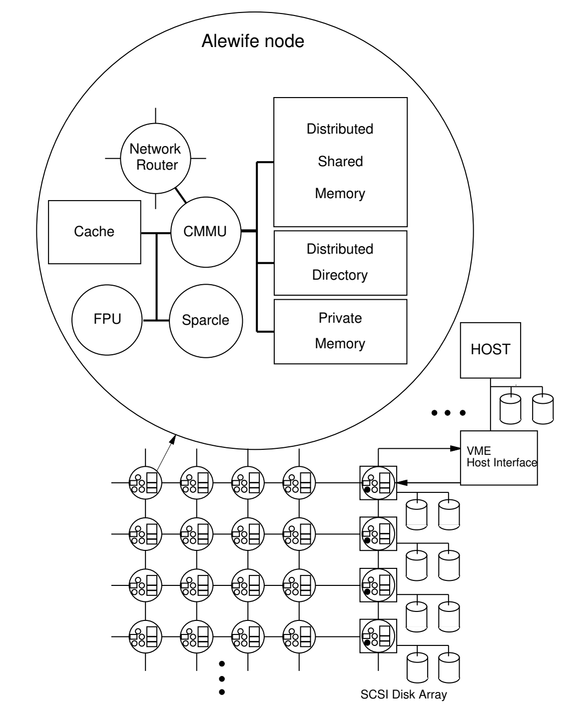


Figure 1-4: Node Diagram and Schematic for a 16-node Alewife Machine.

Each Alewife node consists of a Sparcle[4] processor, 64K bytes of direct-mapped cache, 4M bytes of shared data and 2M bytes of directory (to support a 4M byte portion of shared memory), 2M bytes of private (unshared) memory, a floating-point coprocessor, and an Elko-series mesh routing chip (EMRC) from Caltech. Both the cache memories and floating-point unit (FPU) are off-the-shelf, SPARC-compatible components. The EMRC network routers use wormhole routing and are connected to form a direct network with a mesh topology. The nodes communicate via messages through this network. A single-chip Communications and Memory Management Unit (CMMU) services data requests from the processor and network, as well as performing more mundane tasks such as DRAM refresh and control. I/O is provided by a SCSI disk array attached to the edges of the mesh.

All of the chips in the system are "off-the-shelf" components with the exception of the Sparcle processor and the CMMU. In fact, these two chips form the heart of the Alewife machine: the Sparcle processor, a modified SPARC processor, allows user instructions to be communicated efficiently to the CMMU, while the CMMU implements most of the experimental mechanisms, methodologies, and solutions of Alewife. To put it another way, the uniqueness of Sparcle lies in the *t*ransparency and *efficiency* with which it passes user-level instructions onto the CMMU, while the uniqueness of the CMMU lies in the set of *mechanisms* that it implements. Much could be made about this particular division of responsibility (and in fact [4] takes the implementation advantages of this process and attempts to elevate this to a science), but an equally strong position could be made for simply viewing the process as enhancing the instruction set — the Sparcle/CMMU tradeoff merely represented a convenient method of implementation. In fact, we will begin the first chapter of the Design portion of this thesis with just such an argument. The history of computer design has revealed a tendency for computer architects to begin any foray into a new computation niche by adding a host of new instructions — *i.e.* an initial pressure toward a "CISC" methodology, which later becomes streamlined as the effectiveness of compilers and new instructions becomes better understood. Alewife represents one such point on a foray into multiprocessing. Much attention will be given to Sparcle and the CMMU in later chapters.

Alewife as an academic project was unique in a number of ways, not the least of which was the work that the Alewife team put into packaging for Alewife machines[13]. With the assistance of Information Sciences Institute (ISI) at the University of Southern California, Alewife was designed with relatively compact and easily reproducible backplains and nodeboards. Two different packages were produced: a 16-node package which supported 4 I/O nodes and consumed relatively the same form-factor as a floor-standing workstation, and a 128-node package which supported 32 I/O nodes and filled a standard 19-inch rack. In addition to yielding a relatively neat *looking* machines[14], this served to greatly enhance the reliability of the Alewife asynchronous network[15]. Although I will not say much more about Alewife packaging, suffice it to say that the lessons of the Alewife project indicate that care with packaging can have as much to do with success as everything else.

## 1.6 Overview of Results

The most important thing that show in this thesis is that *the integration of message-passing and shared-memory communication models is possible at a reasonable cost, and can be done at a level of efficiency that does not compromise either model.* In short: integration is possible. To show this result, we will systematically tackle the three challenges of integration, then follow with an actual implementation (the Alewife prototype). In this sense, the solutions proposed during examination

[13]Ken Mackenzie was the driving force behind our packaging effort.

[14]There was plenty of appreciation of this, much to our delight, during the DARPA PI meeting which occurred shortly after we assembled a 16-node machine.

[15]The Stanford DASH multiprocessor used similiar network routing chips as Alewife, but had much more trouble with them. Much of this can be attributed to the relatively longer wires and less shielded packaging used for DASH.

of the three challenges are not simply theoretical, but rather lead to an actual implementation. The *User-Direct Messaging* model is one of the key results of the earlier chapters, providing an easily implemented model of messaging that is, none-the-less, extremely efficient. Two ancillary concepts to this messaging model are (1) the notion of a virtualizable *user-level atomicity* mechanism that is crucial to maintaining direct, user-level access to the network; and (2) the notion of *two-case delivery*, which is a "RISC-like" philosophy that handles exceptional network scenarios (such as deadlock) through software buffering, thereby freeing the hardware implementation to provide fast, common-case network abstractions. In fact, one of the results that we will present in Chapter 6 is that two-case delivery is an appropriate technique under a number of circumstances.

The Alewife prototype integrates shared-memory and message-passing communication models and achieves speedups that meet or beat those from many other multiprocessors (on both shared-memory and message-passing applications). This efficiency is achieved through careful construction of the communication models (as described earlier), and careful implementation. One of the key implementation methodologies that we propose is that of *service coupling*, a stream-based technique that separates manipulation of highly contended resources into separate blocks for *scheduling*, *execution*, and *routing*. The end result is that both shared-memory and message-passing communication mechanisms coexist *without compromising the speed of either type of mechanism*[16].

## 1.7 Overview of Thesis

In addition to this chapter (*Introduction*), the following thesis is divided into two major parts: *Design* and *Consequences*. Each of these parts is, in turn, divided into three individual chapters. The first, Design, explores each of the three challenges to integration introduced earlier: Chapter 2 discusses the User-Level Access problem, Chapter 3 examines the Service-Interleaving problem, and Chapter 4 attacks the Protocol Deadlock problem. The second part, Consequences, explores some of the implementation details of Alewife, as well as exploring performance results: Chapter 5 examines the hardware architecture of Alewife, Chapter 6 evaluates the performance of a 32-node prototype, and Chapter 7 discusses a number of the lessons of the Alewife project as well as touches upon related work.

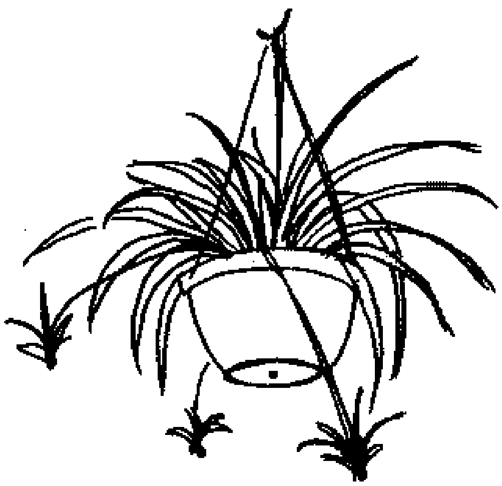
To the extent possible, the Design portion of this thesis is intended to discuss "high-level" or, perhaps, "protocol-level" issues, while the Consequence portion is intended to discuss implementation details and performance consequences. In the tradition of politicians everywhere, I take full credit for the extent to which this division is successful and deny all responsibility for infractions and failures in organization. The Consequences section is, hopefully, appropriately named. However, one of the most important consequences, the unexpected length of time consumed by the Alewife project (and the resulting loss of hair by this author), will not receive too much mention.

Without further ado, let the games begin!

[16]In fact, service coupling permits local and remote memory access to differ by only the pipelined cost of accessing the network.

# Part 1:

# Design



This section of the thesis is titled *D*esign, because it discusses several of the design issues behind the Alewife multiprocessor. In fact, it could equally be called *Challenges to Integration*, since there is one chapter for each of the three different integration challenges as outlined in the introduction. In this part of the thesis, we confront several of the high-level design issues that are confronted by system architects when building a system that integrates both shared-memory and message-passing communication models.

Chapter 2 talks about the User-Level Access Problem, which asks the question: how should communication mechanisms be exported to the user? In this chapter, we introduce our two primary communication models, namely *Latency-Tolerant Shared-Memory* and *User-Direct Messaging*, and the user-visible interactions between them. We also introduce an important threading model, called *featherweight threads*.

Next, Chapter 3 tackles several different interleaving issues which arise in the presence of multiple simultaneous communication streams and which can lead to livelock or deadlock. The *transaction buffer* arises in this chapter as an important centralized resource for tracking processor requests.

Finally, Chapter 4 addresses the problem of protocol deadlock arising from cyclic dependencies in the network. Several different solutions are explored, culminating in the Alewife solution of *two-case delivery*. The advantages and costs of two-case delivery are discussed in depth in this chapter.

All of these issues are approached from a high-level systems standpoint. It is important, however, to keep in mind that all of the high-level interfaces presented in the next three chapters are implemented in a real architecture, namely Alewife. To emphasize this, Part 2 of the thesis (*Consequences*), will examine the way in which our design choices impact the complexity and performance of the Alewife prototype.

# Chapter 2

# The User-Level Access Problem

In the introduction, we motivated the overriding goal of this thesis, namely the integration of message-passing and shared-memory communication. In this chapter, we begin this process by developing detailed *semantics* for the two communication models supported by Alewife, namely *Latency-Tolerant Shared Memory* and *User-Direct Messaging*. The importance of selecting appropriate semantics can not be understated. Among other things, the set of semantics that we choose to support has an important effect on both performance and complexity; ill-defined semantics may lead to fast but unusable communication mechanisms, whereas overly-defined semantics may require inefficient, restrictive, and expensive implementations. We will seek a happy medium. Closely entwined with the choice of semantics is the construction of a concrete set of *interfaces* through which the user interacts with the underlying system. The importance of interfaces lies in the ease with which they are used and implemented, both of which can directly affect performance.

Our current goal, then, is to define a consistent set of semantics and interfaces for the two communication models as well as specifications of the way in which these interact with one another. This complete specification will constitute our solution to the User-Level Access Problem. This task is complicated by the fact that the ultimate choice of semantics and interfaces is driven by a number of interwoven and competing concerns such as *functionality*, *protection*, *performance*, and *implementation complexity*. As a high-level view of this chapter, we will see that our solution to the User-Level Access problem consists of three major facets:

1. A threading model, called *featherweight threads*, whereby threads are inexpensive enough to use for latency tolerance and interrupt handlers.
2. Two communication models, called *Latency-Tolerant Shared Memory* and *User-Direct Messaging* that are compatible with one another, provide reasonable functionality, and whose interactions may be easily characterized.
3. A set of interfaces for these communication models.

With respect to the last of these, we will take the approach that *interfaces are extensions to the ISA*. This approach reflects view that communication is as important as computation and leads us to develop interfaces that can be represented in a small number of actual instructions.

Of the two communication models, it is message passing that presents the greatest challenges for fast interfaces and protection. The shared-memory interface is well-established and easily protected with standard uniprocessor techniques. In fact, our primary additions to "standard" shared-memory semantics have to do with latency tolerance and synchronization, *i.e.* the presence of atomicity operations for rapid context-switching, the presence of non-binding prefetch operations, and the presence of fine-grained synchronization operations attached to load and store requests. Shared memory buries much of the details of communication at a level which the user is unable to influence: cache-coherence messages are of fixed size and well-defined composition, are typically restricted in quantity, and are consumed immediately at their destinations. In contrast, user-level message passing can involve messages of arbitrary size and composition, may permit the network to be flooded, and may expose the system to deadlock since the user may refuse to consume messages. On top of this, the interaction of message-level DMA with shared memory introduces incoherence between data sent by DMA and data seen by the cache-coherence protocol. All of these issues must be addressed in a viable system; as we shall show, these problems can all be handled by choosing the proper messaging model. Unfortunately, previous models are insufficient for this task; thus, our User-Direct Messaging model represents a unique combination of features that meets our goals of flexibility, high-performance, and reasonable implementation cost. One of our key innovations is the notion of *two-case delivery*, which we will introduce in this chapter and explore in depth in Chapter 4.

The solutions that are discussed in this chapter consist of communication models and the interfaces that support them. Our resulting communication interfaces are, not surprisingly, described in the way that they are instantiated in the Alewife machine and, in particular, in the Alewife CMMU. We will be discussing implementation in greater detail in Chapter 5, but one aspect of implementation is important to keep in mind for now. The Alewife CMMU was actually implemented in two different versions (so called "A-1000" and "A-1001"). The first of these (complete with bugs) is the primary component in the machines on which most of the data of Part 3 was acquired. The second version of the CMMU contained updated interfaces and slight bug fixes. In particular, the full atomicity mechanism described in Section 4.3.5 was added in the A-1001 CMMU. Other slight differences are mentioned as appropriated in the text. Unfortunately, some electrical problems with the second version of the CMMU prevented wide-scale use of the A-1001 CMMU. None-the-less, the enhancements (with respect to messaging) provided by this chip are important enough that we will describe them anyway.

As a road map to this chapter, we start by discussing our design philosophy, namely that of discovering an "instruction-set" for multiprocessing. Then, in Section 2.2 we motivate the notion of *featherweight threads* which permit all pieces of a computation (including interrupt handlers) to be view as threads. Next, we discuss our two communication models, namely *Latency-Tolerant Shared Memory* (Section 2.3) and *User-Direct Messaging* (Section 2.4). Finally, in Section 2.5 we describe the interactions between these models.

## 2.1 Interface as an Extension to the ISA

No design process can operate entirely in a vacuum. Thus, we begin with a motivational bias, namely performance. The previous chapter defined the User-Level Access Problem as involving a desire to export hardware communication mechanisms directly to user-level while maintaining a clear distinction between system and user privileges. One of the principal motivations for identifying this problem is the observation that many systems that tout high performance hardware communication mechanisms never actually deliver this performance to applications. Poorly designed interfaces and the resulting inflation of communication-endpoint cost are two of the most insidious reason for such a disparity.

We start by viewing communication as fundamental — in a multiprocessor, communication is as important as computation. Thus, we will view our search for communication interfaces as a process of discovering the proper "communications instruction set", *i.e.* we start by asserting that interfaces are extensions to the ISA. The reasoning behind this viewpoint may be stated simply as:

- The finer the grain of communication which can be supported efficiently, the greater potential parallelism that can be exploited.
- Truly fine-grained communication requires primitives with low-endpoint costs.
- The lowest possible overhead occurs when operations can be launched at user-level via a small number of instructions.

It is a small step to realize that the collective set of communication operations form a type of communications instruction set. Before proceeding further, however, it is important to note that this viewpoint is nothing more than an attitude: it does *not* imply that we must be able to add instructions directly to the processor pipeline. However, if we view our design process as that of constructing an instruction set, then perhaps the resulting mechanisms will be simple enough to result in a low-overhead implementation, regardless of the methodology. Further, if the proper instruction set is discovered in this way, then it can serve as a template for inclusion in some future processor pipeline. It is the belief of this author that both of these goals have been met.

Our viewpoint of interfaces as extensions to the ISA is accompanied by another notion, namely that of the "bag of tricks". This notion states that it is not necessary for the communication operations used by the programmer to be reflected directly by hardware operations; rather, they are constructed by the compiler, library, or runtime system from a set of primitive hardware mechanisms. It is these primitive hardware mechanisms which are provided through the communications instruction set. This point of view seeks a minimal set of hardware operations that provide the most communications performance for the least cost.

As will be discussed in Chapter 5, the Alewife implementation involved enhancements to the processor so that user-level instructions could be passed directly to the memory controller (CMMU) and results could be passed efficiently back. For Alewife, this process was convenient from an implementation standpoint and contributed to the extremely low overhead of Alewife mechanisms. It was by no means *necessary*, however. All of the Alewife operations could have been accomplished via memory-mapped loads and stores (although at greater cost in endpoint time).

## 2.2 Featherweight Threads

One of the key philosophies of Alewife is the notion of the *integrated systems approach*. Simply stated, this recommends the implementation of common cases in hardware and uncommon or complicated cases in software. In Alewife, the functionality that is eligible for migration into software includes a number of asynchronous events, such as exceptional portions of the cache-coherence protocol (for LimitLESS cache coherence), handling of cache-misses (for latency tolerance), synchronization misses (for fine-grained synchronization), handling of messages (for complicated fetch-and-$\phi$ style operations), *etc.*. All of these usages involve invocation of software handlers via hardware interrupts or traps. Consequently, fast exception handling is extremely important.

One source of "pure overhead" in exception handling is the time required to save and restore the state of the interrupted computation. Low overhead entry and exit can be achieved, under some circumstances, through selective saving of processor registers or by choosing to save state only when complete thread generation is desired[114]; however, this is not a general solution and does not serve well for latency tolerance on cache-misses or for general user-level message handling. Instead, we approach the problem from another direction. Our goal will is to view *all* instruction streams, *including* exception handlers, as complete threads (with stacks and the ability to block). The question that we ask, then, is what is required to allow low-overhead generation and destruction of threads, as well as rapid switching between threads. The answer is a combination of hardware and software techniques that we call *featherweight threads*. Featherweight threads consist of the LIFO scheduling of context allocation in a block-multithreaded processor.

### 2.2.1 Block Multithreading

Let us elaborate on this for a moment. Although the actual implementation of featherweight threads will be discussed in more detail in the second part of the thesis, we will start with Figure 2-1 to indicate the primary hardware component of featherweight threads. This diagram shows a *block-multithreaded* processor[5] which contains four separate register sets with associated program counters and status registers. Each register set is a hardware context which can hold one active thread. All of the computational resources (such as ALUs, branch logic, and memory interfaces), are shared by the contexts; consequently, only one thread can be active at any one time. The active context is pointed to by a hardware register called the *context pointer* or CP. Conceptually, a hardware context-switch requires only that the context pointer be altered to point to another context; although this is a fairly low-overhead

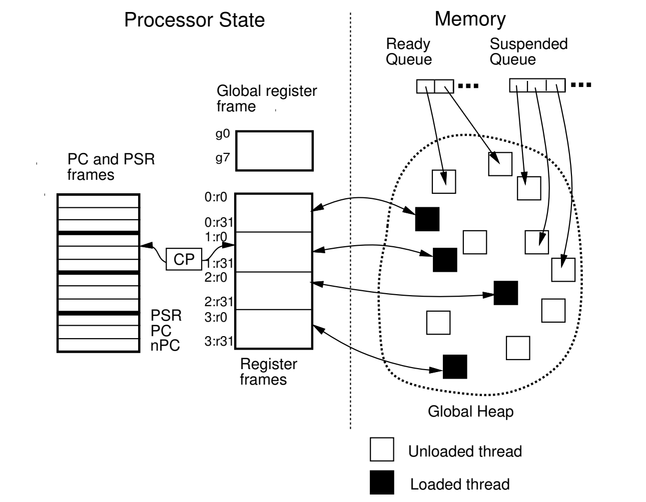



Figure 2-1: Block Multithreading and Virtual Threads

operation, we do not assume that it is instantaneous[1]. Thus, a block-multithreaded architecture ideally executes threads in quantum that are larger than a single cycle (*i.e.* threads are executed in "blocks" of time). This is one of the primary distinctions between a block-multithreaded processor and more hardware-intensive methods of multithreading such as present on MASA [44], HEP [102], or Monsoon [88]. These other architectures can switch between a large number of active "hardware" threads on a cycle-by-cycle basis.

### 2.2.2 Scheduling for Featherweight Threads

Figure 2-1 also shows four threads actively loaded in the processor. These four threads are part of a much larger set of *runnable* and *suspended* threads which are maintained by the operating system. Thus, these four threads can be considered an active cache on the set of runnable threads. In fact, by pursuing this notion of a cache, we can manage these contexts in such a way that empty contexts always contain an idle stack segment and task descriptor. Thus, if an empty context is available, the time to switch from a running thread and generate a new thread can be as short as 10–20 cycles. By scheduling contexts for high-priority thread creation in a LIFO fashion (which mirrors the scheduling of interrupt handlers), we can ensure that short but high-frequency exceptions encounter a free context with high probability (at the top of the "context" stack) and hence encounter very low-overhead thread creation and switching overheads: although a long-running thread may consume the last free context (thus penalizing the next short-running thread), this cost is only seen once, because the short-running thread will leave behind an empty context. This is the essential idea behind featherweight threads: we concentrate the advantages of fast thread creation to shorter threads (where it is needed most), while incurring a modest hardware cost[2].

Alewife implements featherweight threads. Thus, for the rest of this chapter, we will assume that all instructions streams (whether interrupt handlers or not) run as threads. As will be discussed in the next two sections, featherweight threads form the underpinnings of Alewife's latency tolerance techniques. In addition, the blurring between interrupt handlers and non-interrupt handlers afforded by featherweight threads will give us a lot of flexibility when we discuss the user-direct model: we will present a model of atomicity that affords a single-cycle transformation of a user-level interrupt-handler (running at high priority) in a background thread. For a glance at Alewife code that implements featherweight threads for user-level message handlers, see Appendix A (much of this will not make sense until the end of the chapter).

## 2.3 Latency-Tolerant Shared Memory

The programming model for all *hardware* shared-memory systems is a direct generalization of the uniprocessor memory interface: shared data is placed in a monolithic bank of memory that is accessible to all processors in the system through the load/store processor interface. Because of its close relationship to the uniprocessor memory model, shared memory is easily protected with

[1]In Alewife, a context-switch that occurs during a data cache-miss requires 14-cycles. See Section 5.1.2.

[2]In fact, as the implementation of the UltraSPARC demonstrated, multiple register sets are not an implementation bottleneck as had once been assumed[111].

```
load{φ_LSO}(address) ⇒ (result, full | empty)         // Load result from address
store{φ_SSO}(value, address) ⇒ full | empty           // Store value to address

rdpref(address)                                       // Prefetch read-only data at address
wrpref(address)                                       // Prefetch read-write data at address

enable_cswitch_atomicity() ⇒ disabled | enabled       // Prevent context-switching
disable_cswitch_atomicity() ⇒ disabled | enabled      // Allow context-switching
```

Figure 2-2: Latency-Tolerant Shared-Memory Model/Interface. Memory coherence is *sequentially consistent*. $\phi_{LSO}$ and $\phi_{SSO}$ stand for *load-synchronization operation* and *store-synchronization operation*, respectively. (Normal loads and stores are special cases of these more general mechanisms).

virtual memory[3]. Thus, when we formalize our shared-memory model, the load/store interface will be a prominent fixture, for most communication occurs through this interface.

Unfortunately, this simplistic viewpoint of memory becomes greatly complicated when the constraints of a realistic implementation are imposed. Large-scale systems (such as Alewife) must deal with the fact that processor to processor communication and processor to memory communication can take a non-trivial amount of time and may happen in a non-uniform fashion. The notion of a monolithic shared memory quickly falls before architectures which distribute memory among nodes and which exhibit locality of communication. The behavior of such "physically reasonable" systems is much more complicated than that of monolithic PRAM-like theoretical systems, but is better able to exploit locality for performance. Since the shared-memory model itself is well established, our main concern in this section will be defining a set of shared-memory semantics that give us maximum flexibility to tolerate latency and non-uniformities in execution.

Figure 2-2 details the operations that we include in our Latency-Tolerant Shared Memory communication model. Note that we are seeking a happy middle ground with our model: one that places enough functionality in hardware to achieve good performance without undue complexity. The set of operations supported in this model are simple enough that they may be directly implemented via hardware instructions; hence the model *is* the interface and vice versa[4]. The standard shared-memory load and store operations are augmented with fine-grained synchronization specifiers (*load-synchronization operations* ($\phi_{LSO}$) and *store-synchronization operations* ($\phi_{SSO}$) respectively); these specifiers will be discussed in Section 2.3.1. The shared-memory model is *sequentially consistent*, as will also be discussed in Section 2.3.1. Support for latency tolerance involves non-binding prefetch operations and context-switching atomicity operations; see Section 2.3.2. Although caching is a fundamental aspect of latency management, it is effectively transparent to the user; since the mechanisms for coherent caching are partially visible to the Alewife operating system, however, we will touch upon the implementation of caching in Section 2.3.4.

[3]Alewife does not provide virtual memory for a number of reasons. First, and foremost, at the time that the Alewife project was initiated, this particular author was concentrating on a number of other research issues, and virtual memory would have provided one two many things to deal with. Further, there were a number of tricky interactions between virtual memory and user-level DMA that the author was not prepared to deal with at the time. Solutions to these problems are discussed in more detail in [77].

[4]This is in contrast to the User-Direct Messaging communication model, discussed in Section 2.4, where model and interface must be introduced separately.

### 2.3.1 Memory Model and Synchronization

The first issue that we would like to touch upon briefly is the issue of shared-memory model and synchronization mechanisms. As multiprocessors scale in size, the grain size of parallel computations decreases to satisfy higher parallelism requirements. Computational grain size refers to the amount of computation between synchronization operations. Given a fixed problem size, the ability to utilize a larger number of processors to speed up a program is limited by the overhead of parallel and synchronization operations. Systems supporting fine-grain parallelism and synchronization attempt to minimize this overhead so as to allow parallel programs to achieve better speedups.

The challenge of supporting fine-grain computation is in implementing efficient parallelism and synchronization constructs without incurring extensive hardware cost, and without reducing coarse-grain performance. In Alewife, this is done by incorporating synchronization constructs directly into the cache/memory system: each memory word (of 32 bits) has an additional bit (called a full/empty bit [102, 5]) that is employed for synchronization. These full/empty bits are stored in memory and cached along with other data. To access these bits, Alewife attaches a synchronization operation field to each load and store operation. In Figure 2-2, these synchronization operations are shown as $\phi_{LSO}$ and $\phi_{SSO}$; Table 2-1 makes this more explicit by showing examples of Alewife load/store instructions and their corresponding effects on the full/empty bit. As shown in this table, the Alewife operations provide a test and set functionality in which the old value of the full/empty bit is loaded into a testable condition code before the value is set. With this functionality, any of a number of different possible synchronization operations may be constructed.

| Inst | Description |
|---|---|
| `ld` | Normal Load. Leave F/E alone. |
| `st` | Normal Store. 1 $\Rightarrow$ F/E |
| `ldn` | Load, F/E $\Rightarrow$ cond |
| `ldt` | Load, F/E $\Rightarrow$ cond, trap if empty |
| `lden` | Load, F/E $\Rightarrow$ cond, 0 $\Rightarrow$ F/E |
| `stn` | Store, F/E $\Rightarrow$ cond |
| `stt` | Store, F/E $\Rightarrow$ cond, trap if full |
| `stfn` | Store, F/E $\Rightarrow$ cond, 1 $\Rightarrow$ F/E |

Table 2-1: Examples of Alewife synchronizing loads and stores. “F/E” is full/empty bit. “cond” is the testable full/empty condition code.

**The Role of Memory Model:** Alewife supports a sequentially consistent memory model [66]. Among other things, sequential consistency provides one of the easiest memory models to reason about. Further, by providing sequentially consistent hardware shared-memory semantics, Alewife is able to integrate its synchronization mechanism directly into the normal access mechanism; full/empty bits provide good primitives to build a number of synchronization operations, but only if provided on top of a sequentially-consistent memory system. Other weaker memory models must add separate, out-of-band synchronization operations; for example, DASH [41] and Origin [67] provide *release* and *acquire* operations for synchronization; these operations act on locks in a separate “lock space”, that is independent of the shared-memory data space.

Over the years, researchers have developed a number of ways of programming and reasoning about machines that have consistency models that are weaker than sequential consistency[2, 42]. Further, one of the justifications for weaker memory models is that they provide better latency tolerance (under some circumstances) than sequential consistency. As a result, the arguments in favor of a sequentially-consistent memory model are perhaps not as compelling as they once were, except with respect to synchronization. Any further discussion of memory models and full/empty bits is outside the scope of this thesis.

### 2.3.2 Latency Tolerance

One of the single most important barriers to large-scale multiprocessing is memory latency. Since memory in large-scale systems is distributed, cache misses to remote locations may incur long latencies. This, in turn, reduces processor utilization, sometimes precipitously. The general class of solutions to latency tolerance all have an important common feature: they implement mechanisms for allowing multiple outstanding memory transactions and can be viewed as a way of pipelining the processor and the network. The key difference between this pipeline into the network and the processor's execution pipeline is that the latency associated with the communication pipeline is very unpredictable, making it difficult for a compiler to schedule operations for maximal resource utilization. To combat this, systems implement dynamic pipelines into the network, providing hardware to enforce correctness. As mentioned previously, much of the hardware required for sorting out multiple pending operations is already a part of cache coherence protocols.

Alewife provides three hardware mechanisms to tolerate long latencies: *coherent caching*, *non-binding prefetch*, and *rapid context-switching*. With respect to the user model, coherent caching is a transparent optimization (although we will have to revisit issues of fairness in Section 2.3.3). However, both non-binding prefetch and rapid context-switching introduce mechanisms into the user model; hence we will discuss them next.

**Non-binding Prefetch:** Non-binding prefetch operations act as "hints" to begin fetching remote data before it is needed. Non-binding prefetch affects the user model in a very simple way: it introduces a class of instructions that we will call *prefetch instructions*; Figure 2-2 lists the two different prefetch instructions included in Latency-Tolerant Shared Memory, namely `rdpref` (for prefetching data in a read-only state) and `wrpref` (for prefetching data in a read-write state).

To use one of these instructions, the compiler or programmer supplies an address, effectively predicting that data at that address will be needed sometime in the near future. The system then *attempts* to fetch data with the appropriate access type. This is called software prefetching, and has a body of literature behind it (see, for instance [85]). One of the unfortunate aspects of this interface is that instruction bandwidth is consumed for prefetch instructions; in many cases, however, this is more than offset by the performance gain. Note also that over aggressive prefetching can have detrimental effects on program performance because of cache-pollution and excessive memory traffic. Issues with respect to proper use of prefetching are out of the scope of the current discussion.

The real flexibility from the standpoint of both the compiler and the underlying system lies in the fact that these operations are non-binding. For instance, the compiler has the freedom to occasionally request data that it will never use; in fact, the system is even allowed to specify bad addresses without fear – such prefetches will simply be ignored[5]. Similarly, the system is allowed

[5] This is particularly useful for prefetching down a linked list, for instance.

to drop prefetch requests if resources are low; this is an important aspect of guaranteeing forward progress of the system as a whole (see Section 4.3.7).

**Rapid Context-Switching:** Rapid context-switching provides a good dynamic fall-back mechanism for latency tolerance precisely because it is so general. Assuming that sufficient parallelism is available, context-switching can tolerate many different types of latency, from remote memory access latency to synchronization latency. As illustrated in Figure 2-3, the basic idea behind fast context switching is to overlap the latency of a memory request from a given thread of computation with the execution of a different thread. Alewife couples the notion of rapid context-switching with software scheduling as discussed in the previous section on featherweight threads. This is an important difference from other hardware multithreading mechanisms that encode their scheduling mechanisms directly in hardware: it is the invocation of software which makes Alewife's mechanism so powerful. In Alewife, when a thread issues a remote transaction or suffers an unsuccessful synchronization attempt, the Alewife CMMU traps the processor. If the trap resulted from a cache miss to a remote node, the trap handler forces a context switch to a different thread. Otherwise, if the trap resulted from a synchronization fault, the trap handling routine can switch to a different thread of computation. For synchronization faults, the trap handler might also choose to retry the request immediately (spin) or unload the thread immediately.

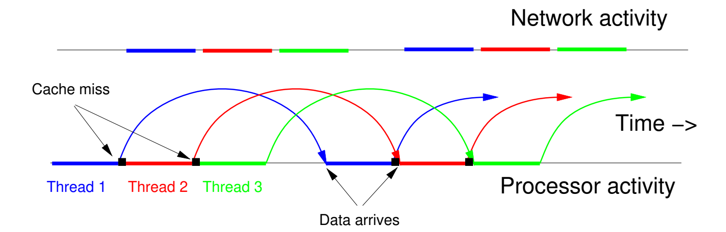


Figure 2-3: Processor and network activity when multiple threads execute on the processor and fast context switching is used for latency tolerance.

Since it is a dynamic mechanism, context-switching is relatively invisible to the programmer and system: context switches occur automatically in response to events such as remote cache misses. As a result, rapid context-switching does not require additions to the user's model in order to permit invocation. What it does require, however, is a mechanism for atomicity; otherwise, race conditions between threads could not be avoided during periods of access to common data structures. In Alewife, this atomicity is provided through simple context switch enable/disable operations that report the previous state of atomicity (thereby permitting a nested stack of enable/disable operations). Figure 2-2 lists these operations: `enable_cswitch_atomicity` and `disable_cswitch_atomicity`. Interestingly enough, the mere presence of context-switching atomicity instructions complicates the design of the memory system, since it can introduce the possibility that different threads on different nodes may deadlock each other; see Section 3.2.6 for more information on this problem.

**Unshared Local Memory:** Unfortunately, certain types of memory operations must be disallowed from causing context-switches under all circumstances. For instance, none of the instruction fetches at the beginning of trap or interrupt handlers can be allowed to cause context-switch traps — in Sparcle this would result in a recursive trap entry (a fatal processor exception). Further, the type of rapid context-switching described above provides a latency-event driven interleaving. Although this is a form of preemptive scheduling, it makes no guarantees of fairness; instead, it tends to maximize cache reuse, letting threads continue to run as long as they are not blocked on memory or synchronization [76]. This means that uncontrolled context-switching during "critical" short interrupt handlers would be deadly (not to mention unwise). Thus, context-switch atomicity must be invoked at all times during handlers and many portions of the kernel.

The simple solution in Alewife is to introduce a distinction between shared and non-shared-memory; by definition, non-shared-memory is local to the processor. (This distinction is very important for other reasons, as will be discussed in Chapter 4.) Context-switching is then disallowed on all accesses to non-shared memory[6]. In this way, kernel instructions and crucial data can be placed in non-shared memory. Kernel writers can rely on the distinction shared and non-shared memory as a way to guarantee that they will not be interrupted.

### 2.3.3 Memory Fairness

One way of viewing the existence of shared memory is as client-server interaction between processor caches (clients) and memory protocol engines (servers). Given this viewpoint, we can ask obvious questions of correctness and fairness — if multiple clients make simultaneous requests from a given server, what is the probability that (1) a correct sequencing of operations happens and that (2) the server eventually satisfies the requests of all clients? Both of these questions fall into the domain of the cache-coherence protocol, but in different ways. The issue of correctness has to do with the cache-coherence problem itself (solved through a variant of directory-based cache coherence on Alewife and DASH and through other methods on other machines). Correctness through cache-coherence is such an obvious issue of the user-level memory model that we assumed its existence as fundamental to the Latency-Tolerant shared-memory model; we will defer further discussion until Section 2.3.4, where we will mention the LimitLESS cache coherence protocol again.

Unfortunately, fairness is not a requisite property of the shared-memory model. In fact, the existence of a cache-coherence protocol that maintains the correctness of cached data does *not* guard against a lack of forward progress due to livelock. There are two different types of livelock that can occur, both of which result in a lack of forward progress by individual processor threads: the *window of vulnerability livelock* and the *multiple-writer livelock*. In one sense, these livelocks are duals of each other, since the first originates at processor side and the second originates at memory side. However, in another sense they are very different: the first results from the premature discarding of data, while the second results from memory-side refusal of service. We will spending much time discussing these problems and their solutions in Chapter 3, when we discuss the *service-interleaving problem*. As we will show in Chapter 3, the window of vulnerability is not really amenable to a software solution and, fortunately, there is a hardware solution that does not require hardware resources. Consequently, we consider the window of vulnerability livelock as something that is fixed in hardware and hence does not enter the programming model (see Section 3.2 for a complete discussion of this).

In contrast, the *multiple-writer livelock*, which is a sub-problem of the *server-interlock problem*, is amenable to a partial software solution. Although complete hardware solutions do exist

[6] In fact, the time to satisfy a local cache-miss in Alewife is actually shorter than the time to perform a context-switch; thus, allowing context-switches on local cache-misses is not advantageous anyway.

for this problem (and will be discussed in Section 3.3), they are more complicated than the combined hardware/software solution. In keeping with the overall Alewife philosophy of exploiting integrated-systems solutions, Alewife makes use of the software. Although this problem and its solution are discussed in more depth in the next chapter, we would like to touch upon them briefly here, since they affect the shared-memory programming model (*i.e.* the software must be aware of the problem and solution).

The root cause of this problem is the fact that many cache-coherence protocols exhibit periods of time in which new requests for data cannot be satisfied immediately. A prime example is the period between the initiation of an invalidation sequence and the its completion. While invalidations are outstanding for a given data block, new read or write requests for this data typically cannot be satisfied because the data itself is incoherent. The fact that something must be done with new requests during this period is referred to as the *server-interlock problem*. To avoid deadlocking the system during these periods, the memory system must either discard or queue incoming requests.

Since queueing is often difficult to do in hardware, the former (discarding of requests by explicitly rejecting them), is the usual solution. This, in turn, forces requesting nodes to retry their requests, leading to the possibility for livelock. To mention our solution to the memory-side livelock briefly, we note that typical coherence protocols (including Alewife's) permit multiple simultaneous readers, but only a single writer. Thus, memory-side livelock arises primarily in the presence of multiple writers (hence the nomenclature *multiple-writer livelock*). In solving this problem, we start by guaranteeing that *at least one writer succeeds* under all circumstances. This hardware property is sufficient to enable the construction of software queue locks, such as MCS locks [84], to combat the multiple writer situation. Thus, we "guarantee" memory fairness through a combined hardware/software solution. One justification to this approach is the observation that the single-write guarantee is more than sufficient by itself to guarantee forward progress for many (perhaps most) uses of shared memory. We will return to this in more detail in Section 3.3.2.

### 2.3.4 Shared-Memory Implementation and LimitLESS Cache Coherence

Having discussed the communication model seen by users of shared-memory, we would like to say a few words about implementation. We have already mentioned that Alewife provides a sequentially-consistent shared-memory space. What is interesting about Alewife's shared memory is that it is partially implemented in software [23]. The basic philosophy espoused by the Alewife coherence protocol mirrors that of many other aspects of Alewife: implementation of common cases in hardware and exceptional cases in software. Although the active set of software exception handlers may be selected on behalf of a user to improve performance (see, for instance [24]), they are always under control of the operating system and, ultimately, transparent to the user. As a consequence, we do not consider the existence of LimitLESS cache-coherence to be a part of the Latency-Tolerant Shared-Memory Model.

However, the existence of LimitLESS *is* a part of the operating system's view of the machine. That is why we have chosen to mention it here. At the lowest levels of the hardware, cache coherence is maintained through the exchange of protocol messages. These messages are exactly like message passing messages except for the value of an "major opcode" field in the header of the message (see Section 2.4.2 and Figure 2-7). When the hardware detects an exceptional condition that must be handled through software, it passes the protocol message up through the message-passing interface (the whole topic of Section 2.4) for handling via software. Thus, cache-coherent shared memory is actually implemented through an integrated combination of shared-memory and message-passing hardware interfaces. This fact will return in other portions of the thesis. Among other things, it introduces an interesting set of problems associated with the handing of events from hardware to software; see Section 3.4.2.

## 2.4 User-Direct Messaging

In this section, we turn our attention to defining a message-passing interface for Alewife. This interface must satisfy three goals: First, it must provide access to the network in a way that is general enough to implement a variety of communication paradigms both at *user level* and *system level*. Included among the necessary features for such an interface is the ability to handle both *polling* and *interrupt* forms of message reception. Second, this interface must interact cleanly with the shared-memory interface, including sufficient flexibility to assist in the implementation of Limit-LESS cache-coherence, which is partially implemented in software (as discussed in Section 2.3.4). Finally, the message-passing interface should lend itself to a natural hardware implementation that can be directly exported to users within the trusted environment of a scalable workstation. This would permit users to invoke protected messaging at raw hardware speeds. Since both transmission and reception of messages is entirely under control of the user, the resulting message model will be called *User-Direct Messaging* or UDM.

As will be illustrated in the following section, message passing on Alewife involves direct processor access to the input and output queues of the network via conventional loads and stores. It also involves a user-level atomicity mechanism, a DMA mechanism, and fast message interrupts via featherweight threads as discussed in Section 2.2. With the integrated interface in Alewife, a message can be sent with just a few user-level instructions[60]. User threads may receive messages either by polling the network or by executing a user-level interrupt handler. As with other aspects of Alewife, the message-passing interface represents a careful balance between hardware and software features. Consequently, scheduling and queueing decisions are made entirely in software.

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. However, as will be discussed in Section 2.5 and in Chapters 3 and 4, this uniformity of resource utilization impacts both the user's programming model and introduces some design complexity.

The message interface itself follows a similar design discipline: it provides a single, uniform communications interface to the processor. This 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. The actions include consuming the data into its registers directly, or issuing DMA-like storeback commands.

The next section (Section 2.4.1) provides an overview of the requirements for Alewife's message-passing interface. Then, Section 2.4.1 formalizes the *user-direct message-passing model*. Section 2.4.2 follows with a discussion of the message-passing interface, *i.e.* the way in which individual operations in the message-passing model reduce to machine operations.

### 2.4.1 User-Direct Messaging Model

Alewife's message-passing interface is unique for three reasons: First, it peacefully coexists with the shared-memory interface and even participates in the implementation of the shared-memory communication model. Second, both message transmission *and* reception occur at user level on the computation processor. This is in marked contrast to the network interfaces of a number of other systems that either force message reception to occur at system level [64, 45] or via a coprocessor [93, 97]. Third, the interface is highly efficient and uses a uniform packet structure. The message-passing interface in the Alewife machine is designed around four primary observations:

1. In many usages of message passing, header information is 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 [14, 35, 113, 46].
2. Blocks of data which reside in memory often accompany such header information. Thus, 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) [58] and distributed block I/O (where both a buffer-header structure and data may reside in memory).
3. Some modern processors, such as Alewife's Sparcle processor [106], MOSAIC [100], the MDP [35], and the UltraSPARC [111], can respond rapidly to interrupts. In particular, as discussed in Section 2.2, Alewife supports fast interrupt delivery via featherweight threads. This couples with efficient DMA to provide another advantage: virtual queuing. Here, the messaging overheads are low enough that a thin layer of interrupt-driven operating-system software, can synthesize arbitrary network queueing structures in software.
4. Permitting compilers (or users) to generate network communications code has a number of advantages, as discussed in [113], [14], and [47]. Compiler-generated code, however, requires user-level access to the message interface, *including* access to some form of atomicity mechanism to control the arrival of message interrupts.

Given these observations, we can proceed to define a message-passing communication model for Alewife. To simplify the explanation, we start by discussing the model from the standpoint of protected user code, in a fashion that is independent of implementation. Since the requirements for user-level access are more strict than those for system level access, this orientation will introduce the complete set of messaging semantics that we would like to support. Later, when discussing

```
send(header, handler, [oper2], ..., [<addr0:len0>], ...)
sendc(header, handler, [oper2], ..., [<addr0:len0>], ...) ⇒ sent | ignored

message_available() ⇒ true | false
receive() ⇒ ([dest0], [dest1], [dest2], ..., [<destaddr0:destlen0>], ...)
peek() ⇒ ([dest0], [dest1], [dest2], ...)

enable_message_atomicity() ⇒ disabled | enabled
disable_message_atomicity() ⇒ disabled | enabled
```

Figure 2-4: The User-Direct Messaging Model. Arguments enclosed in square brackets ("[]") are optional (only the `header` and `handler` arguments are strictly necessary at the source). Any DMA specifiers, such as `<addr0:len0>`, must be after the explicit operands; within the `receive` operation, the length value may be specified as $\infty$, meaning "to the end of the message".

the actual interface, we will illustrate how system-level code gets access to the network. Further, we will assume for now that the network is *dedicated* to a single application; later we will discuss methods for maintaining the appearance of a dedicated network in a multiuser environment[7].

As discussed in Chapter 1, we make the distinction between communication models and implementations, because often the implementation is a hybrid mix of hardware and software. The User-Direct Messaging model has a notion of *messages*, which are the unit of communication, along with operations to *inject* messages into the network and others to *extract* them from the network. It integrates both polling and interrupts for *notification* of message arrival.

**Message Injection Operations:** Figure 2-4 shows the operations that are supported by the UDM communication model. A message is a variable-length sequence of words consisting of two or more *scalar operands*, followed by the contents of zero or more *blocks of memory*. The initial two operands are specialized: the first is an implementation-dependent routing header which specifies the destination of the message; the second is an optional handler address. Remaining operands represent the data payload and are unconstrained; in addition, the contents and sizes of memory blocks (if included) are unconstrained.

The semantics of messaging are *asynchronous* and *unacknowledged*. At the source, messages are injected into the network at any rate up to and including the rate at which the network will accept them. Injection operations are *atomic* in that messages are committed to the network in their entirety; no "partial packets" are ever seen by the communication substrate. As described in Section 2.4.2, injection atomicity is extremely important for sharing of the network interface between user-level and system-level code. Further, as a result of atomicity, injection operations may be described with a procedure-call syntax in which the complete contents of a message are presented to the network as arguments to the injection operation.

Figure 2-4 details two flavors of injection operation that are supported in UDM: a blocking version (`send`) and non-blocking version (`sendc`). These operations, listed in "varargs" format[8],

[7]For all practical purposes, Alewife is a single-user machine. However, these interfaces generalize directly to multiuser systems, as will be discussed later.

[8]I am borrowing this term from the C-language. It means that the total number of arguments is variable (indicated by "..." to indicate that additional arguments may be added).

take two or more operands (the `header` and `handler` are required) and zero or more blocks of memory (specified as address/length pairs such as <`addr0,len0`>). The set of arguments to an injection operation can be thought of as *describing* the message, which is subsequently constructed by concatenating the operands with the contents of each of the blocks of memory (in sequence). In this way, data is *gathered* into the message from disparate locations in memory. Note that DMA is an integral part of the messaging model. *Implicit in this injection syntax is that data is placed into the network from registers and memory with no intermediate copy operations.*

The two different injection operations differ in their response to network congestion: If the network is unable to accept a new message, the `send` operation will block until the message is successfully injected. In contrast, the `sendc` operation does not block, but rather produces a return code to indicate whether or not the message was successfully sent (*i.e. sent* or *not sent*). Since `sendc` may or may not successfully inject a message into the network, it is up to the user to check the return code and retry the operation if it fails. Regardless of the operation used, once a message has been successfully injected into the network, UDM guarantees that it will eventually be delivered to the destination specified in its routing header[9].

**Message Extraction Operations:** At the destination, messages are presented sequentially for reception with no particular ordering. Each message is extracted from the network through an atomic operation (called `receive`), that reads the contents of the first message in the network interface and frees it from the network[10]. It is an error to attempt a `receive` operation when no message is available; hence, the model provides a `message_available` operation which can be invoked to see if `receive` will succeed. The `message_available` operation is the primary vehicle for polling the network.

Figure 2-4 illustrates a procedure-call syntax for `receive` that takes no arguments and scatters the message into a generalized *disposition* vector of l-values. These l-values may be either scalar variables or memory blocks (specified with a <addr:len> syntax); memory blocks must follow the scalar variables. The presence of memory blocks in the disposition vector implies some form of DMA to place the contents of messages into memory. After execution of a `receive` operation, the message is removed from the network and its contents are scattered into the l-values. If the message is longer than the combined length of the l-values, the the remainder of the message is discarded[11]. *Implicit in this syntax is the fact that incoming message contents are placed directly into user variables and memory without redundant copy operations.*

In addition to the `receive` operation, UDM includes an operation, called `peek`, that may be used to examine the contents of the next message without freeing it from the network. Figure 2-4 shows the syntax for this operation. Like `receive`, `peek` scatters the contents of the current message into a disposition vector of l-values. Unlike `receive`, however, `peek` is not allowed to use DMA because it cannot be invoked non-destructively.

[9]This is in marked contrast to other messaging models that may drop messages; UDM is targeted for networks in which the overall failure rate is extremely low (such as the internal network of a multiprocessor).

[10]Note that the atomicity of `receive` is not as important as for the injection operations. In fact, the presence of the `peek` operation (described presently) makes the model of reception explicitly non-atomic.

[11]A special length value of "$\infty$" may be used in the final DMA block to specify that the remainder of the message should be placed in that memory block, whatever the length.

One subtle detail that we have not mentioned about the network injection and extraction operations is that the number of scalar operands (`send` and `sendc`) or number of scalar l-values (`receive` and `peek`) may be limited by the implementation. These limits derive from the number of words of data that the network interface exports directly to the user without DMA. Experience with the CM-5 suggests that five words is far too few; Alewife provides 16 words, which seems to be a good tradeoff between implementation expense and functionality.

**User-Level Atomicity and the Execution Model:** User-Direct Messaging assumes an execution model in which one or more *threads* run concurrently on each processor. As mentioned in Section 2.2, our user-level computation model employs featherweight threads. This threading model permits *both* background computation and interrupt handlers to run in complete threading contexts.

When a message arrives at its destination, some thread must invoke a `receive` operation to extract it from the network (otherwise the network may become clogged and indirectly prevent the forward progress of threads on other nodes). However, threads will not invoke `receive` operations unless they are aware that a message is pending. The process of notifying some appropriate thread of the presence of a message is called *notification*. Notification may occur either passively, during polling, or actively, via the posting of an interrupt.

To control message notification, the UDM model includes a user-level *message atomicity* mechanism. As detailed in Figure 2-4, this mechanism is controlled by two operations: `enable_message_atomicity` and `disable_message_atomicity`. Although the analogy is not exact, the invocation of atomicity is similar to the disabling of message interrupts. When atomicity is enabled, notification is entirely through the `message_available` operation; in this mode notification is passive, and the currently running context must poll by periodically executing `message_available` and extracting messages as they arrive.

In contrast, when atomicity is disabled, the existence of an input message causes the current thread to be suspended and an independent *handler thread* to be initiated, much in the style of *Active Messages* [113]. The handler begins execution with atomicity invoked, at the `handler` address specified in the message. A handler is assumed to extract one or more messages from the network before exiting, blocking, or disabling atomicity. After disabling atomicity, the handler can continue execution; this effectively upgrades the handler to a full thread. When a handler exits, atomicity is disabled (if necessary) and some runnable thread is resumed. This thread might be a thread woken up by the handler, a thread created by the handler, or the interrupted thread; the exact scheduling policy is defined by a user-level thread scheduler, not by the model. Figure 2-5 shows a complete timeline for the injection and extraction of a user-direct message with notification via interrupts.

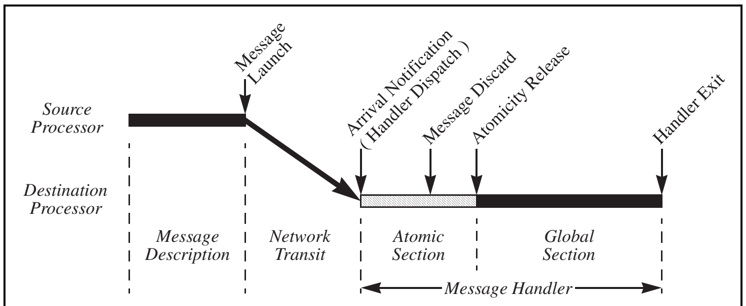



Figure 2-5: Injection of a User-Direct message and later extraction of this message with a user-level interrupt handler.

Periods of execution in which atomicity is enabled (or message-interrupts are “disabled”) are called *atomic sections*. Atomic sections will assume some importance later when we discuss issues of protection in a multi-user environment. For now, however, every interrupt handler begins with an enforced atomic section. This section ends when interrupts are re-enabled (either directly or when the handler exits). The main constraint on atomic sections in a handler is that the handler code must extract one or more messages from the network before re-enabling interrupts or exiting. In addition to atomic sections at the beginning of handlers, atomic sections may be invoked during periods in which data structures are being modified in ways that are incompatible with the arrival of messages and invocation of message handlers.

### 2.4.2 User-Direct Messaging Interfaces

In this section, we discuss the hardware *interfaces* that support the message-passing communication model discussed in Section 2.4.1. Our task is somewhat more difficult here than for the shared-memory communication model, in which the interface was the model (or vice-versa). There are two reasons for this. First, each of the operations for the User-Direct Messaging model are inherently more complicated than those of the shared-memory model. Thus, any reasonable implementation involves a careful mix of hardware and software – hardware for the common cases and software for the complicated or uncommon cases. In the case of Latency-Tolerant Shared-Memory, such an integrated systems approach was employed at the level of the cache-coherence protocol (low-level implementation), but not at the level of interface. Second, by its nature, message-passing communication requires invocation of software at many points: the marshaling of data at source and destination, the handling of message notification events, *etc.*. By design, many of the requisite software components execute at user-level; however, graceful interaction with scheduling and system software require that those operations that appear atomic to the user are actually composed of a number of individual operations.

Having said that, however, we would like to remember that software overheads accumulate rapidly. In Chapter 1, we stressed the importance of fast, user-level access to messaging facilities. Since User-Direct Messaging is a user-level communication model, it can be exploited directly by the compiler, suggesting that unique send and receive code might be generated for each type of communication. However, when a message can be described in less than 10 cycles, the time to cross protection barriers or execute emulation code may easily double or triple the cost of sending that message. Hence, we seek hardware interfaces that may be directly manipulated by the user without compromising protection. It is our goal in this section (and ultimately in Chapter 5 to demonstrate that the User-Direct Messaging model section is amenable a *reasonable* implementation that provides fast, *direct*, *user-level* access to network hardware without violating protection.

Figure 2-6 shows the explicit set of network instructions for User-Direct Messaging. In addition, Tables 2-2 and 2-3 list the set exceptional conditions and the set of network-related CMMU registers respectively. Note that the CMMU registers are memory mapped, but attached to the first-level cache bus. Access to these registers is accomplished via special load and store instructions called `ldio` and `stio`[12]. The interface itself may be summarized in three pieces:

[12]Note that `ldio` and `stio` are just like normal load and store instructions except that they map to a separate address space. Consequently, CMMU register addresses fit entirely within the immediate field of these instructions. In the second version of the Alewife CMMU, this address space was exported to a range of "normal" addresses, allowing

| | | |
|---|---|---|
| `ldio` | `Ra+Offset, Rd` | // Load from CMMU register space. |
| `stio` | `Rs, Ra+Offset` | // Store to CMMU register space. |
| `ipilaunch` | `<numops,length>` | // Inject packet consisting of first `numops` double-words of descriptor followed by (`length` − `numops`) blocks of data via DMA. |
| `ipilaunchi` | `<numops,length>` | // Same as `ipilaunch`, but generates `transmit_completion` interrupt when finished. |
| `ipicst` | `<skip,length>` | // Discard first `skip` double-words of packet, then storeback `length` double-words via DMA to `storeback-address`. If `length` or `skip` = "infinity", discard packet. |
| `ipicsti` | `<skip,length>` | // Same as `ipicsti`, but generates `storeback_completion` interrupt when finished. |
| `enabatom` | <MASK$_{ATDE}$> | // $(\texttt{uatomctrl} \vee \texttt{MASK}_{ATDE}) \Rightarrow \texttt{uatomctrl}$ |
| `disatom` | <MASK$_{ATDE}$> | // **If** `dispose_pending`, cause `dispose_failure` trap.<br>**elseif** `atomcity_extend`, cause `atomicity_extend` trap.<br>**else** $(\texttt{uatomctrl} \wedge (\sim\texttt{MASK}_{ATDE})) \Rightarrow \texttt{uatomctrl}$ |
| `setatom` | <MASK$_{ATDE}$> | // $\texttt{MASK}_{ATDE} \Rightarrow \texttt{uatomctrl}$ |

Figure 2-6: Instructions for the User-Direct Messaging interface. See text for description of user-level restrictions on these instructions. Operands enclosed within "<>" represent compile-time constants. MASK$_{ATDE}$ is a four-bit mask representing *atomicity_assert*, *timer_force*, *dispose_pending*, and *atomicity_extend* in that order. The `uatomctrl` register is in the same format. Conditions `dispose_pending` and `atomicity_extend` represent corresponding bits set in `uatomctrl` register.

- Sending of 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 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.
- Receipt of a message is signaled with an interrupt to the receiving processor. Alternatively, the processor can disable message arrival through a user-level atomicity mechanism and poll for message arrival. The network interface provides an input *window* on each packet that exports the first few words of the packet. Processor actions include discarding the message, transferring the message contents into processor registers, or instructing the CMMU to *scatter* the contents of the message into one or more regions of memory via DMA.
- Mechanisms for *atomicity* and *protection* are provided to permit user and operating-system functions to use the same network interface. We introduce the notion of a *revocable interrupt disable* mechanism. This grants provisional access of network hardware directly to the user, while carefully monitoring access. If the user abuses the network, direct hardware access is revoked (and an emulation mode entered).

In the following sections, we will explore these aspects of the interface in more detail: Section 2.4.3

the network control registers to be accessed with standard `ld` and `st` instructions.

| Interrupt | Event which it signals |
|---|---|
| `bad_launch` | Attempt to launch protected message. |
| `user_message` | User-level message arrival. |
| `system_message` | System-level message arrival. |
| `storeback_completion` | Completion of last `ipicsti`. |
| `bad_dispose`† | `ipicst` without pending message. |
| `transmit_completion` | Completion of last `ipilaunchi`. |
| `space_available` | Requested descriptor space available. |
| `atomicity_congestion` | Network congested |
| `atomicity_timeout` | User-level atomicity timeout. |
| `atomicity_extend`† | Invoke epilogue to atomic section. |
| `stymied_shared_memory`† | Shared memory used during atomicity. |
| `dispose_failure`† | Atomicity exited without dispose. |

Table 2-2: Interrupts and Exceptions for User-Direct Messaging interface. Events marked with † are synchronous exceptions.

| Register | Description |
|---|---|
| `output_descriptor[16]` | Output descriptor array |
| `space_available` | Output descriptor space available |
| `space_request` | Output descriptor space requested |
| `desc_length` | Current descriptor length |
| `traps_pending` | Pending interrupts and flags |
| `input_window[16]` | Input packet array |
| `window_length` | Size of message input window |
| `storeback_address` | Address for next DMA storeback |
| `uatomctrl` | User atomicity control (4 bits) |
| `overflow_timeout` | Network overflow count in cycles |
| `overflow_countdown` | Cycles remaining before overflow |
| `atomicity_timeout` | Atomicity timeout in cycles. |
| `atomicity_countdown` | Cycles remaining before timeout |

Table 2-3: Control registers for User-Direct Messaging interface. Sizes are in double-words. `uatomctrl` is a four bit mask ($\texttt{MASK}_{ATDE}$).

discusses message injection, Section 2.4.4 discusses message extraction, and Section 2.4.5 presents the user-level atomicity mechanism.

First, however, we would like to mention a starting point for the integration of message passing and shared memory: a common packet format. All packets in the Alewife machine have a single, uniform header as their first word. Figure 2-7 shows the format of this header. 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 cache-coherence protocol packets are directly interpreted by hardware to provide coherent shared-memory. Further, packet headers in this format are constructed automatically by hardware for cache-coherence operations; however, users of the message-passing interface must construct headers of the format whenever sending messages.

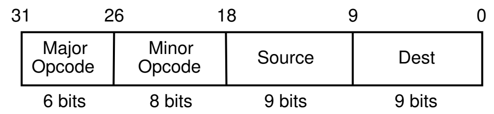


Figure 2-7: A uniform packet header format. The major opcode distinguishes between protocol, system, and user messages.

## 2.4.3 Message Injection Interface

In designing a network output interface with which to implement `send` and `sendc` operations, we have one overriding goal: the interface that we choose must be compatible with simultaneous demands of both user and system code. If `send` or `sendc` are implemented via multiple instructions (which we have already intimated is the case), then this requirement introduces a problem with atomicity since user-code must execute with interrupts enabled[13]. The problem is that multi-instruction sequences can be interrupted at any point; hence, if an interrupt occurs in the middle of

[13]This is necessary since interrupts represent time-critical events that may adversely affect forward-progress or correctness if not handled promptly. Hence, allowing users to disable interrupts is, in general, a bad idea, since incorrect or malicious user-code could compromise the integrity of the machine.

a multi-instruction injection operation and the corresponding interrupt handler must send a message, then the two messages may become intertwined or corrupted. As a case in point, this was a problem with the J-Machine [87], in which message injection was accomplished through instructions that sent data directly into the network (one or two flits at a time). Once user-code had started injecting a message, it could not be interrupted by anything that wanted to send a message (even the operating system!), since this resulted in merged messages.

One solution might be to provide separate output interfaces for the user and supervisor. This solution is undesirable for two reasons: on the one hand, it is overkill, since the chance that both the user and supervisor will attempt to send messages simultaneously is fairly low. On the other hand, it merely defers the problem to a different level: operating systems often have a hierarchy of interrupt levels; as a consequence, any network interface that we choose should be able to handle the preemption of an interrupt handler that is sending a message by another of higher priority.

Consequently, the Alewife machine adopts a more general mechanism for multiplexing the output interface. This mechanism is predicated on the assumption that collisions between message senders are rare, but that the higher priority interrupts should always have access to the network. To accomplish this, we decompose message injection into two distinct phases: *description* and *launch*. During the description phase, all of the information relevant to construction of a message is placed into an *output descriptor array* within the network interface. This array consists of a series of registers on the Alewife CMMU that may be read as well as written by the processor; the interface to this array is memory-mapped. Consequently, the compiler can generate instructions that perform direct register-to-register moves from the processor into this array. In the Alewife implementation, these moves proceed at the speed of the cache bus. Once the message description phase has completed, a single-cycle launch operation commits the corresponding operation to the network. This launch operation is *atomic*: before it is executed, the network is unaffected by modifications to the descriptor array, and after it is executed, the message is committed to the network and completely unaffected by subsequent modifications to the descriptor array. This atomicity is the central feature of the Alewife message injection interface: during the description process, interrupts are free to use the network providing that they save and restore any partially-constructed message descriptors. Thus, our multiplexing mechanism is the familiar "callee-saves" methodology applied to interrupts.

As we will see in Chapter 5, implementation of this mechanism is surprisingly straightforward. However, this simple implementation does not preserve the contents of the descriptor array after launch. Consequently, one additional mechanism is provided: the Alewife CMMU provides a special `desc_length` register. 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. Its value is usually zero.

**Implementation of the `send` operation:** Figure 2-8 makes this interface more explicit by illustrating how an abstract `send` operation is decomposed into explicit UDM instructions. Alewife packet descriptors consist of one or more 64-bit double-words (*i.e.* must consist of an even number of words). Packet description is accomplished by a series of `stio` instructions that transfer data directly from processor registers to the `output_descriptor` array — a 16-word (or 8 double-

```
send(header, handler, op2, op3, [addr0:len0], [addr1:len1]) ⇒
   stio       header, output_descriptor[0]    ; Place header in descriptor
   stio       handler, output_descriptor[1]   ; Place handler in descriptor
   stio       op2, output_descriptor[2]       ; op2 and op3 are additional operands
   stio       op3, output_descriptor[3]       ;
   stio       addr0, output_descriptor[4]     ; addr0 and len0 describe first DMA
   stio       len0, output_descriptor[5]      ;
   stio       addr1, output_descriptor[6]     ; addr1 and len1 describe second DMA
   stio       len1, output_descriptor[7]      ;
   ipilaunch  < 2, 4 >                        ; Launch 2 explicit double-words and
                                              ;    2 memory blocks via DMA
```

Figure 2-8: Translation of a UDM `send` operation into UDM interface instructions.

word) array of registers on the CMMU. The destination addresses for these `stio` instructions are small immediate addresses in the CMMU register space (*i.e.* `output_descriptor[4]` represents a particular CMMU register). The resulting 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. When the resulting packet is constructed, the first word must be a header in the format shown in Figure 2-7, *i.e.* either the first operand must be in header format, or (if there are no operands), the first word of the first DMA block must be in header format.

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 2-6, the opcode fields of an `ipilaunch` specify the number of explicit operands (in double-words) and the total descriptor length (also in double-words), *as compile-time arguments*. Thus, the format of the packet descriptor (in terms of number of explicit operands and DMA blocks) is fixed at the time of compilation. 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[14].

Relating this interface back to the model of Figure 2-4, we can say that, in Alewife, the maximum number of scalar arguments to `send` or `sendc` is 16 (i.e 8 double-words). Further, the maximum number of DMA blocks is 8.

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. Alewife provides two mechanisms to deal with this. First, there is a special version of the launch instruction, called `ipilaunchi`, that 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. Note that this interrupt is conservative: when it occurs, it signals that `all` previously initiated output DMA operations have completed. Second, Alewife provides a special flag bit in the `traps_pending` register that indicates whether or not there are any pend-

[14] In fact, descriptor contents are not preserved across a launch. See Chapter 5.

ing output DMA operations. This flag may be exploited to provide a serializing "wait for launch" operation that will wait until all previous launch operations have completed.

**Protection:** There is often a tension between protection mechanisms and rapid access to hardware facilities. The Alewife network interface is no different. Alewife is a fully integrated architecture in that *all* types of communication travels over the same network: cache-coherence protocol packets, operating-system messages, and user-level messages. To protect the integrity of the machine, we must prevent the user from launching packets that are not user-level message-passing messages. As shown in Figure 2-7, different classes of message are distinguished by the *major opcode* portion of the header. Thus, while the operating system is allowed to construct and inject *any* valid descriptor, the user is a bit more restricted. *In particular, all user-level packets must have at least two operands and must have headers that contain a restricted set of major opcodes.* Note that the first requirement ensures that the header as an *explicit* part of the descriptor rather than in memory, thus permitting the `ipilaunch` or `ipilaunchi` instructions to quickly check the type of a message before injecting it. These restrictions do not apply, since the operating system is allowed to launch any type of message.

Although we will discuss this later, it is interesting to note that the user-level message opcode space is further divided into two separate categories: *active messages* and *remote system calls*. Both of these types of messages may be launched by the user, but cause different interrupts at the receiving side. This reflects different scheduling at the remote side: active messages are handled via polling or user-level interrupt handlers at their destinations, while remote system-calls invoke kernel handlers at the remote side[15].

**Network Congestion and Packet Injection:** If the output network is blocked due to congestion, then it is possible that the CMMU has insufficient resources to launch the next message. As discussed in Section 2.4.1, the `send` operation is *blocking*: it does not complete until sufficient resources are available to inject its message into the network. Given the simple decomposition of `send` shown in Figure 2-8, there are only two places in which this instruction sequence can block: in the `stio` instructions or in the `ipilaunch` instruction. It is, in fact, the former; as will be shown when we discuss hardware architecture in Section 5.2.1, accesses of the `output_descriptor` array translate directly into reads and writes of a circular hardware queue. Thus, the impact of network congestion is that this queue becomes full, and writes to the `output_descriptor` array would overwrite previously queued data. In this case, `stio` instructions block, waiting for the descriptor queue to drain. Consequently, by the time that the sequence of Figure 2-8 advances to the `ipilaunch` instruction, it is guaranteed that the `ipilaunch` will succeed.

In a moment, we will discuss how non-blocking `sendc` operation is achieved with the basic UDM interface, but we would like to first say a few words about philosophy. As we will see in a moment, the UDM interface is optimized for blocking `send` operations, as opposed to non-blocking `sendc` operations. This is in marked contrast to most other direct network interfaces in

[15]Actually, this distinction is better expressed in the second version of the Alewife CMMU: a separate interrupt vector is reserved exclusively for user-level active-message style communication.

which the basic interface is non-blocking and extra instructions (*i.e.* forming a loop) are required to achieve blocking behavior. This particular design decision was made for several reasons:

- Most (if not all) interrupt-driven uses of a message-passing interface use blocking, rather than non-blocking, behavior. The reason for this is that blocking behavior is much simpler handle as as rule (no need to figure out how to retry message injections that fail). When the network input ports are drained via interrupts (especially fast ones such as on Alewife), the need for a non-blocking behavior is much diminished.
- Non-blocking behavior always incurs more overhead, since it involves at least one conditional branch instructions. In particular, the Alewife message interface is used implement portions of the LimitLESS shared-memory coherence protocol; the lowest possible overhead is desirable here.
- The basic hardware architecture is simpler if it is, by default, blocking.

As a consequence, the basic UDM interface requires slightly more overhead to handle non-blocking operations than it would if non-blocking message injection were the default behavior. In fact, this is not really fundamental; at the end of the next section we will discuss how the basic interface can be extended for faster non-blocking behavior.

As a prelude to the next section, we would like to mention the `space_avail` register, in the context of blocking sends. At any one time, this register denotes the number of double-word descriptor slots that are currently free. Effectively, this says, at any one time, the maximum descriptor hat can be constructed without blocking. Thus, to avoid blocking, it would appear that a user could simply check the `space_avail` register before constructing a descriptor. This is certainly true of the operating system, which may explicitly control access to the network interface by enabling and disabling interrupts. Unfortunately, the very fact that the operating system may interrupt user-code and send messages in the middle of the user's description process means that the value in `space_avail` is merely a hint of the amount of space that is available. As we will discuss in the next section, we may combine examination of the `space_avail` register with *atomic restartable sequences* to construct the `sendc` operations.

One other mechanism that exists primarily to assist the operating system (for "virtual queueing" and scheduling large I/O requests) is the `space_request` mechanism. This consists of the `space_request` register and the `space_available` interrupt. To use it, a desired number of double-words of descriptor space are written into the `space_request` register. Then, the `space_available` interrupt occurs as soon as the space in the `output_descriptor` array equals or exceeds the requested amount of space.

**Implementation of the `sendc` operation:** We now have sufficient information to discuss one possible implementation for the `sendc` operation. Figure 2-9 illustrates the translation of a non-blocking `sendc` operation into UDM instructions. This code sequence is a template that is recognized by the operating system as an *atomic restartable sequence*. The restart point is the beginning of the code group (marked by "→"). At the beginning of the sequence, a special "NONBLOCK" bit is set in a user register called `flags`. This is a signal to the operating system that the user is in the middle of a `sendc` operation. Then, the sequence checks the `space_avail` register to see

```
sendc(header, handler, [addr0:len0]) ⇒
→ or         flags, $NONBLOCK, flags        ; Set flag – user nonblocking OP
  ldio       space_avail, temp              ; Get space available
  cmp        temp, 8                        ; We need 4 double-words
  bl,a       %Failure                       ; Not enough space – failure
  and        flags,  $NONBLOCK, flags       ; Clear flag (only on failure).
  stio       header, output_descriptor[0]   ; Place header in descriptor
  stio       handler, output_descriptor[1]  ; Place handler in descriptor
  stio       addr0, output_descriptor[2]    ; addr0 and len0 describe first DMA
  stio       len0, output_descriptor[2]     ;
  ipilaunch  < 1, 2 >                       ; Launch 1 explicit double-word and
                                            ;    1 memory block via DMA
  and        flags,  $NONBLOCK, flags       ; Clear flag (end of sequence).
```

Figure 2-9: Translation of a UDM `sendc` operation into UDM interface instructions. This is an atomic restartable sequence that may be restarted to the beginning.

if sufficient descriptor space is available. If not, it fails immediately, jumping to the "`Failure`" label (wherever that is). If there is sufficient space, it proceeds to construct the descriptor, launch the message, then turn off the NONBLOCK bit.

The danger in this sequence (and the reason that we may need to restart it), is that the check for space could succeed, only to have the description phase interrupted by something that sends a message. If the interrupt simply returns, if may be that sufficient space is nolonger available and one of the `stio` instructions will block. Hence we have three possible situations for an interrupt handler that wants to send a message:

1. The NONBLOCK bit is not set. In this case, nothing special is done.
2. The NONBLOCK bit is set and the interrupted instruction is one of the `and` instructions to turn off the bit. Again, nothing special is done.
3. Otherwise, reset the interrupt point by scanning back until the initial marker `or` instruction is discovered.

Note that the second of these is important to guarantee that the message is not inadvertantly injected twice. This will guarantee that the check for space is reexecuted whenever a handler has sent a message (and hence decreased the amount of space available in the descriptor queue).

**A better `sendc` sequence:** Note that that this sequence has several inefficiencies that are not strictly necessary. First, whenever the sequence is restarted, the operating system must scan backward for the restart point. Second, it is possible for the operating system to restat the sequence when sufficent space is, in fact, available. Third, the check for space is not usually a cost that is incurred by non-blocking interfaces.

Alternatively, the following enhancement is possible as a way to avoid restartable sequences entirely (*this was not done on Alewife but could have been*):

1. Provide "non-blocking" access to the `output_descriptor` array: if an attempt is made to store information to the `output_descriptor` when insufficient space is available, then the store operation is ignored, but the `desc_length` register is set to an "invalid" value (say -1). Otherwise, assuming it is not already invalid, the `desc_length` register is set to the maximum descriptor length, as indicated earlier. This could be accomplished via a special set of addresses or a different store instruction.
2. When the `ipilaunch` or `ipilaunchi` instruction is executed, it injects a message if the `desc_length` is positive and does nothing otherwise. The `desc_length` register is reset to zero and a condition code is returned to indicate success or failure.

The blocking version of `send` would continue to be implemented exactly as before. Now, however the `sendc` operation would be similiar, with non-blocking `stio` instructions instead of blocking ones and a final check to see if the `ipilaunch` instruction succeeded (possibly repeating the entire sequence if necessary).

### 2.4.4 Message Extraction Interface

In constructing an input interface with which to implement `peek` and `receive` operations, we once again need acount for the fact that this interface is used by both the user and the operating system. Further, the design constraints for the input interface are somewhat different than for the output interface. Although we were able to solve the multiplexing problem for message injection by providing an atomic launch mechanism, something slightly different is needed for message reception. The reason for this is that inaction, in the form of refusing to extract messages from the network, can prevent reception of system-level messages on the local node, as well as impeding forward progress in other nodes in the system. In contrast, inaction with respect to injecting messages has no impact other than slowing down the user-program.

The fact that refusal (or inability) to extract messages can cause network congestion has two major consequences. First, in extreme cases, this can cause protocol deadlock — even for bug-free operating-systems code. The issue of protocol deadlock is addressed in Chapter 4, where we discuss *two-case delivery* as a generic mechanism for avoiding deadlock. Second, malicious or incorrect user code can also cause network congestion by refusing to extract messages; this is an issue of protection that is addressed through the user-level atomicity mechanism (Section 2.4.5) which may be coupled with two-case delivery to control errant user code (see Section 4.3.5).

Once consequence of using two-case delivery as a way to gracefully multiplex user and system consumers of the network interface is that there is no need for the equivalent of the user-level atomic injection operation. Hence, we seek the simplest, most efficient possible interface for receipt of messages. An interface that achieves this goal is the primary topic of this section.

**Packet Input Window:** The message input queue produces a serial stream of messages from the network. The message that is at the head of this input queue is presented to the user (or operating system) through an array of CMMU registers called the `input_window`. This array ("window") presents the first 16 words (8 double-words) of the message directly to the user. The basic reasoning behind this is that the first few words of a packet (up to 16) should contain enough information to decide how to dispose of the rest of the packet. In the terminology of Section 2.4.1, this particular interface limits the number of scalar l-values in the disposition vector to 16. If a packet is longer that 8 double-words, then parts of it will be invisible to the processor and may stretch into the network; these trailing parts of the message must be accessed by invoking DMA operations to store them to memory.

*One important aspect of the input interface is that it is independent of timing in the network:* it presents the illusion that all flits of the current packet have arrived, regardless of the speed with which they enter the node. This illusion is accomplished by blocking the processor if it attempts to access words of the packet that are still in transit. Note that blocking in this way is reasonable because of the fact that the message injection interface is atomic; since the complete packet has been committed to the network, any delay in arrival of words from the current packet is guaranteed to be short lived. The fact that this is a blocking interface allows overlap between interrupt or polling entry code and the arrival of the bulk of a message. For instance, message arrival interrupts are posted immediately upon arrival of the first words of messages.

```
receive()⇒(header, handler, op2, op3,[addr0:len0],[addr1:len1])
    ldio    input_window[0], header    ; Get header from packet
    ldio    input_window[1], handler   ; Get handler from packet
    ldio    input_window[2], op2       ; op2 and op3 are additional operands
    ldio    input_window[3], op3       ;
    stio    addr0, storeback_address   ; Set up first DMA address
    ipicst  < 2, len0 >                ; Skip 2 double-words, store len0 double-words to memory
    stio    addr1, storeback_address   ; Set up second DMA address
    ipicst  < 0, -1 >                  ; Store rest of packet via DMA
```

Figure 2-10: Translation of a UDM `receive` operation into UDM interface instructions. Note that a value of "-1" is a special token meaning "$\infty$" or "until end of packet".

**Implementation of the `receive` operation:** Figure 2-10 makes this interface more explicit by illustrating the decomposition of an abstract `receive` operation into explicit UDM instructions. In this example, the disposition vector has four scalar l-values and two blocks for DMA storeback. The scalar operands are loaded directly from the input window into registers. The scalar values are accessed via `ldio` instructions. The source addresses for these instructions are small immediate addresses in the CMMU register space (*i.e.* `input_window[4]` represents a particular CMMU register).

After loading the operands, the processor initiates a DMA storeback operation to extract the message from the network. To do this, it first writes the address for the beginning of the DMA operation to a CMMU register called `storeback_address`. Then, it invokes a single-cycle storeback instruction, called `ipicst` (IPI coherent storeback). As shown in Figure 2-6, this instruction has two opcode fields, *skip* and *length*. The skip field specifies the number of double-words which are first discarded from the head of the packet, while the length field specifies the number of double-words (following those discarded) which will be stored to memory via DMA.

The `receive` operation of Figure 2-10 scatters the packet into two memory blocks via DMA. Hence, after executing the first `ipicst` operation, the processor immediately writes the second address into the `storeback_address` register and executes another `ipicst` operation. This instruction has a skip value of zero, since the data for the two DMA operations is consecutive[16]. Further, the length value is "-1", which is a special token value that means "infinite" or "to the end of the packet." There are two important points here. First, the `storeback_address` register is not a single register, but rather a FIFO of values that become associated with successive `ipicst` instructions (which is why this code works). The particular implementation discussed in Chapter 5 supports up to two outstanding DMA storeback operations before it begins to block the processor. Second, *all* message extraction operations must end with an `ipicst` instruction that has an infinite value in either the skip or length field; this is a signal to the network interface that it is ok to discard all information about the packet. In particular, receive operations that do not involve DMA must end with an `ipicst` instruction that has an infinite skip field.

Since DMA storeback operations may proceed in parallel with processor execution, the processor should not attempt to examine the results of the storeback until after confirming that it has completed. As with output DMA, Alewife provides two mechanisms for controlling the parallelism between DMA and computation. First, there is a special version of the storeback instruction, called `ipicsti`, that requests the generation of a `storeback_completion` interrupt as soon DMA has completed. Note that, similiar to the `transmit_completion` operation, the `transmit_completion` interrupt is conservative when it occurs: it signals that `all` previously initiated input DMA operations havfe completed. The second mechanism that is provided to handle storeback DMA is a special flag bit in the `traps_pending` registe that indicates whether or not there are any pending input DMA operations. This may be exploited to provide a serializing "wait for storeback" operation that will wait until all previous storeback operations have completed.

```
peek()⇒(header, handler, op2, op3)
    ldio   input_window[0], header    ; Get header from packet
    ldio   input_window[1], handler   ; Get handler from packet
    ldio   input_window[2], op2       ; op2 and op3 are additional operands
    ldio   input_window[3], op3       ;
```

Figure 2-11: Translation of a UDM `peek` operation into UDM interface instructions.

**Implementation of the `peek` operation:** As shown above, in Figure 2-11, the peek operation is a natural consequence of our input interface. Essentially, the head of the next message may be examined simply by loading data from the input window. The only distinction between a `peek` operation and a `receive` operation is the absence of `ipicst` instructions in the former. This distinctions clarifies the fact that `peek` operations may not invoke DMA (*i.e.* the disposition vectors for `peek` opertions may not contain DMA blocks): input DMA requires execution of `ipicst` instructions. Another way to look at this is that `peek` operations are intended to be passive and alter nothing about the current message; however, DMA is "destructive" — it advances the input window. Note that the ability to non-destructively examine the head of an input packet is

[16] In fact, the skip value does not *have* to be zero (although this would produce a type of message disposition that is not represented by our model for the `receive` operation).

important for dispatching code, *e.g.* for the thin layers of the operating system that handles setup and execution of user-level message interrupt handlers.

**Notification and the `message_available` operation:** As described in Section 2.4.1, it is "an error" to attempt a `receive` or `peek` operation when no message is available. Thus, it is important to know when a message is available.

If user-level atomicity is disabled, then user-level message arrivals are signified by the execution of user-level interrupt handlers. In some sense, no more notification is necessary: the user-level handlers can rely on the fact that there are a messages available. At the system-level, notification via interrupts occurs through one of two different interrupt vectors: one for system-level messages and another for user-level messages. The division of message types between these two vectors is one of the differences between the A-1000 and A-1001 CMMU. At the time that this author designed the A-1000, the cost of scheduling and dispatch was not fully appreciated. Further, the division of user-*launchable* messages into *active messages* and *remote system calls* was not fully appreciated. The former lead to user-level interrupt handlers (complete with the scheduling issues involved), while the latter need to be dispatched into the operating system in the same way as the system-launched messages. Hence, the A-1001 places user-level active messages on one interrupt vector and everything else on the other; the A-1000 more carelessly placed all user-launchable messages on one vector and system-launchable messages on the other[17].

The user controls notification via the user-level atomicity mechanism (described in Section 2.4.5). The kernel, on the other hand, has a message interrupt disable bit in the `trap_enable` register.

If interrupts are disabled, both the user and kernel can make use of the `traps_pending` register to discover whether or not a message is available and, if so, whether it is a user-level message. Two bits are of interrest — one telling whether the input window has a message available and another indicating the type. However, another mechanism is available that provides greater transparency to the user during two-case delivery (see below): whenever data is loaded from the input window, the CMMU sets the full/empty condition code to indicate whether the data is valid. Hence, by loading from the head of the input window, a user can discover whether or not a message is available by testing the full/empty condition code. The primary advantage of this method is that buffered messages (during two-case delivery) can have their full/empty bits set to full, making the polling methodology completely transparent to delivery method.

**Transparency and the Impact of Two-Case Delivery:** Although we will defer a full discussion of two-case delivery until Section 4.3, it is important to clarify one detail at this time. The policy of two-case delivery selectively removes messages from the input interface and buffers them in memory. The justification for this is that it provides a generic solution to many different types of network deadlock and congestion; that is the topic of a later chapter. For now, however, we would like to stress the most important interface aspect of an implementation of two-case delivery: transparency. In order for two-case delivery to be a reasonable policy, the direct (hardware) and buffered modes of message delivery must appear identical to user software. With transpar-

[17] Of course, the A-1001 has a backward-compatibility mode, but this goes almost without saying.

ent access, the runtime system is free to switch to and from buffered mode at any time, thereby making invocation of two-case delivery tool for resolving many different problems with resource utilization.

This transparency is provided in one of two different ways in Alewife. The first of these was exploited exclusively in the original version of the Alewife CMMU: *message relaunch*. Message relaunch is a type of software "slight-of-hand" that returns the message at the head of the input queue to the exact state that it was in before the buffering process. This is done by "relaunching" messages through the local loopback path (effectively sending messages to the local node through the message output interface). In this way, the network input queue can be completely emptied and restored at any point in a `receive` or `peek` sequence without affecting the results. This solution is conceptually simple and necessary for handling diverted cache-coherence packets. Unfortunately, it entails a number of complexities and can exacerbate problems with network congestion by tying up the local loopback path with long messages. Among other things, the combination of buffering and relaunch copies message data at least twice, possibly three times (when storeback DMA is invoked).

A second method for transparent access was included in version two of the Alewife CMMU: the *virtual input window*. This method requires that the hardware input queue be *memory mapped* into the normal address space, permitting the user to access the hardware input window with standard load instructions[18]. Then, assuming that a known register is shared between the user and operating system, we can use it to point to the base of the hardware input window under normal circumstances and redirect it to memory after buffering. With this type of redirection, two-case delivery results in a single extra copy (over and above any DMA that would have been done anyway). Thus, the buffered copy of the message is treated as a virtual input window (and hence the name). To make this process entirely transparent, we must make sure that `ipicst` instructions are properly emulated; this is done by causing them to trap during buffering mode.

**Protection:** In order to present a protected interface directly to software at userlevel, the message extraction interface has a couple of special features. Protection in the Alewife machine is not intended to hide information, but rather to protect the machine from errant user-code. There are two primary features for the extraction interface, which are as follows:

- The user is not allowed to issue storeback instructions if the message at the head of the queue is a system or coherence message. In a machine for which hiding of information was important, the input window *could* export null information (zeros) whenever the message at the head of the input queue was a system message and the processor was in user mode.

- Although we will discuss multiuser applications of the UDM interface in a moment, this same protedction methodology could be applied in situations for which the message at the head of the hardware input queue belonged to a different process from the one that was currentloy scheduled.

[18]Note that this is no more difficult from an implementation standpoint than providing `ldio` instructions. The primary advantage of `ldio` and `stio` is that the addresses for CMMU registers fit completely into the offset field of these instructions — providing single-instruction access to CMMU registers.

- The user is not allowed to store data into kernel (or other protected) space. This involves 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.

### 2.4.5 The User-Level Atomicity Mechanism

As discussed in Section 2.4.1, atomicity is an important aspect of a message-passing interface. Atomicity in the context of UDM means the ability to disable message arrival interrupts, *i.e.* the ability to disable the automatic invocation of a message handler in response to the arrival of a user-level message. Such control over the network enables efficient message reception by allowing messages to be extracted from the network only when appropriate `receive` operations can be executed; this avoids expensive passes through kernel code and time-consuming copies of data. In fact, disabling of interrupts is a time-honored method for achieving correctness and efficiency in operating systems. However, exporting hardware interrupt disable mechanisms directly to the user is problematic since user-code is never guaranteed to be correct. In particular, there are several reasons not to allow the user to directly block the network interface by disabling hardware interrupts:

- Malicious or poorly written code could block the network for long periods of time, preventing timely processing of message destined for the operating system or other users[19].
- When the user is polling, the operating system may still need to receive messages on the local node via interrupts to ensure forward progress. This is a statement to the effect that we would like to be able to multiplex both user and system messages on the same network.
- Certain combinations of message atomicity and shared-memory access can lead to deadlock in a system that integrates shared memory and message passing. In particular, if message interrupts are disabled and a message is blocking the network input queue, then remomte shared-memory requests may never complete.
- If queue resources are exhausted, attempts to send messages during periods of message atomicity can lead to protocol-level deadlock.
- In a multiuser system, the operating system must demultiplex messages destined for different users. This process should be neither visible to, nor impeded by, any particular user.

In spite of these problems, we would like the user to enjoy similar efficiency (in the common case) to the operating system, extracting messages directly from the network interface. This, in turn, requires that the user be able to disable message interrupts.

[19]The User-Direct Messaging interfaces extend directly to multiuser systems; see Section 2.4.7.

We solve these problems through a *revocable interrupt disable* mechanism that exports virtual operations `enable_message_atomicity` and `disable_message_atomicity` to the user, while maintaining some measure of control for the operating system. The central idea behind this mechanism is that we allow the user to *temporarily* disable hardware message interrupts, as long as the network continues to make forward progress. Should a user-level message stay blocked at the input queue for too long, we *revoke* priviledges to disable network interrupts. The reflects a philosophy of optimistically granting direct access of hardware mechanisms to the user as long as they do not abuse them. To achieve graceful degradation of service, the process of revocation involves switching from physical atomicity (disabling of the actual hardware network interrupt) to virtual atomicity (buffering incoming messages in memory and using the scheduler to maintain the user's notion of atomicity). Thus, proper implementation of revocation is tightly entwined with transparent buffering of messages (i.e. *two-case delivery*, Section 4.3).

Note that the "atomicity" provided by the user atomicity mechanism corresponds to a direct disabling of hardware interrupts only when messages are being retrieved directly from the hardware interface. On the buffered path (second case delivery), atomicity is provided to the user by software in the way the buffered queue and thread scheduling are handled. As discussed in Section 2.4.4, the ability for a system, to transparently switch between a non-buffered and buffered mode is an important aspect of the UDM interface; in the context of the revocable interrupt disable mechanism, this transparency affords freedom to the runtime system to switch into buffered mode when necessary to avoid deadlock. However, we should note that *common case* delivery is intended to occur through direct hardware access to the message input queues and this is what the user-level atomicity mechanism is optimized for; in Chapter 4 we will explore in more detail the process of ensuring that hardware deliver is truly common case.

**Mechanism Requirements:** Given the above discussion, we can now discuss the requirements for the user-level atomicity mechanism. In order to achieve proper revocation, this mechanism must be able to (1) provide a virtual interrupt disable bit that is easily manipulated by the user, (2) distinguish between user-level and system-level messages so that system-level interrupts may be forwarded at all times, regardless of the state of the user-level disable bit, and (3) include monitoring hardware to detect situations in which the user-level disable bit should be revoked. These situations are as follows:

- The user is causing network congestion by refusing to free messages in a timely fashion. To prevent this, we need to monitor the duration of time that a message sits a the head of the input queue, *i.e.* the time from the arrival of the message until the execution of an `ipicst` instruction with an infinite argument. This type of monitoring implies a dedicated countdown timer.
- Attempts by the user to access shared-memory during periods in which atomicity is enabled. In fact, it is only shared-memory accesses that require network traffic that are problematic.
- Proper emulation of atomicity during buffering mode requires that the operating system be invoked when the user *exits* atomicity; this permits the arrival of sub-

| **User Controls** | Description |
|---|---|
| `atomicity_assert` | When set, prevents user-level `message_available` interrupts. In addition, if message is pending, enables atomicity timer; `ipicst` instruction briefly disables (presets) timer if `timer_force` not set. |
| `timer_force` | When set, forces atomicity timer to count down. Otherwise, timer counts down if `atomicity` set and message at head of queue. |

| **Kernel Controls** | Description |
|---|---|
| `dispose_pending` | Set by OS, reset by `ipicst`. |
| `atomicity_extend` | Requests an `atomicity_extend` trap on `disatom`. |

Table 2-4: Control bits for the user-level atomicity mechanism. These bits form the four-bit mask ($\mathtt{MASK}_{ATDE}$) and are controled through the `setatom`, `enabatom`, and `disatom` instructions.

sequent messages to be emulated through software interrupts. We call this the `atomicity_extend` trap.

- Monitoring of the output queue to detect network congestion that might indicate deadlock.

This last item is the duty of a separate mechanism, the atomicity congestion interrupt, that will be discussed in great detail in Chapter 4.

**Priority Inversion:** Note that there is a subtle issue that our atomicity mechanism must address. This is the issue of *priority inversion*. Priority inversion occurs when a low-priority interrupt is allowed interrupt a high-priority handler. In the User-Direct Messaging implemention, this results from the philosophy of allowing user-code to empty its own messages from the network; as a result, portions of a user-level message handler (low priority) may need to be invoked in the middle of high-priority section of kernel code[20]. The solution is to allow the atomic section of the user-level handler to extract the message from the network, followed by a return to high-priority kernel code. To avoid incorrect execution caused by reenabling of disabled interrupts, we will run the atomic section of the message handler with the same set of interrupts disabled as for the high-priority kernel code (with the atomicity mechanism disabling message interrupts as well). However, the execution of user-code is a type of priority inversion in and of itself. To limit the damage caused by this inversion, we need to make sure that (1) the period of time spent executing the atomic section of a user-level handler is bounded and (2) that the kernel regains control after the atomic section has finished. The first of these is addressed by making the countdown timer independent of the message at the head of the input queue. Thus, given our previous discussion, this timer has two modes – one that is tied to the presence of messages at the head of the input queue and another that is independent of such messages. The second issue is addressed by the same `atomicity_extend` trap that we mentioned above in conjunction with atomicity during buffering — it requires a trap that is triggered by the exit from atomicity.

[20]The assumption here is that we are running within the kernel, with message arrival interrupts enabled, but other interrupts disabled (and hence at an interrupt level that is higher than the “background” user-level).

**The Actual Atomicity Mechanism:** Control over user-level interrupts in Alewife is implemented by four atomicity control bits, a dedicated atomicity timer, and three atomicity instructions. Table 2-4 lists the control bits, which reside in the `uatomctrl` register. Two of the bits (tt dispose_pending and `atomicity_extend`) are modifiable only in kernel mode. The other two bits (`atomicity_assert` and `timer_force`) can be set or reset by the user *as long as the kernel-mode bits are zero* (more on this in a moment). The atomicity timer is one of the central features of the revocable interrupt disable mechanism; it is used to detect lack of forward progress in the network. When disabled, it continuously loads a prespecified countdown value. When enabled, it counts down toward zero — generating an `atomicity_timeout` interrupt if it ever reaches zero. The enabling and disabling of the timer is an aspect of the atomicity mechanism that we will discuss below. By dedicating this timer to atomicity detection, the cost of managing it becomes extremely low, permitting extrememly low-cost atomicity "instructions".

The `uatomctrl` bits can be modified by three atomicity control instructions, the `setatom`, `enabatom`, and `disatom` instructions, listed in Figure 2-6. These instructions take a four-bit mask as an operand, and uses that mask to respectively set, enable, or disable bits of the `uatomctrl` register. The `setatom` instruction may be used to directly set the state of all four bits; it is a priviledged instruction. In contrast, the `enabatom` and `disatom` instructions are "differential" in that they affect only the atomicity bits which are set in their corresponding masks; these instructions are "partially priviledged" in that the user is not allowed to modify the kernel-mode bits, and is allowed to clear the user-mode bits only if the kernel-mode bits are both zero. See Figure 2-6 for more details of this behavior of the `disatom` instruction. Note, also, that both `enabatom` and `disatom` return the previous value of the the `atomicity_assert` bit in the full/empty condition code (See Section 2.3.1 and 5.1.1).

The `atomicity_assert` bit is the most basic of the four control bits. When it is clear, the atomicity mechanism is disabled and the other bits are ignored. This means that messages at the head of the input queue generate either `user_message` or `system_message` interrupts depending on their message types[21]. When `atomicity_assert` is set, however, atomicity is *active*. This means that `user_message` interrupts are suppressed, that access to shared-memory is restricted, and the atomicity timer is engaged. Although `user_message` interrupts are suppressed during atomicity, the input queue still generates `system_message` interrupts when system-level messages advance to the head of the queue. Provided that the user continues to extract messages from the network, this means that the operating system can make forward progress even while granting the user full control over atomicity. None-the-less, it is one of our design assumptions that both system and user-level messages are carried on the same network and placed in the same queues. As a result, system-level messages could remain blocked (and unprocessed) behind user-level messages for arbitrary periods of time if it were not for the fact that the user atomicity mechanism bounds the total time that a message stays blocked in the network.

As mentioned earlier, access to shared memory must be restricted when atomicity is enabled since shared-memory requests may become blocked behind messages. Thus, when the `atomicity_assert` bit is set, shared-memory accesses are monitored by the user atomicity mechanism. Should the user attempt a shared-memory access that is incompatible with the current

[21] This is true unless message-interrupts are disabled by the system-level trap mask; however we are not discussing this at the moment.

state of the machine, then it will be faulted with a `stymied_shared_memory` trap. Such a trap acts as an entry into two-case buffering code. Although this is an implementation detail, Alewife supports three different modes of shared-memory monitoring during atomic sections. In the first mode, all accesses to shared memory are faulted. In the second, only shared-memory accesses that require network traffic are faulted. Finally, in the third option, only shared-memory accesses that require network traffic and which are blocked by a message at the head of the input queue are faulted. This third option is the least restrictive of the three. It is important to note that *all restrictions with respect to shared memory and message atomicity are side-effects of our single-network restriction*. This important conclusion of this thesis will be revisited in Chapter 7 when we suggest at least two logical channels in a network.

When the `atomicity_assert` bit is set, the timer is automatically *engaged* (but not necessarily enabled). It is disabled when this bit is clear. When engaged, `timer_force` distinguishes between one of two different behaviors for the timer: monitor or countdown. If the `timer_force` bit is clear, then the timer is in monitor mode, meaning that it is enabled whenever there is a user-level message at the head of the input queue and disabled otherwise. In addition, it is reset briefly whenever a message is discarded, which occurs during the execution of an `ipicst` instruction with an infinite skip or length operand. In monitor mode, the countdown timer ensures that user-level messages never sit at the head of the message queue for longer than the countdown time. This mode is appropriate for polling as well as the atomic sections of non-priority-inverted message handlers. In contrast, if the `timer_force` bit is set, the timer is enabled regardless of the presence or absence of a message at the head of the input queue. This mode is useful for atomic sections of priority-inverted handlers, since it guarantees that the atomic section will be of bounded duration; in this case, it must be coupled with the `atomicity_extend` bit described below.

Note that the actual value of the atomicity timer is a detail unrelated to correctness. However, if it is set too low, it may have a non-trival impact on performance, since it could cause frequent revocation of user-level access to the network. Although the second case (buffered) delivery may be relatively low overhead, it may still increase message overhead by a factor of two or three[22]. Further, if the timeout is too low, then the user may cause excessive network congestion unwittingly. Thus there is a careful balance to be struck. To help in achieving a balance, *the atomicity timer is designed to count only user cycles.* This permits the atomicity timer to be tied directly to the user-level instruction stream, rather than allowing non-deterministic timer behavior caused by kernel interrupt handlers. Further, the user is allowed to set the `timer_force` bit during polling to achieve similiar deterministic behavior in timing — *i.e.* timeouts become tied to the instruction stream rather than to the non-deterministic behavior of the the network.

The final two atomicity control bits may be modified only by the kernel. If either of these bits are set, attempts by the user to clear either `atomicity_assert` or `timer_force` cause synchronous exceptions to the kernel. As we will see in a moment, these bits must be cleared when the user attempts to exit atomicity. Hence, if either of these bits are set, then a user's attempt to exit atomicity will trap into the kernel. The first kernel bit, `dispose_pending` is used to ensure that a user-level interrupt handler extracts at least one message before exiting atomicity. This is to avoid a recursive interrupts. Before calling a user-level interrupt handler, the kernel sets this bit. It is cleared as soon as the user discards at least one message. Hence, user attempts

[22]This may seem like a lot, but the initial hardware delivery overheads are extremely low.

to exit atomicity while `dispose_pending` is set cause a `dispose_failure` exception. The second kernel bit, `atomicity_extend`, is used to cause the kernel to be reinvoked after the user disables atomicity. Hence, user attempts to exit atomicity while `atomicity_extend` is set (and `dispose_pending` is clear) cause an `atomicity_extend` exception. This extension mechanism has a number of uses. For the atomic sections of priority-inverted handlers, the `atomicity_extend` trap is used to return control to the kernel after the end of the atomic section of the handler. In Section 4.3.5 we will see how the `atomicity_extend` mechanism is an integral part of the software emulation of network queues in during two-case delivery.

```
enable_message_atomicity() ⇒
  enabatom   ATOMICITY_ASSERT                      ; Previous atomicity_assert in full/empty.

disable_message_atomicity() ⇒
  disatom    (ATOMICITY_ASSERT|TIMER_FORCE)        ; Previous atomicity_assert in full/empty.
```

Figure 2-12: Translation of UDM atomicity operations into UDM interface instructions.

**Implementation of the UDM atomicity operations:** Given the atomicity mechanism as we have previously described it, translation of the abstract user-level atomicity operations `enable_message_atomcity` and `disable_message_atomicity` into UDM instructions is now trivial. Figure 2-12 illustrates this translation. When user-code enables atomicity (such as for critical sections or during polling), it merely needs to use an `enabatom` instruction with the `ATOMICITY_ASSERT` mask. This instruction sets the `atomicity_assert` bit of the `uatomctrl` register and returns the previous value of this bit as a full/empty condition code. The full/empty code can then be tested with the coprocessor conditional branch instructions (see Section 5.1.1). Other, more complicated entries into atomicity occur through the kernel, and may involve more complicated masks. To exit atomicity, we use the `disatom` instruction, but request clearing of both `ATOMICITY_ASSERT` and `TIMER_FORCE`. Once again, the previous value of the `atomicity_assert` bit is returned as a full/empty condition.

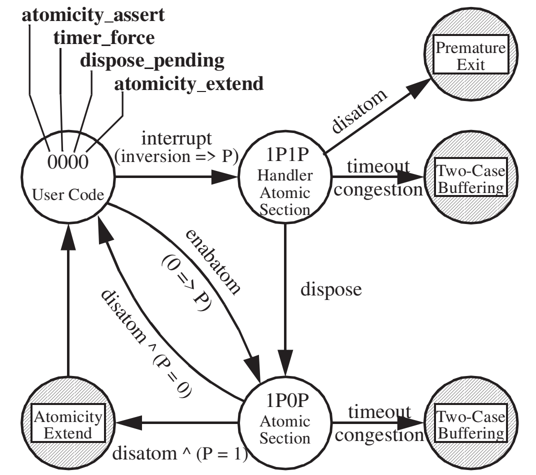



Figure 2-13: Four primary states for the user-level atomicity mechanism.

To the extent that hardware use of the atomicity mechanism is a common-case operation, we can summarize user-atomicity with a state diagram containing three primary user states, an "extension" state for priority inveresion, and several exception arcs. This is shown in Figure 2-13. In this figure, the three primary user states are unshaded and represent code running at user level, while the shaded states represent kernel handlers for exceptional conditions. The first of the user states, *user code*, represents code running outside of a critical section; the second, *handler atomic section* is a critical section which cannot be exited without disposing of a message; the third, *atomic section*, is a "normal" critical section. Each of the primary user states are annotated with values for the atomicity control bits. Note that two of the bits, namely `timer_force` and `atomicity_extend` are labeled symbolically with the letter "P"; this means that these bits are set to the same value, but that this value depends on circumstances[23].

There are three types of kernel exception arcs in Figure 2-13. The first, *two-case buffering*, represents entry into buffering code and can be triggered by the atomicity timeout, as well as shared-memory or network congestion. Section 4.3.5 and Figure 4-8 (page 136) cover buffering in greater detail. The second, *atomicity extend*, is used here exclusively for recovering from priority inversion and is triggered at the close of the atomic section of a priority-inverted handler. The last, *premature exit*, is an error condition that results when the user neglects to free at least one message before exiting the atomic section of a message handler.

This diagram embodies four different scenarios:

POLLING: To poll, the user simply executes a `enable_message_atomicity` operation. This follows the arch from *user code* to *atomic section*, setting the "P" bits to zero. Assuming that no timeout or congestion has occurred, polling is exited by executing `disable_message_atomicity`, which returns the user to *user code*.

CRITICAL SECTION: This is identical to the Polling scenario above, except that it tends to be much shorter.

NORMAL INTERRUPT HANDLER: When a user-level message handler occurs, the kernel causes a transition from *user code* to *handler atomic section*. The "P" value is set to zero. Note also that the `dispose_pending` bit is set to one. Should the user attempt to exit atomicity without disposing of a message, then the exceptional condition of *premature exit* will occur. As soon as the user disposes of a message, then we enter the *atomic section* state. From this point on, the remainder of the atomic section of the handler is identical to a normal critical section.

PRIORITY INVERSION: The only difference between a priority inverted handler and a normal handler is that the `timer_force` and `atomicity_extend` bits are set. This means that (1) the duration of the handler is bounded and (2) after the message is disposed, execution of `disable_message_atomicity` will invoke the *atomicity extend* code to reenter the kernel.

It is important to note that the third item above (`Normal Interrupt Handler`) is a bit different from most other systems. The fact that an interrupt handler is transformed transparently into a "normal" thread is evidenced by the fact that we make no distinction between entry to the *atomic section* state from the background task or and interrupt handler. This is only possible because of our use of featherweight threads to give each interrupt handler a complete thread context. As discussed in Section 2.2, other alternatives (such as Optimistic Active Messages [114]) construct a thread context on the fly, only if necessary. Interestingly enough, the user-level atomicity

[23] See the full-blown state diagram in Figure 4-8 for examples in which these bits are not identical.

mechanism as defined here is perfectly compatible with this methodology – if all passes through the *handler atomic section* state set the `atomicity_extend` bit, then exit of the atomic section would always invoke the atomicity extend handler, which could then construct a thread context.

### 2.4.6 Putting it all together: hardware support for user-level interrupts

The Alewife system supports "user-level interrupts" only because the kernel message interrupt handler quickly converts a system-level interrupt into an upcall to user code and because the Alewife operating system provides a system call to perform a return from interrupt. Thus, the manipulations to `uatomctrl` required for a user-level interrupt handler are added in software at the system level. However, a processor *could* manipulate the `uatomctrl` bits directly: the *interrupt* arc of Figure 2-13 could occur entirely in hardware under those circumstances for which a free featherweight thread was available. When coupled with a user-level return from interrupt, this mechanism would produce an extraordinarily low-overhead, user-level message interupt.

What we have described above is sufficient mechanism to provide an extremely fast thread dispatch to handle user-level interrupts. Unfortunately, this does not address the network output descriptor array. As described in Section 2.4.3, the descriptor array is multiplexed through a process of "callee-saves", which means that an interrupting thread must be careful to save any state from the interrupted background thread before sending messages. Of course we cannot trust a user to do this properly, especially since the background thread may belong to the operating systems under special situations (priority inversion). Consequently, Alewife accomplishes this in the thin layer of operating systems code that is run before dispatching the the user-level handler: this code performs a quick check of the `desc_length` register to see if the interrupted thread had written anything to the output window. If so, it saves the registers, then alters the return address of the interrupted handler to point at a special stub that will reload them. In this way, the machine state becomes a part of the interrupt thread context. Further, the new thread is free to send messages as it wishes.

One of the premises behind using callee-saves to multiplex the descriptor array was that interrupting a background process in the middle of an atomic send sequence is "unlikely" (low probability). Hence, in continuing our exploration of hardware support for user-level interrupts, we note that it would makes sense to continue to dispatch user-level arrival events directly in hardware as described at the beginning of this section, while treating the case of a non-zero `desc_length` register as an exceptional case that interrupted to the operating system[24].

### 2.4.7 User-Direct Messaging in a Multiuser System

As designed, Alewife is a single-user system. There are several reasons for this, but two are particularly compelling: First and foremost, the Alewife project was studying a number of issues of cache-coherence, multithreading, fine-grained communication, and integrated message passing and shared memory; multi-user timesharing would likely have represented one too many areas of

[24] An alternative to this would be to provide a separate descriptor array for each register context, which is somewhat akin to the approach adopted by the M-machine [40] which uses the register file to hold outgoing messages before launch. This, of course, has direct roots in the Alewife design.

focus. Second, the absence of virtual memory made construction of a multiuser operating-system difficult. As described in the footnote on page 44, there were a number of reasons that virtual memory was not included in the Alewife prototype, not the least of which was expedience.

So, to what extent can we generalize the solutions of Alewife to a multiuser system? With respect to communication models, the answer is that the semantics and most of the implementation details are completely sufficient for a multiuser system. Multi-user support for shared memory is easily provided through virtual memory (as has been done for shared-memory multiprocessors for many years now); hence, we do not really need to discuss it further. Multi-user support for message passing, while not as straightforward, is greatly simplified by the presence of two-case delivery. That is the topic of this section.

Before discussing this, we would like to say a brief word about the model of scheduling that we are assuming. One possible model is strictly *space-shared* and *gang-scheduled*. Such machines, of which the CM-5 is one example, are physically partitioned so that individual jobs have exclusive use of a set of nodes during their timeslices. Such machine/operating-system combinations have advantages of simplicity, in that design of the message-passing interface does not have to deal with an intermixing of messages between different processes. Unfortunately, this type of partitioning can provide an extremely inflexible and inefficient use of resources. Note, in closing, however, that User-Direct Messaging *could* be used directly, without modification in such a system.

However, the model of timesharing that we would like to target here is much more flexible: *loose gang-scheduling*. This type of scheduling assumes no restrictions on the intermixing of threads from different processes within a given machine. Processes may expand to occupy an optimum number of nodes; this type of scheduling allows simultaneous execution of large multi-node parallel programs with smaller tasks. The scheduler attempts to schedule threads in groups so that all of the nodes from given process are "usually" running at the same time. What this heuristic ensures is that, while messages from different processes may be intermixed within the same network, the common-case situation is for messages to arrive at nodes that are expecting them, *i.e.* that it is a low probability event for messages from one process to arrive at a node which has another process scheduled. If this type of "message mismatch" is uncommon, then we can take advantage of this in our architecture.

Two-case delivery is an architectural innovation that enables loose gang-scheduling. The reason for this is that buffered delivery of messages is explicitly built into the messaging mechanisms; in the case of a multiuser system, we simply extend the basic buffering scheme to include multiple message buffers, one per process. As a consequence, buffering is always available as an option for a wide array of message delivery problems, including situations in which messages arrive for processes that are not currently scheduled — we handle the mismatched arrival of a message at a node simploy by buffering the mismatched message and delivering it later, when the appropriate process is scheduled. The extent to which mismatched message delivery is an uncommon situation is out of the domain of discussion, but is addressed elsewhere[78].

Given two-case delivery as a method for multiplexing traffic from different users, we can make use of the User-Direct Messaging model unmodified in a multiuser system. In fact, there are only two issues that must be addressed from an *implementation* standpoint:

**1.** A *Process-ID stamp* (PID) must be prepended to each outgoing message by the message injection hardware. The stamp records an identifier of the sending process. At the receiver, the PID of the message is compared to that of currently active process. If it matches, then reception occurs exactly as before. On the other hand, if the PID does not match, then a special `message_mismatch` interrupt occurs which invokes buffering software. This permits direct hardware delivery in those situations for which the scheduler has properly coscheduled senders and receivers.

2. The ability to handle virtual-memory based DMA for messages. To extend the user-level DMA of User-Direct Messaging to a system with virtual memory requires careful handling of various types of page faults that may occur. This can be handled with careful scheduling and manipulation of the free-page cache. This is beyond the topic of the current discussion, however. See [77].

Thus, the User-Direct Messaging model directly generalizes to a multiuser system with little extra mechanism. Note that this result stems from the unique mix of hardware and software represented by two-case delivery.

## 2.5 The Interaction Between Communication Models

Having discussed both the Latency-Tolerant Shared-Memory and User-Direct Messaging communication models in detail, we would now like to explore the consequences of their integration. Both of these communication models are self contained and functional when used separately; thus, we would like to see how they interact when when used together. We are looking for two different things: (1) ways in which the combined use of models can lead to interference or deadlock and (2) ways in which the semantics of the models are sufficiently mismatched that care must be made at interaction boundaries.

We respect to the first of these, it is important to restate an assumption of the Alewife design: the use of a single logical network channel. The fact that Alewife was restricted to utilizing only a single logical network channel was a constraint imposed by the choice of the EMRC network chips from Caltech: these chips form a network with a single (bi-directional) network channel. Note that, as a partial solution of the *protocol deadlock problem* (Chapter 4), the Stanford DASH multiprocessor utilized the EMRC to achieved two independent network channels, but this was done by doubling the amount of hardware and interconnection resources[25]. This was not deemed a reasonable cost in Alewife. Consequently, the Alewife design was restricted to functioning with a single logical network channel.

In some cases, this restriction lead to better solutions (as design within constraints often does). The methodology of two-case delivery, discussed in detail in Section 4.3, was one positive result of restricting the number of network channels to one. Unfortunately, some of the inter-model interactions of this section represent negative consequences of such a restriction. In particular, the presence of a single logical channel impacts the interface between shared memry and message passing, in that the semantics or behavior of one can block or impact the other. In some cases,

[25] Actually quadrupling, since they doubled the width of the network channels, but this is not relevant here.

as we shall see, the user's model of integrated communication suffers directly, becoming more complex as a result.

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

- The DMA coherence problem. The interaction between messages using DMA transfer and cache-coherence. Our solution, *local coherence*, guarantees coherence at message endpoints. This means that data at the source and destination are coherent with respect to local processors. *Global coherence* can be achieved through a two-phase software process.
- Restrictions on shared-memory accesses during message atomic sections. Because of restrictions on network communication, shared-memory accesses made by message handlers and during polling can lead to deadlock. Solution to this problem involves restrictions imposed on system-level users of the network interface, and virtualization of atomicity through the user-level atomicity mechanism.
- The refused service deadlock. The combination of blocking semantics for shared memory and the need for processor handling of messages can cause a type of priority inversion that leads to deadlock. Our solution involves a special type of *synchronous interrupt* called a *high-availability interrupt*.

The following sections explore the first two of these issues, namely the *DMA coherence problem* and *message handler restrictions*. The DMA coherence problem is a natural outcome of attempts to combine DMA and cache-coherent shared memory; our choice of solution (use of *locally coherent DMA*) is driven by considerations of common-case usage, implementation complexity, and characteristics of the network. The restrictions on message handlers, in contrast, derive almost exclusively from restrictions on network communication and requirements for two-case delivery; several of these restrictions could be lifted with hardware enhancements. The third problem, namely the *refused service deadlock*, is one of the service interleaving problems; since it can be solved in a way that is completely transparent to the user, we will defer discussion of it until Section 3.1.

### 2.5.1 The DMA Coherence Problem

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

3. Globally-Coherent DMA: Data at the source and destination are coherent with respect to all nodes. Source memory-lines which 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.

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 which 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 which already supports a cache-invalidation mechanism. Non-coherent DMA can give better performance for messages which 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. While attractive as a "universal mechanism", globally-coherent DMA is not necessarily 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 Chapter 6 demonstrates, message passing is useful as a way of *bypassing* the coherence protocol. In fact, many of the applications of message-passing discussed in [58] 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 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 which may prevent the simple "divert and relaunch" scheme described above for network overflow recovery: input packets which 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:** The above discussion does not mean that globally-coherent DMA cannot be accomplished, however. The key is to note that software which desires 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 needs to access local copies only. 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.

At the time that Alewife was designed, it was an open question as to 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[26]. As a consequence, the Alewife machine provides no hardware assistance for achieving global coherence — it must be driven by software.

However, there is at least one situation for which hardware support of global coherence would be have been desirable: page migration. In particular, one experimental system that was built on top of Alewife was the Multi-Grained shared-memory (MGS) system[120]. This system provided a shared-memory abstraction on top of a cluster of Alewife machines[27]. Communication within each machine was via hardware shared-memory, with software maintained coherence between machines. When moving pages from one machine to another, globally-coherent DMA was necessary; on Alewife, this was accomplished by prefetch loops for page cleaning. In this system, the software overhead of cleaning was a non-trivial fraction of execution for a number of applications. Note also that easy page migration is an important aspect for operating-systems design.

Hence, in retrospect, support for page collection and cleaning would have been good features to include for Alewife.

### 2.5.2 Message Atomicity and Shared-Memory Access

A second interaction between shared memory and message passing that we wish to discuss is between message handlers and shared memory. As described in Section 2.4.1, interrupt handlers begin with interrupts disabled, *i.e.* with an atomic section. This is necessary to avoid recursive interrupts. Unfortunately, during periods in which message interrupts are disabled, no forward progress of the network as a whole may be guaranteed. We will discuss this in detail in Chapter 4. However, one immediate consequence that we would like to mention here is the fact that this lack of forward progress in the network directly impacts the set of operations that may appear in message handlers; in particular, *message handlers may not access shared memory*. This particular constraint is a direct consequence of the fact that both shared-memory and message-passing network traffic are carried on a single logical network channel: if a message is blocked at the head of the network input queue, it can stretch into the network, preventing the passage of shared-memory protocol packets.

This simple interaction has a number of consequences:

- It is one of the causes of the refused service deadlock. We have already pointed out that we will "fix" this problem in the next chapter.

[26] An option which is uniquely available with Alewife.

[27] Actually, it was a simulated cluster — *i.e.* a partitioning of one large machine.

- As we begin to unravel the Alewife implementation, we will see that accesses to the shared-address space must be treated carefully at many levels of the implementation, simply because such traffic has the *potential* to generate network traffic; even if this is not *actually* true (*e.g.* the operating systems designer has carefully reserved a block of memory that is guaranteed to be unshared), this information is buried too deeply (in the protocol directory) for higher levels of hardware to take advantage of. Since all but the most trivial of interrupt handlers must have access to memory, this implies some other class memory, leading to yet another need for *local unshared memory*.
- This prohibition toward access of shared memory extends to all atomic sections; *including periods of polling*.

This last issue is one of the most restrictive interactions, because it prevents integration of shared memory and polling message passing. In fact, as we will discuss in Section 4.3.5, this problem can be masked by appropriate exceptional cases in the user-level atomicity mechanism, triggering two-case delivery when deadlock would otherwise result. Unfortunately, this can make second-case delivery (*i.e.* through software) much more frequent than direct hardware delivery.

**Operating-System Message Handlers:** Although the user-level atomicity mechanism can mask deadlock issues from the user, portions of the operating system must still be able to deal with the raw hardware interface. (This is akin to the fact that portions of all operating systems must operate in real-mode, despite the fact that virtual memory exists.) Further, the software overhead involved in masking deadlock issues issues is an overhead that is neither necessary nor desirable for correctly written code; as an example, the LimitLESS cache-coherence protocol relies on system-level software handlers for part of its implementation; extraneous software overheads add directly to the cost of a remote memory accesses. Thus, for the operating system, we would like to codify a set of rules that interrupt handlers must follow to be correct.

Taking into account the network-overflow methodology of Chapter 4, these rules may be summarized as follows. Before a system-level handler accesses shared-memory, it must:

- Reenable the network overflow interrupt.
- Free the input packet *completely* and reenable network arrival interrupts.
- Release any low-level hardware locks which will defer invalidations in the interrupted code.

The first of these arises because all global accesses have the potential to require use of the network. Consequently, they can be blocked indefinitely if the network should overflow. The network overflow handling mechanism is discussed in Chapter 4. The second is the shared-memory/message-passing interaction mentioned above. Finally, the last condition prevents deadlocks in the thrash elimination mechanism, *i.e.* avoids the *internode deadlock* illustrated in Figure 3-10 and its surrounding text (page 98)[28].

[28] In fact, in the case of user-level message handlers, we treat this as a form of priority inversion; as soon as the atomic section completes, we return to perform the shared-memory access required to release the lock. This is aided by support in the A-1001 CMMU; see Section 6.1.3.

## 2.6 Postscript

In this chapter, we have explored the high-level aspects of integrating message-passing and shared-memory communication models. This involved three major aspects: (1) a threading model, called *featherweight threads*, whereby threads are inexpensive enough to use for latency tolerance and interrupts as well as background processes; (2) shared-memory and message-passing communication models (called, respectively, *Latency-Tolerant Shared Memory* and *User-Direct Messaging*) that are compatible with one another, provide reasonable functionality, and whose interactions may be easily characterized; and (3) development of a set of interfaces for these communication models. With respect to the last of these, we have taken the approach that interfaces are extensions to the ISA. This led us to develop interfaces that can be represented in a small number of actual instructions. In this process we follow in the footsteps of of people such as Bill Dally who have espoused the view that communication should be viewed as being as fundamental as addition and multiplication[34].

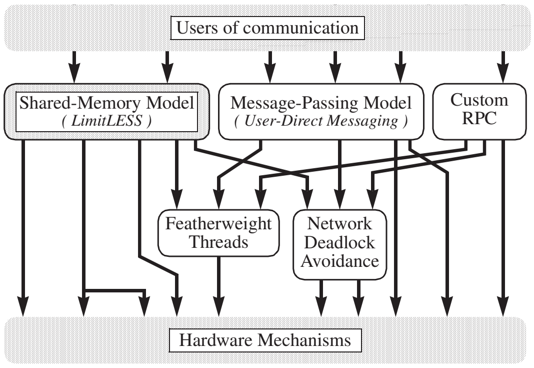


Figure 2-14: Exported communication models in the Alewife multiprocessor.

This is our solution to challenge of user-level access. The result of this chapter may be summarized by Figure 2-14, which reproduces the middle layers of Figure 1-3 (page 27). The only aspect of this diagram that we have yet to discuss is the deadlock-avoidance box; that is the topic of Chapter 4.

---

**Collaborators:** As with any large systems project, Alewife intermixed the contributions of many people. The featherweight threading concepts and code were built on top of existing threading facilities developed by David Kranz, Beng-Hong Lim, and Dan Nussbaum. David Chaiken was instrumental in developing the Alewife cache-coherence protocol and was an important collaborator in the development of hardware mechanisms for implementation of LimitLESS cache coherence. David Kranz provided the original impetus to develop user-level messaging on Alewife. Ken Mackenzie was an important collaborator in the design of the UDM model; he was also responsible for many of the multiuser aspects of UDM. Kirk Johnson and Donald Yeung were the first users of the software version of the user-level atomicity mechanism, providing crucial feedback that lead to the hardware user-level atomicity mechanism presented here (and in Chapter 4).

---

# Chapter 3

# The Service-Interleaving Problem

The *raison détre* of multiprocessing is the exploitation of parallelism to increase the performance of applications. Not surprisingly, parallelism at the application level leads to parallelism in the underlying communication substrate. In this chapter, we will examine several instances for which communication parallelism leads to potential livelocks and deadlocks, especially when shared-memory and message-passing communication are intermixed. We will group this set of problems together and refer to them collectively as the *Service Interleaving Problem*.

The Service Interleaving Problem in an integrated architecture such as Alewife arises from the presence of uncontrolled simultaneity. This simultaneity arises from several sources: First, by their very nature, multiprocessors support multiple instruction streams (at least one per node), each of which can generate shared-memory communication traffic. Second, latency-tolerance techniques such as prefetching and rapid-context switching serve to overlap communication by increasing the number of outstanding requests from each node. Third, message-passing interfaces can give rise to an unbounded number of outstanding requests from each node. Hence, the memory system and network must correctly support many simultaneous requests. Fourth, the boundaries between shared memory and message passing can be explicitly crossed when hardware events are handled by software (such as for the LimitLESS coherence protocol[23, 21]); this can introduce another type of simultaneity in which cache-coherence structures are access by both hardware and software.

Hence, any solution to the Service-Interleaving Problem involves imposing order on this chaos. In particular, two requirements must be met:

1. All communication operations must complete eventually.
2. All communication operations must complete correctly.

The first statement asserts that the system is free of both livelock and deadlock. It requires the solution of three problems, the *refused service deadlock*, the *window of vulnerability livelock*, and the *server-interlock problem*. The second statement implies the existence of a cache-coherence protocol to maintain data consistency, and the presence of explicit mechanisms to deal with message and event reordering. We say that these mechanisms eliminate *protocol reordering sensitivities*.

The refused-service deadlock results from an interaction between the blocking semantics of memory (ultimately resulting from finite processor resources) and the need for processor handling of messages. This is an unavoidable consequence of the integration between message passing and shared memory. The next two issues, the window of vulnerability livelock and the server-interlock problem, are are duals of one another and result from the combined presence of a cache-coherence protocol and multiple threads competing for limited hardware resources. Finally, the set of protocol reordering sensitivities result from simultaneity in the hardware at a number of levels.

This chapter describes each of the four facets of the Service-Interleaving Problem, and presents solutions in the context of Alewife. As we shall see, one of the primary results of this chapter is the notion of a *transaction buffer*. A transaction buffer is a fully-associative store of data and state that sits between the processor and memory (both local and remote). It is used to explicitly track the state of outstanding transactions from the processor's standpoint; this type of tracking provides sufficient information to correct for all sorts of problems: inter-processor data thrashing, cache replacement thrashing, network reordering, mixed hardware/software handling of cache replacement, guarantees that interrupt handlers can execute. In addition, three other aspects of the solution to the Service-Interleaving Problem may be summarized as follows:

1. Guarantee that high-priority asynchronous events can be delivered.
2. Guarantee that *at least one* writer eventually succeeds.
3. Provide locks on contended hardware data structures.

It is our goal in this chapter to show how to solve the service-interleaving problem from a high-level viewpoint; as such, we will present the issues and discuss several different possible solutions. Later, Section 5.2.3 will present the Alewife transaction buffer in detail, showing how the various properties that we have attributed to it are embodied in hardware mechanisms.

## 3.1 The Refused-Service Deadlock

Integrated architectures such as Alewife can be subject to "multi-model interference" caused by interactions between communication mechanisms. Figure 3-1 illustrates one such scenario that arises in the Alewife multiprocessor, called the *refused-service deadlock*. In this figure the processor has exhausted its ability to issue additional memory transactions and is thus blocked awaiting the completion of previously-issued requests. As shown in this figure, the shared-memory responses that it is waiting for are blocked in the network, behind messages. Unfortunately, this situation will not resolve itself because the removal of messages at the head of the queue requires action by the processor, either explicit free operations (if the processor is polling for messages) or the posting of asynchronous interrupts (if the processor is relying on interrupt notification). The fact that the processor will not free messages if it is polling is straightforward, since

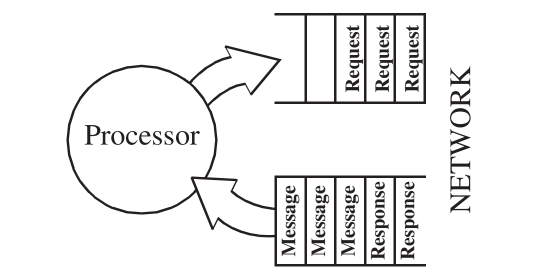



Figure 3-1: The refused-service deadlock. The pipeline is blocked awaiting completion of shared-memory transactions.

the processor is unable to continue executing instructions in the current (polling) thread. However, the processor will not free these messages, *even if it has message interrupts enabled:* since its capacity for issuing memory requests has been exhausted, the processor cannot issue requests for interrupt handler instructions; further, if the pipeline is frozen, the asynchronous interrupt condition may not even be registered by the processor. Hence, Figure 3-1 represents a deadlocked situation regardless of the current mode of message notification.

### 3.1.1 Blocking Memory Operations Cause Deadlock

At the risk of belaboring the point, the situation in this figure arises for two reasons: First, processor pipelines have finite resources that may be used for outstanding requests to the memory system; for example, the pipeline of the Sparcle processor on Alewife can have only one outstanding memory transaction, while more recent processors such as the Power PC or the R10000 support four to eight outstanding requests. Finite resources are a fact of implementation, and hence this potential exhausting of resources is unavoidable. The second reason, however, results from an explicit design decision: the use of a single logical network channel. If shared-memory and message-passing network traffic were completely decoupled, through use of multiple networks or virtual channels[31], then dependencies such as shown in Figure 3-1 would not occur: the shared-memory responses would simply bypass the messages, unblocking the pipeline, and hence avoiding deadlock. As with similar deadlocks, we have two components: finite resources (processor load buffers) and interlocking dependencies (interference between shared memory and message passing).

Although this particular instantiation of the refused-service deadlock may be eliminated by including more than one logical channel in the network, other instances are less avoidable. In fact, the refused service deadlock occurs under any circumstances in which the memory interface is used for operations that (1) may block and (2) may not complete without processor action. In a multiprocessor such as Alewife that makes copious use of software handlers to resolve exceptional situations, examples of this sort of interlocking abound. For instance, as we shall see in Chapter 4, Alewife makes use of a software scheme called *two-case delivery* to recover from protocol deadlock in the network. Once again, this sets up a dependency in which shared-memory operations may not complete without the posting of an atomicity congestion interrupt and the subsequent execution of the network overflow recovery handler. A related example is that of the blocking aspects of the network output interface discussed in Section 2.4.2: attempts to describe a message by writing to the output interface may blocked pending drainage of the network. If the network happens to be deadlocked, then only the execution of the network overflow handler will permit forward progress. One slightly different example involves the LimitLESS cache coherence protocol: since software is an integral aspect of this protocol, then it may be the case that a local operation memory request may not complete without the execution of a local interrupt handler.

### 3.1.2 High-Availability Interrupts

In solving the refused-service deadlock, we would like to separate two different subclasses of this deadlock by the method in which the dependent condition would be resolved: *interrupts vs. polling*. The cases in which interrupts are prevented by pending memory operations is the more pernicious of the two, simply because most exceptional conditions are handled via interrupts. The fact that interrupts can be prevented from occurring is a type of priority inversion: high-priority asynchronous interrupts are deferred pending completion of lower-priority dependent operations. Priority inversions are invariably bad, because interrupt hierarchies is designed with forward progress (or timeliness) in mind; any violation of these hierarchies will likely compromise correctness or cause deadlock. The polling example represented by Figure 3-1 is fundamentally different because polling represents normal (non-exceptional) execution. Thus, the polling version of the refused service deadlock can be avoided in several ways: (1) by requiring the user to avoid those situations that cause deadlock, (2) by increasing the level of resources in, for instance, the network to eliminate this problem, or (3) by recognizing when a polling execution cannot complete and invoking an interrupt to clean up the situation.

Hence, unlike polling versions of the refused-service deadlock, the interrupt version is far more worrisome: it can prevent forward progress regardless of the care with which the user constructs code (exceptional conditions are often related only indirectly to the current execution stream)[1]. To solve the refused-service priority inversion, we must find a way to deliver interrupts during those periods in which the processor pipeline is blocked on memory actions. In order to do this, we must recognize that under "normal" circumstances (those for which typical pipelines are designed), memory operations never block indefinitely — they either complete or fault. Since completion of outstanding requests is not an option, this leaves faulting. Thus, to deliver high-priority asynchronous interrupts under exceptional circumstances (*i.e.* when the pipeline is blocked on dependent operations), we must fault one or more of the memory operations that are currently in progress; further, we must redirect the fault handler for these memory operations so that they vector to an appropriate interrupt handler, rather than to a memory error handler. Later, after the interrupt returns, the faulted memory operations will be retried.

Asynchronous interrupts that can be delivered by faulting in-progress memory operations under exceptional circumstances are called *high-availability interrupts*. A number of Alewife interrupts are high-availability: reception interrupts, atomicity congestion interrupts (for network deadlock recovery), local LimitLESS faults, and even the timer interrupt is high-availability in order to ensure timely delivery of clock events. Note that the presence of high-availability interrupts introduces the possibility of livelock, since in-progress memory operations are faulted and retried. In fact, this situation represents an instance of the window of vulnerability livelock that will be discussed in the next section (Section 3.2).

The memory fault that is employed when invoking a high-availability interrupt is closely related to standard memory faults that are used for memory errors (such as ECC). However, to aid in proper vectoring of high-availability interrupts, some distinguishing characteristic is desirable. As discussed in Section 5.1.1, the Sparcle processor includes several synchronous fault lines for the express purpose of registering high-availability interrupts. However, in the Alewife implementation, there are enough high-availability interrupts that the set of hardware synchronous trap lines are insufficient for complete vectoring; hence, the high-availability trap vector performs an additional level of software dispatch based on a special hardware status register in the CMMU, called

[1]Note that we are not claiming that the polling version of the refused-service deadlock is unimportant, merely that it affects the communication model seen by the user rather than system-level forward progress; we will have more to say about this when we discuss the user-level atomicity mechanism in Section 4.3.5.

the `AuxFault_Vector`. This register is set to reflect the type of high-availability interrupt that has just occurred.

### 3.1.3 Interrupt Promotion Heuristics

Since high-availability interrupts are asynchronous interrupts that are delivered synchronously under special conditions (*i.e.* by faulting memory operations), some heuristic must by used to determine when to promote asynchronous memory operations to synchronous ones. Under the assumption (which is true for Sparcle) that synchronous delivery is more expensive than asynchronous delivery, we would like this delivery heuristic to be as conservative as possible. In practice, some types of interrupts are easier to design conservative heuristics for than others. For instance, with the message arrival interrupt, the set of promotion conditions are straightforward: if the processor is waiting on a remote access and a message arrival interrupt is pending, then this interrupt is promoted to synchronous delivery. As discussed in Section 5.2.2.2, high-availability interrupts are controlled directly from within the primary cache-control state machine on the Alewife CMMU. This permits high-availability interrupts to be promoted to synchronous delivery only in the `Resource_Wait` state of this machine, *i.e.* when all other options for memory access have failed. To choose which high-availability interrupt to deliver, special high-availability priority logic examines the set of pending high-availability interrupts and a set of machine state information (*e.g.* in the case of message-arrival interrupts, the fact that the processor is waiting for a remote access).

## 3.2 The Window of Vulnerability Livelock

This section introduces the *window of vulnerability* and describes how the presence of this window can lead to livelock. The window of vulnerability arises in split-phase shared-memory accesses. The *window of vulnerability livelock* refers to a situation in which an individual thread of control is unable to complete a shared-memory access: each time that the thread requests shared-memory data, it is interrupted before data arrives, only to return to find that the data has been invalidated by another node. This livelock is particularly prevalent in a multiprocessor with rapid context-switching, because such a system is designed to switch threads immediately after making shared-memory requests; as a consequence, two threads on different nodes can prevent each other from making forward progress by attempting to access the same memory-line. However, the window of vulnerability livelock can also occur a system which mixes shared memory and message passing, since message arrival can interrupt threads awaiting shared data.

After introducing the window of vulnerability livelock, this section proceeds by investigating several methods for removing this livelock. It culminates in a unifying architectural framework, the *transaction store*. As will be discussed in Chapter 5, the complete framework has been implemented in the Alewife machine; however, other multiprocessor architects may choose to implement a subset of this framework. To this end, Sections 3.2.5 to 3.2.8 discusses several partial solutions, each of which are appropriate to a different subset of mechanisms.

### 3.2.1 Multi-phase Memory Transactions

Coherent caches are widely recognized as a useful technique for reducing the bandwidth requirements of the shared-memory programming model. Because they automatically replicate data close to where it is being used, caches convert temporal locality of access into physical locality. That is, after a first-time fetch of data from a remote node, subsequent accesses of the data are satisfied entirely within the node. The resulting cache coherence problem can be solved using a variety of directory based schemes [49, 69, 23]. Unfortunately, even with coherent caches, the cost of remote memory actions can be prohibitive. To fetch data through the interconnection network, the processor transmits a request, then waits for a response. The request may be satisfied by a single memory node, or may require the interaction of several nodes in the system. In either case, many processor cycles may be lost waiting for a response. Applying basic pipelining ideas, resource utilization can be improved by allowing a processor to transmit more than one memory request at a time. Multiple outstanding transactions can be supported using software prefetch [20, 85], multithreading via rapid context-switching [115, 5], or weak ordering [1]. Studies have shown that the utilization of the network, processor, and memory systems can be improved almost in proportion to the number of outstanding transactions allowed [63, 50].

Multithreading may be implemented with either *polling* or *signaling* mechanisms. Polling involves retrying memory requests until they are satisfied. This is the behavior of simple RISC pipelines (such as Sparcle) which implement non-binding prefetch or context-switching through synchronous memory faults. Signaling involves additional hardware mechanisms that permit data to be consumed immediately upon its arrival. Such signaling mechanisms would be similar to those used when implementing binding prefetch or out-of-order completion of loads and stores. This section explores the problems involved in closing the window of vulnerability in polled, context-switching processors. For a discussion of signaling methods, see [62].

Systems with multiple outstanding requests may view their requests as *split-phase* transactions, consisting of decoupled request and response phases. The time between request and response may be composed of a number of factors, including communication delay, protocol delay, and queueing delay. Single-threaded processors typically can make little or no forward progress until requested data arrives; they spin while waiting and consume data immediately. In contrast, multithreaded processors, such as Sparcle, may perform other work while waiting for the results of memory requests; as a consequence, data may not be immediately consumed when it returns. In particular, in multithreaded processors that poll for data, memory transactions are *multi-phase*: request, response, and (later) access.

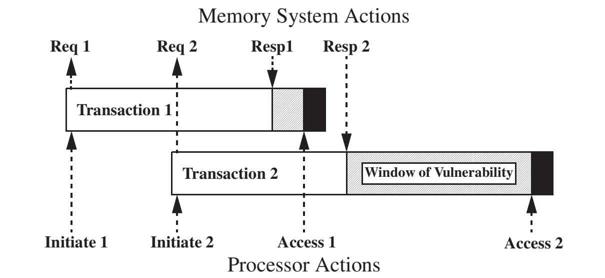


Figure 3-2: Multi-phase memory transactions are subject to a window of vulnerability.

Figure 3-2 shows a time-line of events for two multi-phase memory transactions that originate on a single processing node. Time flows from left to right in the diagram. Events on the lower line are associated with the processor, and events on the upper line are associated with the memory system. In the figure, a processor initiates a memory transaction (Initiate 1), and instead of waiting for a response from the memory system, it continues to perform useful work. During the course of this work, it might initiate yet another memory transaction (Initiate 2). At some later time, the memory system responds to the original request (Response to Request 1). Finally, the processor completes the transaction (Access 1). As mentioned above, a multithreaded processor continues working while it awaits responses from the memory system. Consequently, it might not use returning data immediately. Such is the case in the scenario in Figure 3-2. When the processor receives the response to its second request (Response to Request 2), it is busy with some (possibly unrelated) computation. Eventually, the processor completes the memory transaction (Access 2).

Thus, we can identify three distinct phases of a transaction:

1. Request Phase – The time between the transmission of a request for data and the arrival of this data from memory.
2. Window of Vulnerability – The time between the arrival of data from memory and the initiation of a successful access of this data by the processor.
3. Access Phase – The period during which the processor atomically accesses and commits the data.

The window of vulnerability results from the fact that the processor does not consume data immediately upon its arrival. During this period, the data must be placed somewhere, perhaps in the cache or a temporary buffer. The period between the response and access phases of a transaction is crucial to forward progress. Should the data be invalidated or lost due to cache conflicts during this period, the transaction is terminated before the requesting thread can make forward progress.

As a consequence, the window of vulnerability allows scenarios in which processors repeatedly attempt to initiate transactions only to have them canceled prematurely. In certain pathological cases, individual processors are prevented from making forward progress by cyclic *thrashing* situations. Section 3.2.3 describes four different scenarios in which the processor repeatedly loses data, thus generating livelock. These four scenarios comprise the window of vulnerability livelock.

### 3.2.2 Processor-Side Forward Progress

As shown in Figure 3-3, we will consider the memory system, complete with interconnection network, to be a black-box that satisfies memory requests. While this abstracts away the details of the memory-side of the cache-coherence protocol and ignores the fact that memory is physically distributed with the processors, it permits us to focus on the processor-side of the system, where the window of vulnerability arises. For the remainder of this discussion let us assume that *all requests which are made to the memory-system are eventually satisfied;* we will reconsider this assumption in Section 3.3 when we discuss the *server-interlock problem* and its concomitant problem, the *multiple-writer livelock*. Consequently, in the following few sections,

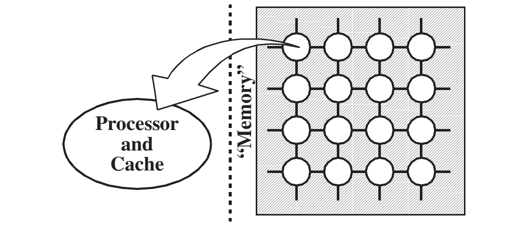



Figure 3-3: Treat the network, coherence protocol, and other nodes as a "black box".

when we say that a processor (or hardware thread) does or does not make forward progress, we are referring to properties of its local hardware and software, assuming that the remote memory system always satisfies requests.

---

Thus, we say that a thread makes forward progress whenever it commits an instruction. Given a processor with precise interrupts, we can think of this as advancing the instruction pointer. A load or store instruction makes forward progress if the instruction pointer is advanced beyond it.

---

**Primary and Secondary Transactions:** Given this definition of forward progress, we can identify two distinct classes of transactions, *primary* and *secondary*. Primary transactions are those which must complete before some hardware thread in the system can make forward progress. Secondary transactions, on the other hand, are not essential to the forward progress of any thread in the system.

The categories of primary and secondary transactions distinguish between binding memory operations (normal loads and stores) and non-binding memory operations (prefetches). Non-binding prefetches are hints to the memory-system; they specify data items which *may* be needed in the future. As hints, they can be ignored without affecting the correctness of the computation.

Thus, when handling a prefetch, the memory system may initiate a secondary transaction. Should this transaction be aborted prematurely, it will not affect the forward progress of the processor. However, if the processor later attempts to access prefetched data via a binding load or store, one of two things can happen:

1. The original transaction has been aborted. In this case the memory system will initiate a new, primary transaction. This is as if the prefetch never occurred.
2. The secondary transaction is still in progress. Since the forward progress of the processor now depends on the successful completion of the transaction, it is effectively "upgraded" to primary status.

This primary-secondary distinction will appear in later discussion.

**Forward Progress and the Window of Vulnerability:** Memory models differ in the degree to which they require primary transactions to complete before the associated loads or stores commit. Sequentially consistent machines, for instance, require write transactions (associated with store instructions) to advance beyond the request phase before their associated threads make forward progress. Weakly-ordered machines, on the other hand, permit store instructions to commit *before* the end of the request phase. In a sense, the cache system promises to ensure that store accesses complete. Therefore, for weakly-ordered machines, *write transactions have no window of vulnerability*. In contrast, most memory models require a read transaction to receive a response from memory before committing the associated load instruction.

As an example, the Alewife multiprocessor uses memory exception traps to cause context switches. Consequently, data instructions are restarted by "returning from trap," or refetching the faulted instruction. If this instruction has been lost due to cache conflicts, then the context may need to fetch it again before making forward progress. Thus, each context can have *both* a primary instruction transaction and a primary data transaction[2]. In contrast, a processor that saved its pipeline state (and faulting instruction) when context-switching would retry only the faulted data access; hence, each context would have at most one primary transaction.

Unless otherwise noted, this chapter will assume that a hardware context can have no more than one primary data transaction. This assumption has two implications. First, any weakly ordered writes that have not yet been seen by the memory system are committed from the standpoint of the processor. Second, a single context cannot have multiple uncommitted load instructions (as in a processor with register reservation bits). Similarly, we allow no more than one primary instruction transaction at a time. In actuality, these restrictions are not necessary for one of our more important results, the *thrashwait algorithm*, but they are required for the *thrashlock mechanism*.

### 3.2.3 Four Window of Vulnerability Livelock Scenarios

This section introduces the four distinct types of livelock or data thrashing which can occur in the processor's cache system. Before describing these livelocks, consider Figure 3-4. This figure introduces the context-timeline diagrams that will be used in this section by illustrating a *successful* multiphase transaction. Across the bottom of this figure is a timeline that indicates which of four contexts (labeled "A" through "D") are currently in control of the processor.

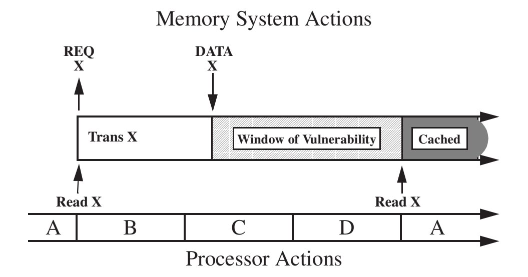



Figure 3-4: Successful multithreading.

At the beginning, context A is in control. When it attempts to read data block X, it discovers that it does not reside on the current node. As a result, the memory system initiates the request phase of a new transaction (called "X") by sending a read request to the current location of the data. Simultaneously, context A relinquishes control of the processor to context B, which begins execution. Later, data for block X returns from the memory system and begins the window of vulnerability phase of transaction X. Finally, context A regains control of the processor (after the succession B $\Rightarrow$ C $\Rightarrow$ D $\Rightarrow$ A) and terminates the window of vulnerability after successfully reading the data. Afterward, block X may reside in a local cache until replaced or invalidated. The fact that context A returns to retry the original read of block X indicates that this is a polling type of multithreading.

We will now proceed to describe the four instances of the window of vulnerability livelock. One of these, *invalidation* thrashing, arises from protocol invalidation for highly contended memory lines. The remaining three result from replacement in a direct-mapped cache. In *intercontext* thrashing, different contexts on the same processor can invalidate each other's data. *high-availability interrupt* thrashing occurs when interrupt handlers replace a context's data in the cache. The last, *instruction-data thrashing*, appears for processors that context-switch by polling and which must refetch load or store instructions before checking for the arrival of data. Section 3.2.5 will discuss methods of eliminating these livelock situations.

[2]Note that factoring instructions into this situation also has some interesting pipeline consequences which will be examined in Chapter 5.

**Invalidation Thrashing:** Figure 3-5 illustrates an interaction between the window of vulnerability and cache coherence that leads to livelock. This figure shows the time-history of two different nodes, interacting through the network. Note 1 is attempting to read memory block X, while node 2 is attempting to write the same block[3]. In this scenario, context A of node 1 requests a read copy of block X (RREQ X), then relinquishes the processor to context B. Later, the data returns (RDATA X), but context A is not in control of the processor. Before context A returns for the data, context C of node 2 makes a write request (WREQ X) for block X, then relinquishes the processor to context D. This write request invokes the coherence protocol, invalidating the block in node 1's cache (INV X ⇒ACK X). When context A of node 1 returns to look for its data, it discovers the data missing, and requests the data again. This time (and for subsequent times), the read request (RREQ X) invokes the coherence protocol and causes an invalidation of the block in node 2's cache (INV X ⇒ACK X). Thus, the process shown in Figure 3-5 is self sustaining: node 1 prevents node 2 from making forward progress and node 2 prevents node 1 from making forward progress. Our experience (both with simulation and real hardware) indicates that this type of thrashing is infrequent, but occurs at some point during the execution of most programs.

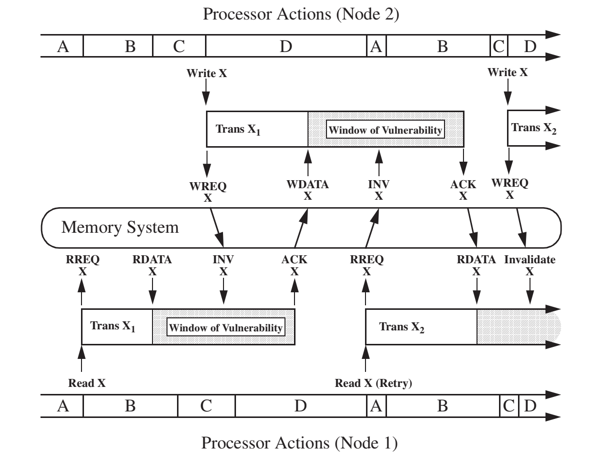



Figure 3-5: Invalidation thrashing between context A on Node 1 and context C on node 2.

**Replacement Thrashing:** Due to the limited set-associativity of a processor's cache, different contexts on the same processor can interfere with each other. Figure 3-6 illustrates this scenario. Contexts A and C try to (respectively) access blocks X and Y which are congruent in the cache: First, block X is requested by context A; later, block Y is requested by context C. Block X arrives first. Since blocks X and Y are congruent with respect to the cache mapping, when the data for block Y arrives, it replaces the data for block X in the cache: in Figure 3-6, the arrival of data block Y terminates transaction X during its window of vulnerability. Hence, context A must retry the read request when it regains control. Similarly, when data for block X returns, it terminates transaction Y by knocking its data out of the cache. Each context prevents the other from making forward progress by replacing cached data during the window of vulnerability. As a consequence of this *replacement* or *intercontext* thrashing, the processor can livelock itself.

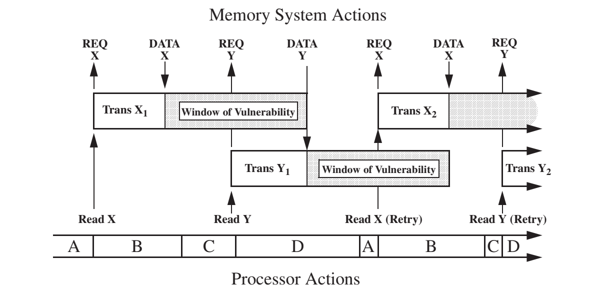



Figure 3-6: Replacement thrashing between contexts A and B on the same node.

[3]Such a situation is not at all contrived, since block X may contain a spin lock that node 1 is attempting to acquire and node 2 is attempting to release.

**High-Availability Interrupt Thrashing:** In Section 3.1, we introduced the use of high-availability interrupts to avoid the refused-service deadlock. In that section we concluded that high-availability interrupts are necessary in machines such as Alewife with integrated shared-memory and message-passing communication. Unfortunately, by their very nature, high-availability interrupts may interrupt a processor that is spinning in wait for the completion of a shared-memory request. This opens the window of vulnerability since the corresponding shared-memory data may be lost or invalidated during the execution of the interrupt handler.

For instance, Figure 3-7 demonstrates how high-availability interrupts can cause a special case of replacement thrashing. The figure shows user code attempting to access memory block X and interrupt code accessing block Y, which maps to the same cache line as X. During a normal memory access, the user code would spin-wait until it received the data associated with block X. However in this pathological scenario, the user code is interrupted by a high-availability interrupt and forced to execute an interrupt handler. While the processor is handling the interrupt, data block X arrives, but is subsequently replaced when an instruction in the handler references block Y. This scenario repeats itself multiple times. With sufficient randomness in the system, context A might *eventually* succeed in completing its access without an interrupt interfering, but this is by no means guaranteed. In particular, this thrashing scenario will never resolve itself if the interrupt handler must execute as a condition to handling the access for block X.

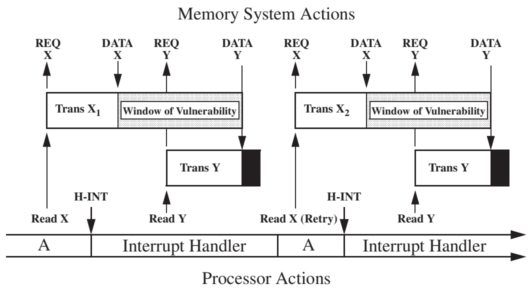



Figure 3-7: High-availability Interrupt Thrashing: Context A is thrashing with an interrupt handler.

Although Figure 3-7 presents an interrupt analog to replacement thrashing, high-availability interrupts may also exacerbate problems with invalidation thrashing. In fact, as we will see in Section 3.2.5, the existence of high-availability interrupts complicates the removal of window of vulnerability livelocks since it permits the loss of data *even when context-switching is disabled.*

**Instruction-Data Thrashing:** As discussed in Section 5.1.1, processors that use polled multithreading such as Sparcle may need to refetch instructions whenever the processor returns to a context to continue execution. Figure 3-8 shows how this type of multithreading is vulnerable to a cache conflict between a load instruction and its data. In this scenario, context A must execute a load instruction to make forward progress, but both the load instruction and its data are congruent in the cache. First, context A initiates a transaction to fetch this instruction (Read I), after which it relinquishes the processor. Later, context A returns and successfully retrieves its instruction, ending the window of vulnerability for the transaction; this instruction will be placed in the local cache for a

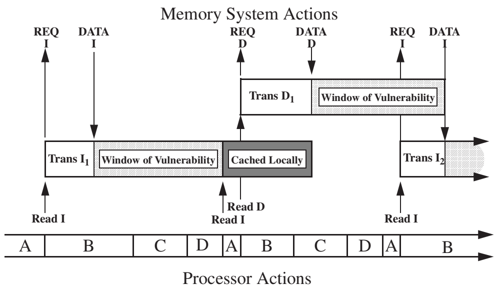



Figure 3-8: Instruction/Data Thrashing: interference between instructions and data in context A.

short period of time afterwards. Having fetched the load instruction, context A proceeds to execute this load, thereby initiating a new transaction for the data (Read D) and once again relinquishing the processor. Unfortunately, when the data returns, it displaces the instruction from cache. Consequently, when context A regains control of the processor and attempts to fetch the instruction, it finds the instruction missing and must request it again. The re-requested instruction, in turn, displaces the data, and a thrashing cycle commences.

Instruction-data thrashing is independent of timing, since a single context is competing with itself. Consequently, if an instruction and its data are congruent in the cache and context-switching is permitted on all cache-misses, then instruction-data thrashing *will* cause livelock. This is in marked contrast to uniprocessor instruction-data thrashing (which merely degrades performance).

**Completeness:** The four types of thrashing presented above represent interference to the forward progress of a given context from four different sources:

- A remote processor (invalidation thrashing)
- Another context (intercontext thrashing)
- Interrupt-code (high-availability interrupt thrashing)
- Itself (instruction-data thrashing).

The later three represent all possible instruction-stream related interference on a context-switching processor. Assuming that invalidation is the only vehicle for external interference, our four types of thrashing represent a complete set. Should we discover ways of limiting each of these types of thrashing, then we will be able to guarantee that each processor context is able to make forward progress (assuming that all available cycles are not consumed by interrupt code).

### 3.2.4 Severity of the Window of Vulnerability

Before embarking on a program of eliminating the various window of vulnerability livelocks, we would like to ask the question: how serious is this problem? To ask this question, the Alewife research group constructed a cycle-by-cycle simulation of an Alewife node (Sparcle, CMMU, and network). This simulation environment, called ASIM[4], permits parallel programs that are written in C or LISP to be compiled, linked, and executed on a virtual Alewife machine. A copious set of statistics-gathering facilities permit post-mortem analysis of the behavior of the program and machine.

One of the statistics that are of particular interest in this section is the size of the window of vulnerability. For each access, ASIM computes the time between the instant that a data block becomes valid in a cache due to a response from memory to the first subsequent access to the cached data. The simulator measures this period of time only for the fraction of memory accesses that generate network traffic and are thus susceptible to the window of vulnerability. Figure 3-9 shows typical measurements of the window of vulnerability. The graph is a histogram of window

[4]For Alewife SIMulator. This simulator has now be supplanted by a simulator which is more faithful to implementation details. ASIM remains a good research tool, however.

of vulnerability sizes, with the size on the horizontal axis and the number of occurrences on the vertical axis. The graph was produced by a simulation of a 64 processor machine (with 4 contexts per processor) running 1,415,308 cycles of a numerical integration program.

For the most part, memory accesses are delayed for only a short period of time between cache fill and cache access: 90% of memory accesses that generate network traffic have windows that are less than 65 cycles long. However, a small number of accesses encounter pathologically long windows of vulnerability. To make the interesting features of the graph visible, it was necessary to plot the data on a logarithmic scale and to eliminate events having a frequency of less than 30 occurrences. Due to a few extremely long context run-lengths, the tail of this particular graph actually runs out to 543,219 cycles! The high standard deviation provides another measure of the importance of the graph's tail. Note that these results reflect a non-preemptive scheduling of contexts, *i.e.* contexts are allowed to continue executing as long as they do not encounter remote accesses.

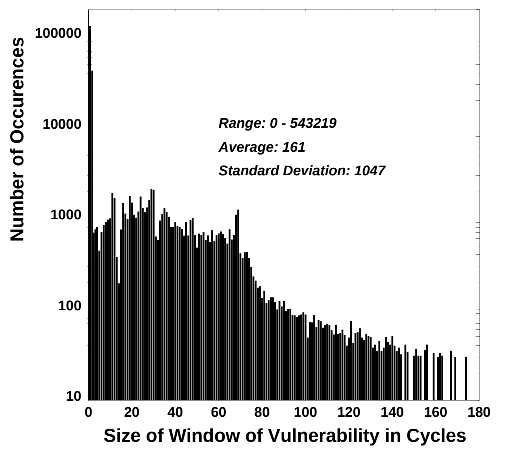



Figure 3-9: Window of vulnerability: 64 processors, 4 contexts.

The sharp spike at zero cycles illustrates the role of context switching and high-availability interrupts in causing the window of vulnerability. The spike is caused by critical sections of the task scheduler that disable context switching. When context switching is disabled, a processor will spin-wait for memory accesses to complete, rather than attempting to tolerate the access latency by doing other work. In this case, the processor accesses the cache on the same cycle that the data becomes available. Such an event corresponds to a zero-size window of vulnerability. The window becomes a problem only when context switching is enabled or when high-availability interrupts interfere with memory accesses.

The window of vulnerability histogram in Figure 3-9 is qualitatively similar to other measurements made for a variety of programs and architectural parameters. The time between cache fill and cache access is usually short, but a small fraction of memory transactions always suffer from long windows of vulnerability. In general, both the average window size and the standard deviation increase with the number of contexts per processor. The window size and standard deviation also grow when the context switch time is increased. We have observed that high-availability interrupts cause the same type of behavior although their effects are not quite as dramatic as the effect of multiple contexts.

For the purposes this section, it does not matter whether the window of vulnerability is large or small, common or uncommon. Even if a window of vulnerability is only tens or hundreds of cycles long, it introduces the possibility of livelock that can prevent an application from making forward progress. The architectural framework described in the next section is necessary merely because the window *exists*. Fortunately, as we shall see, the cost of removing the window of vulnerability livelock is not prohibitive and provides a number of secondary benefits as well (such as providing a framework in which to make the cache-coherence protocol insensitive to network ordering.

### 3.2.5 Closing the Window: Preliminaries

In the following sections, we will examine three different solutions to the window of vulnerability livelock. Our ultimate goal is to discover a framework that eliminates livelock problems associated with the window of vulnerability for systems with multiple outstanding requests and high-availability interrupts. In order to do this, the system must track pending memory transactions in such a way that it can dynamically detect and eliminate pathological thrashing behavior. The complete framework consists of three major components: a small, associative set of *transaction buffers* that keep track of outstanding memory requests, an algorithm called *thrashwait* that detects and eliminates livelock scenarios that are caused by the window of vulnerability, and a buffer locking scheme that prevents livelock in the presence of high-availability interrupts.

Before discussing this framework, however, we would like to make explicit the set of features that we are assuming are part of our shared-memory implementation. By making these features explicit, we can present a range of solutions to the window of vulnerability livelock that address different combinations of these features. These three features are:

1. Coherent caches to automatically replicate data close to where it is needed, and a mechanism to allow multiple outstanding requests to memory.
2. The ability to disable context switching for atomic access to data structures.
3. High-availability interrupts for response to high-priority asynchronous events such as message arrival.

Not all architects will be interested in this full set of mechanisms (for instance, high-availability interrupts would be unnecessary in a shared-memory system that does not integrate message passing and that does not use software exception handlers for partial implementation of the coherence protocol). For this reason, the following three sections describe different subsets of the framework and the mechanisms that each subset will support. These self-contained solutions are called *associative locking* (Section 3.2.6), *thrashwait* (Section 3.2.7), and *associative thrashlock* (Section 3.2.8). Each is appropriate for a different combination of the above shared-memory mechanisms. As summarized in Table 3-1, a system with coherent caches and multiple outstanding requests (*Multi*) is assumed in all cases. To this is added either the ability to disable context switching (*Disable*), the presence of high-availability interrupts (*HAI*), or a combination of both. A *Yes* in Table 3-1 indicates that a given solution is appropriate for the specified combination of mechanisms. During the exposition, two partial solutions are also discussed, namely *locking* and *associative thrashwait*.

| | Multi | Multi + Disable | Multi + HAI | Multi + HAI + Disable |
|---|---|---|---|---|
| Assoc Locking | *Yes* | *No* | *Yes* | *No* |
| Thrashwait | *Yes* | *Yes* | *No* | *No* |
| Assoc Thrashlock | *Yes* | *Yes* | *Yes* | *Yes* |

Table 3-1: Window of Vulnerability closure techniques. *Multi* represents coherent caches and multiple requests. *Disable* represents disabling of context switching. *HAI* represents high-availability interrupts.

*Locking* involves freezing external protocol actions during the window of vulnerability by deferring invalidations. *Thrashwait* is a heuristic that dynamically detects thrashing situations and

| Technique | Prevents Invalidation Thrashing | Prevents Intercontext Thrashing | Prevents HAI Thrashing | Prevents Inst-Data Thrashing | Deadlock Free Context Switch Disable | Free From Cache line Starvation |
|---|---|---|---|---|---|---|
| Locking | *Yes* | *Yes* | *Yes* | *Deadlock* | *No* | *No* |
| Assoc Locking | *Yes* | *Yes* | *Yes* | *Yes* | *No* | *Yes* |
| Thrashwait | *No* | *Yes* | *No* | *Yes* | *Yes* | *Yes* |
| Assoc TW | *No* | *Yes* | *Yes* | *Yes* | *Yes* | *Yes* |
| Assoc Thrashlock | *Yes* | *Yes* | *Yes* | *Yes* | *Yes* | *Yes* |

Table 3-2: Properties of window of vulnerability closure techniques with respect to the complete set of features.

selectively disables context-switching in order to prevent livelock. *Associativity* can be added to each of these techniques by supplementing the cache with an associative buffer for transactions. This yields associative locking and associative thrashwait. Table 3-2 summarizes the deficiencies of each of these mechanisms with respect to supporting the complete set of mechanisms. Associative thrashlock is a hybrid technique that is discussed in Section 3.2.8 and is the only technique that closes the window of vulnerability with the full set of shared-memory mechanisms.

### 3.2.6 The Associative Locking Solution

Our first approach to closing the window of vulnerability involves the *locking* of transactions. A locked transaction is one that ignores (or defers) external events that might otherwise terminate it prematurely. This removes cycles of request and invalidation that characterize the livelocks of Section 3.2.3. Unfortunately, as we shall see, locking must be approached with care since it can transform the window of vulnerability livelocks into *deadlocks*.

As a first attempt at locking, we will assume that returning data (responses) are placed and locked in the cache; this is what we call locking without associativity — *i.e.* the basic associativity of the cache is not supplemented. After we discuss how this removes livelocks, we explore the ways in which it can cause deadlock. Since basic locking causes deadlocks, we do not consider it a complete solution. We then supplement the associativity of the cache with an extra set of buffers for memory transactions. We call this *associative locking*. It produces a viable solution for a system with context switching and high-availability interrupts, although it is unable to properly deal with the disabling of context-switching for atomicity.

**Locking State Bits:** The basic locking solution requires two special state bits in addition to the normal state bits associated with each line in the cache: a *lock* bit and a *deferred invalidation* bit. The *lock* bit signals that a cache line is locked and cannot be replaced. This bit is set when requested data first returns from memory, and cleared when the requesting context returns and successfully accesses data. During the period in which the bit is set (which is the window of vulnerability for the corresponding transaction), the cache line may not be reused for other transactions. This eliminates thrashing because transactions are never terminated before they have completed successfully. In some sense, the role of the lock bit is to eliminate thrashing by *ignoring* attempts to terminate transactions prematurely (or alternatively by preventing conflicting transactions from starting); any operation thus ignored will be later retried, since we are assuming polled multithreading.

The lock bit is sufficient to eliminate replacement, high-availability interrupt, and instruction-data thrashing. We have to be careful in preventing invalidation thrashing, however, because cache-coherence invalidations cannot be simply ignored — they must be deferred. Thus, the second locking bit, the *deferred invalidate* bit, is used to remember that an invalidation arrived for the cache-line in question while it was locked. Such deferred invalidations are later performed (and acknowledged) after the transaction has terminated successfully and the lock bit has been cleared.

**The Transaction-In-Progress State:** One of the consequences of locking cache lines during a transaction's window of vulnerability is that we must also restrict transactions during their request phase. Since each cache line can store only one outstanding request at a time, multiple requests could force the memory system to discard one locked line for another, defeating the purpose of locking. Thus, we supplement the state of a cache line with a *transaction-in-progress* state to prevent multiple outstanding requests to this line. The transaction-in-progress state restricts creation of new transactions, but does not affect data currently in the cache in order to minimize the interference of memory transactions in the cache. Note that the two bits mentioned previously (*lock* and *deferred invalidation*) have only three unique states: unlocked, locked, and locked with deferred invalidation. Hence, the forth state may be used to represent transactions in progress.

**Premature Lock Release:** As described, the above scheme does not quite eliminate all intercontext thrashing, because one context can unlock or touch the data requested by another context. This is called *premature lock release*. Locking can, however, be supplemented with additional bits of state to keep track of which context holds a given lock; then, only the locking context is permitted to free this lock. This can get quite expensive with the simple locking scheme, because these bit must be included in tags-file. However, when introduce associative locking (and move the locus of locking away from the cache), this additional state is far less prohibitive.

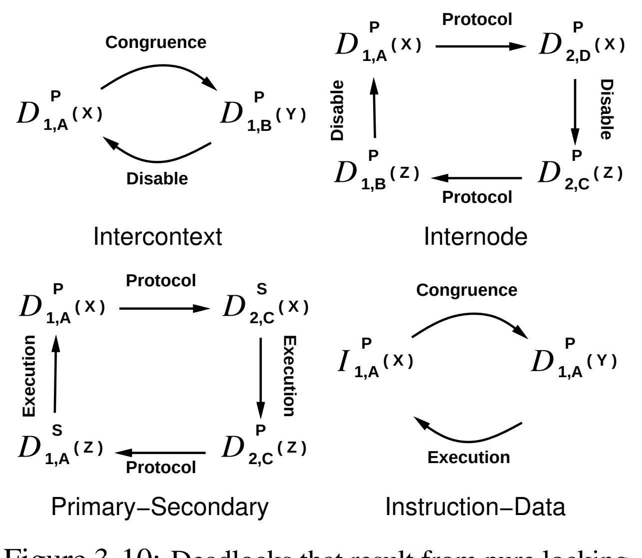



Figure 3-10: Deadlocks that result from pure locking. Addresses $X \equiv Y$ in cache; $X \neq Z$.

**Deadlock Problems:** Unfortunately, the basic locking mechanism can lead to four distinct types of deadlock, illustrated in Figure 3-10. This figure contains four different *waits-for graphs* [11], which represent dependencies between transactions. In these graphs, the large italic letters represent transactions: "*D*" for data transactions and "*I*" for instruction transactions. The superscripts – either "P" or "S" – represent primary or secondary transactions, respectively. The subscripts form a pair consisting of processor number (as an arabic number) and context number (as a letter). The address is given in parentheses; in these examples, $X$ and $Y$ are congruent in the cache ($X \equiv Y$), while $X$ and $Z$ are not equal ($X \neq Z$).

The labeled arcs represent dependencies; a transaction at the tail of an arc cannot complete before the transaction at the head has completed (in other words, the tail transaction *waits-for* the head transaction). Labels indicate the sources of dependencies: A *congruence* arc arises from finite associativity in the cache; the transaction at its head is locked, preventing the transaction at its tail from being initiated. An *execution* arc arises from execution order. *Disable* arcs arise from disabling context-switching; the transactions at their heads belong to active contexts with context-switching disabled; the tails are from other contexts. Finally, a *protocol* arc results from the coherence protocol; the transaction at its head is locked, deferring invalidations, while the transaction at its tail awaits acknowledgment of the invalidation. An example of such a dependence is a locked write transaction at the head of the arc with a read transaction at the tail. Since completion of the write transaction could result in modification of the data, the read transaction cannot proceed until the write has finished. These arcs represent three classes of dependencies: those that prevent launching of transactions (*congruence*), those that prevent completion of a transaction's request phase (*protocol*), and those that prevent final completion (*execution* and *disable*).

Now we describe these deadlocks in more detail. Note that larger cycles can be constructed by combining the basic deadlocks.

- **intercontext:** The context that has entered a critical section (and disabled context-switching) may need to use a cache line that is locked by another context.
- **internode:** This deadlock occurs between two nodes with context-switching disabled. Here, context A on processor 1 is spinning while waiting for variable X, which is locked in context D on processor 2. Context C on processor 2 is also spinning, waiting for variable Z, which is locked by context B on processor 1.
- **primary-secondary:** This is a variant of the internode deadlock problem that arises if secondary transactions (software prefetches) can be locked. Data blocks from secondary transactions are accessed after those from primary ones.
- **instruction-data:** Thrashing between a remote instruction and its data yields a deadlock in the presence of locks. This occurs after a load or store instruction has been successfully fetched for the first time. Then, a request is sent for the data, causing a context-switch. When the data block finally returns, it replaces the instruction and becomes locked. However, the data will not be accessed until after the processor refetches the instruction; however, the instruction cannot be refetched because the requisite cache-line is locked.

Primary-secondary deadlock is easily removed by recognizing that secondary transactions are merely hints; locking them is not necessary to ensure forward progress. Unfortunately, the remaining deadlocks have no obvious solution. Due to these deadlock problems, pure locking cannot be used to close the window of vulnerability.

**Associative Locking:** A variant of the locking scheme that does not restrict the use of the cache or launching of congruent transactions is *locking with associativity*. This scheme supplements the cache with a fully associative set of *transaction buffers*. Each of these buffers contains an address, state bits, and space for a memory line's data. Locking is performed in the transaction buffer, rather than the cache. As discussed above, invalidations to locked buffers are deferred until the data word is accessed. Buffer allocation can be as simple as reserving a fixed set of buffers for each context. More general schemes might keep track of the context that owns each buffer to prevent premature lock release). The use of a transaction buffer architecture has been presented in several milieux, such as lockup-free caching [59], *victim caching* [53], and the *remote-access cache* of the DASH multiprocessor [69].

The need for an associative match on the address stems from several factors. First, protocol traffic is tagged by address rather than by context number. While requests and responses could be tagged with a context identifier inexpensively, tagging invalidations would increase the cost of the directory used to guarantee cache coherence. Second, associativity removes the intercontext and instruction-data deadlocks of Figure 3-10, because it eliminates all of the *congruence* arcs of Figure 3-10.

Third, the associative match permits consolidation of requests from different contexts to the same memory-line; before launching a new request, the cache first checks for outstanding transactions *from any context* to the desired memory line. Should a match be detected, generation of a new request is suppressed.

Finally, the associative matching mechanism can permit contexts to access buffers that are locked by other contexts. Such accesses would have to be performed directly to and from the buffers in question, since placing them into the cache would effectively unlock them. This optimization is useful in a machine with medium-grained threads, since different threads often execute similar code and access the same synchronization variables.

The augmentation of basic locking with associativity appears to be close to a solution for the window of vulnerability. All four thrashing scenarios of Section 3.2.3 are eliminated. Further, the cache is not bogged down by persistent holes. Access to the cache is unrestricted for both user and system code. However, this approach still suffers from internode deadlock when context-switching is disabled. Consequently, as shown in Table 3-1, associative locking is sufficient for systems which do not permit context-switching to be disabled.

### 3.2.7 The Thrashwait Solution

Locking transactions prevents livelock by making data invulnerable during a transaction's window of vulnerability. To attack the window from another angle, we note that the window is eliminated when the processor is spinning while waiting for data: when the data word arrives, it can be consumed immediately. This observation does not seem to be useful in a machine with context-switching processors, since it requires spinning rather than switching. However, if the processors could context-switch "most of the time," spinning only to prevent thrashing, the system could guarantee forward progress. We call this strategy *thrashwait* (as opposed to touchwait). The trick in implementing thrashwait lies in dynamically detecting thrashing situations. The thrashwait detection algorithm is based on an assumption that the frequency of thrashing is low. Thus, the recovery from a thrashing scenario need not be extremely efficient.

For the purpose of describing the thrashwait scheme, assume that the system has some method for consolidating transactions from different contexts. To implement this feature, either each cache line or the transaction buffers needs a transaction-in-progress state. If the transaction-in-progress state is in the cache, as in the pure locking scheme, the system allows only one outstanding transaction per cache line.

Consider, for simplicity, a processor with a maximum of one outstanding primary transaction per context; multiple primary transactions will be addressed in the next section. Each context requires a bit of state called a *tried-once bit*. The memory system sets the bit when the context initiates *primary* transactions and clears the bit when the context completes a global load or store. Note that *global* accesses, which involve shared locations and the cache-coherence protocol, are distinguished here from *local* accesses which are unshared and do not involve the network or the protocol. In addition, there is a single *thrash-wait bit* which is used to retain the fact that thrashing has been detected on the current access. The algorithm can be described in pseudo-code as shown in Figure 3-11[5]: This function is executed by the cache controller each cycle. The return codes (**READY**, **SWITCH**, and **WAIT**) refer to a successful cache hit, a context-switch request, and a pipeline freeze respectively. **RREQ** is a read request and **WREQ** is a write request.

```
DO_GLOBAL_PROCESSOR_REQUEST(Address, Context)
 1: if (data is ready for Address) then
        // Cache HIT!
 2:     clear tried_once[Context]
 3:     clear thrash_wait
 4:     return READY
 5: elsif (Transaction-in-progress[Address]) then
        // Still waiting for in-progress transaction
 6:     if (thrash_wait or switching disabled) then
 7:         return WAIT
 8:     else return SWITCH
 9: elsif (tried_once[Context]) then
        // Detected thrashing!
10:     send RREQ or WREQ
11:     set thrash_wait
12:     return WAIT
13: else // Normal cache miss
14:     send RREQ or WREQ
15:     set tried_once[Context]
16:     if (switching disabled) then
17:         return WAIT
18:     else return SWITCH
```

Figure 3-11: The Thrashwait Algorithm

The key to the detecting of thrashing is in line 9. This says that the memory system detects a thrashing situation when:

1. The context requests a global load or store that misses in the cache.
2. There is no associated transaction-in-progress state.
3. The context's tried-once bit is set.

The fact that the tried-once bit is set indicates that this context has recently launched a primary transaction but has not successfully completed a global load or store in the interim. Thus, the context has *not* made forward progress.

In particular, the current load or store request must be the same one that launched the original transaction. The fact that transaction-in-progress is clear indicates that the transaction had completed its request phase (data was returned). Consequently, the fact that the access missed in the cache means that a data block has been lost. Once thrashing has been detected, the thrashwait algorithm requests the data for a second time and disables context-switching, causing the processor to wait for the data to arrive.

[5] Adapted from Chaiken [25]. The pseudo-code notation is borrowed from [30].

**Multiple Primary Transactions:** Systems requiring two primary transactions can be accommodated by providing two tried-once bits per context, one for instructions and the other for data. Only a single thrash-wait bit is required. To see why a single tried-once bit is not sufficient, consider an instruction-data thrashing situation. Assuming that a processor has successfully fetched the load or store instruction, it proceeds to send a request for the data, sets the tried-once bit, and switches contexts. When the data block finally arrives, it displaces the instruction; consequently, when the context returns to retry the instruction, it concludes that it is thrashing *on the instruction fetch*. Context-switching will be disabled until the instruction returns, at which point the tried-once bit is cleared. Thus, the algorithm fails to detect thrashing on the data line.

As shown in Figure 3-12, the presence of two separate tried-once bits per context (Inst_TO and Data_TO) solves this problem. This figure shows cache requests from context zero (0) during the fetching and execution of a load instruction which is subject to instruction-data thrashing. Note that this ignores pipeline reordering, which will be considered in Chapter 5. The instruction and data accesses are handled independently, according to the above algorithm. In fact, this two-bit solution can be generalized to a system with an arbitrary number of primary transactions. The only requirement for multiple transactions is that each primary transaction must have a unique tried-once bit that can be associated with it each time the context returns to begin reexecution. (This can become somewhat complex in the face of deep pipelining or multiple-issue architectures.)

**Elimination of Thrashing:** The thrashwait algorithm identifies primary transactions that are likely to be terminated prematurely; that is, before the requesting thread makes forward progress. Assuming that there are no high-availability interrupts, thrashwait removes livelock by breaking the thrashing cycle. Thrashwait permits each primary transaction to be aborted only once before it disables the context-switching mechanism and closes the window of vulnerability.

In a system with multiple primary transactions, livelock removal occurs because primary transactions are ordered by the processor pipeline. A context begins execution by requesting data from the cache system in a deterministic order. Consequently, under worst-case conditions – when all transactions are thrashing, the processor will work its way through the implicit order, invoking thrashwait on each primary transaction in turn. Although a context-switch may flush its pipeline state, the tried-once bits remain, forcing a pipeline freeze (rather than a switch) when thrashing occurs.

An example of this would be seen in Figure 3-12 by replacing the first two **READY** responses (both on instructions) into **WAITs** by causing the instruction data to be lost do to conflict with another context. In this absolute worst-case scenario, the instruction would be requested four times and the data would be requested twice; the context would make forward progress, however.

**Freedom From Deadlock:** In this section, we prove that the thrashwait algorithm does not suffer from any of the deadlocks illustrated in Figure 3-10. We assume (for now) that a processor launches only one primary transaction at a time. Multiple primary transactions, which must complete to make forward progress, are allowed; multiple simultaneous transactions, which are caused by a system that presents several addresses to the memory system at once, are not allowed. At the end of the proof, we discuss a modification to the thrashwait algorithm that is necessary for handling multiple functional units and address buses.

| Context | Processor Request | Cache Response | Cache Actions |
|---|---|---|---|
| 0 | Fetch Load Inst(A) | **SWITCH** | set Inst_TO[0], Send Request (**RREQ**[A]). |
| Other | ⋮ | | |
| ⋮ | | ⋮ | Cache[A] ← Instruction |
| Other | ⋮ | | |
| 0 | Fetch Load Inst[A] | **READY** | clear Inst_TO[0] |
| 0 | Read Data (B ≡ A) | **SWITCH** | set Data_TO[0], Send Request (**RREQ**[A]) |
| Other | ⋮ | | |
| ⋮ | | ⋮ | Cache[B] ← Read Data (Displace Instruction) |
| Other | ⋮ | | |
| 0 | Fetch Load Inst(A) | **SWITCH** | set Inst_TO[0], Send Request (**RREQ**[A]) |
| Other | ⋮ | | |
| ⋮ | | ⋮ | Cache[B] ← Instruction (Displace Data) |
| Other | ⋮ | | |
| 0 | Fetch Load Inst(A) | **READY** | clear Inst_TO[0] |
| 0 | Read Data (B) | **WAIT**† | Send Request (**RREQ**[A]) |
| 0 | ⋮ | **WAIT** | |
| 0 | Read Data (B) | **WAIT** | Cache[B] ← Read Data (Displace Instruction) |
| 0 | Read Data (B) | **READY** | clear Data_TO[0] |
| 0 | ⋮ | ⋮ | |

Figure 3-12: Elimination of instruction-data thrashing through Thrashwait. At the point marked with (†), Thrashwait is invoked since Data_TO[0] is set.

The proof of the deadlock-free property proceeds by contradiction. We assume that the thrashwait algorithm can result in a deadlock. Such a deadlock must be caused by a cycle of primary transactions, linked by the dependencies defined in Section 3.2.6: *disable*, *execution*, *congruence,* and *protocol* arcs. Since the memory transactions involved in the deadlock loop are frozen, it is correct to view the state of transactions simultaneously, even if they reside on different processors. By examining the types of arcs and the associated transactions, we show that such a cycle cannot exist, thereby contradicting the assumption that thrashwait can result in a deadlock.

Disable and execution arcs cannot participate in a deadlock cycle because these dependencies occur only in systems that use a locking scheme. Since thrashwait avoids locking, it immediately eliminates two forms of dependency arcs. This is the key property that gives thrashwait its deadlock-free property. To complete the proof, we only need to show that congruence and protocol arcs cannot couple to form a deadlock.

A deadlock cycle consisting of congruence and protocol arcs can take only one of three possible forms: a loop consisting only of congruence arcs, a loop consisting of both congruence arcs and protocol arcs, or a loop consisting of only protocol arcs. The next three paragraphs show that none of these types of loops are possible. Congruence and protocol arcs cannot be linked together, due to *type conflicts* between the head and tail of congruence and protocol arcs.

First, we show that cycles consisting only of congruence arcs cannot occur. Recall that a congruence arc arises when an existing transaction blocks the initiation of a new transaction due to limited cache associativity. A congruence arc requires an existing transaction at its head and a new transaction at its tail. It is therefore impossible for the tail of a congruence arc (a new transaction) to also be the head of a different congruence arc (an existing transaction). Thus, it is impossible to have a loop consisting only of congruence arcs, because the types of a congruence arc's head and tail do not match.

Second, a cycle consisting only of protocol arcs cannot exist. By definition, the head of a protocol arc is a transaction in its window of vulnerability, which is locked so that invalidations are deferred. The tail of a protocol arc is a transaction in its request phase, waiting for the invalidation to complete. Since a transaction in its request phase cannot be at the head of a protocol arc, protocol arcs cannot be linked together, thereby preventing a loop of protocol arcs.

Finally, the tail of a congruence arc cannot be linked to the head of a protocol arc due to another type conflict: the tail of a congruence arc must be a new transaction, while the head of a protocol arc is an existing transaction in its window of vulnerability. Thus, deadlock loops cannot be constructed from combinations of protocol and congruence loops. The fact that congruence arcs and protocol arcs cannot combine to produce a loop contradicts the assumption that thrashwait can result in a deadlock, completing the proof.

The above proof of the deadlock-free property allows only one primary transaction to be transmitted simultaneously. In order to permit multiple functional units to issue several memory transactions at a time, the memory system must provide sufficient associativity to permit all such transactions to be launched. Also, if the memory system stalls the processor pipeline while multiple transactions are requested, then the processor must access a data word as soon as it arrives. These modifications prevent dependencies between simultaneous transactions and make sure that the window of vulnerability remains closed.

### 3.2.8 The Associative Thrashlock Solution

Despite its success in detecting thrashing in systems without high-availability interrupts, thrashwait (Section 3.2.7) fails to guarantee forward progress in the presence of such interrupts. This is a result of the method by which thrashwait closes the window of vulnerability: by causing the processor to spin. This corresponds to asserting the memory-hold line and freezing the pipeline. High-availability interrupts defeat this interlock by faulting the load or store in progress so that interrupt code can be executed. Viewing the execution of high-availability interrupt handlers as occurring in an independent "context" reveals that the presence of such interrupts reintroduces three of the four types of thrashing mentioned in Section 3.2.3. Instruction-data and high-availability interrupt thrashing arise from interactions between the thrashwaiting context and interrupt code. Invalidation thrashing arises because high-availability interrupts open the window of vulnerability, even for transactions that are targeted for thrashwaiting. Only intercontext thrashing is avoided, since software conventions can require high-availability interrupt handlers to return to the interrupted context. Consequently, a system with high-availability interrupts must implement more than the simple thrashwait scheme.

**Associative Thrashwait (Partial Solution):** In an attempt to solve the problems introduced by high-availability interrupts, we supplement the thrashwait scheme with associative transaction buffers. As described in Section 3.2.6, transaction buffers eliminate restrictions on transaction launches. Further, instruction-data and high-availability interrupt thrashing are eliminated. This effect is produced entirely by increased associativity: since transactions are not placed in the cache during their window of vulnerability, they cannot be lost through conflict. Thus, the *associative thrashwait* scheme with high-availability interrupts is only vulnerable to invalidation thrashing. The framework proposed in the next section solves this last remaining problem.

**Associative Thrashlock:** Now that we have analyzed the benefits and deficiencies of the components of our architectural framework, we are ready to present a hybrid approach, called *associative thrashlock*. This framework solves the problems inherent in each of the independent components.

Assume, for the moment, that we have a single primary transaction per context. As discussed above, thrashwait with associativity has a flaw. Once the processor has begun thrashwaiting on a particular transaction, it is unable to protect this transaction from invalidation during high-availability interrupts. To prevent high-availability interrupts from breaking the thrashwait scheme, associative thrashlock augments associative thrashwait with a *single* buffer lock. This lock is invoked when the processor begins thrashwaiting, and is released when the processor completes *any* global access. Should the processor respond to a high-availability interrupt in the interim, the data will be protected from invalidation.

It is important to stress that this solution provides *one* lock per processor. The scheme avoids deadlock by requiring that all high-availability interrupt handlers:

1. Make no references to global memory locations, and
2. Return to the interrupted context.

These two software conventions guarantee that the processor will always return to access this buffer, and that no additional dependencies are introduced[6]. Thus, associative thrashlock has the same transaction dependency graph as thrashwait without high-availability interrupts (as in Section 3.2.7). Processor access to the locked buffer is delayed – but not impeded – by the execution of high-availability interrupts.

Application of the above solution in the face of multiple primary transactions (such as instruction and data) is not as straightforward as it might seem. We provide a lock for both instructions and data (in addition to the two tried-once bits specified in Section 3.2.7). When thrashing is detected, the appropriate lock is invoked.

This locking scheme reintroduces a deadlock loop similar to the primary-secondary problem discussed earlier. Fortunately, in this case the loop is rather unnatural: it corresponds to two processors, each trying to fetch instruction words that are locked as *data* in the other node. To prevent this particular kind of deadlock, a software convention disallows the execution of instructions that are simultaneously being written. Prohibiting modifications to code segments is a common restriction in RISC architectures. Another method for preventing this type of deadlock is to make

[6]As will be shown in Chapter 5, the first condition can be relaxed somewhat, easing the burden of the runtime system.

instruction accesses incoherent. Since invalidations are never generated for instructions, the effect of the lock is nullified (no protocol arcs).

The complexity of the argument for associative thrashlock might seem to indicate that the architectural framework is hard to implement. It is important to emphasize, however, that even though the issues involved in closing the window of vulnerability are complicated, the end product is relatively straightforward. Section 5.2.3 explores the design aspects of the transaction buffer in more detail, with Section 5.2.3.3 discussing the specifics of the *Thrash Monitor* used to implement the associative thrashlock algorithm. Further, as we will see in the next two sections and in Chapter 5, the transaction buffer provides a remarkably flexible interface between the processor and memory system.

## 3.3 The Server-Interlock Problem

One way of viewing the existence of shared memory is as a client-server interaction between a processor and its cache (the client) and a remote protocol engine (the server). Given this viewpoint, we can ask the obvious question of what happens when a server becomes overloaded or interlocked for some reason. By an *overloaded server*, we simply mean one for which requests are arriving faster than they can be processed. In contrast, an *interlocked server* is one which is unable to immediately satisfy an incoming request for reasons of correctness. With the type of networks that we assume for Alewife-type machines, messages are typically *not* dropped for reasons of congestion. Consequently, an overloaded server causes messages to back up into the network, ultimately applying back pressure to the requesters. This is not particularly bad behavior, since the system will eventually make forward progress; in many cases it is preferable to the alternative of dropping messages and forcing retries on timeouts.

However, the situation of an interlocked server is more problematic. To make this situation a bit more concrete, one property of cache-coherence protocols is that they exhibit periods of time in which memory coherence engines are unable to satisfy new requests for particular memory lines because these memory lines are in an inconsistent state. A good example is the period of time between the initiation of an invalidation operation and the reception of the final acknowledgment. During such periods, new requests cannot be handled and must be deferred in some way. Further, because the network does not drop packets, we cannot simply refuse to accept these new requests, since this can deadlock the system. This is a situation very similar to the refused-service deadlock (Section 3.1): the message at the head of the input queue (request) cannot be processed before the arrival of pending acknowledgments (which are behind the request). Thus, we have two possible solutions: (1) queue the request for later processing and (2) send some form of negative acknowledgment (NAK) to the requester, forcing a later retry of the request. Some systems (in particular the Scalable Coherent Interface (SCI) [98]) make use of the first of these (burying the complexity of this solution in an already complex protocol), while the second is employed by Alewife, Dash, and Origin, among others.

### 3.3.1 Queuing Requests for an Interlocked Server

Let us spend a couple of moments exploring the queueing solution. Queueing is attractive from an abstract standpoint because it obviates any need for mechanisms to prevent livelock or ensure fairness. Unfortunately, queueing entails a number of complexities that make it a solution not often taken by hardware designers (software queueing is another story entirely). First, providing storage in blocks large enough to handle the largest number of requesters (likely the largest number of processors in the system), is not an efficient use of space. Hence, hardware queuing usually entails formation of linked-lists directly in hardware. This sort of processing (and the almost inevitable free-list handling) is a level of complexity best avoided in hardware. However, even if the complexities of linked-list processing in hardware are overcome, the problem of allocating sufficient server-side resources for all possible bad situations still remains. As a result, unless sufficient space can be guaranteed server-side to handle the worse-case request patterns, some fallback mechanism would be necessary over and above the queueing mechanism.

There is an alternative, however. Queueing of requests at the server requires a potentially unbounded (or at least unpredictable) amount of memory simply because every node in the system *could* simultaneously request data from a single server. Consider this from the standpoint of the requesting nodes for a moment. The discussion of the transaction-buffer framework in Section 3.2 served to suggest that shared-memory systems should explicitly track outstanding transactions for a number of reasons. This, in turn, puts a well defined hardware limit on the number of outstanding requests that can be present at any one time from a given node[7]. The upshot of this is that a distributed queueing mechanism *could* be guaranteed to contain sufficient resources, since the cost of queue space scales linearly with the number of nodes; in fact, the space that would be used for queueing might already be there (for the transaction buffer).

This is a variant of the argument used to justify the *Scalable Coherent Interface* (SCI) protocol [98]. For SCI, the coherence protocol forms sharing chains through the caches in order to form a distributed coherence directory. What this means is that the coherence protocol forms a doubly-linked list of all cached copies of a given memory line, and that this chain threads through the caches where the data resides. As a result, the space required for maintaining coherence is exactly matched to the amount of cache and thus grows linearly with the number of nodes in the system. This is in contrast to most other coherence protocol schemes (including the LimitLESS coherence scheme of Alewife), that keep track of data sharers at server-side and hence have potentially unbounded coherence memory requirements. There are a number of downsides to SCI, not the least of which is the fact that the protocol is quite complicated, requiring extensive automatic validation in order to ensure correctness (see [110], for instance for a look at the set of bugs discovered in the base protocol after it had been out as a tentative standard). With respect to the server-interlock problem that we have been discussing, however, the fact that SCI performs queueing in hardware means that it naturally handles incoming requests during periods of invalidation: SCI simply builds the so-called *prepend queue* of requesters at the head of a linked list – even while interlocked for invalidation [98, 54].

[7] In fact, modern pipelines with out-of-order execution often have a limit to the number of outstanding requests for a similar reason. For instance, the R10000 has a maximum limit of four outstanding requests (including prefetches) at any one time [92].

While SCI solves the server-interlock problem directly, it has several unfortunate properties in addition to its complexity. First, and foremost, studies of the properties of a number of parallel programs suggest that the "worker-set" or number of processors simultaneously sharing a given piece of data is usually small [22]. Hence, SCI optimizes for the uncommon case rather than the common case pattern of sharing. Further, since it forms linear chains of data sharers, it risks long latencies during invalidation. Attempts to correct latency problems have only lead to more complexity [55, 54]. Finally, the fact that sharing chains stretch through individual caches means that expensive, high-speed storage (in the form of cache-tags) is being used to support sharing rather than less expensive DRAM resources.

**Queueing With the Transaction Buffer:** Hence, for a "complete" queueing solution, we suggest adopting the prepend-queue aspect of SCI while avoiding most of the complexity. Since the server-interlock problem occurs only during shared-memory requests, its solution falls naturally into the domain of the transaction buffer as suggested earlier. A prepend queue could be constructed with a singly-linked list pointing from from one pending operation to the next. Doubly-linked lists (such as for SCI) would be unnecessary, since transaction buffers are normally allocated for the duration of the request process, thus pending requests would not need to be removed from the chain (one of the primary reasons for the doubly-linked lists of SCI). Further, transaction-buffer storage is small; the space used for this solution would not require fast pointer storage that scaled with cache size. Finally, since chains such as this would not impact the method of coherence invalidation, this would not exhibit the increased-latency that is an aspect of SCI. Note that queueing as described here is used primarily to control access to highly-contended locations, optimizing for variables that are under such rapid modification that the worker-sets are small.

This is a natural use of transaction buffers; however, Alewife does not take this approach (although it could have). Instead, it makes use of a non-queueing solution, described in the next section.

### 3.3.2 Negative Acknowledgment and the Multiple-Writer Livelock

As described above, hardware queueing has a number of complexities. An alternative way of avoiding deadlock during server-interlock periods is to discard excess requests by sending NAK messages back to the requestor. DASH did this [69], Alewife did this [23], and others have done this. While this removes a large amount of complexity due to queueing, it forces requesting nodes to retry their requests, thereby leading to the possibility of livelock[8]. Although we have been doing so implicitly it is perhaps interesting to contrast this memory-side livelock with the processor-side livelock that results from the *window of vulnerability* (Section 3.2): these are, in some sense, duals of each other, since both stem from the cache-coherence protocol.

Before proposing ways to fix this livelock, we need to examine typical directory-based coherence protocols in more detail. In particular, most of these protocols are of a multiple-reader, single-writer variety. What this means, is that these protocols handle multiple simultaneous readers naturally (without periods of server interlock). However, once readers are combined with writers,

[8]Nodes that are closer to the server may have a significant advantage accessing data over remote nodes, effectively locking out attempts by remote nodes to access this data.

coherence mechanisms start kicking in, causing periods of server interlock and memory-side livelock. Thus, memory-side livelock arises in the presence of multiple writers (hence the fact that we have called this the *multiple-writer livelock* in other places of this thesis).

As a consequence, Alewife's approach to the multiple-writer livelock is to guarantee that at least one writer always succeeds by retaining the node identifier for the request that triggered an invalidation in the coherence directory. This hardware property is sufficient to guarantee that *test&test&set* spin-locks may be constructed:

- Lock acquisition in the face of contention requires that at least one writer make forward progress, even while many are simultaneously attempting to write.
- Lock release in the face of contention requires that the lock owner be able to write to the lock in the fact of many other readers.

Although this does not fully solve the multiple-writer livelock (since there is no guarantee that such locks will be granted fairly), this hardware property is sufficient to enable construction of software queue locks (*e.g.* MCS locks [84]). Thus, we "guarantee" memory fairness through this combined hardware/software solution. In Note that the single-writer guarantee is more than sufficient by itself for many (perhaps most) uses of shared memory.

**Use of Request Priorities:** The SGI Origin takes a slightly different approach to eliminating the multiple-writer livelock. Each request is given a "priority" value which starts low and is incremented whenever the request is retried. In this way, requests can be guaranteed to eventually achieve a priority value that is higher than any other simultaneous requesters [67]. This technique provides an elegant solution to the multiple-writer livelock, removing it entirely with little hardware complexity or cost in resources. Further, with such a hardware solution, users of the shared-memory model do not need to be aware of this issue.

## 3.4 Protocol Reordering Sensitivities

This section presents two problems that can arise from the reordering of events (where an event is notification that something has happened, either with or without data). The first issue is sensitivity to reordering of cache-coherence messages in the network. We will demonstrate how an uncompensated protocol can be sensitive to reordering of messages and show how the transaction buffer can be used to help solve this problem. The second issue has to do with forwarding of events between hardware and software. In Alewife, the CMMU operates independently of the processor under many circumstances; however, certain exceptional situations are passed to software for handling (*e.g.* the LimitLESS cache-coherence protocol passes instances of wide sharing to the processor). This policy exhibits a tradeoff between parallelism and correctness: we would like the CMMU and processor to operate as independently as possible, while still performing a correct handoff of events from hardware to software. We will show how two mechanisms, the *faultable flush queue* and *directory locks* may be used to ensure a smooth handoff from hardware to software.

### 3.4.1 Achieving Insensitivity to Network Reordering

In machines such as Alewife, cache coherence is maintained by a cache-coherence protocol. Such a protocol consists of a sequence of operations applied to memories and caches to achieve the *illusion* of continuous data coherence within the system. Operations are mediated by messages in the system. A basic assumption of coherence protocols is that processors operate asynchronously from one another and that no global ordering of messages exists. However, a naive view of the network implicitly assumes that there is a *partial* ordering of messages, *i.e.* that two messages sent in sequence from a given node to another node will arrive in the same sequence. This assumption allows the collective state of data in caches and memories to be viewed as a set of interacting state machines, one for each memory line. While this assumption makes cache-coherence protocols easier to reason about, it comes at the cost of requiring the network to provide such ordering. In fact, there are a number of reasons for relaxing this constraint:

- It permits use of adaptive or fault-tolerant networks that route packets non-deterministically.
- It relaxes the design of the processor side of the coherence protocol, since invalidations and data from local memory can be handled independently[9].
- It affords flexibility in choosing the algorithm that is used for two-case delivery (discussed in Section 4.3.5), since software may deliver buffered in a different order from that in which they arrive.

Hence, for the remainder of this section, let us assume that the network does not actually preserve the ordering of messages.

Figure 3-13 illustrates one type of problem that may arrive if a cache coherence protocol is naively designed with an assumption of strict network ordering. This example shows read-only data (RDATA) returning from a remote memory module to the cache. While this data is still in transit, the memory issues a read-invalidation message (INVR) to the same cache. Because the INVR was transmitted after the RDATA, an invalidation acknowledgment should not be sent to memory until after the RDATA has been discarded. However, as illustrated by the third and fourth panels, this does not happen. For whatever reason (network contention, perhaps), the INVR arrives *before* the RDATA, even though it was originally transmitted *after* the RDATA. As a result, the cache sends an invalidation acknowledgment (ACK) without discarding the read-only data. When this

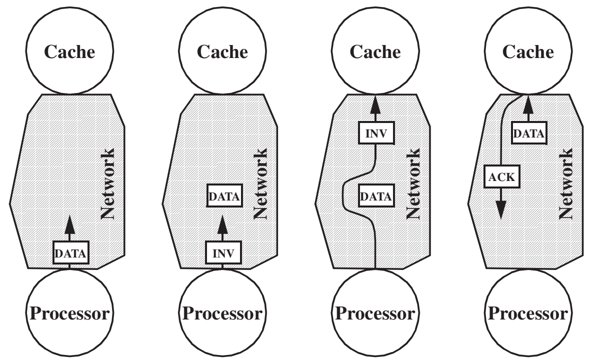



Figure 3-13: The effect of network reordering on an uncompensated protocol.

[9] Protocol invalidations require the examination of the cache tags-file, while returning data requires only placement in the transaction buffer. In the Alewife CMMU, invalidations are handled by one state machine, while returning data is handled by another.

data finally arrives, it would be blithely placed in the cache. Meanwhile, the memory would receive the `ACK` message and grant write permission to another node. Cache incoherence would result.

While independence from network ordering can always be gained by restricting the number of simultaneous messages to one, this restriction requires explicit acknowledgments for every message reception, and this requirement, in turn, increases required network bandwidth (more messages) and increases memory access time (more serialization). Thus, we prefer a solution which has far less communication overhead with respect to a similar protocol that relies on order. To this end, the Alewife CMMU carefully *restricts* the number and type of protocol messages that can be simultaneously present in the network and maintains sufficient state at message destinations so that misordering events can be recognized and dealt with appropriately. To this end, the *transaction buffer* assists by providing state at processor side of the protocol.

**Analysis of the Problem:** There are two possible destinations for protocol packets: memory and cache. Each must deal with reordering. At the memory side, two types of messages can arrive from the processor: requests for data (`RREQ` and `WREQ`) and "returns" (*i.e.* acknowledgments or updates). Acknowledgments (`ACK`) are generated in response to an invalidation request. Update messages (`UPDATE`) contain dirty data and may be returned to memory during replacement of read-write data in the cache or in response to an invalidation request. The ability to deal with reordering depends on two invariants:

- Transaction buffers at the processor side guarantee that no more than one simultaneous request will be issued from a given processor to a given memory line.
- The protocol guarantees that at any one time there is *at most one* return message that might arrive from a remote cache. Examples of periods in which return messages "might arrive" include (1) the period after an invalidation is sent but before an acknowledgment has arrived, and (2) the entire period beginning with the issuing of a read-write copy to a remote cache and ending with the return of dirty data in the form of an `UPDATE` message.

Thus, only the ordering between requests and returns is at issue. This is addressed by an essential feature of the coherence protocol: during periods in which a return message *might* arrive, no new requests are processed; they are either deferred or aborted. As an example, suppose that a node transmits an acknowledgment in response to an invalidation from memory, then proceeds to issue another request to the same memory-line. If that request reaches memory first, then the memory will abort the request by returning a `BUSY` message (the memory is currently busy performing an invalidation at the behest of a different node). The original node will reissue its request at a later time. Consequently, an explicit processing order is enforced, regardless of network order.

At the cache side, two types of messages can arrive: data and invalidations. Correctly handling reordering between these types of messages hinges on several things. First, the explicit recording of transactions guarantees that there is no more than one simultaneous request for a given memory line; consequently, there can be no more than one data item in flight from memory to the processor[10]. Furthermore, data for a given address can arrive from memory *only* during those periods in which the transaction store contains a matching transaction buffer with TIP set.

In addition, the memory management hardware will generate no more than one invalidation for each piece of data that it sends to the processor. Invalidations are divided into two distinct varieties:

1. Read invalidations (`INVR`), which invalidate read copies and generate acknowledgments.
2. Write invalidations (`INVW`), which invalidate write copies (if they exist) and generate updates. If they encounter no data, then they are ignored.

These two varieties reflect two distinct types of cache replacement and are necessary to maintain the "at most one" property of return messages to memory. When replaced from the cache, read copies are simply discarded; thus, `INVR` messages must return explicit acknowledgments. After an `INVR` has been transmitted by the memory, no further messages (either invalidations or data) will be sent by the memory until an acknowledgment has been received. In contrast, dirty read-write copies generate update messages when they are replaced from the cache; since this can happen at any time, `INVW` messages can only be used to accelerate the process and "dislodge" read-write copies. They are otherwise ignored by the cache.

**Achieving Insensitivity:** Thus, we have three types of misordering which may occur between messages destined for the cache (reordering between two data messages cannot occur, since only one data item can be in flight at once):

- Between two invalidations: At least one of the two invalidations must be an `INVW` messages, since no more than one `INVR` can be in flight at once. If both of them are `INVW` messages, then reordering is irrelevant. If one is an `INVR`, then this must have been for a read copy. Consequently, there are no write copies for the `INVW` message to invalidate and it will be ignored. Again reordering is irrelevant.
- Between an invalidation and a data item, where the invalidation is transmitted first but arrives second: The invalidation must be an `INVW` message, since all `INVR` messages must be explicitly acknowledged before the memory will start processing a new request. Consequently, if the data item is a read copy, then the `INVW` message will be ignored. If the data item is a write copy, then the `INVW` message might invalidate the data prematurely, but will not violate cache consistency[11].
- Between an invalidation and a data item, where the invalidation is transmitted second but arrives first: This was the example given in Figure 3-13 and can lead to incoherence if not dealt with properly. Memory will be incorrectly notified that an invalidation has occurred.

[10]In particular, this restriction means that Alewife does not support an update-style protocol. Such a protocol can send update messages at any time; unfortunately, no analog of "transaction-in-progress" would be available at the processor side.

[11]This is a situation in which reordering causes a slight loss in performance.

Consequently, of the three possible types of reordering, only the last must be recognized and corrected. The heuristic that is employed in Alewife is called *deferred invalidation*: invalidation requests that arrive while a transaction is in progress and that are of the same type as this transaction are deferred until the data is received by setting the INV bit in the transaction buffer. When the data arrives, it is discarded and an acknowledgment is returned to the memory[12]. Invalidations that are not of the same type as the transaction cannot be destined for the expected data; they are processed normally. An individual transaction buffer interacts with premature invalidations by setting a special bit in its state vector. See section refsubsec:alewife-transbuffer.

The policy of deferring invalidations is only a heuristic: although it successfully untangles the third type of reordering, it can cause invalidations to be held even when reordering has not occurred. This delay occurs when data has been replaced in the cache and re-requested. Invalidations which arrive for the old data are deferred rather than being processed immediately. Only a returning data or `BUSY` message terminates the transaction and performs the invalidation[13].

### 3.4.2 Achieving a Clean Handoff from Hardware to Software

The second type of protocol reordering sensitivity that we will consider is that of handing off of events from hardware to software. This is of particular interest in Alewife because of Alewife's philosophy of implementing common cases in hardware and uncommon cases in software; the LimitLESS cache coherence protocol is one particularly salient example of this. The issue to be addressed is the tradeoff between hardware/software parallelism and correctness. One of the reasons this is particularly important is the fact that software handling of events is typically several orders of magnitude more expensive than equivalent hardware handling; such a discrepancy immediately rules out stopping all hardware handling during periods in which software actions are pending.

One obvious solution to this type of problem is to decouple the processor and hardware through queues. Decoupled architectures such as the ZS-1 [103] and WM machine [118] were among the first to use queues and register interlocks to decouple load and store operations from computation operations. In these architectures, the potential for parallelism between memory operations and ALU operations is ultimately limited by the maximum load/use distance that can be achieved (number of instructions between the loading of a piece of data and its use). However, in a memory controller, much greater decoupling is possible, since each memory operation is potentially independent from the next. What we would like to do in this section is show how the basic notion of decoupling is applied in Alewife to permit a graceful handoff between hardware shared-memory mechanisms and software. To this end, we will discuss two mechanisms: the *directory interlocks* and the *faultable flush-queue*.

[12] If the buffer is locked (See Section 5.2.3), then returning data will be placed into the buffer and the TIP bit will be cleared. This places the buffer in a transient state. Consequently, the invalidation will be further deferred until the data is accessed by the processor and the lock is released.

[13] This is an example in which the *existence* of a mechanism for reordering can impact performance under normal circumstances.

#### 3.4.2.1 Directory Interlocks

In the LimitLESS cache-coherence protocol, some "corner-case" operations are handled by forwarding them from hardware to software [23]. Since software handlers take nearly an order of magnitude longer to process requests than the hardware does[14], some amount of decoupling is desirable. Two types of decoupling are exploited in this way: First, in most (if not all) cache-coherence protocols, every memory line is independent of every other one. We exploit this independence by allowing the hardware to continue to process unrelated memory requests while a particular request is queued for software handling. Second, data and control operations in a cache-coherence protocol may be separated in some instances to achieve pipelining. In Alewife, while software is busy handling the coherence aspects of a read operation, the hardware will fetch data from memory and return it to the requesting processor; requests for software handling may remain queued awaiting processing long after read data has been returned to the requestor. This is called the *read-ahead optimization*[15].

These two types of decoupling require care in implementation, since they can easily lead to incoherence. As discussed in Section 3.4.1, coherence protocols typically involve one set of interacting state-machines for each memory line. State transitions that are handled in software are similar to hardware transitions, with the exception that they take longer and are not atomic with respect to the hardware: herein lies the danger. Consider the specific case of decoupling hardware and software with respect to requests that arrive across the network. Since the network input queue presents packets one at a time to the CMMU, requests that require software processing must be removed from the network input queue in order to allow in order to achieve parallelism between the hardware and software. During the time in which a software request remains pending, the state transition for that particular state machine has not yet fully occurred and hence the hardware state *is not reflective of the complete state of the corresponding state machine*. If we blindly process incoming requests without accounting for pending software transitions, we can easily perform incorrect transitions (just because one request requires software processing does not necessarily mean that a subsequent one also requires hardware handling).

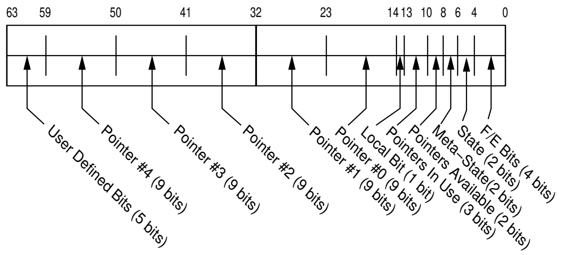



Figure 3-14: The Alewife coherence directory.

| Name of Meta-State | Description |
|---|---|
| `Normal` | dir. entry under hardware control |
| `Trap_On_Write` | reads handled by hw, writes by sw |
| `Write_In_Progress` | dir. entry software interlocked |
| `Trap_Always` | all requests handled by sw |

Table 3-3: The four meta-states of the Alewife coherence directory.

The description of the problem is nearly sufficient to suggest a solution: interlock. When software requests are pending, we can simply leave a mark in the hardware state machine to indicate that a request is pending. These special interlocks are accomplished by "meta-states" in the Alewife coherence directory. Figure 3-14 illustrates the format for the coherence directory and

[14]Although this number is not particularly fundamental, it is amazing how much time can be taken by such simple things as searching for a software directory in a hash table, manipulating directories at the bit level, etc.

[15]The FLASH multiprocessor from Stanford [64] takes this approach in a somewhat more conservative fashion: in that machine, software handling of coherence operations is separated from data prefetching but must simultaneously.

Table 3-3 lists the four "meta-states". The `write_in_prog` meta-state is an interlock state: when a directory entry is marked as `write_in_prog`, all hardware processing of requests to that directory entry are suspended; this means that requests that arrive for such a directory entry are left at the head of the network input queue until a software handler changes the meta-state to something other than `write_in_prog`. The presence of a per-directory-entry `write_in_progress` state means that we get maximum hardware/software decoupling as long we have enough queue space to handle incoming requests and as long as we avoid multiple requests for the same memory line: whenever the hardware decides to pass a request to software, it simply changes the directory to `write_in_progress` state, then passes the request into the network input queue, just as it does for other, message-passing messages.

The `normal`, `trap_on_write`, and `trap_always` meta-states cause slightly different behavior at the interface between hardware and software. The `normal` meta-state indicates a directory that is entirely under hardware control. In this state, the only type of request that might require software handling in this situation is a read request. Should a read request require software handling (because all five of the pointers are in use), we switch the meta-state to `trap_on_write`, place the request in the message input queue, and send data back to the requestor. This behavior produces the read-ahead optimization. The `trap_on_write` meta-state indicates the the hardware directory does not contain sufficient information to handle a write request. However, read requests may continue to be handled as above, producing read-ahead behavior. Finally, when a write request encounters the `trap_on_write` meta-state, it causes a transition to `write_in_prog`, since no further requests may be handled for that memory line until after software has processed the directory.

The `trap_always` state is a generic state for special protocols that require 100% software processing. Whenever an incoming request encounters `trap_always`, it invokes an immediate transition to `write_in_prog`, forwarding data to the processor.

**Directory Access:** The above discussion implied that directory entries could be accessed directly by software. This, in itself, can cause problems if both software and hardware attempt to modify a directory entry at the same time. Thus, Alewife provides special instructions for directory access, called `rldir` (read and lock directory) and `wudir` (write and unlock directory). The `rldir` instruction loads a particular directory entry into the processor, while at the same time invoking the *directory lock*. The directory lock is a flag to the hardware that tells it to block if it is about to process a request for the locked entry. The `wudir` operation writes and unlocks the specified entry. This, combined with the interlock meta-states, provides maximum parallelism between hardware and software, *i.e.* interlocking only when necessary but retaining correctness.

#### 3.4.2.2 Faultable Flush Queue

The second reordering sensitivity that we would like to examine arises during the handoff of events from hardware to software during cache replacement. Alewife complicates the replacement process in two different ways:

- The `trap_always` state requires all protocol processing to pass through software. While this provides a great source of flexibility, it also complicates the replacement process, since items that have been replaced from the cache may require software handling before they can be retired.

- The Alewife philosophy of handling exceptional cases in software requires an unshared address space, since only local unshared memory can be guaranteed to exhibit no dependencies on the network. We discussed the need for unshared memory in the last chapter and will revisit it in the next. Thus, we need to provide a path to the memory system that can be guaranteed to complete, despite the fact that shared-memory accesses may be blocked indefinitely.

Ordering of replacement events is particularly important because an incorrect ordering can effectively deadlock the system: if unshared accesses are unable to bypass shared accesses, then the network overflow recovery process may not be able to complete. Further, replaced memory lines that require software handling must be able to wait somewhere in the memory system for software handling without blocking other accesses.

The transaction buffer provides a particularly clean solution to this problem. In order to avoid blocking all memory actions during pending replacement operations, flush queues require associative matching circuitry (otherwise incoherence of local data can result). This is provided by the transaction buffer already. Further, we can mix unshared-memory and shared-memory replacements in the transaction buffer, as long as we guarantee that some minimum number of entries are reserved for unshared traffic (solving the second of the two problems above). Finally, the transaction buffer provides an ideal place to store replacements that are "faulted", awaiting software processing. The transaction buffer can be viewed as both a queue and as a random access memory, depending on the mode of operation. As shown in Chapter 5, we can get the behavior of two interspersed queues through special transaction buffer flush states and "monitors" that decide which of a set of pending replacements in the transaction buffer should be flushed next.

## 3.5 Postscript: The Transaction Buffer Framework

As discussed above, the transaction buffer provides a natural framework in which to solve many aspects of the service-interleaving problem:

- By monitoring outstanding transactions, it provides a framework in which to prevent inter-processor data thrashing and to close the window of vulnerability. This level of processor-side monitoring is also sufficient to permit make the cache-coherence protocol insensitive to network reordering.
- By providing associativity, it protects against inter-context thrashing on a single node. This enables, among other things, greater flexibility in reflecting exceptional conditions to software.
- It provides a natural interface between the processor and memory system, allowing guarantees to be made of access to unshared memory. Part of the flexibility of providing two "independent" memory systems (shared and unshared) stems from the ability to reorder flush operations (the *faultable flush queue* of Section 3.4.2.

Thus, we could consider the transaction buffer as one of the primary results of this chapter. In Section 5.2.3, we will revisit the transaction buffer from the standpoint of implementation. We will demonstrate how to achieve all of the advantages of centralized resource tracking without compromizing performance.

---

**Collaborators:** David Chaiken was the first to observe and address internode data thrashing problems with the ASIM simulator, back in the early days of the Alewife project. He was a key collaborator in addressing many of the issues arising from the window of vulnerability. He was also an important contributer to the transaction buffer design including its use as a framework for handle misordering of protocol packets in the network.

---

# Chapter 4

# The Protocol Deadlock Problem

This chapter examines the problem of guaranteeing deadlock freedom in a machine that supports both message passing and shared memory. The deadlocks examined in this chapter are all distributed in nature, *i.e.* they involve multiple nodes. Further, unlike the deadlocks of the previous chapter, which could be avoided through careful restriction of resource locking or removed by re-ordering of handlers, the deadlocks of this chapter are present as a consequence of cycles in the communication graph. Hence, a global solution is required.

By way of motivation, we first describe the deadlock problem in cache-coherent shared memory and explore the "mathematical" approach for providing deadlock freedom by breaking cycles in the communication graph. Such an approach has been employed in machines such as the Stanford DASH multiprocessor and the Origin multiprocessor from Silicon Graphics. In fact, as we will show, guaranteeing deadlock freedom has complexities and costs that increase with the complexity of the cache-coherence protocol.

Next, we examine the problem of guaranteeing deadlock freedom in a machine with general message passing. Although machines such as the CM-5 *have* provided mechanisms to avoid deadlock with a restricted class of communication patterns (such as request/response communication), these mechanisms are not powerful enough to guarantee deadlock freedom with general communication patterns. More importantly, users are under no compulsion to make use of deadlock-avoidance features even if they exist. Hence, machines with user-level message interfaces, such as described in Chapter 2 for Alewife, introduce the possibility of user-induced network deadlock — a type of deadlock that operates beyond the domain of carefully constructed hardware or operating-system software to prevent. As a case in point, programmers have had a tendency to ignore the deadlock avoidance features provided on machines such as the CM-5, adopting ad-hoc methods of buffering instead[1].

With these two discussions as a backdrop, we will spend the bulk of this chapter examining an alternate method for deadlock avoidance called *two-case delivery*. Two-case delivery refers to on-demand software buffering of messages at their destination as a method for breaking deadlock cycles. Messages that are buffered in this way are not dropped, but later delivered to the appropriate hardware or software consumer; hence the name "two-case delivery." Since software is invoked

[1]In the specific case of the CM-5, greater aggregate network bandwidth is often achieved by treating the request and response channels as alternate pathways — independent of message type.

to handle exceptional situations, one viewpoint of two-case delivery is that it is a software recovery scheme. Hence, in Chapter 6, we will introduce DeadSIM, a queueing simulator designed with the express purpose of examining the frequency of protocol deadlock under varying machine parameters. The frequency of deadlock is important to us precisely because the performance of software recovery methodologies (such as two-case delivery) is crucially dependent on the frequency with which software must be invoked. We will find that deadlock can indeed be made infrequent with appropriate selection of queue sizes.

Among other things, two-case delivery simplifies the design of the cache-coherence protocol, since the protocol can operate *as if* it has as many virtual channels as necessary to avoid deadlock without actually requiring such channels to exist. This solution is one of the system-level benefits resulting from the efficient message-passing interfaces provided on Alewife.

## 4.1 Cache-Coherence Protocols and Deadlock

By their very nature, cache-coherence protocols contain cycles in their communication graphs. The cycles exist because messages arriving in the input queue (e.g. shared-memory requests) cause the generation of messages on the output queue (e.g. data responses). This dependency, combined with finite queue resources, leaves open the possibility that two nodes could *deadlock* each other. Figure 4-1 illustrates the simplest of such deadlocks. In this figure, both nodes have full input and output queues and both nodes have packets at the downstream side of their output queues that are destined for the input queue of the other node. Deadlock results because neither node is able to remove packets from its input queue since it is unable to place responses on its output queue. This problem is fundamental to cache-coherence protocols and hence must be solved in all shared-memory multiprocessors.

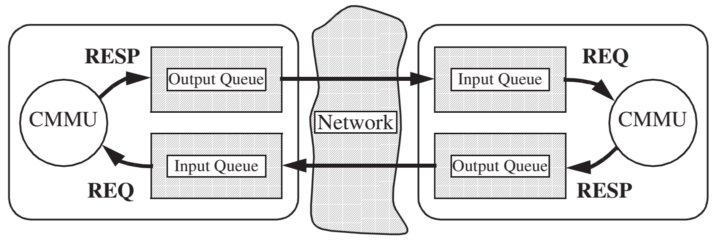



Figure 4-1: Cache-coherence protocols contain cycles in their communication graphs that can cause deadlock.

Figure 4-2 illustrates the queue-level dependencies that lead to the simple deadlock. This figure is a *waits-for-graph* (WFG) such as we encountered in the last chapter. It shows two nodes interacting across a network. Nodal resources are enclosed within rounded boxes, while arrows represent both intra-node dependencies and network dependencies (*i.e.* full input and output queues that are coupled through the network)[2]. Each node contains a source of requests (marked `REQ`), a sink of responses (marked `DATA`), and a protocol processor that takes remote requests and produces responses (marked `REQ`⇒`DATA`). The output queue receives both locally generated requests and responses destined for remote nodes; this fact is symbolized by the box marked `MERGE`. Since the input queue contains a mixture of requests and responses from remote nodes, these must be demultiplexed; this function is represented by the box marked `SORT`. Given this diagram, the source of deadlock is clear: there is a cycle that includes the two protocol processors, the network, and

[2]In Section 4.3, we will revisit the implicit assumption of finite queue resources.

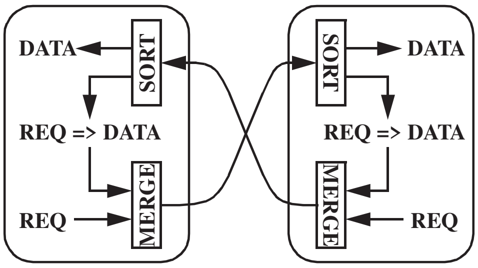



Figure 4-2: Schematic representation for simple request/response deadlock

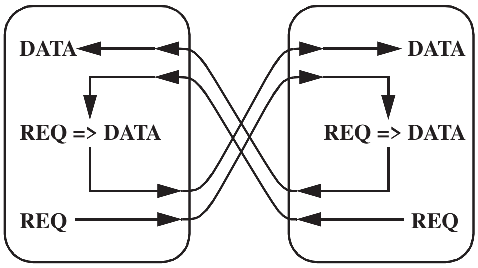



Figure 4-3: Breaking the simple deadlock with multiple network channels

the SORT and MERGE hardware. As a result, it is possible for input and output queues to be full and a request to be at the head of each of the input queues; these requests will emerge from the protocol-processor branches of the SORT elements, but be unable to be freed because of dependencies through the MERGE elements to the output queues.

Note that low-level network hardware is usually designed to avoid deadlock. However, to achieve this guarantee (i.e. that every message injected into the network will eventually be delivered to its destination), users of the network must remove packets as they are delivered. Thus, as we can see from Figure 4-2, it is the dependencies introduced by the protocol processors that are responsible for deadlock in cache-coherence protocols. Hence the name: *protocol deadlock.*

### 4.1.1 Breaking Cycles With Logical Channels

One solution to this particular deadlock, employed by the DASH [69] multiprocessor, is to recognize that responses (such as data) can be guaranteed to be "sinkable" at their destinations: as a precondition to sending a remote request for data, the source of the request can allocate sufficient space to handle the response when it returns. Consequently, we can take advantage of two network channels, as shown in Figure 4-3, to prevent deadlock: we assign requests to one channel and responses to the other. In this diagram, each network path from Figure 4-2 has been replaced by two independent (or *logical*) network channels. Given this topology, no cycles remain in the communication graph; hence, the protocol processor (responsible for the transformation REQ⇒DATA) is never blocked indefinitely. Although DASH used physically separate networks (with independent wires), *virtual channels*[31] represent a more practical implementation methodology. For the remainder of this discussion, we will use the term "logical channel" to denote an independent path between nodes; we will have a bit more to say about implementation later.

Unfortunately, this solution is not sufficient for most cache-coherence protocols, since their coherence protocols contain more complicated dependencies than simple request/response. Figure 4-4 illustrates several protocol message dependencies that appear in the DASH and Alewife multiprocessors. Each of the circles in this diagram represents a physical node while each arrow represents a message sent between nodes. Below each diagram is a *message dependency graph.* The "⇒" symbol indicates a protocol dependence, in which the reception of a message to the left results in the transmission of one or more messages to the right. Each message that is generated can, in turn, give rise to additional chains of dependency, *i.e.* protocol dependencies can form

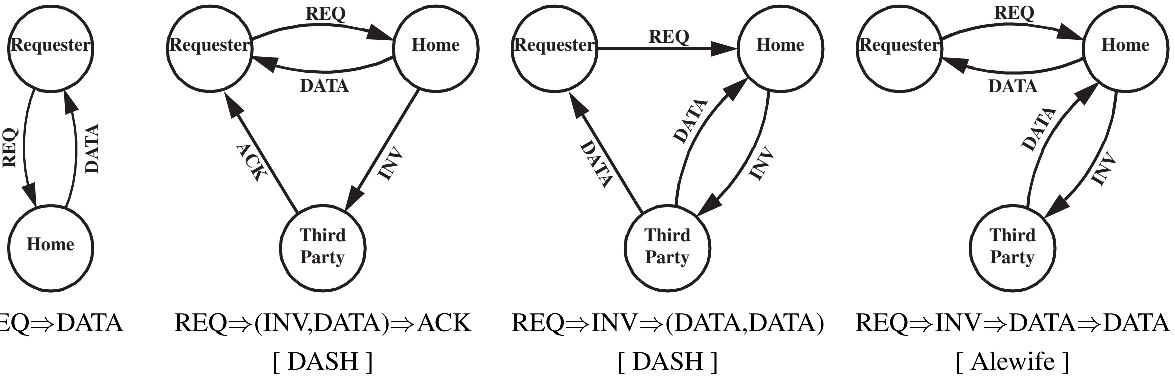



Figure 4-4: Typical coherence-protocol dependencies. Circles represent physical nodes and arrows represents messages. Below each figure is a symbolic representation of the *message dependency graph*. This graph is, in general, acyclic. Each "⇒" represents an action by the protocol processor.

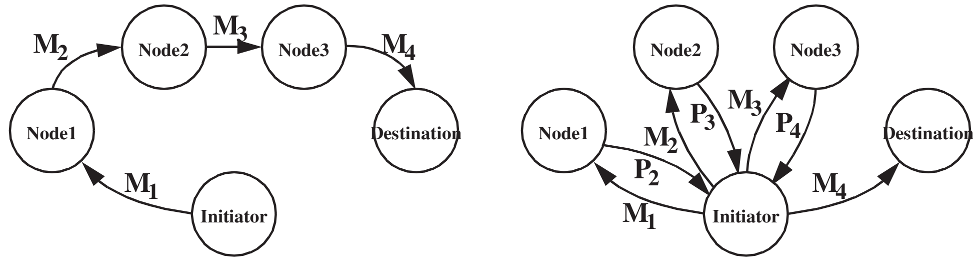
general acyclic graphs rather than just dependence chains.

For instance, the third dependence shown in Figure 4-4 is the data forwarding optimization of the DASH coherence protocol. This is a three way dependence, `REQ⇒INV⇒(DATA,DATA)`, that can be interpreted as follows: a node (*Requester*) starts by sending a REQ message to another node (*Home*). This node receives the REQ and responds with an INV message to the current holder of data (*Third Party*). Finally, this last node receives the INV message, then responds by sending out two copies of the DATA: one to the *Requester* and one to the *Home*. In effect, this implies two separate protocol-level dependencies, *i.e.* `REQ⇒INV` and `INV⇒(DATA,DATA)`. Diagrams similar to Figure 4-2 and 4-3 can be constructed to demonstrate that *three* independent network channels are required for complete deadlock freedom (otherwise there is a possible three-node deadlock).

The upshot of this is that the base DASH coherence protocol is not *quite* deadlock free, despite the fact that two different network channels were provided to "eliminate" deadlock; to avoid physically deadlocking network channels, the memory controller aborts dependencies such as `REQ⇒INV` by sending a NAK message back to the source under certain situations involving full queues[69]. This solution transforms *potential* deadlock into a potential livelock, since the request must be subsequently retried[3]. Note that this livelock is of a different variety from the Multiple Writer Livelock discussed in Chapter 3, since it depends on the vagaries of network behavior and cannot be corrected by software — even in principle. Consequently, the DASH machine does not have a complete deadlock avoidance solution.

In general, the most straightforward implementation of a coherence protocol requires as many independent network channels as the maximum depth of its set of protocol dependence graphs. We will call this the *dependence depth* of the protocol. Note that the DASH protocol has a dependence depth of three, while Alewife has a dependence depth of four. Assuming that we have as many logical channels as the dependence depth, we simply number the channels sequentially and switch to the next sequential logical channel with each protocol action ("⇒"), *i.e.* for each protocol

[3]Implementors of DASH claim that this was never a problem in practice, possibly because of the small number of processors and/or the large network queues (Private communication).




$M_1 \Rightarrow M_2 \Rightarrow M_3 \Rightarrow M_4$
[ Original transaction ]

$M_1 \Rightarrow P_2$, $M_2 \Rightarrow P_3$, $M_3 \Rightarrow P_4$, $M_4$
[ Reduced transaction ]

Figure 4-5: Reducing the dependence depth of a cache-coherence protocol by breaking long dependence chains. In this specific example, a transaction of dependence depth four is transformed into four separate transactions each of dependence depth two.

action initiated by a message, we send outgoing messages on the logical channel whose index is one greater than the channel on which the message was received. This guarantees that there are no cyclic dependencies between logical channels since each channel depends only on the channel whose index is one greater[4]. If we guarantee that messages at the leaves of dependency graphs can always be accepted at their destinations, then the protocol will be deadlock free. Conversely if we have fewer logical channels than the dependence depth, than there will be at least one protocol transaction for which we must reuse one or more of the logical channels. This introduces a cycle in the communication graph and hence the potential for deadlock. The design complexity incurred to eliminate deadlock is in the generation of protocol dependency graphs; from there, the dependence depth and requisite number of logical channels can be derived.

### 4.1.2 Reducing the Dependence Depth

The previous section implied that cache coherence protocols require as many logical channels as the protocol dependence depth. It turns out that, by increasing the complexity of a protocol, we can systematically reduce its dependence depth to a minimum of two, as shown in Figure 4-5. This figure illustrates the transformation of a single protocol transaction of dependence depth four into four individual transactions, each of dependence depth two.

In this figure, the original dependence graph is of the form: $M_1 \Rightarrow M_2 \Rightarrow M_3 \Rightarrow M_4$. For generality, the final message ($M_4$) in this transaction is shown as absorbed by a node that is distinct from the *Initiator* of the transaction; however, both the *Initiator* and *Destination* could be the same. After reduction, this transaction is transformed into four independent transactions of depth two: $M_1 \Rightarrow P_2$, $M_2 \Rightarrow P_3$, $M_3 \Rightarrow P_4$, and $M_4$ (a one-way message transaction). To perform the reduction transformation, we use the *Initiator* node as a proxy to send messages $M_2$ – $M_4$. We do this by encapsulating these original messages in return messages $P_2$ – $P_4$. If this is all that we do, however, we have done nothing more than increasing the dependence depth by a factor of two. To finish the

[4]In fact, there will be no cycles as long as we guarantee that we always use channels in ascending order. We might, for instance, always send DATA messages on a particular set of logical channels that had higher available bandwidth. This approach shares many details with methods for deadlock-free routing; see, for instance, Dally [33, 32].

process of decoupling, we must break the network dependencies $\mathtt{P}_x \Rightarrow \mathtt{M}_x$. These dependencies will be broken if the initiator reserves sufficient space to hold messages $P_2$ – $P_4$ at the time in which it sends the first message ($M_1$); this permits the initiator node to guarantee that it can always sink the proxy messages. It is interesting to note that the dependence-depth-reducing transformation illustrated by Figure 4-5 corresponds roughly to a transformation of the underlying communication pattern from a "message-passing" style to a "shared-memory" style.

Note that Figure 4-5 takes the dependence-depth-reducing transformation to its logical extreme — reducing the dependence depth to two[5]. Intermediate transformations are equally possible, breaking the original dependence graph at any desired point to reduce the dependence depth to any value intermediate between its original value and its minimum of two.

**Adaptive Depth Reduction:** From the standpoint of performance, depth reduction increases the total time to complete a transaction by a non-trivial factor (if for no other reason than the fact that the number of network round trips is increased by nearly a factor of two). This suggests that a reduction transformation should be performed *adaptively*, *i.e.* dependencies should be broken at one of the proxy points only if there are insufficient network resources. In a system with two logical channels, all messages except for leaf messages could be transmitted on one channel. Then, if a node is unable to transmit an intermediate message $M_x$ because of queue congestion, it can construct a proxy message $P_x$ instead and send it off to the initiator on the second channel. This affords the performance advantages of the unreduced protocol when queue resources are available, while falling back onto the reduced protocol when necessary to avoid deadlock.

Interestingly enough, the Origin multiprocessor[67] from Silicon Graphics (SGI) makes use of adaptive depth reduction as described here to avoid deadlock with two logical channels and a DASH-like protocol. The base protocol has a dependency depth of three. When a three-hop dependency encounters network congestion, the Origin packages information and returns it to the requesting node. For instance, the `REQ⇒INV⇒(DATA,DATA)` dependency is handled by placing both REQ and INV messages on the request network and DATA messages on the response network. If an INV message cannot be queued because of a full queue, then a special message is returned to the source with sufficient information to continue the invalidation later, from the source[6].

### 4.1.3 Complexity of Deadlock Avoidance

As discussed in Section 4.1.1 the primary design complexity behind the non-reduced method of deadlock removal is the production of message-dependence graphs for all of the transactions. This type of analysis is slightly more complicated than the bare minimum required to design the protocol in first place, but not by much. Thus, assuming that a sufficient number of logical channels are available, deadlock can be avoided in this way. The presence of invalidations in cache-coherence

[5]Lenoski and Weber's book on multiprocessing mentions the reduction of a depth-three DASH protocol to a depth two protocol. See [72].

[6]In fact, this proxy message is a version of the coherence directory, describing multiple invalidation messages in a compact form.

protocols implies that non-reduced coherence protocols have a minimum dependence depth of three.

Why might an insufficient number of logical channels be available? As mentioned earlier, the most likely method for implementing logical channels is to make use of underlying virtual channels in the network. As with all hardware features, virtual channels have a design and performance cost that grows with the number of channels[26, 8]. Each virtual channel requires a separate input and/or output queue on each network port. Further, viewing the set of hardware virtual channels as a scarce resource, there are a number of ways that we might want to allocate them other than for deadlock avoidance of the cache-coherence protocol. For instance, virtual channels can be used to smooth out the performance of wormhole-routed mesh networks[31] and to achieve deadlock freedom in adaptive networks[32]. These particular uses of virtual channels interact multiplicatively with the deadlock avoidance methods that we have been discussing here. Thus, for instance, if a particular performance improvement scheme makes use of two virtual channels and deadlock avoidance requires three logical channels, then the underlying network requires *six* virtual channels: two for each logical channel. Thus, the number of hardware queues can grow quickly with the dependence depth of a protocol.

As shown in the previous section (Section 4.1.2), we can reduce the number of logical channels that we need for deadlock avoidance by reducing the dependence depth of the protocol. Abstractly, the dependence-depth-reducing transformation represented by Figure 4-5 is straightforward. Unfortunately, this process hides a number of disadvantages and complexities. For one thing, the complexity of the protocol increases over that of a non-reduced protocol because the number and type of messages and protocol operations increases. Although taking our viewpoint here and thinking of the protocol reduction process as one of using the initiator as a proxy makes a reduced protocol easier to think about, such a protocol is still more complex than the non-reduced version. In addition, as pointed out in the previous section, reduced protocols incur a non-trivial performance penalty, suggesting the need for adaptive reduction. This further complicates the protocol by introducing protocol arcs that choose between reduced and non-reduced forms. Given that verification of the coherence protocol can be an onerous task, this type of complexity increase must be approached with care. Finally, the implementation of a reduced protocol becomes more complex because initiator node must be able to allocate space to sink intermediate proxy messages and be able to unpackage them for transmission in a deferred fashion: when there are insufficient queue resources to send unpackaged messages, the initiator must be able to schedule later transmission of these messages. This type of deferred transmission is unlike other message transmissions, which are triggered by a direct processor action or message arrival events.

## 4.2 Message Passing and Deadlock

Since the communication patterns of a cache-coherence protocol are a restricted class of the more general message-passing paradigm, the problem of deadlock avoidance in message-passing systems must be at least as difficult as for shared-memory systems. In fact, message-passing introduces a new twist to the deadlock avoidance analysis: software. Message-passing systems export the lowest (or “link-level”) communication mechanisms directly to software; this is, in fact, one of the advantages of message passing touted in earlier chapters. Unfortunately, software can generate communication graphs of arbitrary dependence depth: for example, node $i$ can send a message to node $i+1$, which can send a message to node $i+2$, *etc.* This means that, regardless of the number of logical channels provided to software, there exists a communication pattern that is subject to deadlock. This means that avoidance of deadlock is out of the hands of the hardware designer.

### 4.2.1 Elimination of Deadlock by Design

Assuming that message-passing software is *trusted* and *bug-free*, then the same type of analysis that was discussed in Section 4.1 for cache-coherence protocols can be applied to a message-passing program (including the operating system) to guarantee that it is free from deadlock. Dependence-depth-reducing transformations *can* be applied to a given communication graph to reduce the number of logical channels required to avoid deadlock, although the performance penalty for for such transformations with software protocols is potentially much larger than for hardware coherence protocols: by increasing the number of messages that must be sent and received during a transaction we increase the amount of handler code that must be processed.

Unfortunately, in a system with user-level message passing, much of the software that is sending messages is neither trusted nor guaranteed to be bug-free. This means that deadlock avoidance is completely out of the hands of both the hardware designer and the writer of the operating system. The CM5, in fact, allowed users to deadlock the network, relying on periodic timer interrupts (at scheduler boundaries) to clear the network. This solution was "reasonable" because the CM5 was strictly gang-scheduled, permitting only one user to access to a given set of network resources at a time. Since we are interested in general sharing of the message network between different users, this solution is undesirable because it allows one faulty user task to negatively impact both the operating system and other users.

### 4.2.2 Atomicity is the Root of All Deadlock

In Section 4.1, we noted that the dependence arcs introduced by protocol processors are responsible for the existence of deadlock in shared-memory systems. It is the goal of this section to uncover similar aspects of message passing that couple the network input and output queues, permitting the occurrence of deadlock. Chapter 2 discussed the need for atomicity in any model of message passing[7]. In fact, the presence of atomicity is precisely what we are looking for: during periods of atomicity, message notification in the form of interrupts is disabled. Hence, message removal from the input queue is entirely under control of the currently active thread (either user or system level). This, in turn, *couples* the input and output queues during periods in which messages are being sent. In addition, user-level messaging may be performed by *incorrect* code that never frees messages from the network; this has the same detrimental effect on the network as actual deadlock. In contrast, when atomicity is disabled, message arrival interrupts can occur. This *decouples* input and output queues since messages can be removed from the input queue by interrupt handlers. The important conclusion here is that deadlock can occur only during those periods in which atomicity is enabled.

[7]Note that atomicity is considered both present and enabled during periods in which a task is polling for messages.

On closer examination, atomicity is present in shared-memory system as well. Earlier, we implied that a dependence graph such as REQ⇒DATA is an *atomic sequence*: the reception of a REQ message and subsequent transmission of a DATA message can not be split by the coherence protocol hardware. This action implies several disjoint operations that must occur together to avoid an incorrectly functioning protocol: dequeueing of the REQ, possible flushing of data from internal buffers, modification of the coherence directory, and transmission of the DATA. Since none of these operations are idempotent, they must be performed once and only once, with no intervening operations. In the language of Chapter 2, we can think of the hardware protocol processor as executing "message-passing handlers" in response to incoming messages; these handlers perform actions (including the sending of responses) without exiting their atomic sections.

Consequently, for both message-passing and shared-memory systems, *atomicity is the source of all deadlock*: atomicity introduces a dependency arc between reception and transmission that, when coupled with finite queue resources, can cause deadlock. Given this insight, we will focus on points of message transmission during atomicity when attacking deadlock in Section 4.3. In Alewife, such atomicity points occur in three different places: during hardware coherence-protocol processing, during message passing by the operating system, and during user-level messaging.

## 4.3 Exploiting Two-Case Delivery for Deadlock Removal

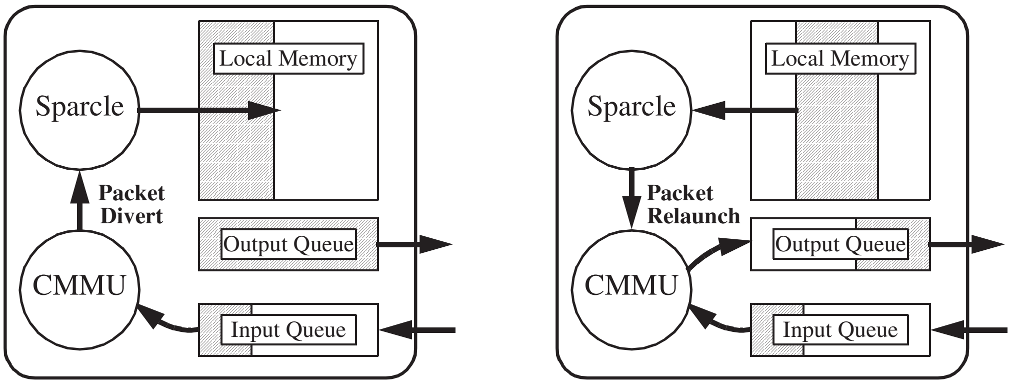
We will now discuss an alternative to the mathematical approach to avoiding deadlock, called *two-case delivery*. The original motivation for considering alternatives to logical channels was the fact that the Alewife network has a *single* logical channel; this constraint derives from the Caltech network router that we employed in Alewife. However, as illustrated in the last section, message passing introduces deadlock issues regardless of the number of logical network channels, since it is impossible to force a user to employ these channels correctly. Thus, we seek another solution.

### 4.3.1 Revisiting the Assumption of Finite Buffering

Discussion in Sections 4.1 and 4.2 was predicated on an assumption that we introduced only in passing, *i.e.* that queue resources are finite. Finite resources introduce cross-network dependencies, such as in Figures 4-2 and 4-3, allowing deadlocks to be introduced by message atomicity. The converse of this statement is as follows: if input or output queues are *infinite*, then none of the deadlocks mentioned in the previous sections would occur, *regardless of the number of logical channels in the network*[8]. This can be seen by the fact that the presence of infinite queues removes all dependencies through the network, preventing the occurrence of cycles in protocol wait-for-graphs. Of course, since hardware resources *are* finite, this statement does not seem to be particularly interesting. However, if we permit software buffering under extreme circumstances, then an *approximation* to infinite buffering can be achieved: when deadlock occurs, we buffer packets in the local memory of the processor, effectively extending the size of the hardware queues by many orders of magnitude[9]. The buffering mode introduces overhead that is not present during

[8] We are still assuming that the network has a deadlock-free routing algorithm.

[9] We will address concerns that local memory is not infinite later.




(A): Phase I: Divert Mode (B): Phase II: Relaunch Mode

Figure 4-6: Two Phases of Network Overflow Recovery

direct hardware delivery, thus we buffer only as needed (on *demand*). As a consequence, we call this a *two-case delivery process*: the first case is direct delivery, while the second case is buffered delivery. Since buffering is invoked to correct deficiencies in hardware resources, we call the buffering process *network overflow recovery*.

Figure 4-6 illustrates the two phases of network overflow recovery: *divert* and *relaunch*. Both of these phases are executed by interrupt handlers on the computational processor. The divert phase is entered by a high-priority *atomicity congestion interrupt* (see Section 4.3.2) and is used to empty the network *input queue*, placing messages into one or more queues in local memory. During divert mode, the output queue is completely full, refusing to accept outgoing packets. The process of emptying the network input queue will eventually lead to elimination of deadlock, since it inserts extra queue space in one of the branches of the deadlocked wait-for-graph; another way of looking at this is that the local output queue is clogged because a message at its head is destined for a node that cannot empty its input queue because of a clogged output queue — the act of emptying the local input queue helps to unclog this remote output queue, freeing the remote input queue, and thus indirectly unclogging the local output queue[10]. Figure 4-6 can be compared with Figure 4-1 to see how this methodology solves the deadlock problem. When network congestion has finally been eliminated, the relaunch phase is entered at software interrupt level. It is used to empty the local memory queues of pending messages, effectively restoring the system to its original state.

---

Two-case delivery is attractive because it eliminates deadlock for shared-memory *and* message-passing communication styles. The ability to exploit two-case delivery is an important system-wide benefit deriving from the close integration of message passing and shared memory in Alewife.

---

**Local Recovery is Good:** In Section 6.3.5, we will use the DeadSIM simulator to explore the nature of actual deadlocks that can occur in a system. One of the striking results that will appear is that the vast majority of deadlocks occur between two nodes in a system. This means that

[10]Of course, in high probability, that other node will enter the overflow recovery process as well.

a local deadlock recovery mechanism (such as provided by two-case delivery) provides a good approach for recovering from deadlock — most of the time, only a small number of nodes need to be interrupted to recover from deadlock.

**Divert Mode:** Although we will discuss the network overflow recovery process in greater detail later, we can give a few details now. At the beginning of the divert phase, the processor places the memory controller (CMMU) into a special mode in which the hardware ignores *all* incoming messages, forwarding them to the message input interface without processing. Note that this process of diversion has a subtle impact on the design of the protocol processor: hardware handling of cache-coherence packets must be *atomic* and *non-blocking*, *i.e.* the memory controller must refuse to start processing a cache coherence request before it has sufficient resources (such as output queue space) to complete the request. We will revisit this requirement in Section 4.3.7 when we discuss the some of the hardware complexities incurred by a machine that is involved in two-case delivery.

The majority of the divert handler is spent transferring packets from the network input queue to local memory. During this process, packets are sorted into several different memory queues based on packet type; among other things, this sorting process separates user-level packets from operating system packets so that different delivery guarantees can be placed on different packet types. Although the sorting process is important for correctness in a way that we will discuss later (*i.e.* for maintenance of user-level atomicity semantics), it can also be thought of as part of a *destination-based flow control* methodology: user packets can be buffered in the "virtual space" of the user *at the destination*, a method being developed by Ken Mackenzie [79]. To make this a complete flow control mechanism, it requires a method to suppress hyper-active users; this can be accomplished through the scheduler. Thus, rather than explicitly preventing users from sending more data than can be received, we let them send as much data as desired and exert back-pressure through the scheduler. This method avoids the explicit overhead incurred through token or credit-based flow control mechanisms. This is explored in greater depth elsewhere [78].

**Relaunch Mode:** The process of packet relaunch (phase two of recovery) can occur in two different ways: either by launching packets to the local memory controller or by passing a pointer to the queued version of each message to its appropriate message handler. The first of these was employed in the original Alewife design and is accomplished by sending buffered packets from the memory queues via the message-passing output interface (hence the term "relaunching"). Since each of the buffered packets has a routing header that points to the local node (packets are not altered by the queueing process), the message interface simply routes them through an internal loopback path, then processes them as if they had arrived from the network. This permits hardware processing of cache-coherence packets, allowing the buffered requests to be "transparently" processed by the memory controller. Message-passing packets can be processed in this way also; they are routed through the loopback path to the message input interface, where they cause an interrupt and invoke appropriate handlers[11]. Thus, the process of relaunch is a generic mechanism.

Unfortunately, relaunch as described above can introduce large overheads in the handling of

[11] An interesting subtlety is present here: since message interrupts caused by relaunched packets must have higher priority than interrupts that initiate message relaunch. More on this later.

small messages. This, coupled with the fact that fine-grained message-passing codes occasionally make heavy use of the network and can incur frequent use of the overflow handler, introduces a desire for an alternative to the relaunch process for message-passing packets. The alternative is relatively simple and is predicated on two simple observations: First, the message input interface is already "memory-mapped" in its behavior, even though it uses special, out-of-band load/store instructions. Second, the messages that are relaunched enter and immediately exit the CMMU; this is a waste of resources and causes the loopback path to be a contended resource. In fact, a key enabling feature that was missing from the original version of the CMMU (A-1000) was the ability to access the network input interface with normal load/store instructions. We explored this in depth in Section 2.4.4 when we discussed *transparency*. This ability was added to the A-1001 version of the CMMU. With this one feature, we can arrange for user-level message handlers to reference the contents of input messages through a reserved register that contains a "base pointer" to the head of the current message. Under normal circumstances, this register points to the head of the hardware input queue. During overflow recovery, the overflow handler can *transparently* switch between different delivery modes by simply altering this base pointer to point at queued messages in memory. Transparency is one of the key assumptions behind the full user-level atomicity mechanism discussed in Section 4.3.5. This straightforward mechanism vastly simplifies the complexity of the relaunch handler and vastly reduces the overhead incurred by messages that pass through the buffering mechanism.

### 4.3.2 Detecting the Need for Buffering: Atomicity Congestion Events

By design, two-case message delivery is a process of detection and recovery, not one of prevention. Since message delivery via the recovery process (second-case delivery) is more expensive than hardware delivery (first-case delivery), we need to be careful about triggering overflow recovery unless it is absolutely necessary. Unfortunately, deadlock is a global (multi-node) phenomena; as a consequence, true deadlock detection requires a distributed algorithm and out-of-band communication between nodes (since the normal communication network may be blocked). To avoid this, we would like to find a reasonable *approximation* to deadlock detection that employs only local information. As discussed in Section 4.2.2, we can focus all of our attention to periods of atomicity. Interestingly enough, Section 6.3 shows that protocol-level deadlocks have a tendency to involve a small number of nodes (two or three) rather than many nodes; this suggests that local detection and correction mechanisms may actually be better than global mechanisms.

One aspect of our deadlock detection mechanism is immediately clear: since the computational processor may be stalled during network congestion (with its pipeline blocked), the process of detecting deadlock must be accomplished by an independent hardware monitor. This hardware would be expected to flag a high-availability interrupt (in the sense of Chapter 3) whenever deadlock was suspected. In the following, the resulting interrupts will be called *atomicity congestion events* and will be used to invoke the first phase of network overflow recovery.

Alewife exhibits three classes of atomicity. Any monitor that we choose for deadlock detection must operate correctly for each of them. Briefly, the types of atomicity that we must consider are:

- **Hardware protocol processing**: The handling of cache-coherence requests that require responses. Hardware-induced problems are completely *invisible* to software.
- **System-level message passing**: Atomicity of system-level handlers that send messages. System-level code *can* be trusted to follow simple procedures to guarantee correctness.
- **User-level message passing**: Atomicity of user-level handlers that send messages. User-level code *cannot* be trusted to follow any deadlock avoidance protocol.

To handle the first of these, the deadlock-detection monitor must watch the lowest levels of hardware queueing for potential deadlock. Under normal execution, operation of this monitor (and execution of the corresponding overflow recovery handler) should be continuous and transparent to users of the network. In contrast, the monitoring of atomicity during system-level messaging must be selective. The reason for this is that system-level code needs to disable interrupts, *including the atomicity congestion interrupt*, at various times to ensure overall system-level correctness (*e.g.* during use of the local memory allocator). This, in turn, implies that high-priority system code (including network overflow recovery handlers) must be able to execute *regardless of the current state of the network*[12]. Furthermore, many system-level message handlers *do not* send messages (*e.g.* remote thread dispatch); such handlers are simpler if they leave interrupts disabled while freeing messages from the network. In essence, since we can trust system-code to follow simple rules of communication, we can allow it the freedom to ignore deadlock detection when it is safe to do so. In marked contrast to system-level code, user-code is not reliable; hence, deadlock detection must be enabled at all times during user-level execution. For example, user-level atomicity warrants special treatment because user-code can introduce deadlock at points of atomicity simply by refusing to free messages from the network.

The above discussion suggests two separate monitoring mechanisms: (1) a hardware mechanism that watches for queue-level deadlock, but which may be disabled when necessary by the operating system, and (2) a specialized mechanism that detects abuse of atomicity by the user. Together, these cover all requisite aspects of atomicity congestion detection. In the following sections, we will discuss both of these mechanisms. First, we will discuss a mechanism for detecting queue-level deadlock. Next, we will discuss how this can be coupled with system-level code to achieve deadlock-free execution. Finally, we will combine the lessons from these two in a single, user-level atomicity mechanism that achieves protection and deadlock freedom in the face of potentially incorrect user code.

[12]This requirement is a non-trivial one to satisfy. We will be discussing it in depth in Section 4.3.7.

### 4.3.3 Detecting Queue-Level Deadlock

Figure 4-7 shows the queue topology of the Alewife multiprocessor. As seen in this diagram, there are a number of queues in the system. At the lowest level are the network input and output queues. Alewife's integration of message passing and shared memory is evidenced by the fact that all network traffic feeds through these two queues. The abstract "protocol engine" used to discuss deadlock in Section 4.1 is, in reality, implemented as three separate protocol processors: the *Cache Management*, *Remote Transaction* and *Memory Management* protocol engines. As a consequence, the network input queue feeds four input paths: the IPI input queue (which forwards messages to the processor), and the three different protocol engines[13]. Furthermore, three separate output queues feed into the network output queue: the IPI Local Queue (which holds messages from the processor), the cache-protocol output queue (which accepts packets from both the Cache Management and Remote Transaction machines), and the memory-protocol output queue (which accepts packets from the Memory Management protocol engine). We will discuss these different entities in a bit more detail in Chapter 5. The two muxes along the input path facilitate looping of packets back from the IPI output descriptor and merging of these packets with incoming network packets; this mechanism is used during network overflow recovery and will be mentioned later. One final aspect of Figure 4-7 that we will discuss in Section 4.3.7 is the existence of block DMA engines; these handle the DMA to and from the network and must be guaranteed to make forward progress in order to avoid introducing additional opportunities for deadlock.

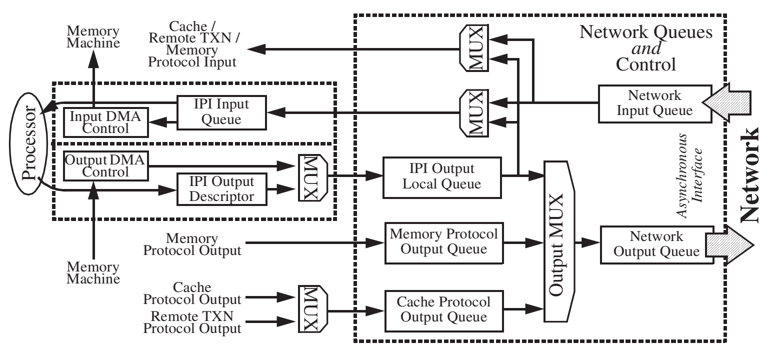



Figure 4-7: Queue topology for the Alewife CMMU.

Let us now return to the topic at hand, namely the detection of queue-level deadlock. Since the network output queue is downstream from all sources of data in the CMMU, we could conceivably monitor this queue in order to make estimates about the local state of the network. In all cases of *actual* deadlock, the output queue will be clogged indefinitely. Hence, one possible detection heuristic would be to monitor the flow of data through the output queue, declaring deadlock in instances for which the output queue is full and not moving. This heuristic would be guaranteed to detect all actual deadlocks. Unfortunately, it is a bit too simplistic because it is subject to many false detections: networks such as those present on Alewife have a wide variance of latency and throughput since applications tend to be bursty and the resulting contention effects cause intermittent (and widely varying) periods of blockage. Consequently, the output queue can be full and not moving under normal circumstances. This suggests that hysteresis is an important component of any detection mechanism[14]; we will illustrate the need for hysteresis directly when we explore the

[13] Figure 4-7 does not show the demultiplexor that feeds incoming protocol packets to appropriate hardware.

[14] As a case in point, the J-machine[87] employed an "interrupt on full output queue" heuristic that tended to be far too sensitive to network behavior[73].

*heuristic offset* (a measure of the effectiveness of a deadlock detection heuristic) in Section 6.3.5. We should note that, although the input queue is clogged during deadlock, it provides a less desirable detection heuristic than the output queue, since software message handlers introduce periods of queue blockage on the input that have more to do with software overhead than actual deadlock (potential or real)[15].

Hence, we suggest the following detection heuristic for detecting queue-level deadlock:

- Monitor the *network output queue.*
- Consider the local network *clogged* if the output queue is *full* and *not moving*.
- If the network is clogged for a preset period (the hysteresis interval), generate an atomicity congestion interrupt.

This relatively simple detection heuristic was implemented in Alewife. As we will discuss in Section 6.3.1, this heuristic is surprisingly robust, but is still subject to a bit of premature triggering: in practice, it tends to trigger in bursty situations for which the output queue is clogged, but the input queue is either empty or partially filled. Such a scenario is clearly not a deadlock (which involves full input *and* output queues). Thus, a simple modification to the definition of *clogged* could be:

- Consider the local network *clogged* if the output queue is *full* and *not moving*, and the input queue is *full* and *not moving*.

We explore the efficacy of these two detection heuristics (with respect to false detections) in Section 6.3.5, and will show that the latter is more effective at detecting deadlock.

### 4.3.4 Software Constraints Imposed by Queue-Level Deadlock Detection

Once we have a queue-level deadlock detection mechanism, we must ask the question of how it should be used. In this section, we will begin to answer this question by exploring the use of the detection mechanism for hardware-protocol processing (*i.e.* cache coherence) and operating-systems-level message passing. We will assume that these two users of the network are "correct" in the sense that both memory hardware and operating-systems code are extensively tested and under the control of sophisticated "implementers"[16]. It is our goal to understand what restrictions need to respected by these uses of the network interface.

Recall that we already broached this topic in Section 2.5.2, where we discussed the interactions between shared memory and message passing in interrupt handlers and other atomic sections. That section explored interactions at the level of the communication models and thus described interactions that were visible to the user. Such interactions (*e.g.* no shared memory in message handlers), were discussed in general terms, and attributed to the fact that Alewife has only a single logical

[15] Alternatively, the hallmark of deadlock is the inability to process messages due to a lack of output queue resources; this is distinct from network congestion caused by long handler occupancies.

[16] The extent to which this is true is a recurring matter of debate as systems get more complicated, but we will assume it as an axiom for the time being.

network channel. However, that section was written from the point of view of the operating system or other users that are not insulated from the raw network hardware. As we pointed out in Section 4.2.2, all deadlock centers around atomic sections. Hence, one of the results that we will achieve in this section is to codify the constraints on operating-systems by queue-level deadlock detection, couple them with the model-level constraints of Section 2.5.2, and ultimately (in Section 4.3.5) produce a user-level atomicity mechanism that can remove these constraints through virtualization.

In this context, it is perhaps interesting that we have not yet directly mentioned atomicity, even though we stressed the importance of atomicity earlier. In fact, atomicity comes into play with the enabling and disabling of atomicity congestion interrupts. If atomicity congestion interrupts are disabled, then code *must not:*

- Touch or access shared memory.
- Send messages.

The first constraint arises because the shared-memory system is always active, regardless of the state of the interrupt enable mask; thus, shared-memory requests might not make forward progress if atomicity congestion detection is disabled[17]. However, the attentive reader might ask why the second of these constraints is necessary if message arrival interrupts are enabled (thereby guaranteeing that the messaging software *is not* in an atomic section). This interaction between shared memory and message passing is a consequence of the decision to rely on a single logical network channel to carry both styles of communication. It may be understood by examining Figure 4-7: if message at the head of the network input queue is a shared-memory request, then it can provide a deadlock-causing dependency, preventing messages behind it in the input queue from being received by the processor. The use of a single logical channel effectively couples the shared-memory and message-passing atomic sections.

A complementary coupling between message passing and shared memory was introduced in Section 2.5.2 and is also caused by the presence of a single logical network channel: shared-memory accesses cannot occur during periods in which message arrival interrupts are disabled. We can appeal to Figure 4-7, once again, to note that the reason for this restriction is that a message (or group of messages) may stretch through the IPI input queue into the network input queue, preventing the arrival of shared-memory data. If message arrival interrupts are disabled, then there is no guarantee this situation will resolve itself. This means that *no accesses to shared memory are allowed during the atomic sections of message handlers*. Note that this constraint is, in fact, a "law of programming" that is designed to avoid the refused service deadlock (Section 3.1). When we revisit the assumptions of the Alewife design in Chapter 7, one of our observations will be that the presence of a single logical network channel provides an unfortunate coupling between shared memory and message passing that restricts integrated use of these mechanisms.

Exploring these restrictions a bit further, our prohibition toward access of shared memory when *either* atomicity congestion interrupts *or* message arrival interrupts are disabled extends even to data that a careful operating-systems designer has guaranteed will not be accessed by remote

[17]Understanding why this is true was, in fact, one of the points of Section 4.1.

users[18]. The reasons for this are a bit subtle, but are basically a constraint imposed by reasonable implementation. We examine this in a bit more depth in Section 4.3.7.

### 4.3.5 User-Level Atomicity and Two-Case Delivery

Now that we have examined the constraints on system-level atomicity, we can discuss features of the complete user-level atomicity mechanism. Section 2.4.5 covered many of the salient points, but we like to examine them in the complete context of two-case delivery. The goal of the user-level atomicity mechanism is to provide direct hardware access to the user, but to invoke two-case delivery as a special case whenever the user violates any of the atomicity constraints that we have mentioned above. To summarize the high-level constraints of Section 2.4.5 and the hardware constraints of Section 4.3.4, we must handle the following special circumstances:

1. Whenever the user refuses to free messages from the network we must invoke second-case delivery. Failure to do so could cause deadlock for any of the reasons mentioned in earlier sections of this chapter. Further, we would like to invoke buffering when any message been stalled at the head of the network input queue for some period of time, event if the user will eventually free the message. This is simply to decongest the network.
2. Whenever the user attempts to access shared memory in an atomic section and a message is pending at the head of the input queue we must invoke second-case delivery. We must invoke second-case delivery in order to free the network for shared-memory arrival.
3. Should a user-level handler be invoked with priority inversion, then we must be able to revert to the higher priority code when the user exits atomicity. Note that we also enforce a maximum time limit on priority inverted handler code before triggering second-case delivery, thereby minimizing the interference between user-code and the higher-priority code. Included in our classification of “priority inversions” is the case in which the interrupted context has invoked the thrash-lock mechanism to avoid thrashing (Section 3.2.8). In that case, we must consider the shared-memory protocol lock to be of higher priority than the user-level handler.
4. Whenever the queue-level deadlock detection mechanism of Section 4.3.3 triggers, we must invoke second-case delivery.

Note that the last of these mechanisms is independent of the other three and must be functioning at all times since queue-level deadlock may occur even under “correct” operation. This is why we have provided the queue-level deadlock detection mechanism as an independent mechanism (which is not considered a part of user-level atomicity).

Section 2.4.5 discussed the the user-level atomicity mechanism and provided a simple state diagram of the “common-case” states of this mechanism (Figure 2-13). Adding two-case delivery to handle the corner cases produces Figure 4-8. In this diagram, user-code states are unshaded

[18] Consider the user of shared-memory addresses for operating system stacks.

and annotated with a four-bit `uatomctrl` value, while OS states are shaded. This is divided into three distinct sets of states: hardware atomicity (left), queue emulation (middle), and software atomicity (right). The first category is comprised of the four primary states from Figure 2-13). These are labeled as "First-Case Delivery". In this regime, state transitions are under direct control of the user. Further, the atomicity extension mechanism is used to trigger a return to the operating system to handle a return to high-priority code during priority inversion. The middle states (shaded and circled) represent operating-system's code that emulates network queues during second-case delivery, while the right three states are active when the user is consuming buffered messages. Collectively, these seven states are labeled "Second-Case Delivery". Note that the freedom to move back and forth between the left and right halves of this diagram stems from the network access transparency that we made a point of discussing in Section 2.4.4.

This diagram is best understood by starting with the user-level states. For simplicity, all of the error arcs have been omitted. The background `user_code` state is executed when there are no pending messages; hence it appears only on the left half of the diagram. The other two user states, `handler_atomic_section` and `atomic section` are mirrored on both sides of the diagram. The `handler_atomic_section` state is invoked when a message handler first starts, and serves enforce the constraint that a user must free at least one message before exiting the atomic section. Message disposal transfers into the appropriate `atomic_section` state (possibly crossing from right to left if there are no further buffered messages). The key distinction between these two states is the value of the `dispose_pending` bit of the `uatomctrl` register. In addition to handling the second half of a handler atomic section, the `atomic_section` states are entered during polling and simple background atomicity requests.

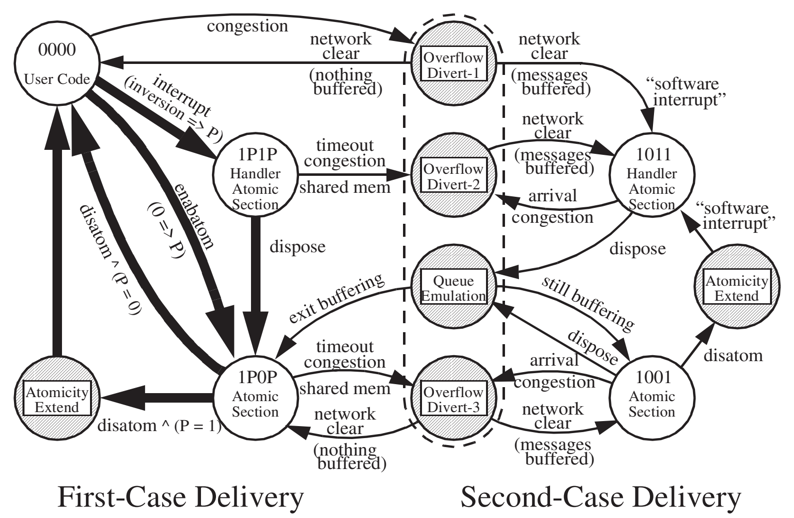



Figure 4-8: The interaction of the user-level atomicity mechanism with two-case buffering. The four bit value is the `uatomctrl` register; these bits are *atomicity_assert*, *timer_force*, *dispose_pending*, and *atomicity_extend* in that order.

Revocation of the interrupt disable mechanism represents a transition from the left to the right half of the diagram. Revocation starts by entry into one of the three `overflow_divert` states; these three perform identical buffering functions with the exception of their exit arcs. There are three distinct types of revocation events that can trigger entry into buffering: *network congestion* (*i.e.* queue-level deadlock detection), *atomicity timeout*, and *shared-memory access*. Exit from these buffering states occurs after buffering one or more messages as long as the network is deemed "sufficiently clear".

During second-case delivery, the user is not directly impacting the network. Hence, the atomicity timeout is never invoked. Further, for similar reasons, shared-memory access is not restricted. Thus, reentry into buffering from the right of the diagram is triggered either by network congestion (always a potential problem) or by the arrival of user-level messages. The arrival of new user-level messages invokes buffering for the simple reason that the user is already occupied processing buffered message. Hence, we simply add new messages to the end of the software queue at this point. Transitions from the right to the left of the diagram occur when the user has processed all pending buffered messages. Should the user exit second-case delivery by consuming all buffered messages, then new messages will once again be consumed directly.

Note that *transparency* during second-case delivery means three things here: First, that user code can read from the message input array without caring whether or not it is buffered in memory; second, that disposal operations (`ipicst` instructions) cause traps into the buffering emulation code; and third, that atomicity extend operations are always invoked at the end of an atomic section in order to emulate a message arrival interrupt (during buffering). Further, as mentioned above, message arrival interrupts are redirected into buffering code rather than invoking the featherweight threading code.

This is the complete user-level atomicity mechanism. It is simple in its high-level details, if perhaps requiring a bit of care in its implementation.

### 4.3.6 Virtual Queueing and Second-Case Delivery

In the previous section, we discussed the user-level atomicity mechanism from a relatively high level. In this section, we would like to explore the process of buffering and second-case delivery in a bit more detail. Given the presence of software in the buffering process, we are free to demultiplex the single incoming message stream into a number of different software queues. This particular operation, which we call *virtual queueing*, was mentioned in Section 2.4.7 in the context of providing UDM-style messaging a multi-user environment. Message passing in a multi-user environment occurs simply by buffering messages that arrive for descheduled users (*i.e.* mismatched messages) into reserved buffers, one for each user. When one of these formerly idle users is rescheduled, we simply begin their execution in the buffered half of the atomicity diagram. It is up to the scheduler to guarantee that the number of such "mismatched" messages is relatively small, and hence the amount of time that applications spend in buffering mode at the beginning of a schedule quantum is relatively small. Ken Mackenzie's PhD thesis explores this aspect of buffering in more detail [79] as does [78].

However, virtual queueing is important even in a single user machine such as Alewife. Let us return to the question of what happens when the queue-level deadlock detection mechanism is invoked. In Section 4.3.1, we introduced two distinct phases of second-case delivery, namely *divert* and *relaunch*. When the hardware queues are deadlocked, much of the system is inoperative. Thus, the *divert* handler is a very high-priority handler (in fact, it is the highest-priority handler). Once it is invoked, the first thing that it must do is preserve any state that may be required to continue later. In particular, since the operating system accesses the network input queue directly (especially in the A-1000 CMMU, where transparency was not available), the divert handler must keep track of the message at the head of the hardware input queue so that it can restore this message before returning from interrupt.

Next, the divert handler must unclog the network, sorting packets as it does so. Alewife supports several distinct types of message traffic: cache-coherence protocol messages, system-level messages, and user-level messages. Furthermore, given the presence of the LimitLESS coherence protocol, cache-coherence packets may be seen as divided into two categories: those that have not been touched by hardware (unprocessed) and those that have been queued for software handling (processed); during buffering, these latter two types of messages may be mixed in the same input queue[19]. Each of these four types of messages require distinct processing and hence must be queued separately.

The reasons for virtual queueing are straightforward: cache-coherence messages that have already been processed must be handled by software and (for protocol correctness), must be handled as a group before any unprocessed coherence messages. In contrast, unprocessed messages must be *relaunched*, *i.e.* passed back to the hardware for processing (this results in more efficient processing than any software handler could accomplish). Further, user and system messages must be buffered in different queues because they have different memory requirements: system-level messages must be buffered in system space and can be "guaranteed" not to overflow system resources, whereas user-level messages must be buffered in user-space and are not subject to any constraints on memory usage[20].

When the network is sufficiently unclogged (as noted by the draining of the output queue), the divert handler restores the initial message by launching through the loopback queue, schedules a software interrupt, then returns to the interrupted code. The *relaunch* process is triggered later at software-interrupt level. The various ordering constraints (such as handling of processed messages before unprocessed ones) is handled by providing prioritized software interrupts. Note that the fact that software interrupts are all lower than other interrupts (such as message arrival interrupts) means that the process of relaunching messages through the loopback path (the only option in the original Alewife machine) may be gracefully interspersed with interrupts to handle these messages.

### 4.3.7 What are the Hidden Costs of Two-Case Delivery?

Having spent a number of pages extolling the virtues of two-case delivery (both from a hardware complexity standpoint and from the standpoint of guaranteeing freedom from deadlock for user-level message passing), we now turn to some of its hidden complexities. Certainly, two-case delivery was the right choice for the Alewife machine. At the time that Alewife was initiated, networks with a single logical network channel were the only ones available; in point of fact, the EMRC was the only viable choice. However, a complete implementation of Alewife yielded a number of complexities that can be traced directly back to the requirement of two-case delivery. Some of these constraints, as we will see, are intrinsic to two-case delivery, while others are specific

[19] There is a bit in the header that distinguishes these types of messages.

[20] In fact, the potentially unbounded buffering requirements in a multi-user multiprocessor system can be handled by buffering user messages in virtual memory backed by a second logical network for paging to disk if necessary; the scheduler can then be used to squelch processes that are being overzealous in their sending of messages in order to minimize that actual amount of paging that occurs. This notion of *virtual buffering* is examined in detail by Ken Mackenzie in [79].

to interactions with other Alewife features. There are four different issues that we would like to discuss here:

- Software complexity. Two-case delivery introduces a layer of software which is below even the cache-coherence protocol.
- Hardware-level atomicity. Every action undertaken by the hardware must be guaranteed to complete atomically, in a bounded amount of time.
- Flexible network topology, *i.e.* the ability to "divert" messages normally processed by hardware into the software input queues.
- Local unshared memory space. Two-case delivery introduces a need for access to memory that may be used under all circumstances.
- Guarantees that network DMA can always complete, regardless of the state of the network.

The first item is a general comment about the complexity of divert and relaunch handlers, while the second three items are hardware requirements that stem directly from two-case delivery.

**Software Complexity:** First and foremost, the complexity of the divert and relaunch handlers in Alewife was much greater than anticipated. The network overflow process is buried at a level of abstraction below most of the layers of the operating system and, for that matter, below the cache coherence protocol. As a result, problems with this code were non-deterministic and hard to debug. In addition, the presence of two-case delivery added a distinction within the operating system between code that was protected against network deadlock (when the queue-deadlock detection interrupt was enabled), and code that was not protected. However, much of the complexity in the actual divert and relaunch handlers derived from the fact that all messages needed to be relaunched in the original Alewife machine and hence, queues had to be cleared out in a way that avoided deadlock between the relaunch process and users of the output descriptor array. Adding the ability to transparently extract messages from the network and buffer them in memory without relaunching them would greatly simplify this code. Further, elevating issues of queue-level deadlock to software ensures that the deadlock process *is* debuggable — the occurrence of deadlock is visible (and non-fatal) with two-case delivery; contrast this with a hardware-level deadlock removal scheme, for which bugs in the deadlock avoidance scheme results in fatal network deadlocks. Thus, while two-case delivery adds another layer of complex software, it has a number of advantages as well.

**Hardware-Level Atomicity:** On requirement that is introduced by the existence of two-case delivery is the need for atomicity at all levels of the hardware. In particular, hardware handling of cache-coherence packets must be *atomic* and *non-blocking*, *i.e.* the memory controller must refuse to start processing a cache coherence request before it has sufficient resources (such as output queue space) to complete the request. There are two reasons for this requirement in the context of two-case delivery: First, since network congestion is relieved through the execution of an interrupt handler, the memory controller must be free to handle memory requests from the local processor (such as instruction cache misses) at all times; thus, it cannot be blocked awaiting the completion of a half-processed coherence protocol request. Second, since we may divert and relaunch packets from the input queue, we must make sure that coherence actions are not executed multiple times (most coherence protocol processing is not idempotent). Thus, the memory controller must never stop in the middle of processing a request.

In fact, as it turns out, enforcing atomicity of operations makes for much better scheduling of the memory controller than would occur otherwise. This is because the amount of time required to process a request becomes bounded, and the scheduler can maximize use of its cycles (or eliminate unnecessary dead-time blocking on resources). We will return to this again in Chapter 5 when we discuss scheduling of the memory controller.

**Flexible Network Topology:** In addition to the atomicity of operations, a machine that relies on two-case delivery must also support a flexible network topology, permitting the redirection of packets normally destined for hardware processing into the message input queues. In Alewife, this consists of special network control bits in one of the control registers that puts the internal network into "divert mode". One of the impacts of hardware atomicity on this mechanism is that second-case delivery software can place the network into divert mode at any time without risking partial processing of coherence protocol packets.

**Local Unshared-Memory Space:** As mentioned previously, the existence of two-case delivery introduces a need for two classes of memory: shared and unshared. The reason for this distinction is that shared memory is not guaranteed to be make forward progress when the network is congested (or deadlocked). Since the *divert* handler must be able to execute in order to remove network deadlock, some other memory space must be available, both for access to instructions and for temporary data/software queues. That was the local unshared-memory space. What we would like to explore here is that fact that the distinction between shared and unshared memory must be visible in the physical addresses, not merely in the coherence directory. The reasons for this are a bit subtle, but may be described as a constraint imposed by implementation. By tagging accesses with a "local address space" tag, various levels of the hardware can provide special streamlined services.

There are three different areas in which this this tag comes in handy: the memory controller itself, the local request queue to memory, and the transaction buffer. The first of these stems directly from the atomicity constraint mentioned above: the need for atomicity of memory operations is best implemented by assuming that *every* shared-memory request processed by the local memory system requires a minimum amount of queue resources (possibly for invalidation traffic, *etc.*). Chapter 5 will discuss this in more detail, but the end result is a memory system that will refuse to process shared-memory requests during periods of network congestion, even if the satisfaction of these requests ultimately does not require network resources. These constraints are enforced by scheduler. By providing a "local memory" tag, the scheduler can avoid checking for queue resources, knowing that local memory operations are guaranteed not to require the network.

The second area in which the unshared-memory tag is useful is in the request queue to local memory. The local memory system includes a special request queue between the processor and local memory. This queue is of depth two, so it is possible for up to two prefetch requests for global memory to be pending at any one time. As a result, if the network becomes deadlocked, it is possible for these prefetches to become clogged, preventing the success of subsequent requests to local memory. This would defeat the whole purpose of having a special memory space. Thus, during network deadlock, all requests to the global memory system are aborted: non-binding prefetches may be aborted in general, and non-prefetch requests are simply NAKed. This has the effect of terminating any outstanding transactions to local memory, hence clearing out the request queue (and freeing up the corresponding transaction buffers). Hence, the unshared-memory tag helps to guarantee the success of local requests.

The third area in which the unshared-memory tag is useful is in the management of the transaction buffer. Architecturally, the transaction buffer lies between the processor and the memory system. In order to guarantee that accesses to local memory can complete at all times, the transaction buffer must be managed by reserving buffers for local accesses. The presence or absence of local memory tags in the transaction buffer is used directly by the transaction buffer allocation mechanism to ensure that a minimum number of buffers are always available for local memory accesses. We mentioned this in the context of the faultable flush queue in Section 3.4.2. The set of transaction buffer constraints for local accesses are discussed in detail in Section 5.2.3.

Thus, the separation of memory addresses into shared- and unshared-memory ranges is exploited by a number of levels of the hardware.

On subtle aspect of providing a local memory system that is always available is the need to demote cached accesses to uncached accesses on demand. The reason for this is that the first-level cache is a write-back cache. This means that, at any one time, the cache may contain many dirty cache lines, including lines for the shared-memory space. Under normal circumstances, dirty lines are simply flushed back to the memory system or network on demand in order to free up space in the cache. Unfortunately, this means that portions of the cache may become unavailable for reuse during periods of network congestion. This situation, in turn, would require all instructions and data for the divert handler to be uncached in order to ensure that it can recover from network deadlock. This would require distinguishing the divert handler from all other code in the system (so that its instruction fetches could be issued as uncached), not to mention causing this handler to run very slowly.

However, as we have discussed in previous paragraphs, forward progress of accesses to the local memory system are guaranteed by the transaction buffer and other portions of the memory system. Thus, an alternative to running the divert handler instructions uncached is simply to demote cached accesses to uncached accesses automatically when replacements cannot occur for one reason or another. “Demote” here means that the processor accesses data directly in the transaction buffer rather than the first-level cache. Thus, this feature allows accesses to “go around” the first level cache in those instances for which it is necessary, but permits the instructions and data to be cached in other circumstances.

**Guarantees that DMA Can Complete:** The final impact of two-case delivery on implementation results from message DMA. As discussed in Chapter 2, DMA is an integral part of the user-level message-passing interface of Alewife. This is tremendously useful from the standpoint of functionality, but has consequences in the presence of two-case delivery. First, since the basic tenet of two-case delivery is to recover from deadlocked situations by emptying the network input queue, it is extremely important that DMA operations that have been started on input packets be able to complete at all costs. If it were possible for a congested network to prevent input DMA from completing, then there would be scenarios in which the divert handler would be unable to perform its primary function of emptying the network. In addition, since relaunching of packets through the loopback queue is one of our methods of recovering from deadlock (and could, alternatively, be requested by a user), it must be possible for both input and output DMA to be operational simultaneously. Thus, there must be two independent DMA engines, one for input messages and another for output messages.

Further, when two-case delivery is coupled with DMA coherence (see Section 2.5.1), the requirement that DMA engines make forward progress becomes complicated by the coherence aspects: data being fetched or invalidated from the local cache must have a guaranteed path back to the memory system, regardless of the state of the network. Thus, the DMA engines perform a type of cache invalidation that must be guaranteed to succeed. As we will discuss in Chapter 5, this is accomplished by passing data from the cache through the transaction buffer, invoking the same constraints and buffer reservation mechanisms that guarantee the forward progress of local memory[21]. This methodology ensures a path to local memory simply by ensuring a minimum number of transaction buffers devoted to local memory access.

Unfortunately, this solution is not sufficient if data has been DMA'ed from the shared address space. Since DMA in Alewife is only guaranteed to be locally coherent, the primary issue here is ensuring that the local coherence directories are kept coherent with the local cache (so as not to confuse the normal memory access mechanisms). Practically speaking, this means that the transfer of write permission between the cache and local memory must be handled properly by DMA. As discussed previously, transaction buffer constraints and memory scheduling are applied based on the type of memory line that is being handled (shared or unshared), but in a fashion that is otherwise oblivious to the actual sharing state of this memory line. Hence the difficulty: although we would like to guarantee that invalidated data can pass through the transaction buffer and memory scheduler under all circumstances we really have no way to to this without vastly complicating the memory scheduler and related mechanisms[22].

The solution that is employed in Alewife is to *separate* the data and coherence information from a pending DMA invalidation when the transaction buffer contains too much global data: we take the data from the cache, mark it as "unshared", and place it in the transaction buffer; at the same time, we leave the cache-coherent "write permission" behind in the cache as a special *invalid write permission* (IWP) state. This IWP state behaves exactly like a read-write copy of a memory-line, but without the actual data. The transaction buffer and memory scheduler can treat the data as strictly local, since it no longer contains any coherence information; in fact, the cache-coherence directory does not need to be accessed. As far as the coherence protocol is concerned, the cache still owns a dirty copy the memory-line; hence it can be flushed or replaced from the cache in the normal way (with special provisions for the fact that it does not actually contain data). Should the processor attempt to access a cache-line that is in the IWP state, it blocks while the IWP is flushed back to memory and a new copy of the data fetched.

[21] Passing data through the transaction buffer is important: the processor will be able to find it again should it attempt to fetch it again before it has returned to memory.

[22] If the network is not congested and the transaction buffer is unclogged, then transfer of write permission from the cache to memory can happen in exactly the same way that it normally does for other types of cache replacement.

## 4.4 Postscript: Two-Case Delivery as a Universal Solution

This chapter has explored some of the issues involved in preventing cache-coherence protocols and message-passing applications from deadlocking in the network. As we saw, cache coherence protocols *can* be designed to be deadlock free, one of the principal requirements being the presence of two or more logical network channels. In some cases, the cost of multiple virtual channels can be high, leaving a designer wondering if there are alternatives. User-level message-passing applications, while in theory amenable to the same sort of analysis as cache-coherence protocols, cannot be guaranteed to be deadlock free, simply because the user cannot be trusted to produce bug-free code.

In this chapter, two-case delivery was introduced as an alternative to deadlock-removal via multiple virtual channels for both cache-coherence protocols and message-passing applications. Although it has a number of complexities, two-case delivery also has a number of advantages. Utilizing buffering as a mechanism for eliminating dependencies through the network, two-case delivery is an ideal mechanism to include in machines that integrate message passing and shared memory; it subsumes other types of deadlock avoidance mechanisms by absorbing deadlock arcs into the software queues. Further, two-case delivery provides a natural framework in which to implement a user-level message-passing system such as UDM in a multi-user environment. Finally, two-case delivery embodies the integrated-systems philosophy of handling common cases directly in hardware (first-case delivery), while relegating uncommon or congested situations to software.

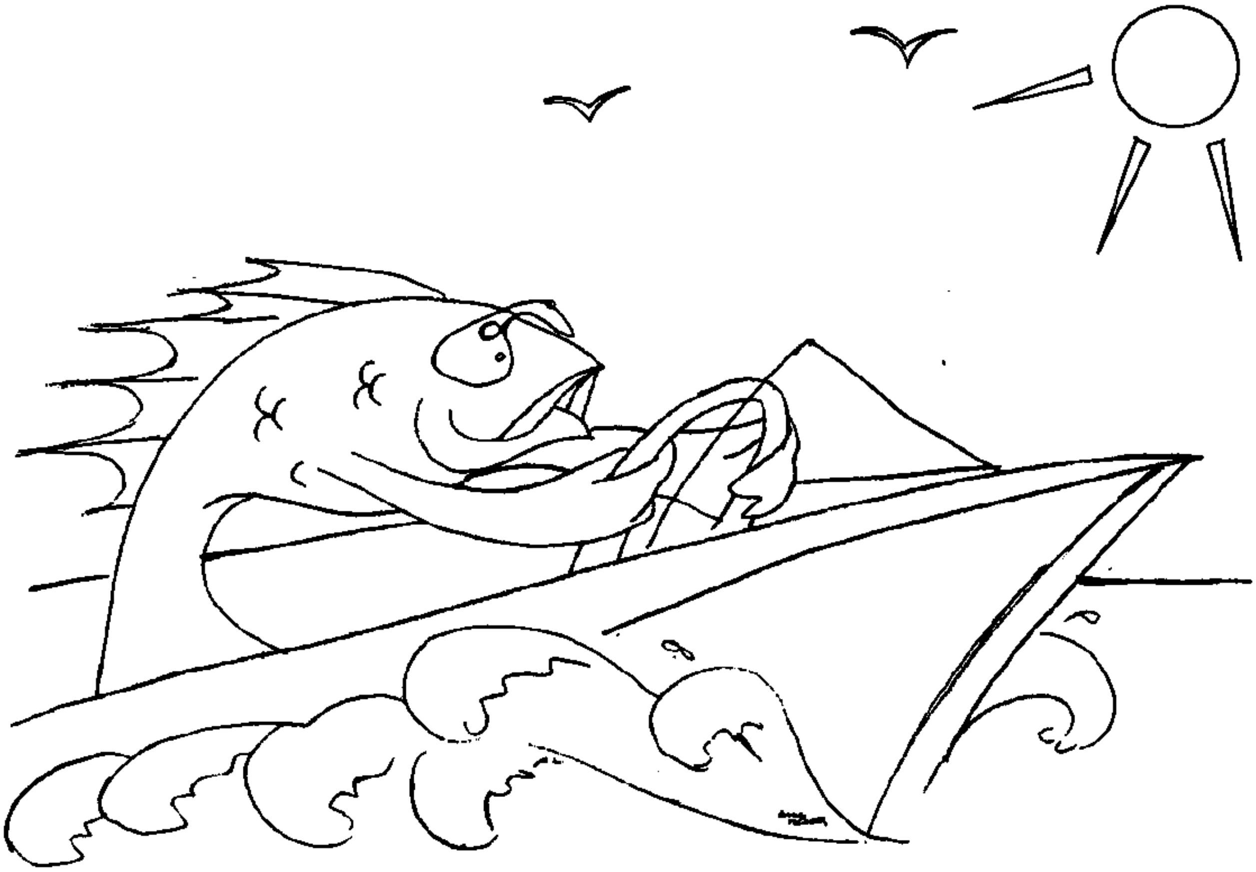
---

**Collaborators:** Ken Mackenzie was an important contributor to many of the advanced aspects of two-case delivery. In particular, the concept of virtual buffering was developed in great detail by Ken in his thesis [79]. Further, Ken was instrumental in transforming two-case delivery from a deadlock-removal mechanism into a universal delivery paradigm. His thesis illustrates the use of two-case delivery to handle everything from page-faults in message handlers to incorrect message delivery in a multiuser multiprocessor. Donald Yeung served as a complex user of many of the exceptional arcs of Figure 4-8, helping to debug the interactions between user-level atomicity and network overflow recovery.

---

# Part 2:

# Consequences



In the second part of this thesis we will explore the implementation and performance of an integrated multiprocessor. In particular, we describe how the three challenges of integration are embodied in the complete system architecture of the Alewife machine. Hopefully this part of the thesis can provide a reality check on the discussions of the past three chapters. Alewife is *not* just a paper architecture — real machines exist in the lab and are actively used for research.

Many of the mechanisms present in the Alewife machine (both hardware and software) are limited if viewed in isolation. Thus, the high-level, hierarchical viewpoint of mechanisms first illustrated by Figure 1-3 (page 27) must be remembered at all times. It is the communication models (presented to higher levels of software) which ultimately matter. So, even as we present the decomposition of Alewife into mechanisms, state machines, and data-path diagrams we will attempt to keep the resulting mechanisms in context – the context of our desired communication models. This being said, however, the implementation details *are* important in that they impact performance in a non-trivial way, and serve to illustrate that a machine such as Alewife *can* support fast mechanisms.

**Why bother to implement?** Before continuing, we should address an important issue, namely the omnipresent question: "Why implement at all?" This question is especially important today, when many universities have succumb to the notion that implementation in an academic setting is no longer practical. Alewife operated under severe constraints of limited manpower and finite resources[†]. Despite these limitations, members of the Alewife team (and in particular this author) believe quite strongly in the value of a working implementation. These convictions led to the working Alewife prototype described in this thesis.

To expand on this point for a moment, there were a number of reasons for our belief in the importance of implementation. First, the speed and realism of real hardware permits a much greater variety of application studies and problem sizes than would be available from a simulator. This permits a proposed set of architectural features to be explored in a more realistic context than would otherwise be possible. In a real implementation, the level of detail is absolute; in a simulator, there is the omnipresent tradeoff between speed and accuracy. Although simulation technology has improved much in the last decade, it is still not entirely suited to the execution of large problems running on top of a complete operating system with all of the real effects present in a real machine. Execution-driven simulators, such as SimOS [95], Proteus [15], and others have made the greatest strides in that direction. These simulators are good at reducing the number of simulator instructions executed per simulated instruction (hence increasing simulation speed), but have trouble when confronted by effects such as fine-grained interrupts, interaction of local cache misses with global cache misses, etc. The very nature of some of the proposed Alewife features, such as LimitLESS cache coherence and the integration of fine-grained message passing and shared memory, requires a level of faithfulness in simulation that would have defeated many of the advantages of execution-driven simulation methodologies. Lest this author should give the wrong impression, simulation *is* an important tool throughout the design process, serving to choose among alternatives in the early stages of design and helping to validate during the later stages.

[†] Although similar constraints are perhaps observed in industry, they are often not as severe — the rise of the "Intel paradigm" (hundreds of engineers working on a single chip) would tend to suggest that they are by no means necessary.

Second, the act of implementation is a form of discovery in and of itself — many problems and complications that are unappreciated during the initial stages of a design become painfully obvious during the implementation process. This is a fact that was demonstrated several times during the course of the Alewife implementation and is one that is often ignored by those claiming that careful simulation studies are sufficient. Many of the details discussed in Chapter 3 on Service Interleaving, for instance, were apprehended only during the course of implementing the Alewife CMMU. A designer with a well-tuned intuition might be able to divine some of these problems and solutions, but would be hard-pressed to stumble upon the final result without embarking on a complete implementation effort. In this vein, there are those who propose that academicians should save the time and expense of testing and fabrication by treating the implementation process as an exercise which does not result in actual hardware. However, as one who partook in the implementation of Alewife, I would be the first to state that this point of view seriously underestimates the motivational factors (desire for a final product) involved in implementation.

The final and perhaps most important reason for embarking upon an implementation project is the fact that it is only with the existence of a reasonable implementation that ideas in computer architecture graduate to viable solutions. Too many paper designs propose hardware mechanisms that would be impossible or unreasonably complicated to implement. Features such as fully-associative caches, huge cross-bars, and other mechanisms abound in untested architectures. In contrast, the Alewife multiprocessor exists and is in active use as a research vehicle. The existence of a working architecture provides validation for the ideas described in the first few chapters. Whether or not these ideas apply in future situations, they have become viable architectural solutions.

With the previous discussion as motivation, we now embark on a description of the implementation and performance of the Alewife multiprocessor. The first chapter, Chapter 5, explores the implementation of the Alewife machine, the most important component of which is the Communications and Memory Management Unit (CMMU). The next chapter, Chapter 6, shows the performance of the Alewife machine. Finally, Chapter 7 looks back at some of the lessons of the Alewife project. Figure 1-3 in Chapter 1 (page 27) served as a road-map for the Design portion of this thesis; it will do so again as we begin the process of exploring the Alewife implementation.

# Chapter 5

# The Hardware Architecture of Alewife

In this chapter, we will unravel some of the details of the Alewife implementation. As a starting point, we reproduce the Alewife node-diagram in Figure 5-1. This figure gives a high-level picture of the Alewife implementation. Each node of Alewife consists of two major components, the Sparcle processor and the communications and memory-management unit (A-1000 CMMU). Other elements of the system are subordinate to these two components. The Sparcle processor contains the basic execution pipeline and is responsible for all instruction-fetch and control-flow, as well as execution of most instructions (with the notable exception of floating-point instructions, handled by the FPU, and communication instructions, handled by the CMMU). Sparcle is best described as a conventional RISC microprocessor with a few additional features to support multiprocessing. The CMMU, on the other hand, is responsible for all memory and communication-related operations: it generates and consumes network traffic from the mesh router; it keeps the global caches coherent; it performs DRAM refresh and control; it generates interrupts and gathers statistics. In some sense, Sparcle handles all of the "standard" or "non-research" elements of an Alewife instruction stream, while the CMMU is responsible for everything else.

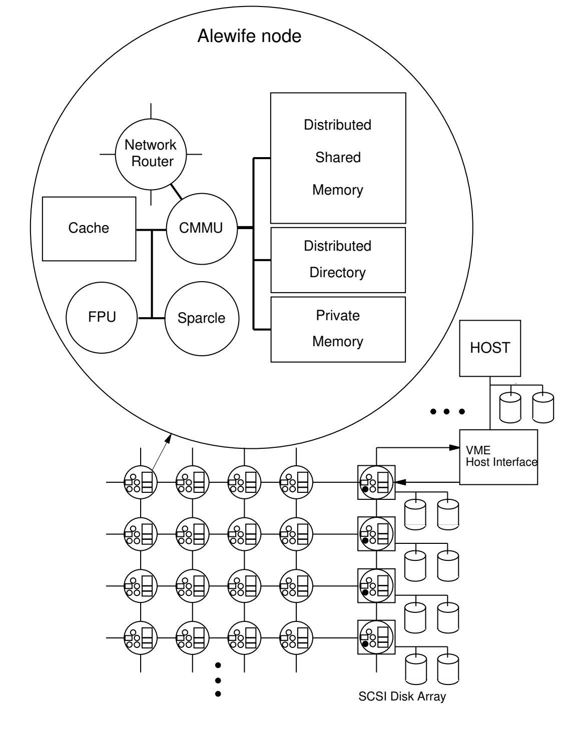


Figure 5-1: Node Diagram and Schematic for a 16-node Alewife Machine.

This particular division of Alewife into Sparcle and CMMU was as much the result of convenience of implementation as of anything more fundamental. Since Sparcle was derived from an industry-standard SPARC integer pipeline, this division reduced the probability of introducing bugs into a functioning and debugged element of the system. Further, MIT's control over the Sparcle design process was limited, since all modifications were implemented off-site, at LSI Logic and Sun Microsystems. Hence, most of the research interest of Sparcle lies in its *interfaces* to the CMMU, *i.e.* the way in which it permits user-level communication operations to be passed directly to the CMMU for processing. In the next section, we will spend some time discussing these interfaces.

It is important to note that the CMMU (and hence the communication interfaces) is connected directly to the first-level cache bus. Much debate has waged about cost and feasibility of placing communication interfaces so close to processor pipeline. In many cases, this debate degenerates into a philosophical debate about convenience and the degree to which the processor pipeline is untouchable. The point of view of the author was expressed in Section 2.1, when we discussed viewing communication interfaces as extensions to the ISA: communication operations are as important as computation instructions. Thus, it has been our goal throughout this thesis to explore interfaces that are simple and general enough that they would be included in the processor design from the beginning. This is in marked contrast to many other multiprocessor designs that tack communication interfaces onto existing pipelines and consider this a virtue.

Before we move forward, however, we would like to start with a bit of focus. Inasmuch as the Alewife multiprocessor is a complete, functioning system, many of the mechanisms presented in this chapter serve as scaffolding on which the research mechanisms are built or serve to provide functions required for a complete machine. Elements such as DRAM refresh and control or the fast interrupt controller fall into this category of scaffolding. Thus, before diving into the architecture, we would like to briefly touch upon the mechanisms or *primitives* that are directly related to the integration of message passing and shared memory. To do so, we call to mind the "hardware primitives" portion of Figure 1-3 on page 27, reproduced here as Figure 5-2.

In this figure, we see five semi-independent classes of mechanism, namely *shared-memory support (LimitLESS)*, *rapid context-switching support*, *message-passing support*, *deadlock detection*, and *revocable interrupt disable support*. The high-level aspects of each of these mechanisms have appeared in discussions of the previous three chapters. In this chapter, we will look in more detail at the actual implementation of these mechanisms.

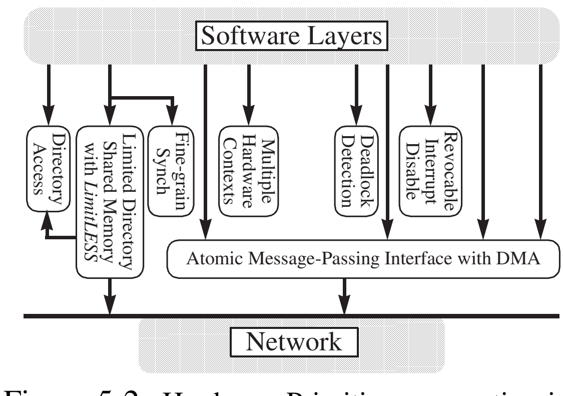


Figure 5-2: Hardware Primitives supporting integration in the Alewife architecture.

All told, Alewife provides a plethora of mechanisms and communication instructions. This would, at first glance, seem to be a regression to the days of CISC. However, the actual set of communication operations is a fairly concise set. In addition to tightly coupled facilities that might be considered "instructions", the CMMU provides many functions normally handled by other aspects of a system, such as interrupt control, memory initialization and control, timer facilities and statistics. Further, some of the facilities are experimental in nature. In this chapter, we will discuss the facilities that Alewife provides. In the chapter on Lessons of the Alewife Project (Chapter 7), we will revisit the question of essential mechanisms, attempting to get a better handle on the elusive "instruction set for multiprocessing".

# 5.1 Sparcle Architecture and Implementation

Our goal in this section is to show how Sparcle enables the fast interfaces of Chapter 2, and hence permits the tight integration of shared-memory and message-passing communication. As mentioned above, Sparcle is best thought of as a standard RISC microprocessor with extensions for multiprocessing[4]. One of the aspects of Sparcle that was exploited in Alewife was the fact that it had no on-chip caches, something which is unheard of today. In this sense, Sparcle represents microprocessor implementation technology from two or three generations ago. It is important to keep in mind, however, that this was of *implementation convenience only*. It permitted close, but external, access to the processor pipeline, thereby making it much easier to add instructions and mechanisms. One of the primary goals (and results) of Alewife was an "instruction set for multiprocessing" that was compatible with standard RISC pipelines. The ease with which Sparcle was supplemented with multiprocessing capability would translate directly to modern pipelines; these interfaces would simply be present on-chip rather than off-chip.

This section is divided into two major pieces. First, we discuss the interface between Sparcle and the CMMU. This interface is flexible enough to permit the construction of a number of powerful "instructions for multiprocessing". Then, we discuss the additions to SPARC to support rapid context-switching and featherweight threads. An important thing to remember during this discussion, is the fact that Sparcle is a derivative of a SPARC Version-7 processor from LSI Logic. The additions were relatively minor — they increased the number of gates by about 10% over the basic SPARC[1]. Modifications to this processor were specified at MIT, but implemented by LSI Logic and Sun Microsystems[2]. Thus, the set of modifications that we requested were necessarily limited in scope. Hence, Sparcle represents a tradeoff between desired functionality and practicality.

## 5.1.1 Sparcle/CMMU Interfaces

Sparcle implements a powerful and flexible interface to the CMMU. Figure 5-3 illustrates how this interface couples the processor pipeline with the CMMU. As seen in this figure, the interface can be divided into two general classes of signals: *flexible data-access mechanisms* and *instruction-extension mechanisms*. Flexible data access refers to the ability to distinguish different "flavors" of data access, thereby enabling the memory system to react differently to each flavor. Instruction-extension refers to the ability to add new instructions to the Sparcle pipeline.

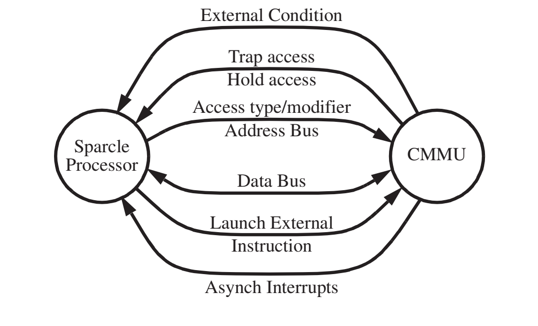



Figure 5-3: High-level interface between the Sparcle pipeline and the CMMU.

**Flexible Data Access:** There are several important features of flexible data-access mechanisms as used in Alewife. First, each load or store to the first-level cache is tagged with a "flavor", allowing

[1]Which may sound like a lot until one realizes that this is only about 2000 gates.

[2]We are eternally in debt to both LSI Logic and Sun, but more specifically to Godfrey D'Souza (LSI Logic) and Mike Parkin (Sun).

a system to possess a range of load and store *operations*, each with slightly different semantics. Note that these differing semantics are provided by the underlying memory system (CMMU) and not by the processor. Second, the ability to return condition information from the memory system to the processor. Third, the ability to extend operations into software by generating inexpensive exceptions on load/store operations. Inexpensive here means the ability to enter handler routines in a small number of cycles. This combination of features allows a dazzling array of operations to be synthesized: non-binding prefetch, fine-grained synchronization, rapid context-switching, supervisory control over things like cache-coherence directories, *etc.*.

Together, the Access Type, Address Bus, Data Bus, and Hold Access line form the nucleus of data access mechanisms and comprise a standard external cache interface. To permit the construction of other types of data accesses for synchronization, this basic interface has been supplemented with four classes of signals:

- A *Modifier* that is part of the operation code for load/store instructions and that is *not* interpreted by the core processor pipeline. The modifier provides several "flavors" or "colors" of load/store instruction.
- An *Access Type* that denotes the type of access that is occurring. Critical information includes the supervisor state, the size of the access, and whether or not atomicity is requested (for operations like atomic swap, etc).
- Two *External Conditions* that return information about the last access. They can affect the flow of control through special branch instructions.
- Several vectored memory exception signals (denoted *Trap Access* in the figure). These synchronous trap lines can abort active load/store operations and can invoke function-specific trap handles.

These mechanisms permit one to extend the load/store architecture of a simple RISC pipeline with a powerful set of operations.

**Instruction Extension:** An instruction extension mechanism permits one to augment the basic instruction set with external functional units. Instructions that are added in this way can be pipelined in the same fashion as standard instructions. The mechanism that Alewife exploits for adding instructions is commonly present in older-generation microprocessors (and Sparcle in particular) to permit close coupling with coprocessors: a special range of opcodes is reserved for external instructions and a *launch external instruction* signal is provided to initiate execution. Coprocessors are required to snoop on the instruction bus, grabbing the same set of instructions as the integer unit[3]. The coprocessor is allowed to begin decoding instructions that it recognizes, but is not allowed to commit any of them to execution until signaled to do so by the processor. For this reason, the coprocessor may need to buffer one or more pending operations, allowing for bubbles in the integer pipeline. Since launching of these instructions occurs under control of the integer pipeline, all of the complexities of branch-delay slots, pipeline flushing during exceptions, instruction commit points, *etc.*are focused in a single place, namely the integer execution unit.

[3]This mechanism, by its very nature, requires access to the first-level cache-bus.

The ability to add new instructions was exploited in Alewife to provide extremely low-overhead mechanisms for interrupt control, manipulation of the message-passing interface, and profiling statistics. In addition, the ability to snoop on the floating-point coprocessor interface allowed the synthesis of a floating-point unit with multiple register sets. Other than the latter, which was truly an implementation hack, the aforementioned uses of the instruction-extension mechanism could have been replaced with memory-mapped operations (albeit at an increase in cost of the corresponding mechanisms).

**Implementation of the Interface:** Figure 5-4 illustrates specific, SPARC-compatible names for the interface between Sparcle and the Alewife CMMU. As mentioned above, Sparcle is derived from a SPARC Version-7 processor from LSI Logic. Most of the "features for multiprocessing" that were added to this processor were already present to some degree. For instance, the mechanism for instruction extension was already present as part of a coprocessor interface, as was the external condition mechanism (complete with coprocessor "branch on condition" instructions). These interfaces were co-opted directly and consist of the `CINS` and `CCC` signals respectively.

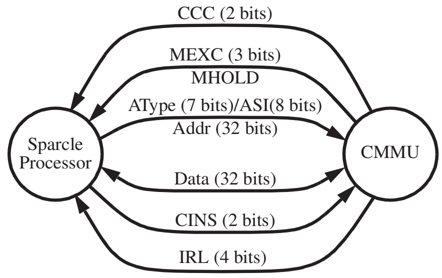



Figure 5-4: SPARC-compatible signal names for the Sparcle/CMMU interface.

The data-access modifier used to provide different flavors of load/store operations was already present as a relatively obscure feature of SPARC architectures, called the *Alternate Space Indicator* or ASI. Although this 8-bit tag was attached to all load and store operations, its potential was wasted to a large extent: gratuitous use of the ASI was made by memory and cache controllers to alter page tables and to control other supervisor state. Further, in the original architecture, user-code was allowed to produce only two of the 256 different ASI values. The Sparcle processor enhanced the ASI functionality in two ways:

- User-code is allowed to produce the top 128 of the 256 different ASI values. This means that the standard `lda` and `sta` instructions provided by SPARC may be executed by the user as long as they specify an ASI of 128 or greater.
- Sparcle provides a number of new load/store instructions that act exactly like standard load/store instructions except for the fact that they produce different ASI values. These were added because the `lda` and `sta` instructions (mentioned above) only exist with register/register addressing modes; the ASI value takes up space normally occupied by the offset in the register/offset mode.

These additions turned a superfluous feature into a powerful mechanism for adding operations.

Most of the access-type information is part of the standard SPARC bus interface. However, Sparcle adds the current state of the supervisor bit as an extra signal. This is important in that it allows selective protection to be placed on data accesses, as well as to instructions added with the instruction-extension mechanism. Alewife uses this feature to protect CMMU registers, as well as to provide enhanced capabilities to some of the communication instructions (*e.g.* allowing certain classes of communication to be initiated only by the supervisor). The existence of an external version of the supervisor bit provides much more flexibility in extending the functionality of the basic integer pipeline.

Finally, the synchronous trap mechanism was enhanced for Sparcle. The basic processor interface included a single synchronous trap line (called MEXC). One of the predominant Alewife philosophies (discussed in earlier portions of this thesis) is that of handling common cases in hardware and extending complicated or uncommon operations into software. The impact of transferring operations from hardware into software depends greatly on the cost of the hand-off mechanism. Thus, Sparcle adds two additional synchronous trap lines to minimize the overhead of entering into handler code. The three different lines (now called MEXC0, MEXC1, and MEXC2) behave identically, except for the fact that they invoke different trap handers. In Alewife, two of these lines (MEXC0 and MEXC1) are dedicated to context-switching and fine-grained synchronization respectively. The third is used for all remaining synchronous exceptions. In retrospect, more synchronous trap vectors would have been better. As discussed in Chapter 3, the distinction between synchronous and asynchronous exceptions (traps *vs.* interrupts) becomes greatly blurred in the presence of high-availability interrupts. SPARC (and Sparcle) provides 15 different asynchronous trap vectors, encoded with four different trap lines; a similar set of synchronous trap lines would have been desirable[4].

**Pipeline Effects on the Interface:** The coupling of the Sparcle pipeline with the CMMU results in user-visible delay slots on the external condition codes and interrupt controller. In particular, Sparcle requires a delay slot between instructions that modify the external condition codes and the testing of such codes. For instance:

```
lden       %r8, %r9, %r10  ; Load value, set empty.
noop                       ; Do something else
cbfull,a   GotLock         ; If full, got lock. Branch.
add        %r10, 3, %r11   ; Other random operation.
```

This *external condition delay slot* is required regardless of whether the setting instruction is a data operation or an external instruction. The second effect is a bit more obscure, but we will mention it here as an introduction to later "pipeline hacking". There is an asymmetry required in order to be conservative in the interaction between external instructions and asynchronous interrupts: in order to disable interrupts "immediately", we need to act in the decode stage, while in order to reenable interrupts, we need to act after the execute stage. The end effect is that loops such as the following never actually enable interrupts:

```
enabint TIMER  ; Take a timer interrupt here (if necessary).
disint  TIMER  ; Disable it again.
```

These effects are best described with the assistance of pipeline timing diagrams. Such diagrams will be of use to us later in discussing the CMMU, so we introduce them here. During the course of the following discussion, we will demonstrate the origin of the condition delay slot. The subtleties of the interrupt controller behavior we will save until the later discussion of the interrupt controller.

[4]The A-1001 CMMU included 25 different synchronous traps that were dispatched through these three vectors.

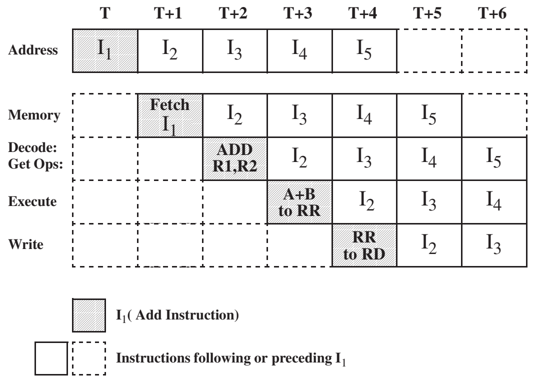



Figure 5-5: Pipelining for arithmetic instructions.

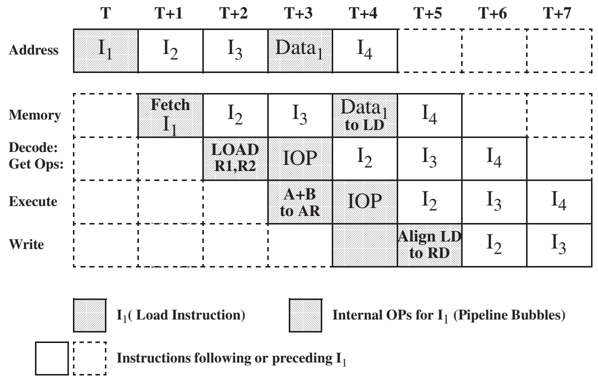



Figure 5-6: Pipelining for a single-word load.

The Sparcle pipeline varies in length from four to seven stages, depending on the type of instruction that is being executed. This variation in delay is a direct result of the fact that Sparcle has a single unified first-level instruction/data cache. Arithmetic instructions take four cycles while data access instructions take five (single-word loads), six (double-word loads, single-word stores), or seven (double-word stores and atomic, single-word swaps). Figure 5-5 shows timing for an arithmetic instruction, while Figure 5-6 shows timing for a single-word load[65]. Both of these diagrams present time across the top. Each line represents a different "stage": *address bus*, *memory bus*, *decode*,*execute*, *writeback*. The address bus (top line) represents the cycle before a data access in which the address is presented to the CMMU; hence we are not really counting it as a stage.

As is shown by Figure 5-5, the memory bus is 100% occupied fetching new instructions during arithmetic operations. This figure shows the execution of an `add` instruction, identified as $I_1$. Several following instructions are also shown in this diagram. With respect to $I_1$, this diagram can be interpreted as follows:

- *T:* The address of the `add` instruction appears on the address bus.
- *T+1:* The `add` instruction fetched from the cache.
- *T+2:* The `add` is decoded and operands are fetched from the register file.
- *T+3:* The `add` is executed.
- *T+4:* Results are written back to the register file.

This diagram can be used to illustrate the need for an external condition delay slot: assume for a moment that instruction $I_3$ is a `branch` instruction. Since Sparcle has one branch delay slot, this means that instruction $I_5$ might be the branch target. Since the address for $I_5$ goes out on cycle *T+4*, which is the decode cycle for $I_3$, this means that branch instructions make their decisions *during the decode stage*. This, in turn, means that any external conditions on which this decision is based must be available while the branch is still in the fetch (memory) stage[5]. Note that $I_1$ is in the execute stage when $I_3$ is in the fetch stage; thus, if $I_1$ produced an external condition, $I_3$ would be the first instruction that could act on that condition (hence the delay slot mentioned at the beginning of this section).

[5] This is not true for *internal* conditions, since internal bypassing is fast enough.

Figure 5-6 shows behavior for a load instruction, $\mathtt{I}_1$. Since the memory bus is 100% occupied with instruction fetches, we must pause the pipeline mechanism long enough to insert an extra data load cycle. This is accomplished by inserting placeholder instructions into the pipeline (shown as IOPs). The end result is that, on cycle *T+3*, instruction $\mathtt{I}_2$ is prevented from advancing into the decode stage and the address of the load operation is presented on the data bus instead of the address of instruction $\mathtt{I}_4$. Finally, on cycle *T+4*, the actual data access is performed. Note that cycle *T+3* illustrates that the fetch of $\mathtt{I}_3$ continues in spite of the fact that $\mathtt{I}_2$ has not advanced. This is the origin of the fact that a coprocessor may need to buffer more than one instruction at a time (this was mentioned above in the discussion of the instruction extension mechanism). Should this load produce an external condition, it would do so on cycle *T+4*. Given the discussion of the previous paragraph, instruction $\mathtt{I}_3$ would be the first instruction that could act on this condition (again showing the need for a single delay slot).

### 5.1.2 Support for Rapid Context-Switching and Featherweight Threads.

Chapter 2 discusses the advantages and high-level justifications for featherweight threads. The principal barrier to fast switching between threads is the the amount of per-thread state that must be saved and restored during such a switch. As a result, the more state that is duplicated in hardware (registers, program counters, *etc.*), the less software overhead that is incurred during a context switch. A given architecture will support some amount of state duplication (*e.g.* four contexts in Alewife); this yields a maximum number of threads that may be loaded and ready to execute at any one time. Since context switching is invoked automatically under some circumstances (during a cache miss, for instance), some method must be present to control which of the available hardware contexts are *active*. Thus, in this section, we would like to discuss two aspects of Sparcle that enable fast context switching: the explicit duplication of state (or techniques for duplicating state), and instructions for switching between active hardware contexts. Our goal is to permit rapid context switching and context management in as few instructions as possible.

As a starting point, Sparcle builds on the existence of multiple integer register sets in SPARC. The particular SPARC design that was modified for Sparcle contains eight overlapping register windows. Rather than using the register windows as a register stack (the intended use for a SPARC processor), Alewife uses them in pairs to represent four independent, non-overlapping contexts. This change requires use of a custom compiler, since standard SPARC compilers change window pointers on (almost) every procedure call[6]. In this scheme, the SPARC *Current Window Pointer* (CWP) serves as the context pointer and the *Window Invalid Mask* (WIM) indicates which contexts are disabled and which are active. This particular use of register windows does not involve any modifications, just a change in software conventions.

Although this is close to what is needed for block multithreading, it is not quite sufficient. In the following paragraphs, we will discuss the various issues involved. The modifications in Sparcle that were directed toward block multithreading were, in fact, aimed at minimizing the overhead of context switching as used for latency tolerance. It turns out, however, that these modifications-line benefited all uses of block-multithreading, including the more general featherweight threads.

[6] This was not an issue, since Alewife used its own compiler for many other reasons as well.

**Use of Traps:** One deficiency in SPARC with respect to block multithreading is the fact that SPARC does not have multiple sets of program counters and status registers. Since adding such facilities would impact the pipeline in a nontrivial fashion, we chose to implement context-switching via a special trap with an extremely short trap handler that saves and restores these pieces of state. A trap is necessary, since this is the only way to get direct access to the two active program counters. Further, on SPARC, modifications of the CWP (used in Sparcle for context management) and the processor status register (PSR) are protected operations. As a result, all context management operations (and hence featherweight thread operations) must pass through supervisor level. Hence, to achieve latency tolerance, the CMMU generates a *context-switch trap*, that causes Sparcle to switch contexts while simultaneously sending a request for data to a remote node.

**Multithreading the FPU and Context-Sensitive Memory Operations:** Up to this point, we have been talking about an unmodified SPARC processor. Unfortunately, such a processor has several deficiencies with respect to context switching that cannot be addressed with software. First and foremost, the SPARC FPU contains only one set of registers and a single status register — complicating the implementation of context-switching. Second, as discussed in Chapter 3, block multithreading introduces a number of service-interleaving problems whose solutions require per-context state within the memory controller. Sparcle addresses both of these issues by exporting the current context identifier (top two bits of the CWP) to the pins; this information is part of the *Access Type* and permits the CMMU to maintain context-dependent state. Among other things, this modification enables the "FPU multithreading hack", whereby the CMMU modifies FPU instructions on-the-fly, inserting the context identifier into the top two bits of the register specifiers. This transforms an FPU with one 32-register context into an FPU with four 8-register contexts[7].

**Context Management Instructions:** Another deficiency in SPARC for block multithreading is the fact that manipulation of the current window pointer is oriented toward a linear, stack-like usage. This has consequences: for one thing, movement to the next Alewife context requires execution of two SPARC instructions (since Sparcle uses register windows in pairs). For another, these instructions invoke *window overflow* or *window underflow* traps when they encounter a disabled register set. Sparcle addresses both of these problems by adding new instructions, called `nextf` and `prevf`, for context management. These instructions advance the CWP by two, matching the Alewife usage of register windows. Further, they use the WIM to indicate which contexts are considered enabled. When executed, they will move to the next active context (either forward or backward respectively) as indicated by the WIM register. If no additional contexts are active, they leave the window pointer unchanged. Hence, these instructions decouple context management from context switching.

In retrospect, these instructions were close to ideal; other features would have been desirable however: First, the ability to tell quickly whether or not all contexts were allocated. Second, the ability to move quickly to the next *disabled* context. (These two could have been combined into a single operation which set a condition bit.) Third, the ability to mark a context as enabled or

[7] It is important to realize that this was of implementation convenience only — any "real" system would implement this functionality within the FPU without restricting the number of per-context registers.

disabled without have to compute the proper bit mask for operations on the CWP. The presence of these three operations would have shortened context management to a few instructions in the common case. Nonetheless, `nextf` and `prevf` provided a reasonable set of operations.

**Miscellaneous:** Two other slight modifications were made to assist block multithreading by minimizing extraneous trap overhead. One of these was mentioned above, namely the addition of multiple synchronous trap lines. This allows the dedication of a synchronous trap line (and corresponding vector) to cache-miss handling. Second, the default size of a trap vector on SPARC is four instructions (16 bytes). Sparcle increases this size to 16 instructions (64 bytes), thereby providing enough space for a number of short interrupt handlers to fit entirely within the vector.

```
rdpsr %r16            ; Save PSR in reserved register
nextf %r0, %r0, %r0   ; Move to next active context
wrpsr %r16            ; Restore PSR.
jmpl  %r17, %r0       ; Restore PC.
rett  %r18, %r0       ; Restore nPC, start new thread
```

Figure 5-7: Context switch trap code for Sparcle

| Cycle | Operation |
|---|---|
| 1 | Fetch of Data instruction (load or store) |
| 2 | Decode of Data instruction (load or store) |
| 3 | Execute of instruction (compute address) |
| 4 | Data cycle (which will fail) |
| ⇒ 5 | Pipeline freeze, flag exception to processor. |
| 6 | Pipeline flush (save PC) |
| 7 | Pipeline flush (save nPC, decr. CWP) |
| 8 | Fetch: `rdpsr %r16` |
| 9 | Fetch: `nextf %r0, %r0, %r0` |
| 10 | Fetch: `wrpsr %r16` |
| 11 | Fetch: `jmpl %r17, %r0` |
| 12 | Fetch: `rett %r18, %r0` |
| 13 | Dead cycle from `jmpl` |
| 14 | First fetch of new instruction |
| 15 | Dead cycle from `rett` |

Figure 5-8: Breakdown of a 14-cycle context-switch on data for a load or store. Note that cycle 15 performs useful "post context-switch" work.

**Resulting Context-Switch Behavior:** With these changes, the context-switch trap handler is shown in Figure 5-7. When the trap occurs, Sparcle switches *one* window backward (as does a normal SPARC). This switch places the window pointer *between* active contexts, where a few registers are reserved by the Alewife operating-system for context state. As per normal SPARC trapping behavior, the PC and nPC are written to registers r17 and r18 by the hardware. This trap code places the PSR in register r16. Then, the `nextf` instruction advances to the trap frame of the next active context. Finally, a return from trap sequence `jmpl/rett` restores the PC and nPC of that context, causing execution to continue at that point. The net effect is that a Sparcle context switch can be done in 14 cycles as depicted in Figure 5-8. This illustrates the total penalty for a context-switch on a data instruction. Cycle four (marked with "⇒") is the cycle in which the actual miss is flagged via a synchronous memory fault. A request to memory is initiated at the same time[8].

By maintaining a separate PC and PSR for each context, a more aggressive processor design could switch contexts much faster. However, even with 14 cycles of overhead and four processor-resident contexts, multithreading can significantly improve system performance [115, 63].

Appendix A shows a more detailed use of the Sparcle context-switching mechanisms to implement featherweight threads for user-level active message handlers. The code shown in this appendix illustrates how the value in the WIM is combined with the `nextf` and `prevf` instructions to achieve fast allocation and deallocation of contexts, as well as simple multi-level scheduling.

[8] Although 15 cycles are shown in this diagram, one of them is the fetch of the first instruction from the next context.

## 5.2 The Communications and Memory Management Unit

The Alewife Communications and Memory Management Unit (CMMU) [61] is the heart of the Alewife machine. Whereas Sparcle facilitates the low overhead communication interfaces of Alewife, the CMMU houses the actual interfaces as well as all of the communications functionality. This chip contains approximately one million transistors, consisting of 100,000 gates (four-transistor inverter equivalents) and about 120,000 bits of SRAM. By today's standards, this is a small chip; however, this was no less challenging to produce in an academic environment. During the course of the Alewife project, two different versions of the CMMU were produced, designated the A-1000 and A-1001. The primary difference between these chips is that the A-1001 fixes bugs present in the A-1000, as well as including an implementation of the user-level atomicity mechanism (Section 2.4.5) that was absent in the original. Unfortunately, the A-1001 also had some noise problems not present in the original, so no large systems were built with this chip[9].

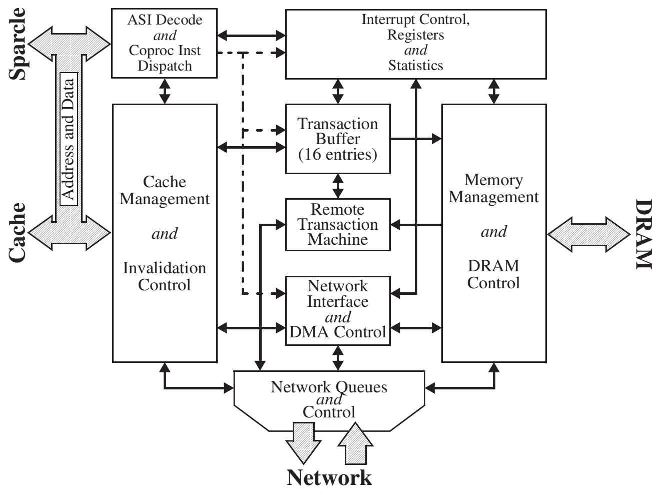



Figure 5-9: Block diagram for the Alewife CMMU.

Figure 5-9 shows a block diagram for the CMMU. In an Alewife node, the CMMU is connected directly to the first-level cache bus and provides much the same functionality as a cache-controller/memory-management unit in a uniprocessor: it contains tags for the cache, provides DRAM refresh and ECC, and handles cache fills and replacements. Most importantly, however, it also implements the architectural features for multiprocessing discussed in the first part of this thesis. The fact that both uniprocessor and multiprocessor features are integrated in the same package is one of the reasons that Alewife has such low cycle counts for various operations.

Examining this diagram for a moment, we see that the external network connects to the *Network Queues and Control* block. This block is responsible for multiplexing and demultiplexing data between the CMMU and the network, and provides the central point for the integration of message-passing and shared-memory communication. Also in this diagram, we see that the Sparcle processor interfaces directly to two different logic blocks, the *ASI Decode and Coprocessor Instruction Dispatch* block and the *Cache Management and Invalidation Control* block. The first of these is responsible for interpreting and responding to the Sparcle/CMMU interfaces discussed in Section 5.1.1. Among other things, this block contains a mirror of the processor pipeline that handles coprocessor instruction extensions (such as `ipilaunch`) and contains logic to interpret and respond to ASI values (such as Full/Empty operations). The second block implements the processor side of the cache-coherence protocol, as well as managing the cache and allocation of *Transaction Buffer* entries. This block is mirrored by the *Memory Management and DRAM Control* block, that handles the memory-side aspects of the cache coherence protocol, as well as providing

[9] However, see Ken Mackenzie's thesis [79].

DRAM refresh and control.

At the heart of the CMMU is the *Transaction Buffer*, which was one of the primary results of Chapter 3. This block provides a 16-entry fully-associative data store that tracks outstanding cache-coherence transactions, holds prefetched data, and stages data in transit between the cache, network, and memory. The *Transaction Buffer* is at the center of the CMMU and is provides a 16-entry, fully-associative data store that tracks outstanding cache-coherence transactions, holds prefetched data, and stages data in transit between the cache, network, and memory. The *Remote Transaction Machine* is responsible for handling those aspects of Transaction Buffer manipulation that do not require direct intervention of the Cache Management block; this includes retiring transient (flush) entries (Section 5.2.3) as well as accepting data that arrives from the network in response to a data request.

Finally, the *Interrupt Control, Register, and Statistics* block provides a general, memory-mapped interface to much of the internal state of the CMMU, including those registers required for sending and receiving user-direct messages. In addition, this block contains all of the interrupt control functionality, as well as a wide array of statistics gathering facilities.

Since the Alewife CMMU is a large, multi-faceted chip, a complete discussion of the hardware architecture is somewhat beyond the scope of this document[10]. Instead, we would like to focus on a few key aspects of the architecture that are relevant to discussions in the first part of the thesis and that represent challenging aspects of the implementation. First, Section 5.2.1 discusses how the five major CMMU state machines interact with the network queues. This particular section serves to highlight the integration of message passing and shared memory that is present in the Alewife CMMU. Next, Section 5.2.2 discusses the combined Sparcle/CMMU interface from the standpoint of the CMMU; this is tightly coupled with cache-management. Among other things, we will see how high-availability interrupts are forwarded to the processor. Next, Section 5.2.3 discusses the states and functionality of the Alewife transaction buffer. Section 5.2.4 follows with a discussion of memory scheduling and atomicity, introducing a technique called *service coupling*. Finally, Section 5.2.5 discusses the implementation of local coherence in Alewife, pulling together aspects of Cache Management, the transaction buffer, and memory scheduling.

### 5.2.1 The Division of Labor and the Network Topology

Aside from interface logic (which we will discuss briefly in the next section), the Alewife CMMU contains five major loci of control: the *Cache Management Machine*, the *Remote Transaction Machine*, the *Memory Management Machine*, the *IPI Input Machine*, and the *IPI Output Machine*. It is the interaction between these five state machines that exemplifies the integration of message passing and shared memory in Alewife. The first three of these are responsible for handling general memory access and cache coherence, while the later two are responsible for implementing the network interface and providing locally coherent DMA.

This division of labor into five different state machines was natural for a number of reasons. First, as discussed in Section 4.3.7, the DMA engines must be independent of one another and the shared-memory system in order to avoid deadlock in a machine that relies on two-case delivery. This argues for independent operation of the two IPI machines. In addition, to permit pipelining

[10] If for no other reason than it would stretch for more pages than we have here.

in DMA coherence, the independence of these two state machines from the cache and memory handling is necessary. Second, division of cache-coherence protocol handling into processor-side and memory-side state machines naturally reflects the "client-server" nature of cache-coherence protocols. Finally, the separation of the processor-side of the cache coherence protocol into two machines, the Cache Management and Remote Transaction machines was natural because it split processor-side operations into those that required access to the cache tags and those that did not; this freed the processor to continue execution during the return of remote data and during transaction buffer management operations (such as garbage collection; see the discussion of transient states in Section 5.2.3.1).

To expand for a moment on the distinction between these latter two state machines: The Cache Management Machine is responsible for any operations that touch the first-level cache directly, including invalidation, cache fills, cache replacements, *etc.*. Further, it is the sole initiator of new transactions (and allocator of transaction buffers). The Remote Transaction Machine, on the other hand, is responsible for accepting returning global data from the network[11], flushing transaction buffers into the network (if they are for remote addresses), and garbage collecting cached transaction buffer entries.

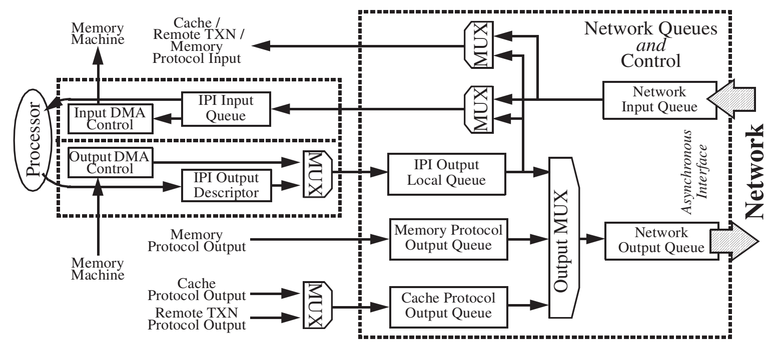



Figure 5-10: Queue topology for the Alewife CMMU.

Not surprisingly, the presence of these five state machines is reflected in the topology of the network queues. Figure 5-10 illustrates this topology. This diagram was introduced briefly in Chapter 4 in order to emphasize the potential conflicts between shared memory and message passing. Here we would like explore this topology in greater detail. The first thing to note is that two of the blocks from the CMMU block diagram (Figure 5-9) are shown as grouped sets of queues (surrounded by dashed boxes) in Figure 5-10: the *Network Queues and Control* block and the *Network Interface and DMA Control* block (unlabeled and split in two at the left of the diagram).

The *Network Queues and Control* block consists of five different queues and several muxes. It also contains inter-queue routing logic that is not shown. The queues closest to the network (labeled *Network Input Queue* and *Network Output Queue*) are driven asynchronously by the EMRC routing chips. Three separate queues feed into the Network Output Queue: The *Cache Protocol Output Queue*, the *Memory Protocol Output Queue*, and the *IPI Output Local Queue*. Two of these carry shared-memory traffic (from the cache and memory sides of the protocol respectively)[12], while the third carries messages from the user-level message-passing interface to the network. Note that this

[11]Note that returning data is *not* placed directly into the first-level cache, since the processor is likely doing something other than waiting for this data (because of rapid context-switching). If the processor *does* happen to be waiting for data, however, the data is pipelined directly into the first level cache. See Section 5.2.3.5.

[12]The fact that the Cache Protocol Output Queue is shared between the Cache Management and Remote Transaction Machines directly mirrors the sharing of the associative access port of the transaction buffer (see Section 5.2.3).

output queue doubles as a “loopback” queue, feeding back into the protocol and message-passing input queues; this provides a short-cut path for messages that are launched though the message interface but destined for the local node; one of the reasons that this path was included was for two-case delivery (see Section 4.3), since it aids in the relaunching of messages that have been buffered in memory without requiring passage through a possibly congested network.

The input side of the cache coherence protocol is shown as a single, non-demultiplexed input path. This single path leads to all three of the cache-coherence state machines. Message sorting hardware (not shown) takes each incoming message and decides whether to route it to message-passing interface or the coherence protocol interface; if it is the latter, this circuitry additionally notifies the appropriate state machine that a message is waiting for processing. What is advantageous about this arrangement is that one of the protocol machines can choose to reject an input message without freeing its contents from the network; in that case the message may be redirected to the message input port (possibly with slight modifications of the header flit). This is the principal mechanism for forwarding cache-coherence protocol messages to software for LimitLESS[13].

**User-Level Network Interfaces:** One additional aspect of Figure 5-10 that we would like to discuss is the message-passing interface. This is shown to the left of the figure, and corresponds to the *Network Interface and DMA Control* block of the CMMU. Queues in this block are mapped directly to the message-passing interfaces of Section 2.4.2; in particular, the *IPI Input Queue* is directly mapped to the `input_window` array while the *IPI Output Descriptor* is mapped directly to the `output_descriptor` array (both adjusted for the circular nature of the physical queues). In fact, the design of interfaces for UDM were targeted toward mapping queues in this way; for instance, Section 2.4.3 discusses the fact that injection of network packets is stalled during network congestion through blocking writes to the `output_descriptor` array. As another example, given the direct mapping of of the `output_descriptor` array to the IPI Output Descriptor queue, the atomicity of the `ipilaunch` operation is achieved simply by advancing the head pointer of a circular queue. Thus, our statement in the introduction that the UDM interfaces were chosen for fast simple implementation is accurate.

As one final note on this mapping between user interfaces and physical queues: the mapping is done in a way that maximizes parallelism between the processor and network; thus, for instance, the illusion is maintained that all flits of a message have arrived at the input interface, even when they have not. This permits the CMMU to post notification for an incoming message as soon as the first few flits arrive, thereby allowing interrupt handler or polling code to overlap the arrival of data (see Section 2.4.4).

Also shown in this portion of Figure 5-10 are the two DMA engines that are responsible for providing locally-coherent DMA for messages. We will be discussing DMA coherence in more detail in Section 5.2.5. For now, however, note that message DMA and the related tasks of constructing outgoing messages and deconstructing incoming messages are activities that are performed by the IPI Output Machine and IPI Input Machine respectively.

[13]In fact, the memory system is even more clever than that; to achieve the *read-ahead* optimization of Section 3.4.2.1, the Memory Machine will process a read request while simultaneously requesting that it be redirected to the message input queue. This provides quick turn around on data — returning it immediately to the requesting node — while still invoking software to perform protocol actions.

## 5.2.2 The Sparcle/CMMU Interface Revisited

Section 5.1.1 discussed the Sparcle/CMMU interface from the standpoint of the Sparcle processor. In this section we would like to explore this interface from the standpoint of the CMMU. As mentioned earlier, there were two classes of interface mechanisms: *flexible data-access mechanisms* and *instruction-extension mechanisms*. These are provided by a combination of two blocks in the block diagram of Figure 5-9: the *ASI Decode and Coprocessor Instruction Dispatch* and the *Cache Management and and Invalidation Control*. One of our goals in this section is to see how these two blocks cooperate to simultaneously satisfy the needs of the Sparcle processor without neglecting the demands of other users for cache invalidation and coherence services. Both of these blocks operate by tracking aspects of the Sparcle pipeline.

### 5.2.2.1 The Coprocessor Pipeline

Because it is the most straightforward, we will start by discussing the instruction-extension mechanism. In the SPARC Version-7 specification, coprocessors add instructions to an ISA by mirroring the instruction-fetch process of the integer unit. The coprocessor is responsible for the decode and execution of only those instructions that it is providing. In addition, the integer unit retains control over execution flow by controlling which instructions are fetched in the first place and which of the fetched instructions should actually be executed by the coprocessor.

In the Alewife CMMU, this *coprocessor pipeline* (part of the *ASI Decode and Coprocessor Instruction Dispatch*) is responsible for decoding and launching a number of different operations. Included in this set are the network interface instructions such as `ipilaunch` and `ipicst`, as well as interrupt controller instructions such as `enabatom`, `disatom`, `enabint`, and `disint`. Instructions added in this way have the advantage of possessing the same timing as arithmetic instructions (Figure 5-5). As a consequence, Alewife supports single-cycle instruction controller operations, single cycle message launch, *etc.*. It is important to note that this particular implementation detail is not *required* for the interfaces of Chapter 2; memory-mapped operations would suffice. However, this level of integration with the processor pipeline is in keeping with our view of interfaces as an extension to the ISA. The coprocessor pipeline is also responsible for controlling the FPU context-switching mechanisms mentioned in Section 5.1.2.

### 5.2.2.2 The Cache Management Machine

Whereas the coprocessor pipeline operates by tracking the state of the Sparcle pipeline in detail (at least for coprocessor and FPU instructions), the Cache Management Machine (CMM) operates by tracking Sparcle memory operations in detail. Figure 5-11 illustrates that this process can be view as consisting of five primary meta-states (each of which consists of multiple individual states). Two of these (the `Normal_Access` and `Resource_Wait` meta-states are shaded, indicating that they may persist for many cycles. In contrast, the `Cache_Fill`, `Cache_Replace`, and `Flag_Exception` states are transient, lasting for short periods of time.

The `Normal_Access` meta-state (an exploded view of which is shown in Figure 5-12) explicitly tracks normal, cached memory operations. The CMM returns to the `Home` state at the

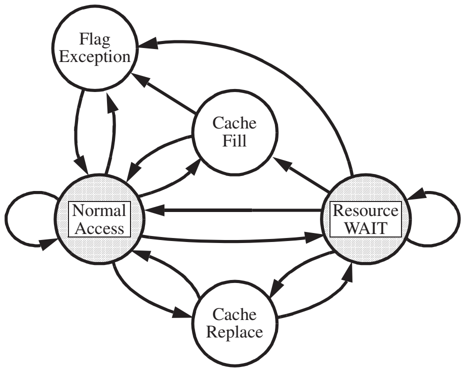



Figure 5-11: Meta-states of the Cache Management Machine. Each of these states is actually a conglomerate of states (see, for instance, diagram to the right). Shaded meta-states are potentially long-lived.

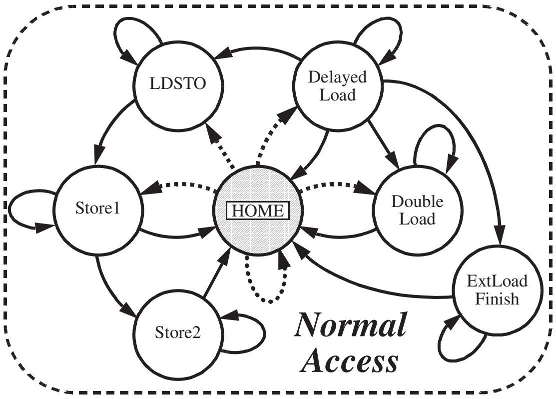



Figure 5-12: An exploded view of the normal "state" of the Cache Management Machine. The Cache Management Machine operates entirely within this set of states as long as it is processing cache hits. Dashed arcs represent transitions that may occur indirectly, by passing first through a cache-fill or remote access operation.

beginning of each access to the first-level cache. Multi-cycle memory operations (such as double-word loads, stores, and atomic swap operations) are handled by brief deviations away from the `Home` state. In addition, the `Delayed_Load` state is used to insert an extra bus-turn around cycle for uncached accesses from the transaction buffer. The different states of the `Normal_Access` meta-state are used to control such things as the acquisition of a new address, the direction of the bus, cache control signals, *etc.*. In addition, the presence of states other than the `Home` state guarantees that multi-cycle address operations are atomic and unbroken by other operations.

Since Sparcle has no first-level cache, it makes requests to the memory system (for either instructions or data) on every cycle in which it makes forward progress. Arcs that are dashed in Figure 5-12 represent transitions that may occur indirectly through other meta-states. For instance, in the case of store that misses in the cache but hits in the transaction buffer (more about cached transaction-buffer states in Section 5.2.3), the CMM would transition into the `Cache_Fill` meta-state to fill the cache, then finish by transitioning to the `Store1` state.

The `Resource_Wait` "meta-state" (which is actually a single state) is entered whenever Sparcle becomes blocked awaiting resources. Thus, for instance, this state is entered whenever the processor initiates a cache miss (remote or local). It is also entered for exceptional conditions, such as the presence of insufficient transaction buffers to initiate some new request. The CMMU asserts a hold signal whenever the Cache Management Machine enters the `Resource_Wait` state.

**Multiplexing of the Cache:** Multiplexing of the first-level cache between the processor and other "auxiliary" users (such as remote invalidations or the DMA engines), occurs on one of two clock edges: either at the beginning of a new access (when the CMM is transitioning to the `Home` state) or when the CMM is transitioning to the `Resource_Wait` state. Because these are the only two points at which ownership of the CMM can change, memory operations which the processor considers "atomic" (such as swap) remain atomic to operations of the CMM as well. Call clock edges in which ownership of the something can be scheduled *reschedule points*. The CMM has an accompanying scheduler that is responsible for taking the current set of requests for the cache (from the processor, network, local memory, and DMA engines), deciding which have sufficient resources to complete, then choosing a new request at each reschedule point from the set of "schedulable" requests. The scheduler is responsible for making sure that all requesters eventually make forward progress; this is done by scheduling them in an LRU fashion. This type of scheduling is a version of *service coupling*, that we will discuss in Section 5.2.4.

**Achieving Flexible Access:** Given our discussion of the Cache Management Machine, achievement of flexible data access (as defined in Section 5.1.1) is straightforward. At the beginning of each new access (signaled by a transition *into* the `Home` state), the *ASI Decode and Coprocessor Instruction Dispatch* (ADCID) block latches the address, access type, and ASI value for the new memory request[14]. Then, based on the access information, the ADCID block chooses one of several different sources/destinations of data on the Sparcle data bus. This permits, for instance, the CMMU registers to be accessed by choosing an instruction with the appropriate ASI value; the ADCID block is responsible for routing data to/from the *Interrupt Control, Registers, and Statistics* block. If the type of access is an access to global memory with fine-grained extensions, the ADCID block is additionally responsible for decoding the ASI value to detect the type of synchronization operation being requested, then modifying the current value of the synchronization bit and possibly generating a request for a `full_empty_trap`.

Hence, as asserted at the beginning of this section, the Sparcle/CMMU interface is implemented by a combination of the ADCID and CMM blocks.

**High Availability Interrupts:** Finally, the implementation of high-availability interrupts (as discussed in Section 3.1) is straightforward, given the Cache Management Machine. As a brief reminder, high-availability interrupts are asynchronous interrupts that may be delivered synchronously under specialized circumstances. In particular, those specialized circumstances are precisely those periods in which the processor pipeline is blocked on memory operations. In the CMM, periods in which normal asynchronous interrupts must be transformed into synchronous interrupts are precisely those points at which the CMM is waiting in the `Resource_Wait`.

Under normal circumstances, the Alewife interrupt controller takes a vector of pending asynchronous interrupts and modifies it by the enable mask[15]. The resulting set of "active" interrupts are priority-encoded to produce a single asynchronous interrupt that is presented to Sparcle (this is the `IRL` signal in the interface diagram of Figure 5-4). In addition to generating the highest-priority asynchronous interrupt, the interrupt controller examines the set of active interrupts for ones that are eligible for high-availability delivery[16]. When the CMM is waiting in the `Resource_Wait`

[14]The CMM is carefully synchronized with Sparcle so that access information is present on this particular edge.

[15]The enable mask is modified by three instructions, `enabint`, `disint`, and `settemask`, as well as the user-level atomicity operations, `enabatom`, `disatom`, and `setatom`.

[16]The set of asynchronous interrupts that are considered eligible for synchronous delivery varies based on the current state of the CMMU and the pending Sparcle memory operation. Thus, for instance, high-availability interrupts are not delivered when Sparcle is blocked awaiting a cache-fill from the unshared memory. Also, to ensure forward progress, asynchronous events are promoted to synchronous delivery only if forward progress of the system can be guaranteed.

state, it will deliver the highest-priority eligible high-availability interrupt by asserting one of the Sparcle synchronous trap lines (MEXC2) and by recording an identifier for the type of trap in the `AuxFault_Vector` register.

### 5.2.3 The Transaction Buffer Framework

The transaction buffer lies at the heart of the Alewife machine and was one of the major results of Chapter 3. As noted in that chapter, the presence of the transaction buffer provides a ready solution to a number of problems facing an integrated architecture such as Alewife. This centralized module keeps track of outstanding data transactions, both local and remote. In doing this, it combines several related functions:

- Window of vulnerability closure: The transaction buffer implements the thrashlock mechanism of Section 3.2.
- Reordering of network packets: The explicit recording of each outstanding transaction permits the Alewife coherence protocol to be insensitive to network order.
- Flush queue to local memory: When dirty lines are replaced from the cache, they are written to transaction buffers for later processing by the memory management hardware. As discussed in Section 3.4.2, this has a number of important advantages over a FIFO replacement queue.
- Transient data storage for DMA coherence: Data flushed from the cache during the process of providing DMA coherence passes through the transaction buffer, thus remaining under control of cache-management mechanisms. This is really an extension of the previous item.
- Small, fully-associative cache: Under normal circumstances, the "access" phase of a transaction involves transferring a complete cache line from a transaction buffer to the cache. It is also possible, however, to perform "uncached" reads and writes which access data directly in the transaction buffer rather than filling the cache; afterwards, the corresponding memory lines remain in the transaction buffer.
- Repository for prefetched data: This is a simple extension of the previous item: non-binding prefetch operations return data to the transaction buffer.

The transaction buffer thus provides an important centralized resource. To permit this single interface to be used for all data movement between processor, network, and memory, the access ports are fully bypassed to allow the pipelining of from one port to another.

The block diagram of Figure 5-9 captures the fact that the transaction buffer lies at the heart of the three primary cache-coherence protocol machines (the Cache Management, Remote Transaction, and Memory Management blocks). Also, as will be discussed in Section 5.2.5, the transaction buffer plays a pivotal role in achieving DMA coherence[17].

[17]Figure 5-9 does not show the transaction buffer connected directly to the DMA Control block, since DMA coherence occurs indirectly: the DMA engines request invalidations from the Cache Management block and retrieve data from the Memory Management block.

The transaction buffer is comprised of two major blocks: the *transaction state CAM* (content-addressable-memory) and the *transaction data store*. Together, these pieces comprise sixteen transaction buffers. Figure 5-13 shows the key aspects of the transaction state CAM. This module consists of dual-ported storage for the state bits, an arbitrated *associative match* for locating data, a *thrashlock monitor* which implements the thrashlock mechanism of Section 3.2, and a *transaction monitor* for managing the pool of buffers. The transaction data module (not shown in this figure) contains sufficient storage for one complete, four-word memory-line for each transaction buffer. Each memory-line is physically divided into two 64-bit sub-lines.

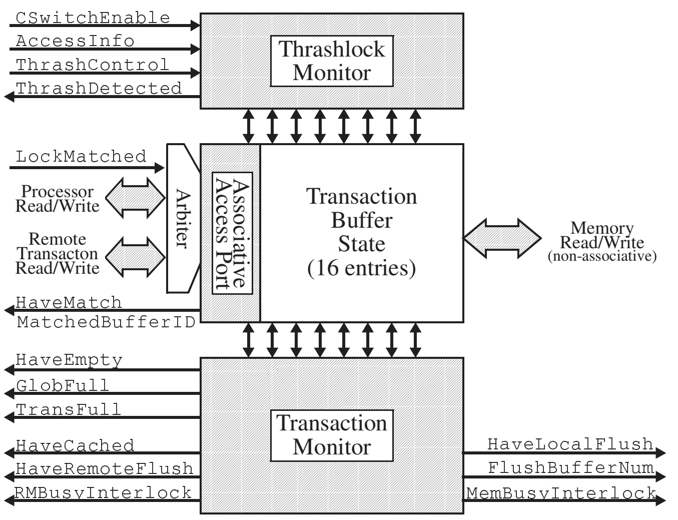


Figure 5-13: The transaction state CAM and associated transaction monitors.

The choice to implement sixteen transaction buffers arose from sufficiency arguments and space limitations. Each of the four hardware contexts can have two outstanding primary transactions; consequently, at least eight buffers are necessary. Then, since the transaction buffer serves as a flush queue to local memory, a few additional entries are necessary. Finally, to guarantee access to interrupt code and unshared variables (for network overflow recovery), another buffer is required. Thus approximately eleven buffers are sufficient for operation. Remaining buffers are targeted for uncached accesses, prefetched data, and victim cached memory lines. Note that in code which is amenable to software prefetching, fewer primary transactions will be necessary (since the compiler is orchestrating communications). Section 5.2.3.4 discusses the ability to trade buffer usage between primary and secondary transactions.

#### 5.2.3.1 Transaction Buffer States

Figure 5-14 illustrates the state of a transaction buffer. Several of its components were discussed in Chapter 3. The *address* field is 28 bits[18]. An address which has its top bit set belongs to the global shared address space; consequently, buffers with such global addresses are referred to as *global buffers*. The four *full/empty bits* hold the synchronization bits for the four data words. Finally, the 10 remaining state bits are divided into seven different fields. Some of these fields encode stable states indicating that transactions are in progress or that

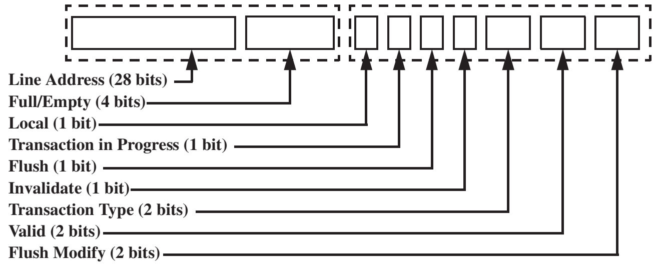


Figure 5-14: The state of a transaction buffer.

[18]Sparcle addresses are 32 bits and memory-lines are 16 bytes long.

data is valid. Others encode transient states. Briefly, these seven fields are as follows:

- The *Local* (LOC) bit caches information on whether the address is local (either shared or unshared) or remote. It is computed when the buffer is allocated.
- The *Transaction-In-Progress* (TIP) bit indicates a transaction that is in its request phase, *i.e.* that a request has been sent but a response has not yet been received.
- The *Flush* (F) bit indicates that this buffer is logically part of the flush queue. Any data which is present is invisible to the processor. If TIP is clear, then this buffer is transient and scheduled for processing by either the local Memory Machine or the Remote Transaction Machine.
- The *Invalidate* (INV) bit indicates that this address has an outstanding protocol invalidation. It corresponds directly to the deferred invalidate bit that was discussed in Chapter 3 and is used to delay invalidations to locked buffers until after the requesting contexts return for their data. As shown in Section 3.4.1, it is also used to defer premature invalidations which result from reordering in the network.
- The *Transaction-Type* (TT) field consists of two bits. It has three primary states, `RO` (read-only), `RW` (read-write), and `RWD` (read-write-dirty). When TIP is set, this field indicates the type of request which was sent to memory and is restricted to `RO` or `RW`. When TIP is clear (and data present), all three states are legal. `RWD` indicates that the buffer contains data which is dirty with respect to memory. A final state, `BUSY` is only valid with an empty buffer; it indicates that a transaction has been terminated by a `BUSY` message from memory[19].
- The *Valid* (VAL) field contains a two-bit Gray code which indicates the degree to which buffer data is valid: `INVALID`, `ARRIVING`, `HALF_VALID`, and `VALID`. The `INVALID` and `VALID` states are stable, indicating that the transaction buffer has either no data or valid data respectively. The remaining two states are transient. `ARRIVING` indicates that protocol transitions have occurred, but no data has yet arrived. `HALF_VALID` indicates that the first 64-bit chunk of data has been written.
- The *Flush Modify* (FM) field contains a two bit field that provides the faultable flush queue functionality of Section 3.4.2.2. The valid states are `NORMAL`, `FAULTED`, `FORCE`, and `DISCARD`. States other than `NORMAL` are valid only for *local* buffers marked for protocol-level flushing to the local memory. `FAULTED` represents a replacement that has pending software action; `FORCE` and `DISCARD` represent buffers that need post-software cleanup by the hardware.

Table 5-1 illustrates how combinations of these bits form transaction buffer states. This table shows legal combinations only; states which are not shown are illegal. For simplicity, the Valid field is shown with two values, `VALID` and `INVALID`. The transient states of `ARRIVING` and `HALF_VALID` are logically grouped with `VALID`.

[19] This message is a negative acknowledgment on the request. See [25] for more information.

| | Buffer State | | | | | | |
|---|---|---|---|---|---|---|---|
| Note(s) | TIP | F | INV | TT | Val | FM | Description |
| A, G | 0 | 0 | 0 | — | INVALID | NORMAL | Empty and available for allocation. |
| A | | | | RO | VALID | | Read-Only data present |
| A | | | | RW | VALID | | Read-Write data present (clean) |
| A | | | | RWD | VALID | | Read-Write data present (dirty) |
| B | 0 | 0 | 1 | RO | VALID | NORMAL | Read-Only data/pending invalidate |
| B | | | | RW | VALID | | Read-Write data (clean)/pending invalidate |
| B | | | | RWD | VALID | | Read-Write data (dirty)/pending invalidate |
| C, H | 0 | 1 | 0 | RWD | VALID | NORMAL | Flush dirty data (no protocol action) |
| C | 0 | 1 | 1 | RO | INVALID | NORMAL | Send Acknowledgment |
| C | | | | RW | INVALID | | Send Acknowledgment |
| C, H | | | | RWD | INVALID | | Flush F/E bits and write permission |
| C | | | | RWD | VALID | | Flush dirty data and write permission |
| H | 0 | 1 | 1 | — | — | FAULTED | Buffer faulted, awaiting software processing |
| C, H | | | | — | — | FORCE | Flush dirty data if RWD (skip protocol action) |
| C, H | | | | — | — | DISCARD | Discard buffer with no action |
| D | 1 | 0 | 0 | RO | INVALID | NORMAL | Read transaction |
| D | | | | RW | INVALID | | Write transaction |
| D | | | | RW | VALID | | Write transaction/read data valid |
| E | 1 | 0 | 1 | RO | INVALID | NORMAL | Read trans/premature read INV |
| E | | | | RW | INVALID | | Write trans/premature write INV |
| E | | | | RW | VALID | | Write trans/read data valid/prem write |
| F | 1 | 1 | 0 | RO | INVALID | NORMAL | Read trans/flush on arrival |
| F | | | | RW | INVALID | | Write trans/flush on arrival |
| E, F | 1 | 1 | 1 | RO | INVALID | NORMAL | Read trans/flush on arrival/prem read INV |
| E, F | | | | RW | INVALID | | Write trans/flush on arrival/prem write INV |

| Note | Text of comment |
|---|---|
| A | These correspond directly to states of a full-associative cache. |
| B | If unlocked, these states are transient and scheduled for flushing. |
| C | Transient and scheduled for flushing. |
| D | These states are entered at the beginning of transactions. |
| E | Entered by rare protocol actions and network reordering. |
| F | Only entered by execution of a flush instruction during a transaction. |
| G | If TT=`BUSY`, then the previous transaction was terminated by a `BUSY` message. |
| H | For local memory traffic only. |

Table 5-1: Legal combinations of transaction buffer state bits. Combinations of bits that do not appear in this table are invalid.

Before exploring Table 5-1 in more detail, let us examine a simple cache-coherence transaction. Assume that the processor requests a remote data item which is not in the local cache. This causes the *Cache Management Machine* to allocate an empty transaction buffer. It sets the transaction-type field to either `RO` or `RW`, depending on whether the processor has executed a load or store respectively. It also sets the TIP bit to reflect the fact that a request is in progress. Finally, it sends a context-switch fault to the processor while simultaneously transmitting a read request (`RREQ`) or write request (`WREQ`) to the remote node.

When data returns from memory, the *Remote Transaction Machine* gives the address from the returning packet to the associative access port in order to locate the appropriate transaction buffer. Then, this machine clears the TIP bit, places the data into the transaction buffer, and sets the Valid field to `VALID`. This data will now be available the next time that the processor requests it.

The following paragraphs explore different classes of states and their uses. These include states for tracking transactions in progress, states for caching data items, transient states, and states which permit insensitivity to network reordering.

**Transaction in Progress States:** States in Table 5-1 that have their TIP bits set indicate transactions that are in the request phase, *i.e.* transactions for which requests have been sent but data has not yet been received. The presence of such transaction buffers prevents duplicate requests from being sent either by the original context (when it returns to check for data) or by other contexts. States marked **D** in Table 5-1 are entered when a transaction is first initiated. Other transaction in progress states are entered by exceptional events: For instance, states marked **E** are *deferred invalidation* states that arise from network reordering (see Section 3.4.1). States marked **F** arise when one thread executes a "flush" operation on a memory line that another thread has recently requested; such transaction buffers are marked for immediate discard when data returns.

**The Transaction Buffer as a Cache:** States in Table 5-1 that are marked **A** correspond directly to states of a fully-associative cache. For this reason, we refer to these as *cached states*. In the simple transaction model of Chapter 3, these states can be used to hold data which has just returned from memory, i.e. during the window of vulnerability. When the original requesting context returns to look for this data, it can be transferred to the primary cache and its buffer emptied.

However, cached states are far more versatile. Since cache-coherence is fully integrated with the transaction buffer, this data is kept coherent in the same way as the primary cache. Thus, cached states can persist outside of primary transactions causing the transaction buffer to act like a small second-level cache. Consequently, non-binding prefetches simply return their data to the transaction buffer[20]. Further, cache-lines can be victim cached [53] in the transaction buffer; this means that lines that have been replaced from the primary cache are placed into the transaction buffer rather than dumped into the network. This simple behavior is of great value for ameliorating the occasional long access latency caused by LimitLESS cache coherence (see Chaiken [24]). Finally, special "uncached" loads and stores can access data directly in the transaction buffer without first transferring it to the primary cache. As discussed in Section 4.3.7, the ability to perform an "on-the-fly" demotion of cached accesses to uncached accesses is a crucial aspect of providing a local unshared memory that is available under all circumstances.

Use of the transaction buffer as a generic cache has two consequences. First, it requires garbage collection or replacement, since buffers that are in the cached state can persist indefinitely. During periods of insufficient resources, the CMMU considers buffers that are in the cached state and not part of a primary transaction to be *reclaimable*. Garbage collection is performed on reclaimable buffers by the Remote Transaction Machine.

Second, the issue of duplicate data between the cache and transaction buffer must be addressed. The buffer maintenance policy ensures that the only possible duplication is the simultaneous existence of a read-only copy in the primary cache and a clean read-write copy in the transaction buffer. Since the protocol guarantees that these two memory-lines will have identical data, no coherence problem ensues[21]. As soon as the data is written by the processor, the read-only copy is invalidated or overwritten.

[20] In fact, the only difference between transaction buffers used for primary and secondary transactions is that primary transactions have live tracking vectors pointing at them. See Section 5.2.3.3.

[21] This situation is allowed because the coherence protocol explicitly checks to see if a requester of write permission is already one of the readers; if so, it will not invalidate the requester's copy before granting write permission.

**Transient States:** States in Table 5-1 that are marked **B** and **C** are transient, *i.e.* scheduled to be processed and emptied by either the Memory Management Machine or the Remote Transaction Machine, depending on whether the Local bit is set or clear. Section 5.2.3.4 discusses the mechanism behind this scheduling. The one exception to immediate scheduling is that buffers in states marked **B** can be protected by buffer locks (from the *thrashlock* mechanism of Section 3.2.). Such buffers remain unmolested in the transaction buffer until their locks are released, at which point they become scheduled for processing. Buffer locks are discussed in Section 5.2.3.3.

Transient buffers which have their INV bits set are implicit protocol messages. Since protocol action occurs only on shared memory lines, these buffers must have global addresses. Those with types `RO` and `RW` are invalidation acknowledgments, while those with type `RWD` are updates. Thus, when a buffer of this nature is processed by the memory management hardware, it invokes protocol actions as if the local processor had marshaled a protocol packet and sent it through the network to the local memory. Similarly, when the Remote Transaction Machine processes such a buffer, it generates an appropriate protocol message, destined for the home node of the buffer's address, and sends it into the network.

As discussed in Section 3.4.2.2, the transaction buffer serves as the "flush queue" to local memory. The cache management machine replaces local data items from the processor cache by writing them into transaction buffers with their Flush bits set. Additionally, replacements which require protocol action (*i.e.* global addresses) have their INV bits set. A similar function is performed by the Remote Transaction Machine for remote data items[22].

The use of transient states in this way has three advantages over a more conventional FIFO queue. First, memory can process flushed cache-lines out of order during network overflow recovery. Since network congestion may prevent the processing of global data items, it can process local unshared data replacements and thus guarantee that the overflow recovery handler can execute. Second, the difference between transaction buffers which are on the flush queue and those which are not is a matter of state; in fact, locked transient items become scheduled for flushing at the time that their locks are released. Consequently, items which are cached in the transaction buffer can be discarded by setting their Flush bits. The Remote Transaction Machine uses this method to discard buffers during garbage collection.

Finally, flush entries that incur software exceptions (*i.e.* that require software processing to complete) can be held in the transaction buffer for further processing. A flush buffer that is held in this way is marked with a *flush-mode* (FM) state of `FAULTED`. Such a buffer will remain in this state indefinitely until software acts upon it; it is thus not a transient state. The existence of one or more buffers in a `FAULTED` state flags a high-availability `flush_fault` interrupt.

#### 5.2.3.2 Transaction Buffer Associative Match

The associative access port of the transaction buffer supports a parallel search for buffers. It is the presence of the associative match that makes the transaction buffer a powerful mechanism for tracking transactions. This port is shared between the Cache Management Machine and the Remote

[22] When remote data items are replaced from the cache, they can either be placed into the transaction buffer or sent directly into the network. This choice depends on whether or not victim caching is enabled.

Transaction Machine[23]; this is the reason that there is only one network output queue between these two machines: possession of the transaction buffer is necessary in order to initiate transactions. The matching address satisfies one of the following conditions (in order of precedence):

1a) $\text{TIP} = 1$
1b) $(\text{TIP} = 0) \wedge (\text{Flush} = 0) \wedge (\text{Valid} \neq 0)$
2) $(\text{TIP} = 0) \wedge (\text{Flush} = 1)$

If multiple buffers match, the one with the highest priority is the one which is returned[24]. Buffers of type (1a), with TIP = 1, correspond to active transactions in progress. Buffers of type (1b) are cached buffers with valid data which is visible to the processor. Buffers of type (2) are transient and represent return messages (either `ACK` or `UPDATE` messages).

The Cache Management Machine and coherence protocol maintain two invariants which permit a unique buffer to be located for each address. The first invariant is that there can be no more than one buffer which falls in the combined categories of (1a) and (1b). Such buffers hold processor-side state for transactions which are either in the request phase (1a) or window of vulnerability (1b). The second invariant is that there will be no more than one buffer in category (2). Such buffers are implicit return messages (acknowledgments or updates); consequently, this second invariant is merely the *at most one* invariant for return messages discussed in Section 3.4.1. Matches of type (2) correspond to the associative matching facilities often included in uniprocessor write-buffers. They facilitate the location of unshared dirty data during the initiation of new requests to local memory; without this ability, local memory operations could return incorrect results and local shared-memory operations might have to invoke unnecessary invalidations[25].

Given these two invariants, associative matching is straightforward. Each transaction buffer has an associated comparator which is used to compare its address with the requested address. Then, the result of this match is combined with a small amount of logic to produce two signals per buffer: one that indicates a match of type (1a) or (1b) and one that indicates a match of type (2). The set of match signals are then combined to produce a unique matched buffer.

On each cycle, the machine which has successfully arbitrated for the associative matcher may also choose to modify state in the transaction buffer. It may select one of two buffer destinations: either the next empty buffer or the current matched buffer. The transaction monitor, described in Section 5.2.3.4 manages empty buffers.

#### 5.2.3.3 Thrashlock Monitor

The function of the *Thrashlock* monitor is to provide support for the thrashlock mechanism. Its keeps track of per-context state information which aids in the recognition and resolution of thrashing scenarios. As shown in Figure 5-13, the thrashlock monitor takes as input information about the current access; this information includes the current context number (0 – 3), the current access

[23]While satisfying a memory request from the local processor, the Memory Machine passes its address to the Remote Transaction Machine, which proceeds to locate the pending transaction buffer and perform the protocol state transitions. This localizes the data response protocol transitions entirely within the Remote Transaction Machine.

[24]An associative matching option which is provided by the CMMU but not discussed above, is the ability to ignore buffers which have TIP set but do not have data. This is useful for "locally coherent" DMA operations.

[25]The result of such a match is sent with requests to local memory.

type (instruction or data), and whether or not context switching is enabled. It combines these pieces of information with internal state and the current associative match to produce the THRASHDETECTED signal, which is used to indicate that context switching should be avoided. It also takes two THRASHCONTROL signals, called CLEARTHRASHINFO and UPDATETHRASHINFO to decide how to alter the subsequent state of the Thrashlock monitor. This interface will be described later, after the introduction of the internal state of the Thrashlock Monitor, namely the *tracking vectors*.

**Tracking Vectors:** Figure 5-15 shows the state which is maintained by the thrashlock monitor. It consists of eight *tracking vectors*, corresponding to one primary instruction transaction and one primary data transaction for each of four hardware contexts. These tracking vectors are responsible for associating specific primary transactions with transaction buffers. In a sense, tracking vectors reflect the pipeline state of the processor by tracking the state of its primary accesses. It is because of this close coupling between tracking vectors and the pipeline that care must be taken to interpret the stream of access requests from the processor in a fashion which guarantees forward progress.

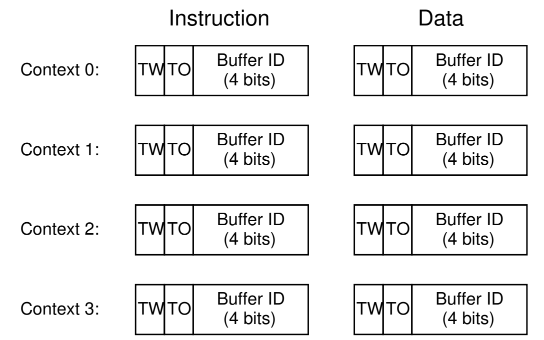


Figure 5-15: Tracking vectors for implementing the thrashlock mechanism.

Each tracking vector has two control bits and one four-bit pointer. The two control bits are the *tried-once* bit (TO) and the *thrash-wait* bit (TW). Both of these were introduced in Section 3.2.7, during the discussion of the Thrashwait mechanism. When a tracking vector has its tried-once bit set, this state indicates that a primary transaction has been initiated and that the identity of the associated transaction buffer resides in its four-bit pointer. Such vectors are considered *live* and the transaction buffers which they point at are denoted *primary transaction buffers*. The thrash-wait bit indicates that thrashing has been detected on this primary transaction. It is set whenever thrashing is detected and cleared whenever the corresponding primary transaction completes successfully.

Thus, to initiate a new primary transaction, the following operations must be performed:

1. Allocate an empty transaction buffer from the pool of free buffers.
2. Set the address and state of this buffer to reflect the transaction. This involves setting the TIP bit and the transaction type.
3. Perform any external action (such as sending a request message).
4. Record the transaction buffer ID in appropriate tracking vector and set its TO bit.
5. Finally, if thrashing was detected on a previous incarnation of this primary transaction, set the TW bit.

These operations occur in parallel in the CMMU. To permit thrash detection across interrupts, tracking vectors *must not* be altered by accesses to local, unshared memory. Note that there are a couple of additional pipelining issues with respect to updating these vectors that arise because they are not directly integrated with the integer pipeline; for more information on this, see [62].

If either of the TW bits for the current context is asserted, then the processor pipeline is frozen with context-switching disabled. This policy makes thrash-detection persistent across the servicing of high-availability interrupts. Since primary transactions are initiated in a unique order (*e.g.* instruction then data), thrashing on primary transactions which are later in this ordering will force context-switching to remain disabled even when transactions which are earlier in the ordering have to be retried. For example, when the processor is spinning on a thrashing data access and is interrupted to service a high-availability interrupt, it will return to refetch the interrupted instruction *with context-switching disabled*. This, behavior, along with the buffer locks described below, guarantee that the latest thrashing transaction will eventually complete. Since completion of a given primary transaction permits the initiation of the next primary transaction, all primary transactions eventually complete and forward progress follows.

**Buffer Locks:** The Thrashlock algorithm requires as many buffer locks as simultaneous primary transactions. Recall from Section 3.2.7 that a buffer is locked during the thrashwait phase to prevent it from being lost to invalidation. These locks are activated whenever thrashing is detected on a primary transaction and are deactivated whenever the transaction is successfully completed by the processor. However, the thrash-wait bits behave in exactly this way. From these bits, the CMMU generates a 16-bit vector of locks, called the NOINVALIDATE vector. A bit in this vector is set if the corresponding transaction buffer has a live tracking vector whose TW bit is asserted.

The NOINVALIDATE vector has two functions. First, it causes invalidation requests for locked buffers to be deferred. This deferral is accomplished by returning an appropriate lock bit with the results of an associative match, so that the state machines which are processing invalidation messages can make the appropriate state transitions.

Second, the NOINVALIDATE vector is used to prevent buffers with deferred invalidations from being visible to those machines which would normally dispose of them. As shown in Table 5-14, there are three transient states in which INV is set and both TIP and Flush are clear. These states are for buffers which have data (in their windows of vulnerability) and which also have pending invalidations. Normally, they are transient, i.e. they immediately generate acknowledgments or updates and become empty. If locked, however, they must remain as they are, waiting for an access from the requesting context. The transaction monitor, described in Section 5.2.3.4, is responsible for scheduling transient states in accordance with current buffer locks.

Note that Section 3.2 showed that multiple buffer locks can lead to deadlock. However, the Thrashlock Monitor described herein has *eight* buffer locks! This large number of locks is not a problem as long as interrupt handlers adhere to the following constraints:

1. Interrupt handlers must return to the same context that they interrupted.
2. Global memory accesses must be from a context other than the interrupted context.
3. Global memory accesses in trap handlers must be to variables which will not thrash.

The second condition ensures that the tracking vectors of the interrupted context are not affected by the interrupt handler. This occurs naturally if the software reserves a hardware context for rapid processing of messages. The first and third conditions restrict buffer locks to a single context, since context switching is disabled when locks are active. Note, in particular, that this is a relaxed form of the constraints presented in Section 2.5.2, which simply disallowed global accesses when locks are present (automatically satisfying condition (3) above).

Unfortunately, for user-level interrupt handlers, we are unable to "trust" the user to avoid incorrect use of shared memory. Hence, the presence of a buffer lock in an interrupted context is treated as a form of priority inversion: the user is allowed to execute the atomic section of a user-level handler, as long as they do not touch shared memory. When they exit the atomic section, the atomicity extension mechanism briefly returns to the interrupted context to finish the locked transaction (see Section 2.4.5 for a discussion of the atomicity-extension mechanism). If, on the other hand, the user touches shared memory in the atomic section, this will invoke two-case delivery (this is normal behavior during a user-level atomic section; see discussion in Section 4.3.5).

**Protection of Primary Transaction Buffers:** During the period in which a tracking vector is live, the buffer which it points to is an integral part of the ongoing primary transaction. Consequently, this buffer must be protected against garbage collection and reallocation, even if it should become empty before the requesting context returns to examine it. This protection is accomplished through the 16-bit NORECLAIM vector — a bit in this vector is set if the corresponding transaction buffer is pointed at by a live tracking vector. The NORECLAIM vector is passed to the transaction monitor, which invokes garbage collections and chooses empty buffers for reallocation.

Protection of primary transactions in this way has a number of advantages. First, transactions which have entered their window of vulnerability (i.e. have data cached in a transaction buffer) are not aborted by the garbage collector. Without this protection, important cached buffers would be considered reclaimable.

Second, premature lock release can be detected. Here it refers to a particular thrashing scenario in which a context (which we will call the *requesting context*) initiates a transaction to a memory-line, which is subsequently satisfied and invalidated. Later, a new transaction is started for the same memory-line by another context. From the standpoint of the requesting context, this is a thrashing situation, since the data has come and gone. Unfortunately, when the requesting context returns, it performs an associative lookup and discovers a transaction buffer for the desired address with TIP set. Without some additional mechanism, there would be no way to distinguish this situation from the one in which the original transaction was long-lived. However, by protecting the original primary transaction buffer from reallocation, we can ensure that the new transaction resides in a different buffer. Consequently, this situation can be detected.

Finally, protection of primary transaction buffers allows us to flag transactions which are aborted by `BUSY` (or NAK) messages: we simply mark them as empty, with the special transaction type of `BUSY`. Such transactions are not considered to be thrashed. This is appropriate since `BUSY` messages are generated during periods in which the memory is sending or waiting for invalidations, which can last for a number of cycles. If thrashing were detected on busied transactions, it would be possible for the cache system and memory system to get locked into a cycle of sending requests and `BUSY` messages[26]. It is much better to conserve memory and network bandwidth by making a single request and switching to another context.

[26]Note that the presence of `BUSY` messages impacts forward progress at the memory side (See Section 3.3).

```
THRASHDETECTED =
    CONTEXT_SWITCHING_DISABLED ∨
    THRASH_WAIT_Data ∨ THRASH_WAIT_Inst ∨                       // Thrashing previously detected
    (TRIED_ONCE ∧ ¬ MatchedTXB ∧ TT(POINTER) ≠ BUSY) ∨          // Inter-processor thrashing
    (TRIED_ONCE ∧ MatchedTXB ∧ ID(MatchedTXB) ≠ POINTER)        // Premature lock release
```

Figure 5-16: Composition of the THRASHDETECTED signal. MatchedTXB signals a matched transaction buffer; ID(MatchedTXB) is its index. POINTER and TRIED_ONCE are fields from the active tracking vector; TT(POINTER) is the trans-type of buffer POINTER; and $\text{THRASH\_WAIT}_{Data}$ and $\text{THRASH\_WAIT}_{Inst}$ are from the current context.

**Thrash Detection and the Processor Interface:** We are now in a position to discuss the interface between the thrashlock monitor and the cache management hardware of the CMMU. The current context number and access type are used to select an appropriate tracking vector. Call this the *active* tracking vector. Then, based on this and on information from the associative matching circuitry, the thrashlock monitor computes the THRASHDETECTED signal, as shown in Figure 5-16.

Two operations that can be used to manipulate the thrashlock monitor are the UPDATETHRASHINFO and CLEARTHRASHINFO signals. These operations are defined as follows:

```
UPDATETHRASHINFO ⇒
  THRASH_WAIT ← THRASHDETECTED
   TRIED_ONCE ← 1
      POINTER ← WRITETXB

CLEARTHRASHINFO ⇒
  THRASH_WAIT ← 0
   TRIED_ONCE ← 0
```

Here, THRASH_WAIT (with no subscript) is either $\text{THRASH\_WAIT}_{Data}$ or $\text{THRASH\_WAIT}_{Inst}$ depending on the type of the current access. Further, WRITETXB denotes the transaction buffer which the processor is about to modify this cycle; WRITETXB is either an empty buffer or the current associatively matched buffer.

```
DO_GLOBAL_PROCESSOR_REQUEST(Address,Context,Type)
if (cache-hit for Address[Type]) then
    assert CLEARTHRASHINFO
    return READY
elsif (TXB-hit for Address[Type]) then
    fill cache from transaction buffer
    clear transaction buffer
    assert CLEARTHRASHINFO
    return READY
else // Data not currently available
    if (¬MatchedTXB) then
        send RREQ or WREQ
        allocate new transaction buffer (NewTXB)
        TT(NewTXB) ← Type, TIP(NewTXB) ← 1
    elsif (¬TIP(MatchedTXB)) then
        // Insufficient access (TT = RO, Type = RW)
        send MREQ // Request RW copy
        TT(MatchedTXB) ← RW, TIP(MatchedTXB) ← 1
    endif
    assert UPDATETHRASHINFO
    if (THRASHDETECTED) then return WAIT
    else return SWITCH
endif
```

Figure 5-17: Data Access with the ThrashLock Algorithm

Finally, an updated version of the processor-access pseudo-code of Section 3.2.7 (Figure 3-11) can now be constructed. Let TIP(MatchedTXB) be the TIP bit from the matched transaction buffer. Then, a processor access is as shown in Figure 5-17. This pseudo-code is the original ThrashWait algorithm, updated with ThrashLock semantics and targeted at the Tracking Vector implementation described in this section.

#### 5.2.3.4 Transaction Monitor

The transaction monitor, shown in Figure 5-13, is responsible for the allocation and management of transaction buffers. It takes state information from each of the 16 buffers and combines it with the NORECLAIM and NOINVALIDATE vectors (see Section 5.2.3.3) to perform a number of supervisory tasks. These are listed below:

**Buffer Allocation Constraints:** Since transaction buffers are a limited resource in the CMMU, some constraints must be placed on their allocation. Two independent (and overlapping) counts of transaction buffers are generated. The first, TRANSACTIONCOUNT, is a combined count of outstanding transactions and cached data. Placing a limit on this count forces a tradeoff between these two buffer classes. The second, GLOBALCOUNT, is used to limit the number of buffers which might be indefinitely occupied during network overflow.

Let GLOBALBUFFER be true if a transaction buffer has a global address. Let RECLAIMABLE be true if a buffer is not protected by the NORECLAIM vector. Then, two allocation invariants which are maintained by the cache management machine can be defined as follows:

$\text{TRANSACTIONCOUNT} \leq 12$ Where TRANSACTIONCOUNT is the count of buffers for which:
$(\text{GLOBALBUFFER} \wedge \text{TIP} = 1) \vee$
$(\text{Flush} = 0 \wedge \text{INV} = 0 \wedge \text{Valid} \neq 0) \vee$
$\neg\, \text{RECLAIMABLE}$

$\text{GLOBALCOUNT} \leq 14$ Where GLOBALCOUNT is the count of buffers for which:
$(\text{GLOBALBUFFER} \wedge \text{TIP} = 1) \vee$
$(\text{GLOBALBUFFER} \wedge \text{INV} = 1) \vee$
$(\text{GLOBALBUFFER} \wedge \text{Flush} = 0 \wedge \text{Valid} \neq 0) \vee$
$\neg\, \text{RECLAIMABLE}$

Note that these invariants are different from the constraints on the number of buffers with identical addresses which were discussed in Section 5.2.3.2. These allocation invariants are maintained by the cache system since:

1. The cache management machine never allocates buffers that violate them.
2. Both are monotonically decreasing with respect to non-allocation transitions[27].

While the actual limits (12 and 14) are somewhat arbitrary, the important orderings are given in parentheses. The fact that the maximum GLOBALCOUNT is larger then the maximum TRANSACTIONCOUNT insures that buffers can be available for flushes to global memory, even when the maximum TransactionCount has been filled with global buffers. The fact that the maximum GLOBALCOUNT is less than the total number of buffers ensures that buffers will always be available for accesses to unshared local memory; when the maximum GLOBALCOUNT has been reached,

[27] Actually, the TRANSACTIONCOUNT limit may be violated briefly by one transition: unshared transaction in progress $\rightarrow$ unshared cached. This causes the TRANSACTIONCOUNT to increase, since unshared transactions are not counted. However, unshared cached buffers can always be garbage-collected; in fact, the invariant will be immediately restored by the garbage-collector.

the remaining buffers must be flushes without protocol action (which can always be completed), unshared cached values (which can be garbage-collected and flushed), or unshared transactions (which can be completed locally).

To expedite the process of maintaining the invariants, the transaction monitor produces two boolean signals, TRANSFULL and GLOBFULL which are defined as follows:

$$\text{TRANSFULL} \Rightarrow (\text{TRANSACTIONCOUNT} \geq 12)$$
$$\text{GLOBFULL} \Rightarrow (\text{GLOBALCOUNT} \geq 14)$$

These signals are asserted whenever their corresponding invariants are in danger of being violated. As an example of their use, when the Cache Management Machine is about to initiate a new remote transaction on behalf of the processor, it checks to make sure that *both* TRANSFULL and GLOBFULL are deasserted (the new buffer would fall into both categories). If not, it asserts the processor hold line and enters the `RESOURCE_WAIT` state until resources become available.

**Allocation of Empty Buffers:** Those buffers that are in the empty state and not protected by the NORECLAIM vector are candidates for allocation. The transaction monitor generates a 16-bit vector with one bit set for each allocatable buffer. Combining this vector with a base pointer, it chooses the next available empty buffer in round-robin fashion. This buffer may be written (allocated) by the state machine that has control of the associative matching port. Only the cache management hardware actually allocates new buffers. To indicate that free buffers are available, the transaction monitor generates a signal called HAVEEMPTY.

**Identification of Buffers for Garbage Collection:** As mentioned in Section 5.2.3.1, use of the transaction buffer as a generic cache requires some form of garbage-collection. Otherwise, cached states quickly consume all available resources. Those buffers that are cached (class **A**) and not protected by the NORECLAIM vectors are considered *reclaimable*. The transaction monitor generates a 16-bit vector that has one bit set for each reclaimable buffer. As with empty buffer allocation, it combines the vector of reclaimable buffers with a base pointer[28] to choose the next candidate for garbage-collection. This information is passed to the Remote Transaction Machine which will perform the actual garbage-collection when resources are low.

Invocation of the garbage-collector is straightforward: the TRANSACTIONCOUNT invariant forces a trade-off between transactions in progress and cached buffer entries. Thus, whenever TRANSFULL is asserted, the remote-transaction machine checks to see if any reclaimable buffers exist. If so, it takes the next candidate from the transaction monitor and reclaims it. Buffers with transaction-type `RO` are discarded. Buffers in states `RW` or `RWD` are transformed into transient states by setting their Flush bits; the INV bits of global buffers are set as well. These buffers then become transient and are scheduled as described in the next section.

**Scheduling of Transient States:** As discussed in Section 5.2.3.1, some transaction buffer states are transient, namely those marked **C** in Table 5-1 and those marked **B** which are not locked.

[28]This is actually the same base pointer as is used for empty buffer allocation. Empty buffers are allocated a fixed number of buffers ahead of where garbage-collection is occurring.

The transaction monitor combines the NOINVALIDATE vector, described in Section 5.2.3.3, with transaction state to produce two classes of transient buffers: those with their Local bits set and those with their Local bits clear. Those with their Local bits set have local addresses and are processed by the memory management machine. Those with their Local bits clear are remote addresses and are processed by the Remote Transaction Machine. These two classes are scheduled separately, in round-robin fashion. Each machine is informed when it has one or more transient buffer to process (signals HAVEREMOTEFLUSH and HAVELOCALFLUSH in Figure 5-13) and is informed of which is the next buffer to process.

The transaction monitor has one addition feature which modifies the scheduling of local buffers during network overflow. To process transient buffers which are implicit protocol messages (global buffers), the memory machine may need to send messages. Consequently, during periods of scarce network resources, it may be impossible to process such buffers at all[29]. Thus, to guarantee that the processor has access to unshared portions of memory, the scheduling of local buffers is modified during network overflow to ignore transient buffers with global addresses. Thus, unshared flushes can always proceed.

#### 5.2.3.5 Transaction Data

Although we have not said much about the transaction data module, we would like to say a few words at this time. The transaction data module contains sufficient storage for one complete, four-word memory-line per transaction buffer. The physical memory cell supports 64-bit data paths, dividing each 128-bit memory line into two 64-bit chunks. This is matched to the fact that the memory bus and internal network queues are also 64 bits. As we have noted, the transaction buffer serves as an extremely flexible interface between the processor, memory, and network. One of the goals that impacted the design of the transaction buffer was a desire for it to serve as a conduit for *all* data accesses, including unshared instruction cache misses to local memory; as such, it should introduce little or no delay over a more direct, hard-wired path.

This goal was accomplished by pipelining data directly through transaction buffers. In the CMMU, pipelining of transaction data is accomplished by three things:

- Separate read and write ports, permitting data to be read at the same time that it is being written. In fact, the CMMU employs a three-port memory file for transaction buffer data with two read ports and one write port. One of the read ports is dedicated to the memory management machine for processing flushed buffers.
- Bypass logic to permit data to be read on the same cycle that it is written.
- A per-buffer Valid field which is a two-bit Gray code. The four Valid states, `INVALID`, `ARRIVING`, `HALF_VALID`, and `VALID` were introduced in Section 5.2.3.1. Logic to modify and examine the Valid bits is integrated directly into the data access ports of the transaction buffer.

[29] As discussed in Section 5.2.4, the Memory Machine always verifies that it has sufficient resources to finish a request *before* starting to process it.

The key to pipelining is in the Valid bits. The `ARRIVING` state indicates that protocol action has completed and that data arrival is imminent. This is used to initiate scheduling of any entity that may be sleeping in wait for data. The `HALF_VALID` state indicates that the first of the two 64-bit sub-lines has valid data, while `VALID` indicates that both lines are valid. Two transitions, (`ARRIVING` → `HALF_VALID`), and (`HALF_VALID` → `VALID`), are invoked directly by writes to the transaction data.

In the next section (Section 5.2.4) we introduce *service coupling*, an implementation methodology that allows the Alewife CMMU to takes advantage of the transaction-buffer pipeline to achieve extremely low-latency local memory access.

### 5.2.4 Service Coupling and Memory Scheduling

The Alewife memory system must handle a diverse set of requests: cache-coherent shared memory, message passing with locally-coherent DMA, and unshared memory access. Each of these categories of communication place different demands on the memory system. Ideally, we would like to find a way to support these different mechanisms without losing efficiency in terms of achievable latency or bandwidth.

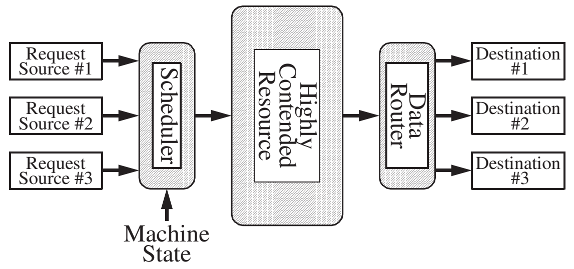


Figure 5-18: Use of service coupling to maximize utilization of a highly-contended resource. Basic components are stream-based *scheduling*, *processing*, and *delivery*.

In this section, we would like to introduce a simple implementation technique, called *service coupling*, that gives us the flexibility to smoothly integrate all of the Alewife communication mechanisms while giving up very little in terms of performance. As shown in Figure 5-18, the key insight behind service coupling is that we can achieve maximal utilization of some highly contended resource (in this case the DRAM system) by transforming all requests for that subsystem into *streams* of requests that eventually yield streams of responses. The components of a service-coupled resource are, therefore:

1. A *scheduler* that takes as input several distinct streams of requests, combines these through some appropriate scheduling criteria, and produces a single multiplexed stream of requests. Note that, although the source of a request is coded in the output stream, this information might be examined only by the output router or ignored entirely.
2. A *stream-based resource* that takes a stream of requests and produces a stream of responses.
3. A *router* that takes as input a multiplexed stream of responses and demultiplexes these to a set of destination streams. Note that the set of response destinations is not necessarily the same as the set of request sources.

There are a number of things to note about this. First, stream-based processing can offer more efficient packing of cycles than other options, since it offers a "future look" at the next operation in the request stream. Second, stream-based processing naturally leads itself to pipeline optimizations: the results of an operation is a serial stream of data that lends itself to handling in piecemeal fashion. Finally, by treating all types of operations in the same way (shown by the fact that all operations stream through a single resource processor), we can remove "irrelevant" complexity from the act of processing requests (having to do with the destination of the request); in Alewife, this is particularly important in that the destination of a cache-coherence packet may be either remote or local, with appropriate pipelining handled by the output router, *not* the protocol processor. In many cases, this allows us to make all operations equally efficient[30]. Note that, in order to use service coupling, all requests must be cast in a way that resembles a stream input, whether or not they come from queues. Similarly, all destinations must accept stream-like outputs, regardless of the actual format of the destination (*e.g.* network queue, transaction buffer, cache line, *etc.*).

**Scheduling:** The service-coupling framework is utilized in the Cache Management Machine (CMM), the Remote Transaction Machine (RTM), and the Memory Management Machine (MMM) of the Alewife CMMU because it offers a clean framework for satisfying the multiple demands of an integrated architecture. One of the consequences of separating operations into the phases of scheduling, processing, and routing is that it highlights scheduling as an important task. Both the CMM and MMM schedule requests in a way that guarantees *atomicity of operations* and *fairness of access*. We would like to discuss these briefly.

The first of these, atomicity of operations, refers to the guarantee that all operations have sufficient resources to complete before they begin execution. As discussed in Section 4.3.7, atomicity of operations is very important in a machine that supports two-case delivery, since the memory system must never lock up awaiting network queue space. In Alewife, this takes the form of a set of conservative heuristics that guarantee that requests can complete before they pass through the scheduler. For example, all accesses to shared memory are assumed to require sufficient network output queue space to support the maximum output traffic (which, on the memory side, is five outgoing invalidations, encoded in a special format that is unpacked by the output hardware). In addition, depending on which of the input streams was a source of a given request, other requirements may be necessary: for instance, memory requests from the network input queue require space for one invalidation on the local memory-to-processor invalidation queue. These scheduling heuristics must be conservative enough to guarantee atomicity, but not so conservative that they introduce deadlock. Input streams that that pass their resource requirements are considered "schedulable", *i.e.* are potential sources for the next request. Note that the filtering of requests for those that have sufficient resources to complete has the important side-benefit of guaranteeing the highest possible utilization of a contended resource (such as the Memory Management Machine) by greatly limiting (or removing) the amount of time that it wastes to interlock cycles; in other words, this guarantees the lowest possible "occupation" of the contended resource.

The second guarantee, namely fairness of access, addresses forward progress concerns. Its basic tenet is to ensure that all sources of requests eventually make forward progress. In Alewife, both the CMM and MMM employ schedulers that provide "essentially" LRU scheduling of streams. This means that, aside from overrides to ensure that the local processor makes forward progress

[30] Of course, if we are not careful, we could make all operations equally inefficient.

as quickly as possible, these schedulers select the oldest input stream from the set of schedulable streams. This is done with a hardware priority structure that takes into account the set of required resources and the current position in the LRU priority to decide which stream should execute next. Note that to ensure forward progress, this must be done in a way that prevents streams from "stealing" resource from one another. For instance, if stream A needs resources X and Y, but stream B needs only X, the scheduler will prevent stream B from executing if it is of lower LRU priority (has executed more recently) than stream A. It is here that the designer must be careful to avoid cases in which this type of "fair" scheduling prevents forward progress (*e.g.* a situation for which resource Y never becomes available until stream B executes).

Thus, within the service-coupling framework, scheduling has a pivotal role. Assuming that resource requirements are properly formulated and that sources of requests are properly cast as input streams, this type of scheduling can do much toward maximizing utilization of contended resources. The Alewife Memory Management Machine supports six different sources of requests: the processor request queue, the directory operation queue[31], the network protocol input queue, the transaction buffer flush-queue, and the two network DMA engines.

**Data Routing:** The metaphor for the demultiplexing stage of service coupling is that of network routing; this is not accidental. By separating the routing of results from the actual production of results, we can greatly simplify the core processing operation. A good example of this process is the routing of cache-coherence packets within the CMMU. The Memory Management Machine can simply generate a result (such as a response to a read request) without worrying about whether or not this result is destined for the local processor. The design methodology that this engenders views result streams and pipelining as natural and fundamental. Thus, everything that is downstream from the router (in this example, either the transaction buffer or network) can make maximal use of pipelining of results.

The basic mechanism of routing is much like a wormhole-routed network: when the "header" of a particular operation first arrives (this is the first piece of data to emerge from the resource processor in response to a request), it is used to latch in a particular destination data path, much like throwing a switch on a train track. In some cases, this routing decision can be quite simple: for instance, cache-coherence packets with destinations other than the local node are clearly destined for the network output queue; for these cases, the data switch is thrown so that subsequent data items get directed into the network. In contrast, the routing decision *can* be more complex such as for the case of a local memory request. When the Memory Machine is satisfying a local request, the *Remote Transaction Machine* is actually consulted by the routing hardware to discover which transaction buffer is appropriate to fill. Once the correct buffer is discovered, the switch is thrown so that data fills the appropriate transaction buffer. One particularly amusing (and simple) example of routing exists in the Cache Management Machine: if a particular processor write operation is "squashed", then the destination is set to DEVNULL, causing data to be tossed away.

One important point to stress is that results are not necessarily routed to the same functional block that generated the original request. In the CMMU, for instance, a data replacement that came from the transaction buffer (as a transient state) might generate an output data message that is sent

[31]A special, out-of-band interface for handing directory operations up to the processor. This is in support of the `rldir` and `wudir` operations discussed in Section 3.4.2.1.

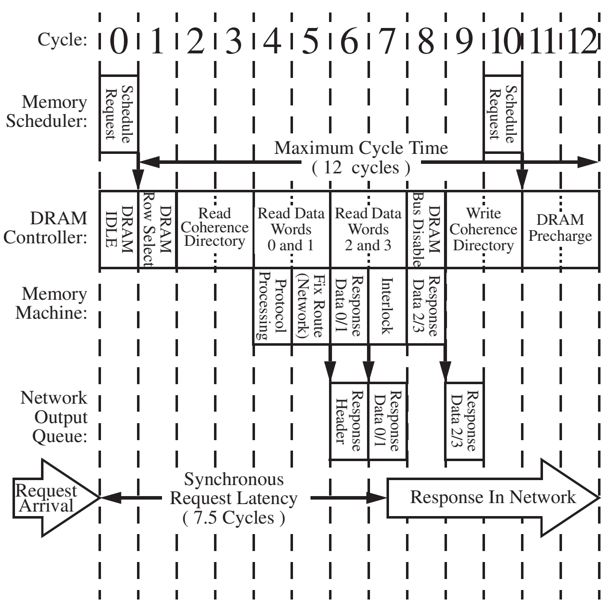



Figure 5-19: Memory-side scheduling of a remote cache-coherence request. The assumption is that the Memory Machine is idle when the request first arrives.

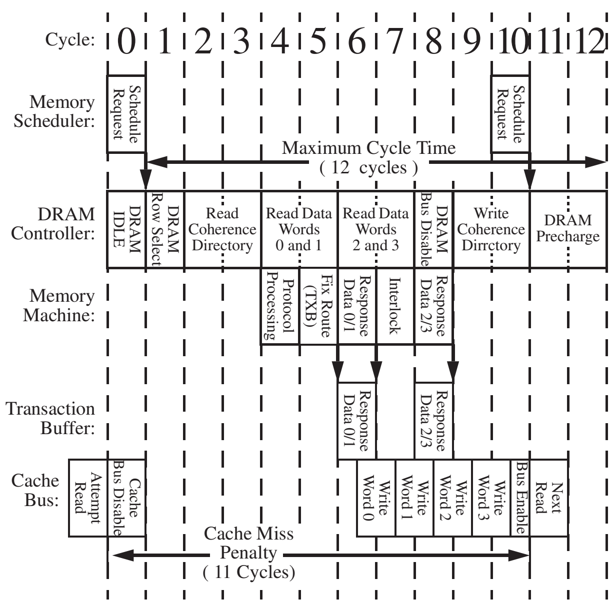



Figure 5-20: Memory-side scheduling of a local cache-coherence request. Timing is given along the bottom for the resulting cache fill (assuming memory initially idle).

to the network; this example might arise as the second half of a data invalidation that resulted from a remote request for data that is dirty in the local cache. The Alewife Memory Management Machine supports four different output destinations: the transaction buffer, the local invalidation queue, the network output queue, and the IPI Local Queue.

**The Myth of Expensive Remote Access:** To give a more concrete example of the benefits of service coupling, we would like to tackle a myth. That myth is the notion that remote memory access must necessarily be expensive. A number of architectures seem to have accepted that a 10 to 1 (or higher) ratio of remote to local access latency is inevitable (for instance, Dash [70]). Unfortunately, such high remote to local access latencies greatly limit the maximum parallelism that can be exploited by applications. In fact, with service coupling, this does not have to be true: Alewife supports a 3 to 1 ratio. Figures 5-19 and 5-20 illustrate this point. These figures show memory scheduling for two "similar" operations that might be considered quite different in less integrated architectures: the satisfying of a remote cache-coherence request and the handling of a local cache miss. The top three rows of these figures, showing the behavior of the memory scheduler, DRAM controller, and Memory Management Machine, are *identical* for both of these operations. Because this handling is identical, we can afford to invest more design effort optimizing the DRAM interface. The bottom two rows of these figures show the "routing" aspects of service coupling, *i.e.* the consumers of transaction results.

To briefly describe these figures, note that we start with the assumption that memory is initially idle. The beginning of Cycle 0 marks the arrival of either a remote memory request (Figure 5-19) or a local cache-miss (Figure 5-20). After one cycle of scheduling (assuming that sufficient resources exist to satisfy the request), the DRAM Controller/Memory Management Machine is invoked. In the service-coupling model, these two interlocked machines are the stream-based resource that we are multiplexing[32]. As shown by these figures, the DRAM controller is busy for the next 10 cycles handling directory reads/writes and data reads for the local request. Every possible DRAM cycle is utilized here. At the end of Cycle 10, the scheduler is consulted again for the next request (if there is one). Assuming that the next request references a different DRAM page than the current one, Cycles 11 and 12 serve to precharge the DRAM (so that Cycle "13" could begin the DRAM Row select for the next request). This gives a maximum DRAM cycle time of 12 cycles; less would be possible if successive requests were to the same DRAM page. Note that references to local unshared memory would shorted this process by an additional 5 cycles, eliminating two cycles of cache-coherence directory read, two cycles of directory write, and one bus turn-around cycle.

Note that the cache-coherence directory becomes available at the end of Cycle 3. In Cycle 4, therefore, the Memory Management Machine performs the cache coherence actions, producing the output header. As a result, Cycle 5 is the cycle in which we perform the data routing operation, based on the result header produced at the end of Cycle 4. In Figure 5-19, the data route gets fixed to the network output queue; allowing for one cycle of pipeline in the network queues, we see that the first output flit starts entering the network in Cycle 7 (since this is an asynchronous network, no timing boundaries make sense after that).

In Figure 5-20, the data routing cycle performs a bit more complicated routing operation. At the end of Cycle 4, a "heads-up" signal is sent to the Remote Transaction Machine, requesting that it preferentially service the Memory Management Machine if possible. Assuming that everything works out, the Remote Transaction Machine will perform an associative lookup on the address during the routing cycle (Cycle 5), returning a transaction buffer identifier to the Memory Management Machine. It proceeds to fix its route to point at this buffer. As a result, the subsequent data words are routed into the transaction buffer. Figure 5-20 illustrates the effect of the pipelining and bypass features of the transaction buffer: the first double-word of data is written into the transaction buffer in Cycle 6 and is bypassed through the buffer directly into the cache (if the processor is waiting)[33].

The end result is an 11-cycle cache-miss penalty for local operations to shared memory. In addition, access to local unshared memory has almost the same timing, as long as one removes the two coherence directory access periods (for a total of 5 cycles – two 2-cycle directory access cycles and a 1 cycle bus turnaround). This affects the latency by shorting the cache-miss penalty for an unshared access to 9 cycles; it is interesting to note that this 9-cycle latency meets or beats the best cache-miss penalties for SPARCstations$^{TM}$ with the same generation of processor. Thus, service coupling allows fast remote access without compromising the latency of local accesses.

It is also interesting to note that the remote cache fill operation implied by Figure 5-19 will have very similar timing to the latter portion of Figure 5-20. This is due to the service-coupling behavior of the Remote Transaction Machine at the destination of the message (which pipelines data directly from the network into the transaction buffer). What this says is that, in an architecture

[32]The separation of memory handling into two separate loci of control was chosen to simplify the issues of refresh and ECC handling, separating them from the higher-level issues such as cache-coherence protocol handling.

[33]The Sparcle cache is written on the falling edge of the clock, hence the half-cycle timing shown here.

which integrates the network and memory controller, *the difference between a remote and local cache-miss is no more than the two-way network latency plus a couple of extra cycles that may be attributed to queueing*. Thus, the myth that cache-coherence operations must be expensive is just that: a myth. Remote to local access latencies on the order of 2 to 1 or lower are currently achievable (see the Origin [67], for instance). In fact, we can expect that remote to local access latencies will be driven even lower than 2 to 1, given current trends of increasing cache-line sizes and increasing relative DRAM delays. This merely reflects the fact that the efficiency of communication in a service-coupled architecture is dependent on fundamental properties of the network and memory system rather than an artificial distinction between remote and local access.

### 5.2.5 Implementation of Local DMA Coherence

One final topic related to the hardware architecture of the Alewife CMMU is the implementation of locally-coherent DMA. This topic is of interest for at least two reasons: First, it contains the details of one of the high-level interactions between message passing and cache-coherent shared memory; hence, it is particularly apropos to the topic of this thesis. Second, the DMA mechanism combines the functionality of several of the blocks of the CMMU, tying them together with multiple levels of service coupling to achieve a highly-efficient, locally-coherent stream of data.

We have already mentioned the two state machines that provide the user-level message-passing interfaces in Section 5.2.1, namely the *IPI Input Machine* and the *IPI Output Machine*; these were grouped together as the *Network Interface and DMA Control* block in the block-diagram of Figure 5-9. What we would like to achieve in this section is a description of how the DMA portions of these machines interact with the Cache Management Machine (CMM), the Transaction Buffer (TXB), and the Memory Management Machine (MMM) to achieve locally coherent DMA. Given service coupling as discussed in the previous section, locally coherent DMA is surprisingly easy to implement in a highly-pipelined fashion.

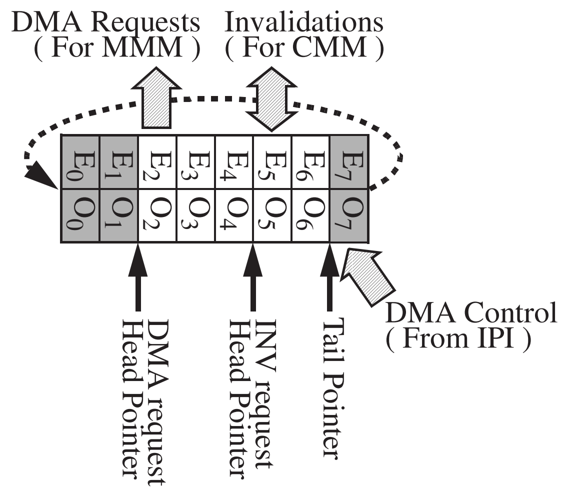



Figure 5-21: The double-headed *coherence queue* that is used to implement local DMA coherence. This structure is repeated for each of the two DMA engines.

The key to this implementation is the "double-headed" queue structure shown in Figure 5-21; one of these queues is associated with each of the two DMA engines (and IPI machines). This structure is a circular queue that contains records describing DMA operations that must occur[34]. Each entry in the queue is processed twice, once for cache invalidation and a second

[34]Each entry in the queue can handle up to two memory-lines simultaneously (a so-called even (E) and odd (O) memory-line). Memory-lines are handled in pairs during DMA in order to minimize the impact of DMA coherence on the processor: cache tags must be consulted for each memory-line to maintain coherence. Note that this implies that cache tags in the Alewife CMMU are divided into two banks — one for even addresses and another for odd addresses.

time to invoke necessary DRAM operations. Because the invalidations must precede memory operations (to achieve local coherence), this queue is organized with one tail pointer (for new records) and two head pointers (the one closest to the tail being the *Invalidation Head* and the other head pointer being the *DMA Request Head*.). Entries are removed from the queue after they have passed both head pointers.

When entries are first placed on this coherence queue, they contain information about the addresses that are to be invalidated. A difference between the Invalidation Head and the Tail indicates an address that has not yet been invalidated from the cache; in the language of service coupling, this forms a request queue to the Cache Management Machine (it is one of those "auxiliary" streams mentioned in Section 5.2.2.2). In the example of Figure 5-21, entries 5 and 6 are awaiting handling by the CMM (entry 5 is the head request). When the CMM scheduler gets around to scheduling this invalidation stream, one of two things may happen: either the requested memory line(s) are dirty in the cache, in which case the invalidation operation writes them into transaction buffers or they are not in the cache in which case nothing happens. Then, the "routing" phase of this operation causes the Invalidation Head to advance, optionally writing the identifiers of modified transaction buffers into the coherence queue (this is why there is a double-headed arrow pointing at the CMM in Figure 5-21).

A difference between the DMA Request Head and the Invalidation Head represents a DMA request that has passed invalidation and is awaiting DRAM operations. This forms another request stream to the Memory Management Machine. In the example of Figure 5-21, entries 2 – 4 are awaiting handling by the MMM (entry 2 is the head request). Note that these requests are actually compound operations whose behavior depends on whether or not the invalidation phase put data into the transaction buffer. For the IPI Output DMA stream, for instance, if data was placed into the transaction buffer during invalidation, then the requested operation is to take data from the transaction buffer and write it to both DRAM and the output queue. In contrast, if data was not placed in the transaction buffer during invalidation, then the request is to read from the DRAM and place the results into the output queue. For IPI Input operations, either data from the transaction buffer is merged with incoming data and written to DRAM or incoming data is simply written to DRAM. Note that the efficiency of the DRAM operation is the same whether or not data came from the cache or memory during local coherence.

This is the complete story on DMA coherence, except for one slight detail in the case of IPI Input: there is a danger that the processor will re-request a memory-line after it has been invalidated but before data has been written to memory. This situation can arise from the above of DMA requests and would represent a violation of local coherence. A reasonable programming style that would lead to this behavior would be the polling of a region of memory that is being overwritten by an incoming message to know when the data is complete; another would be some form of chaotic simulation that was operating on data that was periodically updated by incoming messages. The solution to this problem is to watch the processor request queue for addresses that are in the post-invalidation, pre-DMA coherence stage (between the two Head pointers above). When this happens, we simply interlock the processor request until the DMA operation is complete. This mechanism effectively closes the "local coherence hole".

Thus, locally-coherent DMA takes maximal advantage of service coupling, providing complete pipelining of the invalidation and DRAM accesses.

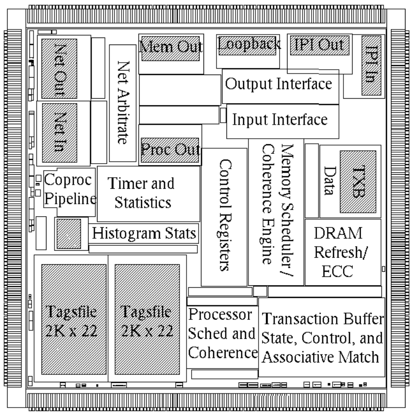



Figure 5-22: Floorplan for the Alewife CMMU (15mm $\times$ 15mm). Shaded regions are standard-cell memories. Remaining blocks are formed from sea-of-gates transistors.

| Category | Gate Count | % | Mechanism SM | MP | LT | FG |
|---|---|---|---|---|---|---|
| Processor Requests | 11686 | 12 | √ | | √ | √ |
| Full/Empty Decode | 2157 | 2 | | | | √ |
| Memory Machine | 13351 | 13 | √ | √ | | |
| DRAM Control | 8720 | 9 | √ | √ | | |
| Transaction Buffer | 17399 | 17 | √ | √ | √ | |
| Tracking Vectors | 2108 | 2 | √ | | √ | |
| Network Interface | 11805 | 12 | √ | √ | | |
| Network Queueing | 7363 | 7 | √ | √ | √ | √ |
| CMMU Registers | 9308 | 9 | √ | √ | | |
| Statistics | 11958 | 12 | | | | |
| Miscellaneous | 4627 | 5 | | | | |

Table 5-2: Functional block sizes (in gates) for the Alewife CMMU, as well as contributions to shared memory (SM), message passing (MP), latency tolerance (LT), and fine-grain synchronization (FG). Total chip resources: 100K gates and 120K bits of SRAM.

## 5.3 Mechanics of the Alewife Implementation

In this section, we would like to mention a few details of the "implementation mechanics" of the Alewife machine, *i.e.* the methods by which the Alewife machine (and in particular the CMMU) was implemented. Construction of the Alewife machine was a careful balance of industrial collaboration and local implementation by members of the Alewife team. LSI Logic, SUN Microsystems, and the APT group at ISI were instrumental in bringing the Alewife prototype to fruition. However, by dint of foresight or (more likely) luck on the part of Alewife, we opted for exactly the right balance of industrial support: help with relatively minor modifications to an existing product (resulting in the Sparcle processor) and assistance with the NRE and mechanics of fabrication for the CMMU. At no point did we rely on major implementation efforts from our industrial collaborators, hence avoiding the "cancelation" phenomenon that has plagued other architecture groups.

### 5.3.1 Implementation of the Alewife CMMU

The CMMU was implemented in LEA-300K hybrid gate-array technology from LSI Logic. This technology has a $0.6\mu$ feature size, three layers of metal, and enables intermixing of dense standard-cells (for memory) amongst sea-of-gates transistors (for general logic). Figure 5-22 shows the floorplan for the A-1000 CMMU. In this figure, shaded regions represent memory and remain fixed in the final layout. Unshaded regions, on the other hand, represent the starting locations for various functional blocks; the place-and-route tool uses this placement of blocks as an initial seed for the simulated annealing placement process. The chip consists of approximately 120,000 bits of SRAM (for the tags-file and network queues), and 100,000 gates of random logic.

Table 5-2 gives a cost-breakdown for the gate-array portion of the chip. Note that gate counts in this table are skewed a bit because they represent aspects of the chip that were not implemented as standard-cell memories. For instance, the Transaction Buffer listing includes latches for implementation of the state bits, since the specialized CAM functionality described in Section 5.2.3.2 was not available as a standard block from LSI Logic.

Several interesting things can be gleaned from this table. First, the network interface portion of the Alewife CMMU (including DMA) represents only about 12 percent of the chip, approximately the same size as either the processor or memory coherence-protocol blocks and only about 50 percent larger than the DRAM controller. This suggests that the addition of the user-direct messaging interface described in Chapter 2 was not a significant aspect of the implementation, especially given the large number of advantages. Table 5-3 gives a more detailed breakdown of the message-passing portions of the chip, illustrating the asynchronous network interfaces and queues for local DMA coherence.

| 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) | $32\times65$ bits |
| Network Output Queue (1R/1W) | $32\times65$ bits |
| IPI Input Queue (2R/1W) | $8\times64$ bits |
| IPI Output (descriptor) Queue (2R/1W) | $8\times64$ bits |
| IPI Output Local Queue (1R/1W) | $8\times65$ bits |
| IPI Input Coherence Queue (2 entries) | 1610 gates |
| IPI Output Coherence Queue (2 entries) | 1306 gates |
| IPI Input Interface | 4181 gates |
| IPI Output Interface | 3393 gates |

Table 5-3: Module sizing for the message-passing portions of the Alewife CMMU.

Second, the transaction buffer was one of the largest blocks of random logic on the chip. This certainly reflects the importance of this block (as does the quantity of text associated with it on this thesis), but also reflects the unfortunate nature of the implementation. In a "real" (non-academic) implementation, much of this random logic would be replaced by a specialized CAM cell that would be more compact *and* much faster (the transaction buffer was in the critical path of the final chips). Third, the CMMU registers listing includes such things as the interrupt controller, as well as all of the user-interface logic to access portions of the chip. Finally, note that statistics formed an non-trivial portion of the Alewife CMMU; this reflects the academic nature of the chip. The resulting functionality was priceless and more than worth the time and effort involved in implementation and debugging.

Unlike Sparcle, which was a collaborative effort between MIT (for specification), and LSI Logic/Sun Microsystems (for implementation), the Alewife CMMU was designed and implemented almost entirely by one person at MIT (this author)[35]. With a project of this size, some concessions must clearly have been made; in Alewife, these concessions involved use of high-level synthesis and gate-array design. Unfortunately, this design methodology leaves much to be desired in term of final hardware speed: generated logic tends to have more levels of logic than strictly necessary, and automatic route and placement tools tend to add delay in inconvenient places[36]. Thus, meeting timing constraints was far more challenging in Alewife then it would have been with more custom design methodologies.

[35] David Chaiken played a pivotal role in design verification, as well as serving as designer of the cache-coherence protocol. The asynchronous network interface was a collaborative effort between the author and Jonathan Babb.

[36] A not-so-amusing example of this was the tendency of the LSI Logic placement tool to separate high-drive buffers from the logic that they were buffering, leaving low-power logic to drive long wires. This apparently followed from some heuristic that tried to place multi-fanout logic as close as possible to the things that it was driving.

Figure 5-23 illustrates the overall build tree for the Alewife CMMU. At the top of this diagram are the three types of input to the design process. Most of the design was formulated in a high-level synthesis language called LES from LSI Logic (at the inception of the Alewife project, this was all that was available for our use). By the end of the project, there were about 36,000 lines of high-level synthesis code. This code included the cache-coherence protocol tables, interpretation of full-empty bits, the coprocessor pipeline, transaction buffer control logic, statistics, *etc.*Since high-level synthesis is notoriously bad at producing fast logic, output from the synthesis tool was post-processed by the Berkeley SIS logic optimization tool. SIS has a high usage-overhead in terms of customized control scripts; the resulting output, however, is often better than Synopsis. Unfortunately, the combined output of high-level synthesis and optimization still produces obviously inferior logic in the case of data paths and critical control logic; in these cases, logic (*e.g.* muxes and AOI gates) was coded explicitly in the LES code[37].

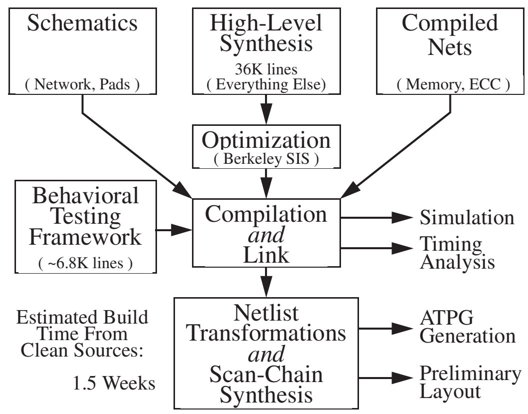



Figure 5-23: Build tree for the Alewife CMMU. Input includes schematics, synthesis, and data-path compilation.

Two other inputs to the design process were schematics and data path compilers. Schematics were used for the asynchronous network, since the delay and ordering of logic had to be explicitly controlled. Schematics were also used for the connection of logic around the pads and stubs for the ATPG scan chain. Data path compilers were the most desirable technique for design, simply because they required little in the way of designer input, and tended to produce optimized designs. Unfortunately, LSI Logic had a limited set of compilers of this type. The prime example of compiled logic was SRAM cells. Other examples included counters, large muxes, and adders. In addition to the LSI Logic compilers, this author wrote a compiler that produced ECC logic[38].

The output from all three types of design input were netlists (consisting of primitive cells and interconnection information). These netlists were compiled together to produce an image that could be simulated. This image was combined with the testing framework, described in Section 5.3.2, to perform functional test. In addition, the compiled image was used in conjunction with the block diagram (Figure 5-9) to perform a preliminary place and route. Further, an automatic tool by LSI Logic was used to synthesize a scan chain from those aspects of the design that were sufficiently synchronous to be tested via ATPG.

LSI Logic takes as input the netlists, block diagram, other design files (specifying pinouts), as well as sets of generated hardware vectors (the ATPG output in combination with hand-constructed parallel test vectors). They return working chips[39].

---

[37] In order to prevent the optimizer from modifying these gates, they were coded with names that were opaque to both the LES compiler and the optimizer.

[38] This handled a variety of data widths and two classes of codes. It was only used once in the CMMU, however.

[39] This statement is idealized, but essentially true. What I am ignoring here is several months of design file preparation and layout tweeking to produce reasonable hardware speeds.

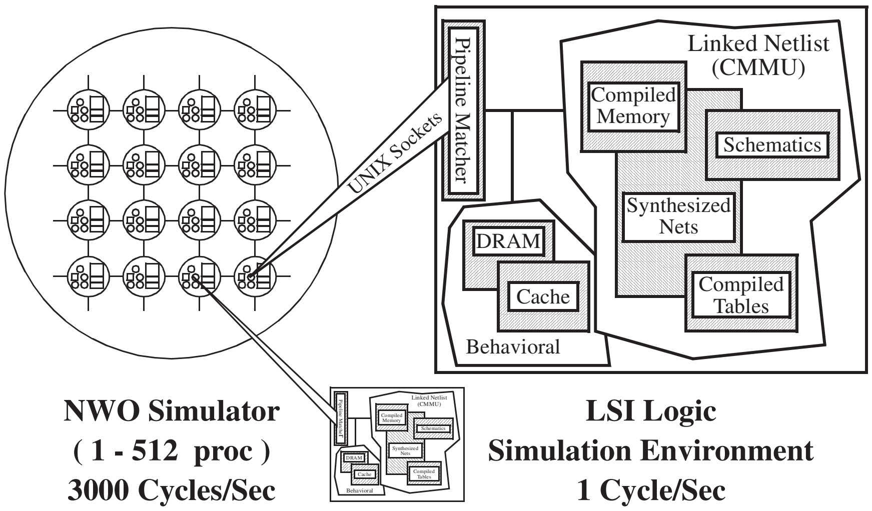



Figure 5-24: The Alewife hybrid testing environment supports multiple levels of simulation. NWO is a high-level simulation which is correct to the level of of queuing topology and machine states. The LSI Logic simulator, on the other hand, is a compiled, gate-level simulation of the actual CMMU.

### 5.3.2 Validation Through Multi-Level Simulation

First-silicon for the Alewife CMMU was remarkably bug free. This can only be attributed to our testing effort. While the Alewife CMMU was designed and implemented by one person, it was tested by seven graduate students, one professor, and one research scientist. This testing effort was coordinated by the author, who retained all control over sources for the controller; in some sense, you could say that this author was ultimately responsible for the generation and removal of bugs. However, the testers had the much more difficult task of finding behavior that did not fit specifications.

Figure 5-24 illustrates the hybrid testing environment utilized in the design of the Alewife CMMU. As shown by this diagram, the Alewife testing process occurred at two separate levels of simulation. On one end of the spectrum was the NWO simulator (standing for *New World Order*), written by David Chaiken. This was a high-level simulation, written in Scheme, that was accurate to the level of Sparcle instruction set, CMMU queue topology, cache-coherence protocol tables[40], and internal CMMU state machines. At the other end of the spectrum was a gate-level simulation of the final CMMU and Sparcle netlists (post optimization and other netlist transformations). The difference in speed of these two types of simulation was over three orders of magnitude. Further, NWO could be configured in multiprocessor fashion (multiple simulated nodes), whereas the LSI simulation could only handle one node at a time.

NWO was sufficient to catch high-level design flaws, queue-level deadlocks, cache-coherence protocol problems, *etc.*It was also the primary vehicle for developing operating systems and ap-

[40]David Chaiken developed a compiler which took as input the Espresso format for the protocol tables (output by the LES hardware synthesizer), and generated scheme code that was subsequently loaded into NWO. As a result, the protocol tables in NWO exactly matched the protocol tables coded into the hardware.

plications code. The LSI simulation, on the other hand, was necessary to detect more detailed problems, such as pipelining interlock problems, misorderings caused by the actual implementation, *etc.*As shown in Figure 5-24, the two simulations could be linked via UNIX sockets to achieve a multi-level simulation that leveraged the best of both worlds. Because sockets were the primary form of communication, multiple LSI simulations could be run on separate machines and interact with a primary NWO simulation. This allowed, for instance, a 64 node NWO simulation to include one or two "real" nodes, thus effecting a more realistic environment, protocol traffic and all.

So how was this simulator used for testing? One of the key features of the CMMU design that we have not discussed previously, was that it was "designed for test". What this means in a practical sense is that most of the internal state of the CMMU is exposed in a way that is accessible to the processor. As a result, all of the Alewife test "vectors" consisted of code running on the Sparcle processor. Much of the Alewife testing effort was inspired by a paper by Douglas Clark [29]. In this paper, he makes a convincing argument for large-scale, complete-system testing. He argues that, since the interfaces between modules in a large system are as likely to be flawed as the modules themselves, time spent testing modules independently is not necessarily the best use of resources. As a result, most of the Alewife testing was in the form of complete system-level tests, utilizing the hybrid simulation environment.

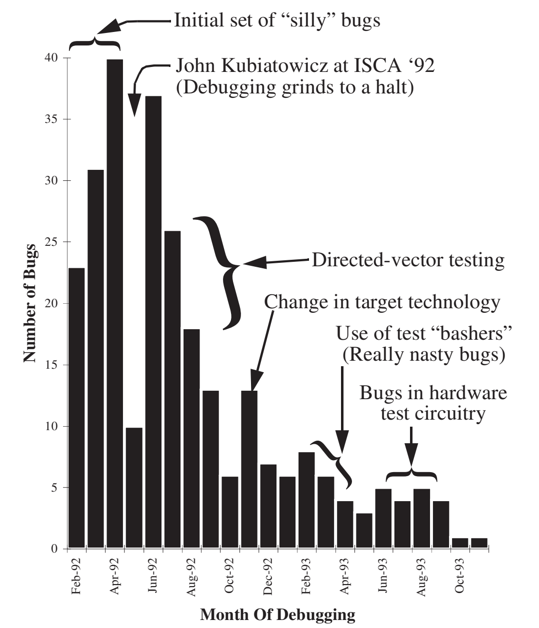



Figure 5-25: Number of bugs detected in the Alewife CMMU as a function of time.

The tests utilized in Alewife fell into roughly three categories (also inspired by the Clark paper):

1. Directed Vectors
2. Bashers and Daemons
3. Full Applications

The first category of tests were written in Sparcle machine language and targeted at exercising individual features of the CMMU. These were the most labor intensive tests to generate and required the most supervision by the author. However, these types of tests were very good at uncovering gross implementation errors and even some subtler pipeline issues. Note that, although these tests are for specific features, they meet the criteria of "complete-system test", since they execute against a completely compiled and linked version of the CMMU, not against specific submodules. Directed vectors were responsible for a large fraction of the total bugs detected in the CMMU.

While simple first-level interactions may be targeted by directed vectors, combinations of features rapidly grow out of hand, since the cross-product of all features and interactions is a very large set indeed! Thus, the second category of tests is more automated and attempts to take over where the first category left off. Ken Mackenzie was extremely successful in developing randomized tests that combined high utilization of shared-memory and message-passing mechanisms with random mechanism selection and timing[41]. Further, David Chaiken developed a generic testing framework that permitted the combining of various directed vectors in random order. Both of these types of tests explored operation orderings and portions of the testing space that were never addressed by explicit vectors. Since these types of test had a tendency to stress the system in many ways, they were dubbed "bashers". Closely related to bashers were "daemons" that would insert non-fatal errors into a simulation; one of the primary examples for Alewife included inserting single-bit errors into the simulated DRAM. Although correctable, single-bit errors would invoke an extra pipeline interlock for correction, thereby changing timing. The basher/daemon strategy was tremendously successful at discovering complex interactions[42]. Although the number of bugs discovered this way was much smaller than the number discovered through directed vectors, these bugs tended to be far more subtle and hazardous.

Finally, the last category of tests involved execution of complete applications running under the Alewife operating system. These detected problems in interrupt handlers (such as LimitLESS protocol handlers) as well as problem due to higher stress use of the coherence protocol. Combined with daemons, complete application tests adopted some of the characteristics of bashers/daemons combinations.

Figure 5-25 illustrates a histogram of bug-find-rate over the course of the roughly two years of CMMU development. This data was taken from bug logs kept by the author[43]. Ignoring the upslope at the beginning (which was as much a result of a learning curve on the part of testers as anything else), this curve has a roughly decaying exponential shape. Deviations from that basic shape represent interesting events in terms of new testing techniques, changes in technology, or a different testing focus (*e.g.* a switch from functional test to generation of manufacturing test circuitry). Some of these are labeled in the figure.

### 5.3.3 Hardware Test Methodology

A brief word about the hardware test vectors (used during manufacture) is in order. Most of the Alewife CMMU was sufficiently synchronous that it could be tested via automatic test pattern generation (ATPG) targeted at a single scan chain. As mentioned above, a tool from LSI Logic was used to connect all of the flip-flops in the chip into a scan chain. Unfortunately, there were two aspects of the chip that were not amenable to this type of test. First, SRAM memories were not edge-triggered and hence not easily accessible via the scan chain. This was handled by making them accessible via the processor data bus. A set of parallel vectors was then targeted to perform fault coverage of the SRAM modules. Second, the asynchronous network was not amenable to full testing via any synchronous methodologies. Instead, it was tested in two modes: a synchronous test, in which all of the control signal transitions were synchronized with the clock, and an asynchronous loopback test, in which the output network interface was coupled to the input network

[41] As a tester of the CMMU, Ken Mackenzie single-handedly found more bugs (and more interesting bugs) than any other member of the testing team.

[42] Ken Mackenzie was the king of the basher writers. He discovered more complex bugs than anyone else on the testing team.

[43] One amusing accompaniment to this histogram was the "bug chart" that was posted outside of this author's office. Each of the testers had a slot on this chart. Whenever they would find a bug, they got a star.

interface. This permitted reasonably sufficient coverage of both the asynchronous SRAM queues and the control logic[44].

## 5.4 Postscript: Implementation is Possible and Necessary

In this chapter, we touched upon some of the many details of the Alewife implementation, attempting to focus on those aspects relevant to an integration of message passing and shared memory. In the next chapter we will examine some performance results to to understand the overall validity of the Alewife approach and in Chapter 7, we will try to glean a few lessons from Alewife's integration of message passing and shared memory.

Section 5.3 condensed (in very short form) a number of years of implementation and debugging work. There is absolutely no question in the mind of this author that the design and implementation of computer architectures is becoming increasingly difficult to achieve in an academic environment. There are, perhaps, three possible responses to this situation: One is to "give up" and perform only simulation studies of future architectures. While there is plenty of room for simulation studies and not all research groups are capable of launching large-scale design projects (for lack of resources if not lack of expertise), this is a very dangerous course if taken by all researchers; it is a sure way to make academic computer architecture irrelevant. A second response is to carefully choose to alter a small number of pieces of an existing architecture, implementing only a simple "widget" which attaches to a much larger system. Many would say this is the wisest course, for it has the potential to achieve interesting results with the least investment of time and effort. In addition, it does not suffer from the same lack of "reality" that plagues the simulation-only approach. Unfortunately, by its very nature, this approach is incremental: major paradigm shifts are unlikely. Further, it often generates results that are not fundamental, but rather sensitive to simple implementation details (*e.g.* the speed of some external bus to which the widget was stapled).

A third response to the increasing difficulty of implementation: "work smarter". This response is similar to the previous response, but involves a very different attitude. First, the sights of an implementation project should always be high enough to target something fundamental. An important accompaniment to this attitude is that all aspects of a proposed system should be considered to be changeable (or "nonstandard") at the inception of a project[45]. The essential difference between this third approach and the previous two is that *all* aspects of a system are considered open to innovation. Of course, research groups have limits of one form or another. Thus, having decided what key things must be changed, the trick is to figure out (1) how to best change those crucial details and (2) how to leave as many other things unchanged as possible. For Alewife, a small addition of functionality to Sparcle allowed the implementation of very fast interfaces without altering much of the integer pipeline. Today, many ASIC manufacturers offer "core" technology that contains complete RISC pipelines, floating-point units, *etc.*that can be dropped into larger chip projects.

[44]Of course, one of the banes of asynchronous design is that it is not really amenable to any of the standard synchronous test methodologies.

[45]To say, for instance, that integer pipelines are so fixed that they cannot be altered is to ignore the reality that the last few years have seen a great increase in the functionality (and complexity) of pipelines.

Also, forward looking silicon companies understand that they should be looking farther than a couple of development cycles into the future; such companies can make good allies (within careful limits) in academic projects[46]. Finally, software tools for chip design and implementation continue to improve, although slowly. The use of high-level synthesis in Alewife permitted a relatively large project to be tackled by a single person.

The bottom line is that implementation is still possible in an academic setting. More importantly, *implementation is necessary*: without far reaching and forward looking architectures, backed by reasonable implementations, academic computer architecture research is doomed to irrelevance. It is the job of computer architects to lead, not follow.

[46] Although the important thing here is to have other options and/or keep the time horizons of industry participation short – a number of academic projects have been canceled because some crucial industrial partner decided that it was no longer interested in the research.

# Chapter 6

# The Performance of the Alewife Prototype

To some extent, many of the results of this thesis have already been presented as solutions to the Three Challenges of Integration, complete with the hardware architectural structures presented in Chapter 5. However, one of the exciting things about the Alewife architecture is that prototypes were actually built *and* successfully utilized for research. Figure 6-1 shows two incarnations of the Alewife machine: a 16-node Alewife system, and a 32-node system within a chassis scalable to 128 nods. The 16-node system, complete with two internal $3\frac{1}{2}$-inch disk drives, is about $74\times12\times46$ cm, roughly the size of a floor-standing workstation. Packaging for a 128-node machine occupies a standard 19-inch rack. In this chapter, we would like to explore the performance of this prototype. Timing numbers that we will present reflect a 32-node Alewife machine, packaged in the lower right quarter of the 128-node chassis (visible in Figure 6-1).

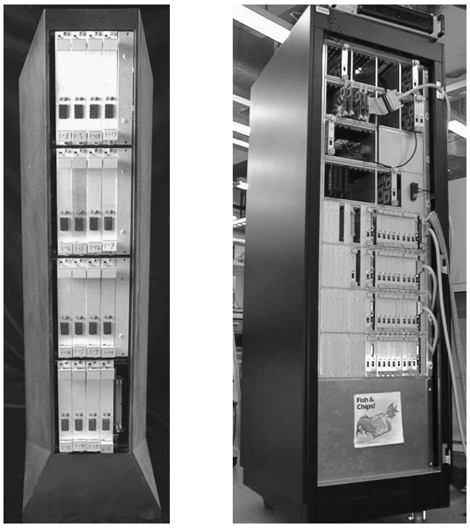


Figure 6-1: 16-node machine and 128-node chassis populated with a 32-node machine.

The first few A-1000 CMMU chips were received in May of 1994. Because complete software development (including development of the operating system, boot software, and applications) occurred simultaneously with the development of Sparcle and the CMMU, two benefits were immediately apparent upon arrival of these chips: First, the CMMU chips were substantially bug-free on first silicon: although there were a few bugs in the A-1000 CMMU, all of them had software work-arounds (each of which involved many hours of this author's time, but that is another story). Second, the maturity of the software system became evident when we had a two-node Alewife machine operational within a week of receiving the first chips. When the next eight chips arrived, this uncovered another bug (including a problem with DMA coherence), but a subsequent week produced an operational eight-node Alewife machine. And so on . . . One conclusion to be drawn from our experience during the physical integration process is that many of the design and testing decisions that were made at the inception of the Alewife project were tremendously effective.

**On Clock Rates and Technology Curves:** Before embarking on a exploration of the performance of the Alewife prototype, a brief discussion of clock rates is in order. To be specific, the Alewife machine was originally targeted for 33MHz operation. At the beginning of the CMMU implementation (1990), clock rates of 33 MHz were pretty much state of the art (or soon to be so). Unfortunately, with the long design schedules of an academic project (not to mention the time involved in "exploratory implementation", *i.e.* the time involved in figuring out how to implement something that had not been implemented before), the Alewife machine was much behind the technology curve when it was finally operational in 1994. However, this particular author makes no apologies for this result; it is simply one of the costs of construction in an academic environment.

With respect to the final prototype, how fast did it actually operate? For the most part, the 33MHz speed goal was met, however a bug in the coprocessor pipeline caused the "FPU Hack" of Section 5.1.2 to miss its bus-turn around cycle during write operations, thereby causing a bus conflict at full speed. This conflict caused us to generate most of our performance numbers at 20MHz. Note, however, that these speeds do not, in any way, reflect critical paths of the CMMU or the scalability of mechanisms within the CMMU; integer operations operated much closer to the speed target, *i.e.* at 31MHz. Critical patches in the A-1000 had much more to do with limitations of the implementation methodology (gate array with random-gate CAM cells, *etc.*) than the base technology. Further, as indicated by the last chapter (and in particular the discussion in Section 5.2.4 on service-coupling), critical portions of the CMMU were pipelined in a way that would be easily scalable to higher clock rates and newer technologies.

Thus, performance numbers in this chapter for Alewife must be interpreted with two crucial mind-sets:

- These numbers reflect *balance* between the processor and network. At 45 MBytes/sec, the Alewife network reflects early 1990's technology in the same way that the processor and memory controller reflect this technology.
- Cycle counts *are* important. Keep in mind that one of the goals of this thesis was the exploration of the notion that communication interfaces should be considered as extensions to the ISA.

This author fully realizes that cycle counts taken out of the context of clock rates can be meaningless; however *a modern processor is implemented by 100s of engineers; the Alewife CMMU was implemented by one engineer (plus the crucial testers).*

**On Numbers and Attribution:** One important point that should be mentioned before diving into an exploration of performance is to mention the source of numbers. Many of the numbers in the following sections were actually taken by others in the Alewife research group; I will do my best to reference these people as I go. In many cases, I helped to debug the applications that produced these numbers, helped to design interesting experiments, and to enhance the operating system as necessary to support these studies. I was also (in conjunction with Sramana Mitra) responsible for the statistics software used to collect them. Thus, although I did not sit down and physically take many of the numbers, I will none-the-less use them to discuss the performance of the prototype. However, I do not want to minimize the tremendous amount of work that went into writing or porting the applications by others in the group. It is upon their efforts that I build this chapter.

## 6.1 Microbenchmarks and the Alewife Prototype

In this first section, we would like to discuss some microbenchmarks that indicate the performance of various communication mechanisms in the Alewife prototype. In the following paragraphs, we will characterize the raw performance of the Alewife network, as well as the uncontended performance of shared-memory and message-passing communication. Kirk Johnson was instrumental in many of these measurements. Also, many of these numbers have been published elsewhere [3, 16, 60].

### 6.1.1 Performance of the Network

In this section, we perform a characterization of the network performance as a whole. This will yield a baseline from which we can evaluate the other communication mechanisms. Three numbers are of relevance here: (1) the raw, per-flit bandwidth of a network channel, (2) the minimum fall-through latency of a "0-hop" message, and (3) the per-hop latency of a message. These results are summarized in Table 6-1.

| Parameter | Count |
|---|---|
| Network Bandwidth: | 44.5 MBytes/sec |
| Minimum routing latency (one-way): | 140 nsec |
| Per-hop latency (one-way): | 39 nsec/hop |

Table 6-1: Three critical network parameters

The raw network bandwidth number is straightforward, since the Alewife DMA mechanisms are fast enough to saturate the network. Thus, our basic mechanism for measuring bandwidth will be to send large messages between two nodes via DMA for a variety of different message sizes. The slope of the "number of bytes sent" *vs.* "round-trip time" will give us twice the latency of sending one byte. (Note that this particular method is completely independent of endpoint cost). Varying message sizes from 32 – 1024 bytes at 20MHz yields a slope of 0.898 bytes/cycle, which is about 22.5 nsec/byte or 44.5 MBytes/second.

Measurement of the fall-through latencies is more difficult. However, one useful consequence the asynchronous network in Alewife is that it is possible to vary the processor's clock rate, thereby separating the contributions of the network and the CMMU to remote access time[1]. This was done for a clean read-miss to a remote node. In addition, a number of different communication distances were explored as well (to get the per-hop cost). By treating the round-trip time and the clock period in nanoseconds, we can do a two-variable linear regression: round-trip time *vs.* clock period and number of hops. The constant term in this result is the "constant" portion of the latency, *i.e.* consists of the sum of the basic delay involved in opening up a round-trip channel in the network (which we want) and the raw delay involved in pushing flits through the network. By using our

[1] This is perhaps the only advantage to an asynchronous network — among a plethora of disadvantages in design and test.

22.5 nsec/byte value from the previous paragraph and the 32 bytes involved in a clean read, we can factor out the second term to yield our minimal routing time. The slope of the round-trip delay with respect to number of hops is the per-hop latency. Results are quoted in Table 6-1. Note that there is an implicit assumption here that only "Manhattan distance" matters in the network; in fact that is true: the mesh network chips route through two levels of dimensional routing ("X" and "Y") regardless of the relationship between source and destination.

### 6.1.2 Performance of Shared Memory

In exploring the performance of shared-memory, we must first take into account the behavior of Sparcle with respect to cache hits. Sparcle employs a single-ported, unified first-level cache, with no on-chip instruction cache. Consequently, 32-bit loads and stores that hit in the cache take two and three cycles, respectively (one cycle for the instruction fetch). Figure 5-6 on page 155 illustrated this timing for a load instruction. Doubleword (64-bit) loads and stores that hit in the cache take one additional cycle over and above the performance for normal loads and stores.

| Miss Type | Home Location | # Inv. Msgs | hw/ sw | Miss Penalty Cycles | Miss Penalty $\mu$sec |
|---|---|---|---|---|---|
| Load | local (unshared) | N/A | hw | 9 | 0.45 |
| | local | 0 | hw | 11 | 0.55 |
| | remote | 0 | hw | 38 | 1.90 |
| | remote (2-party) | 1 | hw | 42 | 2.10 |
| | remote (3-party) | 1 | hw | 63 | 3.15 |
| | remote | – | sw† | 425 | 21.25 |
| Store | local (unshared) | N/A | hw | 10 | 0.50 |
| | local | 0 | hw | 12 | 0.60 |
| | local | 1 | hw | 40 | 2.00 |
| | remote | 0 | hw | 39 | 1.95 |
| | remote (2-party) | 1 | hw | 43 | 2.15 |
| | remote (3-party) | 1 | hw | 66 | 3.30 |
| | remote | 5 | hw | 84 | 4.20 |
| | remote | 6 | sw | 707 | 35.35 |

† This sw read time represents the throughput seen by a single node that invokes LimitLESS handling at a sw-limited rate.

Table 6-2: Typical nearest-neighbor cache-miss penalties at 20MHz under a variety of circumstances. For longer distances, add 1.6 cycles/hop.

Once we know the times of basic operations under cache hits, we can characterize the costs of shared-memory in terms of *cache-miss penalties*, *i.e.* the the number of additional cycles incurred by an access that misses in the cache and must perform some type of communication. Table 6-2 shows cache-miss penalties for a variety of circumstances. These values were obtained with a sequence of experiments run on an otherwise idle Alewife system. All remote misses or invalidations are between adjacent nodes. As discussed in Section 6.1.1, each additional "hop" of communication distance increases these latencies by $2\times39$ nsec or approximately 1.6 cycles.

Each operation in Table 6-2 is categorized by the "Home Location" of the access; this refers to whether or not the directory for the access resides on the same node that makes the access (*i.e. local*) or some remote node (*i.e. remote*). These locations are further modified with additional information. Here, *unshared* refers to an access to the unshared-memory space; *2-party* refers to an access for which the most up-to-date copy of the data is dirty in the cache on the home-node; *3-party* refers to an access for which the most up-to-date copy of the data is dirty in some node different from either the requesting node or the home node (*i.e.* there are "three-parties" to the transaction). The column marked "#Inv. Msgs" indicates the number of invalidation messages sent in order to satisfy a request.

This compact table encodes a lot of information, some of which we will indicate here. First, the difference between the penalty to local unshared memory and to local shared memory is two cycles. These two extra cycles derive from the time to read the coherence directory, which is not necessary for local unshared memory (see Section 5.2.4). Note that the 9-cycle read-miss penalty for local unshared memory matches or beats the best times for SPARCstations$^{TM}$ of the same generation. As we discussed in Section 5.2.4, this shows that the service coupling methodology permits fast remote access without compromising local access times.

The next thing to note is the difference between the *remote* and *remote (2-party)* numbers: 4 cycles. This extremely short value reflects the efficiency of scheduling and pipelining within the Alewife Memory Management Machine and Cache Management Machine, because this is implemented as two *separate* transactions: one that takes the incoming request and generates an invalidation to the local Cache Management Machine, and another that results from scheduling the resulting flush operation from the transaction buffer[2].

Finally, cache-miss entries marked as “sw” represent the access time seen when a cache line is shared more widely than is supported in hardware (five pointers), so that the home node processor must be interrupted to service the request. In the case of a load, the software time represents the maximum throughput available when every request requires software servicing. Because of the read-ahead optimization (Section 3.4.2.1), and the fact that software handling can be amortized by emptying the coherence directory (hence allowing the subsequent five read requests to be handled in software), this latency number will rarely be experienced by a requesting node. The software store latency represents an actual latency seen by a writer; it includes the time required for the software handler to send six invalidations, for these invalidations to be received by the hardware, and for an exclusive copy to be returned.

**Breakdown of a Read Request:** Finally, we would like to illustrate the breakdown of a 38-cycle clean read-miss to remote memory. Such a breakdown is given in Table 6-3. The first thing to note about the timings in this table is that they express non-overlapped latencies. Thus, for instance, the processing and setup time of the Remote Transaction Machine for the cache fill is completely overlapped. Similarly, the first half of the cache fill (2 cycles) is completely overlapped with reception of the final 8 bytes; that is why “Cache Fill Time” is listed as 2 cycles. The origins of the 7-cycle timing for the memory response was shown in the previous chapter (Figure 5-19, page 183); this is the latency from the arrival of the request at the memory controller to the queueing of the header in the asynchronous network output queue. At

| | Action | Count |
|---|---|---|
| | Cache-miss to request queued in network | 2 |
| | Routing Flit Computation | 0.5 |
| @ | Request transit time (8 bytes) | 7 |
| | Input Synchronization Time | 2.5 |
| | Request at memory to output header transmit | 7 |
| | Routing Flit Computation | 0.5 |
| @ | Data return in network (24 bytes) | 14 |
| | Input Synchronization Time | 2.5 |
| | Cache fill time (non-overlapped) | 2 |

Table 6-3: Rough breakdown of the non-overlapped portions of a 38-cycle clean read-miss to neighboring node at 20MHz. Note that there are 21 cycles of asynchronous network communication (marked with “@”).

[2] The CMMU contains a directory cache that allows us to skip the DRAM directory read for the second transaction.

this point, we should probably explain where the various half-cycle synchronous delays came from. For output operations, the "Routing Flit Computation" is the extra half cycle used to compute routing flits in the network routing fabric; the first flit of a message is released into the network on the falling edge of the clock. On the input side, the two entries marked "Input Synchronization" represent the *average* delay through a two-stage synchronizer; the extra half cycle originates from the fact that, on average, a message arrives half-way through the cycle.

The numbers given in this table are derived from a combination of measurements and examination of the design. For instance, by varying the clock rate of Alewife (and performing a simple regression of read-miss time in cycles *vs.* clock frequency), we discover that the 38-cycle miss time is composed of the 17-cycles of clock-rate independent latency that we have attributed to various operations within the CMMU (these are all operations that are not marked with "@").

The remaining 21 cycles are considered "asynchronous" and are attributed to the network (they are marked with "@" in the table). These cycle counts can be derived by using our numbers from Section 6.1.1 to compute the network transit times. In that section, we discovered that a one-way latency to a node that is one hop away is about 140 nsec + 39 nsec or 179 nsec. This is 3.6 cycles. Further, the 8-byte request requires $8\times22.5$ nsec or 180 nsec, while the 24-byte response requires $24\times22.5$ nsec or 540 nsec. This is 3.6 and 10.8 cycles respectively. Thus, we get 7.2 and 14.4 cycles for the outgoing and incoming values respectively[3]. Since these cycle counts represent *fixed* delay times, independent of clock rate, they simply scale by the clock frequency to yield the corresponding values for other clock rates. For example, at 33MHz, we would have 11.9 and 23.8 cycles respectively.

Note that, if we remove all cycles devoted to interface with the network ( $2\times2.5 + 2\times0.5$ cycles) as well as the network cycles (21 cycles), we are left with 11-cycles! This is the same as the local cache miss penalty[4], and illustrates that service-coupling enables efficient utilization of resources by both local and remote accesses.

### 6.1.3 Performance of Message Passing

To explore the performance of message passing on the Alewife prototype, this section looks at the cost of basic message-passing mechanisms. Table 6-4 gives a rough breakdown of the costs of sending and receiving a null active message with notification via interrupts. Note that sending and receiving overheads are given separately and that network transit time is *not* included in this table; data from Section 6.1.1 should be invoked to estimate a complete one-way latency.

The sending process portion of Table 6-4 is the most straightforward. The sending of a null active message requires two words to be stored in the output descriptor array (the *Header* and *Handler* arguments), followed by the execution of an `ipilaunch` instruction. From the user's standpoint, this takes a total of 7 cycles; this number should be increased by 3 cycles for each additional argument. Of course, this description of the sending process vastly oversimplifies the software overheads that lead up to any realistic use of active messages; there is time involved in

[3]This is consistent with the fact that 38 cycles (total) minus 17 cycles (synchronous latency) is 21 cycles (asynchronous latency).

[4]Careful examination of these cycles indicates that a couple of the memory-interlock cycles of the DRAM controller have been converted into pipeline-delays into the network queues, *i.e.* cycles have slightly redistributed.

loading the handler address so that it can be stored into the network interface (perhaps 2 cycles); there is time involved in constructing a header with the appropriate destination (perhaps 3 or 4 cycles),*etc.*Consequently, the times given in this table represent a lower bound on overhead. Note that the message begins to appear in the network 2.5 cycles after execution of the `ipilaunch`.

The receiving process is at once more complicated and more interesting. Table 6-4 gives complete timings for the scheduling of a user-level active message: from the arrival of the message at the network input port, to the scheduling of a user-level thread, to the execution of the null handler, to the freeing of the thread and processor return from interrupt. The user-level atomicity mechanism as discussed in Section 2.4.5, was not fully designed at the time that the original (A-1000) CMMU was implemented. In fact, the need for such a mechanism was fully appreciated only after this author wrote the runtime system for user-level active messages. As a result, the original Alewife system provided user-level atomicity through software emulation. Subsequently, given the lessons learned in the process of providing atomicity in software, the hardware version of the user-level atomicity was designed and implemented in the A-1001 CMMU. Portions of the existing runtime system were rewritten to use this mechanism. As a consequence, Table 6-4, gives two separate columns of numbers: a "soft atom" column (for software atomicity) and a "hard atom" column (for hardware atomicity). Comparison of these two columns will illuminate some of the advantages of the hardware version of the user-level atomicity mechanism. As shown by these tables, the receive overhead is 99.5 and 51.5 cycles respectively. Unlike the send mechanism, these numbers do not hide other overheads: they *do* reflect timings for a null handler execution. Obviously, timings will increase for more complicated handlers; for instance, there will be a cost of at least 2 cycles for each additional argument (to load them into registers).

| Message Send | cycles |
|---|---|
| Descriptor construction | 6 |
| `ipilaunch` | 1 |
| ⇒Network Queueing | 2 |
| ⇒Routing Flit Computation | 0.5 |
| *send total:* | 9.5 |

| Message Receive (interrupt) | soft atom | hard atom |
|---|---|---|
| ⇒Input Synchronization Time | 2.5 | 2.5 |
| ⇒Input queueing & interrupt generation | 2 | 2 |
| Hardware interrupt overhead | 4 | 4 |
| Register allocation & first dispatch | 16 | 13 |
| Atomicity check | 2 | 0 |
| Timer setup | 11 | 1 |
| Thread dispatch | 15 | 6 |
| *dispatch overhead:* | 52.5 | 28.5 |
| Null handler (`ipicst` + return) | 5 | 5 |
| Return to supervisor | 5 | 6 |
| Detect handler state | 5 | 0 |
| Restore system timer | 14 | 0 |
| Check for message free | 6 | 0 |
| Return from trap | 12 | 14 |
| *cleanup overhead:* | 42 | 20 |
| *receive total:* | 99.5 | 51.5 |
| *Atomic Section → Global (optional):* | >135 | 1 |

Table 6-4: Overheads for message send and receive of a null active message. Operations marked with "⇒" do not consume processor cycles. The "soft atom" numbers are for software emulation of the atomicity mechanism.

In examining these numbers, however, we must keep in mind that the "hard atom" column details a scheduler that has more flexibility and functionality than the original scheduler. To expand on this for a moment, there are two primary differences between these two schedulers, along with a handful of other minor differences. First and foremost, of course, the "hard" version makes use of the hardware atomicity mechanism. One big source of savings from this mechanism is the existence of an additional timer that is automatically managed by the hardware. Since the A-1000 CMMU contains only a single hardware timer that is scheduled to produce a general system timer facility, the "soft" version must save and restore this timer across atomic sections. As seen by Table 6-4, manipulation of this timer is single-handedly responsible for over 24 cycles of additional overhead. Another savings that results from the hardware atomicity mechanism is its automatic tracking of whether or not the user has freed a message during the handler. This is responsible for 7 cycles of difference (there is a hidden setup cycle that is not shown in the table).

A final savings that results from use of the hardware mechanism is the fact that the user is not allowed to enable and disable message interrupts in the A-1000; in general, permitting a user to do so is a serious security risk, especially in a machine such as Alewife for which the operating system relies on use of the network for correct operation. As a result, any disabling of message interrupts on behalf of the user must pass through the kernel; this is one of the origins of the huge cost should a user decide to exit the atomic section and enter the global section of a handler (128 cycles, at the bottom of the table); we will discuss the other major component of this in a moment. Further, when the user requests atomicity (i.e. that user level threads will not run), this is done indirectly: the user sets a bit in a global register. If a message arrives during this period of atomicity, the message interrupt occurs; then, the fact that the user has requested atomicity is recognized before a new thread is scheduled. At such a point, the hardware interrupt is disabled by the kernel and the timer is bypassed to time the resulting atomicity. To exit atomicity, the user executes a special system-call that restores the timer and reenables the interrupt. Note that the check for atomicity is shown in the "soft" breakdown (2 cycles).

The second major difference between these two schedulers is the fact that the "hard" scheduler makes use of featherweight threads. In the original Alewife scheduler, a special context is left free for interrupts. For user-level message handlers, this context is temporarily consumed during the atomic section. Should the user exit the handler without exiting the atomic section, this context can be freed again without violating the normal "interrupt-level" rules of context usage. However, if the user chooses to exit the atomic section, then something else has to happen, since the context must now be allocated on a long-term basis. What the original Alewife scheduler does is check for a free context; if one is not available, it proceeds to unload some other context in order to maintain one free for interrupts. Further, in the process of creating a full "thread", this scheduler must allocate a free thread descriptor structure. This is the primary cost of the *Atomic Section* $\rightarrow$ *Global* operation in Table 6-4. In contrast, the "hard" scheduler caches thread information and preschedules threads so that the handler need only turn off atomicity to enter its "global" section; more on this in a moment. Note that the "soft" scheduler's method of deferring allocation of special thread structures until after the atomic section means that the atomic section must run at higher priority than background; thus, the original scheduler has a number of cycles of manipulation of the interrupt mask that are not present in the "hard" version scheduler.

Use of the user-level atomicity mechanism requires use featherweight threads. The reason for this is that the transition from atomic section to global section is done with a simple, user-level operation. Thus, the kernel is given no chance to reschedule the thread should it choose to reenable message interrupts (*i.e.* exit the atomic section). Thus, the thread must be fully scheduled on entrance to the handler. It is only through the use of featherweight threads that the costs of this are kept low. Even though it would seem that the runtime system must perform *more* work to schedule threads in this way, in fact it performs less. Since much state is cached with featherweight threads, the thread dispatch is much less expensive. Further, since the atomicity section of the thread is at the same priority as a normal user thread (with atomicity enabled), no manipulation of the interrupt mask is necessary. This makes the entrance code (first dispatch) cheaper, as well as the final exit code for trap return. Interestingly enough, the “hard” scheduler detailed in Table 6-4 even keeps better track of background *vs.* message-level priorities and schedules them accordingly. With the general scheduling of threads (and the hardware atomicity mechanism), the transition from atomic section to global section is extremely inexpensive: 1 cycle.

Two final changes with respect to the A-1001 CMMU were exploited by the “hard” scheduler: first, user-level active messages arrive on their own interrupt vector. This saves about 8 cycles of dispatch over the “soft” version (some of which are reclaimed for more functionality). Second, the A-1001 exports information about whether or not the system is in priority inversion and whether the the `desc_length` register is zero to the coprocessor condition codes on loading of this register. This saves about 10–12 cycles in the fast interrupt path[5].

Unfortunately, for reasons of time and problems with the A-1001 CMMU (it runs perfectly well at low speeds, but exhibits strange I/O noise problems at higher speeds), the “hard” scheduler was never incorporated into a complete system. Consequently, the numbers presented in later sections of this chapter are from an Alewife machine with a “soft” scheduler. Appendix A shows the fast-path assembly code for the “hard” scheduler.

## 6.2 Macrobenchmarks and the Alewife Prototype

In this section, we would like to accomplish two things: First, we would like to see the extent to which the Alewife mechanisms lead to good speedups with a variety of different programs. Second, we would like to perform a few comparisons between the same program written in shared-memory and message-passing styles. Many of these numbers have been published elsewhere [3, 28], hence this is a summary of the results. One of the key contributers to the numbers presented in Section 6.2.1 was Ricardo Bianchini, who single-handedly ported a large number of the SPLASH [101] benchmarks. Fredric Chong was instrumental in performing explorations of message-passing applications, as well as explorations of the tradeoffs between shared-memory and message-passing performance (see also Chong’s PhD thesis [27]).

### 6.2.1 Performance of Shared-Memory Programs

As reported in [3], shared-memory programs perform well on Alewife, proving the viability of the overall memory controller architecture, the shared-memory interface, and the two-case delivery mechanism; the operating-system resulted from the combined effort of a number of individual researchers (David Kranz, David Chaiken, Dan Nussbaum, and Beng-Hong Lim, in addition to this author, were among the principal contributers). Table 6-5 summarizes the overall performance of a number of shared-memory applications.

[5]In fact, the “soft” scheduler does not even attempt to detect that a shared-memory invalidation lock is present (see footnote on page 81). This is handled automatically by the A-1001 priority-inversion detection mechanism.

| | Running Time (Mcycles) | | | | | | Speedup | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Program | 1P | 2P | 4P | 8P | 16P | 32P | 1P | 2P | 4P | 8P | 16P | 32P |
| Orig MP3D | 67.6 | 41.7 | 24.8 | 13.9 | 7.4 | 4.3 | 1.0 | 1.6 | 2.7 | 4.9 | 9.2 | 15.7 |
| Mod MP3D | 47.4 | 24.5 | 12.4 | 6.9 | 3.5 | 2.2 | 1.0 | 1.9 | 3.8 | 6.9 | 13.4 | 21.9 |
| Barnes-Hut | 9144.6 | 4776.5 | 2486.9 | 1319.4 | 719.6 | 434.2 | 1.0 | 1.9 | 3.7 | 6.9 | 12.7 | 21.1 |
| Barnes-Hut * | – | 10423.6 | 5401.6 | 2873.3 | 1568.4 | 908.5 | – | 2.0 | 3.9 | 7.3 | 13.3 | 22.9 |
| LocusRoute | 1796.0 | 919.9 | 474.1 | 249.5 | 147.0 | 97.1 | 1.0 | 2.0 | 3.8 | 7.2 | 12.2 | 18.5 |
| Cholesky | 2748.1 | 1567.3 | 910.5 | 545.8 | 407.7 | 398.1 | 1.0 | 1.8 | 3.0 | 5.0 | 6.7 | 6.9 |
| Cholesky * | – | – | 2282.2 | 1320.8 | 880.9 | 681.1 | – | – | 4.0 | 6.9 | 10.4 | 13.4 |
| Water | 12592.0 | 6370.8 | 3320.9 | 1705.5 | 897.5 | 451.3 | 1.0 | 2.0 | 3.8 | 7.4 | 14.0 | 27.9 |
| Appbt | 4928.3 | 2617.3 | 1360.5 | 704.7 | 389.7 | 223.7 | 1.0 | 1.9 | 3.6 | 7.0 | 12.6 | 22.0 |
| Multigrid | 2792.0 | 1415.6 | 709.1 | 406.2 | 252.9 | 165.5 | 1.0 | 2.0 | 3.9 | 6.9 | 11.0 | 16.9 |
| CG | 1279.2 | 724.9 | 498.0 | 311.1 | 179.0 | 124.9 | 1.0 | 1.8 | 2.6 | 4.1 | 7.1 | 10.2 |
| EM3D | 331.7 | 192.1 | 95.5 | 46.8 | 22.4 | 10.7 | 1.0 | 1.7 | 3.5 | 7.1 | 14.8 | 31.1 |
| Gauss | 1877.0 | 938.9 | 465.8 | 226.4 | 115.7 | 77.8 | 1.0 | 2.0 | 4.0 | 8.3 | 16.2 | 24.1 |
| FFT | 1731.8 | 928.0 | 491.8 | 261.6 | 136.7 | 71.8 | 1.0 | 1.9 | 3.5 | 6.6 | 12.7 | 24.1 |
| SOR | 1066.2 | 535.7 | 268.8 | 134.9 | 68.1 | 32.3 | 1.0 | 2.0 | 4.0 | 7.9 | 15.7 | 33.0 |
| MICCG3D-32-Coarse | – | 36.6 | 21.7 | 11.7 | 6.9 | 4.4 | – | 0.5 | 0.8 | 1.5 | 2.5 | 3.9 |
| MICCG3D-32-Fine | – | – | 11.7 | 5.8 | 2.9 | 1.5 | – | – | 1.5 | 3.0 | 5.9 | 11.5 |
| MICCG3D-64-Coarse | – | – | – | – | – | 32.2 | – | – | – | – | – | 4.3 |
| MICCG3D-64-Fine | – | – | – | – | – | 12.5 | – | – | – | – | – | 11.1 |

Table 6-5: Performance of shared-memory applications on Alewife.

The results show that Alewife achieves good application performance, especially for the computational kernels, even for relatively small input sizes. In particular, MP3D (an application with a difficult shared-memory workload) achieves extremely good results. In contrast, a comparison between the two entries for Cholesky in the table demonstrates the importance of the input size for the performance of this application; a 5-fold input size increase leads to a significant improvement in speedup. The modest speedups of CG and Multigrid result from load imbalance and bad cache behavior, which can be addressed by using larger input sizes and the latency tolerance mechanisms in Alewife.

The important point to glean from this table is that the Alewife shared-memory mechanisms are efficient enough to yield good speedups, even though they are fully integrated with message-passing mechanisms. One of the principal factors contributing to this result is pipelining in the transaction buffer and service-coupled scheduling of hardware resources.

The other thing to note from the large number of ported applications is that Alewife provides a good environment for developing applications. Programs can be easily ported to the machine and can achieve good performance.

### 6.2.2 Performance of Message-Passing Programs

The set of message-passing applications written over the course of the Alewife project was much smaller than the set of shared-memory applications. One of the reasons for this is that the message-passing development environment was somewhat later in arriving than the shared-memory environment. However, the message-passing capabilities of Alewife did inspire several independent studies by individual Alewife members. We will simply summarize these results here and forward the reader to the corresponding references.

David Chaiken made extensive use of small and fast messages to complete and enhance the

Alewife cache coherence protocol [24]. The speed of the message-passing interface is evidenced by the fact that fact that Alewife shared memory performs well across a range of applications (previous section). Thus, the success of LimitLESS provides insight into the speed of message-passing mechanisms.

Further, Alewife members developed two complete programming systems as user-level libraries on top of the Alewife user-level message-passing system: the C Region Library (CRL) by Kirk Johnson [52, 51] and the Multi-Grain Shared-Memory system (MGS) by Donald Yeung [120, 119]. Both of these systems make extensive use of the UDM messaging model, complete with software versions of the user-level atomicity mechanisms (the "soft" runtime system described in Section 6.1.3). Both of these systems achieved good performance while introducing software cache-coherence protocol overhead and explicit message-passing. These two systems represent the ideal target for the message-passing model of Alewife, because they operate at user level, make full use of atomicity mechanisms, and (in the case of MGS) make frequent use of message handlers that disable atomicity after freeing their messages.

Finally, Fred Chong wrote a number of fine-grained message-passing applications for Alewife. Among other things, he explored sparse-matrix kernels, achieving speedups in message-passing versions on Alewife that surpassed the best known speedups at the time. These explorations are shown in greater detail in [27]. What is interesting about Fred Chong's work (and particularly apropos for this thesis), is that he coded several of these high-communication-volume applications in different message-passing and shared-memory styles and discovered that their performances are, in many cases, comparable.

### 6.2.3 Integration of Message Passing and Shared Memory

As we have indicated in the previous sections, both message-passing and shared-memory communication mechanisms perform well in the Alewife multiprocessor. What we would like to show in this section is that Alewife mechanisms are efficient enough that some of the inherent properties of the individual mechanism (such as bandwidth requirements) start effecting performance. We will restrict our attention to the characteristics of one application (EM3D). This is joint work with Fred Chong (who took most of the actual numbers). See [28, 27] for more details and other examples.

EM3D from Berkeley models the propagation of electromagnetic waves through three-dimensional objects [80]. It implicit red-black computation on an irregular bipartite graph. EM3D is iterative and is barrier-synchronized between two phases. Communication takes place because data updated within one phase is used by other nodes in the subsequent phase. In [28], EM3D is written in five different versions (three message-passing and two shared-memory versions):

1. Fine-grained message-passing with interrupt-driven active messages ("int-mp"). This version communicates five double-words at a time.
2. Fine-grained message-passing with polling ("poll-mp"). This version also communicates five double-words at a time.
3. Bulk-transfer message-passing ("bulk-mp"). This version explicitly gathers communication data into a contiguous buffer for subsequent DMA transfer. It makes use of interrupt-driven active messages with DMA. EM3D allows for large enough

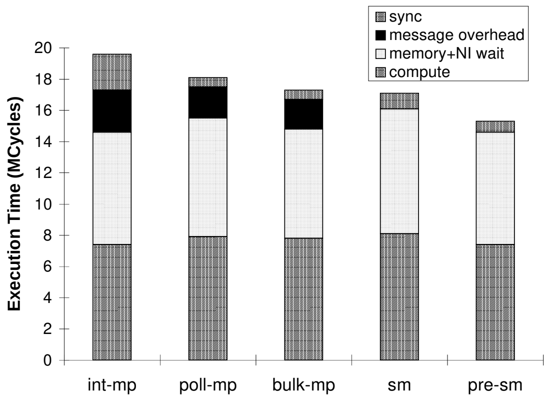



Figure 6-2: Performance breakdown for the five different versions of EM3D.

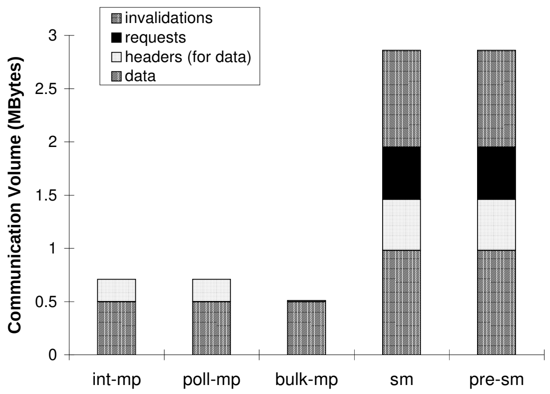



Figure 6-3: Communication volume for the five different versions of EM3D.

DMA transfers to cover data gathering overhead. Preprocessing of outgoing data allows it to be used in-place after it is DMA'ed into memory at the destination. This preprocessing code, however, is extremely complex and represents high coding effort.

4. Shared memory without prefetching ("sm"). The shared-memory implementation of EM3D is much simpler because neither preprocessing code nor pre-communication steps are required. Barriers provide synchronization between phases.

5. Shared memory with prefetching ("pre-sm"). A write-prefetch was issued to get write-ownership of data structures before the computation for that data begins. In addition, read prefetches were used to fetch values two computations ahead of use. Inserting of these prefetches was straightforward, requiring only three lines of code.

Each of these versions of EM3D were carefully tuned so that they made best use of the target communication mechanisms.

Figure 6-2 shows that shared memory and message passing perform competitively with one another (within 30%). Bulk-transfer gains from lower overheads of DMA transfer, but pays the costs of message aggregation and software caching. The fine-grained versions suffer from higher message overhead, which offset the benefits from avoid the costs of aggregation. Also, as seen by this figure, EM3D benefits from prefetching. It is interesting to compare these performance results with the communication volume of Figure 6-3. As we see from this figure, the shared-memory version of EM3D consumes more than a factor of five (5.6) over the bulk message-passing version. Thus we can see that, although shared-memory and message-passing versions of EM3D may perform roughly equivalently on Alewife, they have vastly different communication requirements.

These differences in communication behavior can be expected to cause differences in performance for systems that have more restricted bandwidth, *i.e.* message-passing programs would be expected to perform better than shared-memory programs in low bandwidth environments. Further, message-passing programs have a tendency to be far more latency tolerant than shared-memory programs simply because they are written with unidirectional communication: this encourages

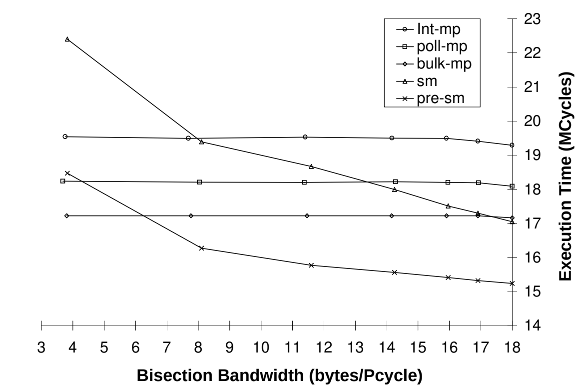



Figure 6-4: Variation in performance of the five different versions of EM3D as a function of the available bandwidth across the bisection of a 32-node Alewife machine. Normal Alewife bandwidth is at 18 bytes/cycle (far right of graph).

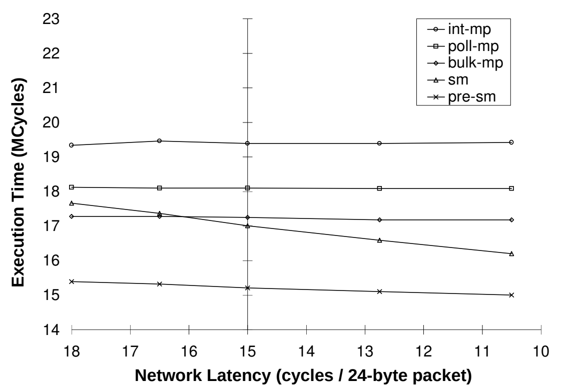



Figure 6-5: Variation in performance of the five different versions of EM3D as a function of one-way network latency. At 20 MHz, a 32-node Alewife machine takes approximately 15 cycles to transmit a 24-byte message across the bisection.

a coding style that has many messages in flight at any one time, each of which may invoke an independent computation. In contrast, shared-memory programs tend to have fewer outstanding requests and have performance that is often directly tied to the round-trip latency of the network. Thus, message-passing programs would be expected to show a performance advantage in systems that have higher network latencies as well. What is interesting about the Alewife machine is that the communication mechanisms that it supports have small enough overheads that they are limited in performance primarily by the network and DRAM. This means that we can derive simple experiments to show differences in the fundamental behavior of the mechanisms themselves. Such experiments are reported in detail in [28], but we will show samples of them for EM3D.

Figure 6-4 show the performance effects on EM3D of restricting the total amount of bandwidth across the bisection of a 32-node Alewife machine. The unrestricted number is approximately 18 bytes/processor cycle across the bisection (which is why the axis is at 18). Numbers to the left of this represent restricted bandwidth. The bandwidth restriction experiment is performed by using independent I/O nodes to send message traffic (small messages) across the bisection of the Alewife machine. For any particular run, the total number of messages successfully sent that way is combined with the runtime to compute the total bandwidth taken from the bisection; this is used to compute the bandwidth remaining for the application, which ultimately produces the X-axis value in this figure. As can be seen by this figure, although the shared-memory versions of EM3D are initially faster than the message-passing versions, this situation changes for restricted bandwidth. In fact, for sufficiently restricted available bandwidth, both of the shared-memory versions cross over the message-passing versions of this program. Given the communication volumes of Figure 6-3, this is not surprising. Further, since this graph is showing performance as a function of bandwidth/processor cycle, a general leftward trend is expected as processors become faster relative to their underlying communication networks.

In fact, given the asynchronous network of Alewife, we can alter the relative performance between the network and processors by changing the processor clock rate. This is, perhaps, a more faithful variation of the relative performance between processor and network than the bandwidth restriction experiment. Obviously, the clock rate cannot be varied by too large an overall fraction[6]. However, given the variation that is possible, we can see a bit of the crossover between unprefetched shared-memory and message-passing versions in Figure 6-5. In this graph, the 20MHz Alewife machine comes in at approximately 15 cycles/24-byte message across the machine (our measure for the X-axis).

Thus, the important fact to glean from both Figures 6-4 and 6-5 is that Alewife supports mechanisms that have low enough overheads that their performance is limited by network characteristics and DRAM bandwidth; as a result we can contrive simple experiments to show variations in the *fundamental* behavior of these mechanisms. It is the service-coupled hardware scheduling that is responsible for this efficiency.

## 6.3 How Frequent IS Deadlock?

In this final section, we would like to explore two-case delivery from the standpoint of several questions: First, how frequent *is* deadlock anyway? If protocol level deadlock arises frequently under normal circumstances, then the software overhead of two-case delivery could easily become prohibitive. Along similar lines, the frequency of deadlock will be affected by the a number of machine and application characteristics; which of these is most important to avoiding deadlock? Second, for a given frequency of deadlocks, how well do the atomicity congestion heuristics of the previous section (Section 4.3.2) detect deadlock? Since our heuristics are conservative by design, this question is more properly stated: to what extent do local detection heuristics avoid triggering network overflow recovery for non-deadlocked (but badly congested) communication patterns. Finally, from a purely academic point of view, is there any pattern that gleaned in the distribution of occurrences of deadlock?

### 6.3.1 Alewife: A Brief Case Study.

In answering these question, we start our examination of the frequency of deadlock by asking the question: how well does two-case delivery perform on Alewife? This question is addressed by Figures 6-6 and 6-7. Figure 6-6 presents the runtimes (in billions of cycles) of kernels for a number of different applications. Some of these are from the Stanford SPASH benchmark suite[101], while others were written during the course of the Alewife project. Figure 6-7 shows the impact of two-case delivery on these execution times. The vertical axis in this case is in percentage of execution. Interestingly enough, none of the applications experience more than a 0.5 percent slowdown, and, more importantly, *most of them experience no invocations of network overflow recovery at all* – the missing bars are actually zero.

Several observations can be made about this data. First, some of the results are sensitive to the setting of the timeout (or hysteresis) value in the network deadlock detection monitor. In general, message-passing applications (ones that have "mp" in the communication designation after

[6] In fact, the only reason that the clock-rate can be raised relative to the "base" rate of 20MHz is because 20MHz does not represent the maximum rate of the CMMU – see discussion of the FPU bug at the beginning of this chapter.

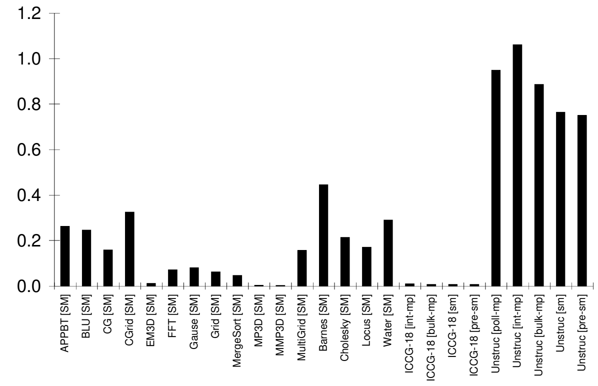



Figure 6-6: Runtimes for Alewife applications (in billions of cycles).

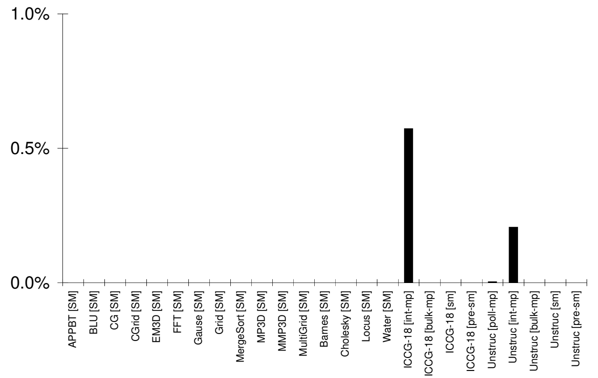



Figure 6-7: Percentage of execution time spent in overflow recovery.

their name) are more subject to triggering of the network overflow heuristic than shared-memory programs. Figure 6-7 present results obtained with a timeout value of 4096 cycles. For the shared-memory applications, this timeout value can be lowered to 128 cycles before *any* atomicity congestion interrupts occur; in fact, the atomicity congestion timeout may be completely disabled and the shared-memory applications will continue to run. In contrast, the message-passing programs of Figure 6-7 incur an increase of *an order of magnitude* in the number of cycles for network overflow recovery when the timeout value is lowered to 2048. We can draw two conclusions about two-case delivery on Alewife from these results:

- For the particular combination of number of processors, queue sizes, packet sizes, and number of outstanding requests present on Alewife, shared-memory rarely runs into deadlock.
- Although message-passing applications have a tendency to cause more network congestion than shared memory, much of this congestion does not cause actual deadlock but rather reflects weaknesses of the simple congestion detection heuristic.

These results are encouraging and suggest that two-case delivery was the correct choice for Alewife (perhaps with a better detection heuristic). In [78], two-case delivery is further explored in the context of a multi-user system.

However, to understand two-case delivery in more detail, we need the ability to vary parameters in a way that is not possible with any particular hardware platform. Further, answers to some of the questions introduced at the beginning of Section 6.3 require a greater level of visibility into network queues than possible with actual hardware.

### 6.3.2 The DeadSIM Simulator

As illustrated in the previous section, two-case delivery works reasonably well in at least one machine, namely Alewife. One of the most important determinants of its success relative to other more design intensive methods (such as explicit virtual channel management), is the extent to which protocol deadlock is rare. Unfortunately, the circumstances and frequency with which protocol-level deadlock will occur is highly dependent on a number of application and system-level parameters, such as request rates, access patterns, queue lengths, and network topology. As a consequence, the feasibility of two-case delivery as a general design methodology is hard to evaluate on any one architecture. Further, the efficacy of local detection and correction techniques depends on the nature and topology of deadlocks once they occur — information that is hard if not impossible to acquire on a "real" machine. All of this argues for simulation as a technique for exploration.

Of course, simulation is notorious in its ability to produce reams of meaningless data. This tendency may be countered in one of two ways:

1. Build detailed (and complicated simulations) that are carefully validated against some "real" architecture. Detailed questions can then be asked, but the results may not be particularly generalizable.
2. Build simple simulations (not necessarily "realistic" in all of their details), then ask broad questions.

Since we would like to ask some general questions about two-case delivery in general, we will choose the later option — a simple queue-level simulation with enough realism to model wormhole routed networks and simple, depth-two coherence protocol. Enter DeadSIM.

DeadSIM is a simulator that was specifically designed to explore protocol-level deadlock. Among other things, it seeks to address the question that was the title of major Section 6.3, "How frequent *is* deadlock?" We would like to get an overall sense of whether protocol deadlock is, in general, a rare occurrence or not. However, as stated, this question is perhaps a bit misleading. Since no one number would be sufficient for all situations, we will seek, instead, to ask questions like: given such-and-such an access pattern, how does the frequency of deadlock vary as a function of queue size, or number of nodes, etc. Further, we would like to explore the extent to which deadlocks are localized (*i.e.* involve a small number of nodes). Last, but not least, we would like to discover whether or not the queue-level deadlock detection heuristics mentioned in Section 4.3.2 are good at avoiding false detections. This is broad set of goals that could easily get out of hand: we will attempt to use DeadSIM in moderation.

So, what exactly is DeadSIM? DeadSIM is a "message-driven," queue-level simulator that simulates a wormhole routed network of processors in one of two different topologies: crossbar and mesh (of many different dimensions). Queues are of *fixed* size, so resource contention effects are fully simulated. Messages work their way through the network; portions of these messages can be simultaneously present in multiple queues (this is the result of wormhole routing).

Only *direct* networks are simulated, *i.e.* networks for which each intermediate node supports a processor. The *mesh network* routers are somewhat idealized in that they have complete internal crossbars (within the number of dimensions) and block only on their output queues. Like Alewife, network connections are bidirectional but support only a single logical channel in each direction. The *crossbar network* represents a congestion-free network. It has no network routers, but rather provides a small input queue from every source to every destination; hence, the only contention occurs within the source and destination nodes.

**Simulation Assumptions:** In this section, we are going explore simulated processor nodes that perform request/response-style communication (*i.e.* shared memory). The reason that we explore shared memory is three-fold: (1) shared memory is relatively easy to characterize in terms of number of outstanding requests; (2) studies have been done about the communication behavior of shared-memory programs in terms of communication rates and number of outstanding requests; we will make use of such results; and (3) deadlock recovery with message passing can be handled efficiently via two-case delivery if we have a system that has a UDM-style network interface which supports transparency (see Sections 2.4.4 and 4.3.5). In DeadSIM, the total number of outstanding requests is tunable, as is the size of request and response messages. As with Alewife, we assume that the memory system operates independently of the processor and provides deterministic service times for requests (moderated by queue blocking).

Processor requests are to random destinations and are assumed occur after computation sections whose length is exponentially distributed. The processor will always wait to send a request if the current number of requests is at its maximum, roughly modeling shared-memory applications that have some maximum communication parallelism that they can exploit (or alternatively, modeling architectures that have some maximum number outstanding requests that can be exploited). The mean length of the computation sections (*i.e.* the mean of the exponential distribution) is often called the *run-length* and is very application dependent. We will be exploring a range of run-lengths and sanity checking this range with results from Lim and Bianchini [76] in which they measure run-lengths and prefetching behavior from real applications on Alewife.

**Deadlock Detection:** Since our primary interest in this section is in the frequency of deadlock, DeadSIM includes a deadlock-detection algorithm that detects cyclic dependencies between messages. The simulation starts with an empty network. After a brief startup delay, the network reaches a steady state, during which the deadlock detection algorithm continually monitors the state of the network. Whenever a cycle in dependencies is detected, the simulation is stopped and information is recorded about the deadlock (such as number of cycles that the system successfully completed before the deadlock occurred (we call this the *deadlock-free interval*), the number of messages participating in the deadlock, and the number of nodes involved in the deadlock). To develop a statistical profile, each set of parameters is run multiples times with different random seeds, allowing us to collect the averages and standard deviations of parameters[7].

### 6.3.3 On the Character of Deadlocks

Our first task with DeadSIM is to explore the nature of deadlocks, focusing on their topology and consumption of resources. Simulation crucial for this type of exploration, simply because it affords direct access to all of the network queues, a feat that is difficult (if not impossible) to achieve with real hardware. The following, high-level patterns emerge in our explorations:

[7]The actual heuristic for this is to run until the total number of cycles simulated is equal to some preset value (10 million cycles). Although this yields a varying number of sample points for each parameter set (depending on the length of the deadlock cycle), it was more convenient for generating a large variety of results — especially since many combinations of parameters never enter deadlock at all.

**Deadlock Involves a Small Number of Nodes:** One of the most striking results that arises immediately from this type of exploration is that, as long as network queues are large enough to hold a small number of complete messages, *the majority of deadlocks are between two nodes in a system, regardless of the number of nodes or the set of machine parameters.* The one case for which this is not necessarily true is for a crossbar network which exhibits no network congestion. In that case, many-node deadlocks are possible, although *much* harder to come by. Any network contention at all (*e.g.* from a mesh network) seems to restrict the number of nodes to two[8]. This indicates that a *localized* deadlock recovery algorithm (such as two-case delivery) is a good choice for recovering from deadlock, since only a small number of nodes must be interrupted to remove deadlock.

**Deadlock Is Memoryless:** The second general result that arises is the fact that the process of generating deadlock is memoryless. This property can be seen from two facts: (1) that the mean and standard-deviation of the deadlock-free intervals are roughly equal (minus a slight startup transient) regardless of the set of simulated parameters and (2) the resulting distribution is basically exponential in appearance. Although the first of these hints at the second, it is really the second that confirms this conclusion. For example, Figure 6-8 shows a histogram of the number of deadlocks for a particularly stressful set of parameters (with a mean deadlock-free interval of approximately 18306 cycles and a standard-deviation of 14235). Note the startup transient at the beginning; this is the reason that the standard-deviation is somewhat less than the mean. Interestingly enough, this memoryless property is reasonably *independent* of the distribution of run-lengths that is used; for instance, it appears with a normal and deterministic distribution as well; obviously, however, *some* randomness in machine behavior is necessary (in this case, random message destinations), lest overly deterministic behavior produce a discrete deadlock distribution[9].

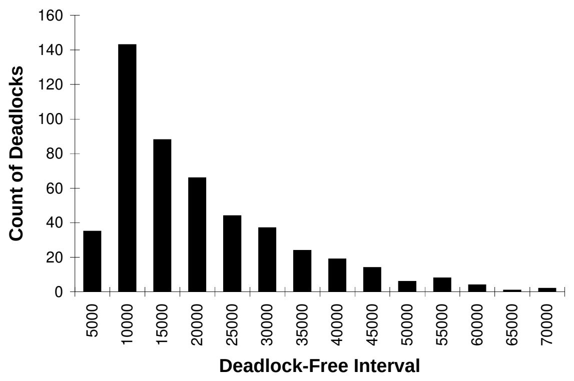



Figure 6-8: Distribution of deadlock-free intervals for a particular set of parameters. The mean of this distribution is 18306 cycles and the standard-deviation is 14235.

One conclusion that we might draw from the memoryless property of deadlocks is that a deadlock recovery mechanism should buffer a small number of messages, then quickly exit; there is no particular advantage to waiting for a long period before restarting the computation on a given node. In the original Alewife system, the deadlock recovery mechanism did not behave this way; instead, it waited until the output queue drained before continuing. This was done since buffering was a relatively expensive operation to initiate (see the discussion of Software Complexity in Section 4.3.7). However, with transparency, the network overflow process could be much lower overhead, and quick recovery would be a more attractive prospect. Note that one aspect that is not

[8]The standard-deviation of the number of nodes is effectively zero: The occasional three-node deadlock is extremely rare and no examples appear of deadlocks with four or more participating nodes.

[9]In fact, there were one or two well-defined, deterministic patterns, used during testing of the CMMU, that were guaranteed to deadlock the network; this allowed verification of the deadlock-detection mechanism.

taken into account by the DeadSIM simulation is the fact that the overflow buffering process has a bit of a retarding effect on overzealous producers of messages; thus, the ideal behavior of two-case delivery with respect to system-level messages merits additional study.

**Deadlock Depends on Routing Freedom:** As shown in Figure 6-9, another interesting property of deadlock formation is the fact that the frequency of deadlock decreases with increasing network dimensionality. This result is similar to results by Timothy Pinkston and Sugath Warnakulasuriya with respect to message deadlocks in networks with adaptive routing [90]. It is, perhaps, not entirely surprising given the fact that we are choosing random destinations for our messages. However, it does say something about the formation of deadlocks. The experiment that produced Figure 6-9 was done with very little buffering in the network (to avoid confusing effects due to additional buffering from those due to increased routing freedom). As a result, packets typically stretch across the network; thus, one advantage to be gained from additional network dimensions, is that fewer packets are likely to be delayed behind packets that happen to be blocked in the network. This suggests that the formation of deadlocks begins with packets that are blocked *on their way to their destinations*, rather than at their destinations. Although the final dimensionality of a network is fixed by many constraints, our result indicates that high-dimensional networks are more desirable from the standpoint of avoiding deadlock cycles.

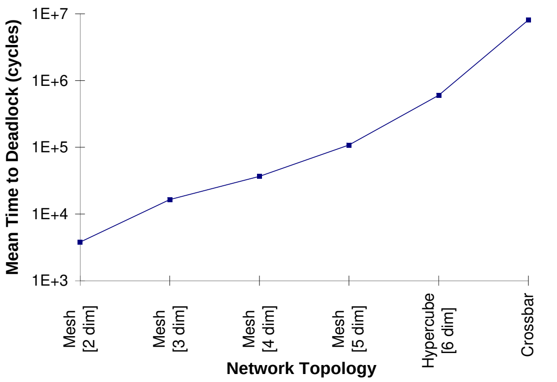



Figure 6-9: Deadlock-free interval as a function of routing freedom for high-stress communication pattern. This simulation uses 64 processors.

**Deadlock Depends on Queue Size:** The final property of deadlocks that we would like to mention here is (perhaps) obvious: the probability of deadlock depends *strongly* on the amount of resources in the network input and output queues of the node. This is illustrated by Figure 6-10. The simulation reported in this figure makes use of 256 processors, each of which can have a maximum of 4 outstanding requests. It uses Alewife-sized cache-lines (8-byte request packets and 24-byte response packets). In this figure, we see that the mean time to deadlock varies by over three orders of magnitude as we vary the queue size by less than a factor of eight. Looking at this another way, at 384 bytes, we have enough queue space in either the input or output queues to hold three times the number of outstanding requests (and their responses). At that design point, deadlock occurs every two-million cycles (or, given the mean run-length of 20 cycles, every 100,000

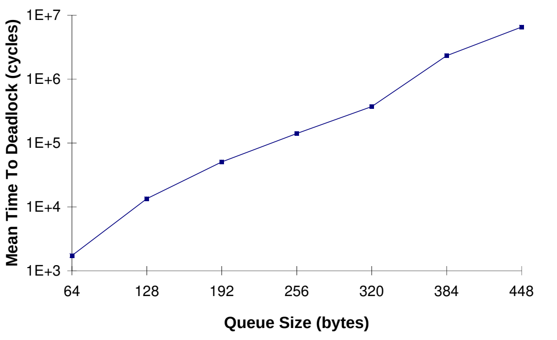



Figure 6-10: Deadlock-free interval as a function of network queue size. This simulation uses 256 processors, a maximum of 4 outstanding requests, and cache-lines of Alewife proportions.

cache-misses). This means that a deadlock-recovery process could take 20,000 cycles and increase runtime no more than one percent. Note further that these results are for a particularly stressful set of parameters: with the DRAM and network parameters used here, the maximum throughput of remote memory requests is one cache-line every 12 cycles (without contention); hence, with a 20-cycle average run-length, we are running pretty close to maximum memory throughput.

This relationship between frequency of deadlock and queue resources is, perhaps, obvious. However, two developments make this particularly apropos: (1) Modern, dynamically-scheduled processors tend allow a small number of outstanding requests since they track pending memory operations with special, transaction-buffer-like hardware structures; the R10000 [92], for instance, allows no more than four memory requests to be outstanding at any one time, and (2) The integration of memory and processor technology continues to improve, easily supporting large network queue sizes. Together, these suggest that implementing network queues with enough space to hold several times the number of outstanding requests is not only possible, but reasonable. Of course, this more difficult when long message-passing style messages are present in the network; however, we have already noted that deadlock recovery for message-passing messages is a relatively low-overhead operation, given transparency and two-case delivery.

### 6.3.4 The Deadlock-Free Interval with Alewife Parameters

So now, we turn to our primary question: how frequent *is* deadlock? In attempting to answer this question, we can vary a large number of individual system parameters, so we must be a bit careful about what we choose to study. The key parameter that we would like to measure is the deadlock-free interval, *i.e.* the mean time between deadlocks for a given set of parameters. Our exploration of deadlock with respect to queue size in the previous section has already indicated that deadlock can be made relatively infrequent by increasing queue sizes. However, it is perhaps more interesting to stick within one particular set of sizings and see the variation that arises. Thus, we will use Alewife parameters for a set of experiments: 2-dimensional mesh, 256-byte network input and output queue sizes, 19-byte queues within the network (an EMRC value), 8-byte request packets, 24-byte response packets, and 2-bytes/cycle of network latency.

In [76], Beng-Hong Lim and Ricardo Bianchini explore typical run-lengths for a number of Alewife applications written in both multithreading and prefetched style. They discovered that most of these applications have run-lengths over 100 cycles and can take advantage of no more than four outstanding requests. Given these results, it is interesting to note that:

---

*With Alewife parameters, a 100-cycle run-length, and four outstanding requests, neither 64-node machines nor 256-node machines experience deadlock.*

---

In fact, with a 100-cycle run-length, a 64-node machine does not experience deadlock even with 12 outstanding requests! With a 40-cycle run-length (less than all measured applications in [76]), a 64-node machine does not start experiencing deadlock until it reaches six outstanding requests, and then the mean time to deadlock is at the limits of our 10-million-cycle cutoff.

Figure 6-11 illustrates the deadlock-free interval for 256 processors, with 4 and 6 outstanding requests, at a variety of run-lengths. We have truncated this data at a 125-cycle run-length simply because it becomes "infinite", *i.e.* deadlock no longer occurs for either configuration. Further, reflecting our discussion of deadlock and routing freedom, *these same run-lengths produce* no *deadlock for a 4-dimensional mesh network.*

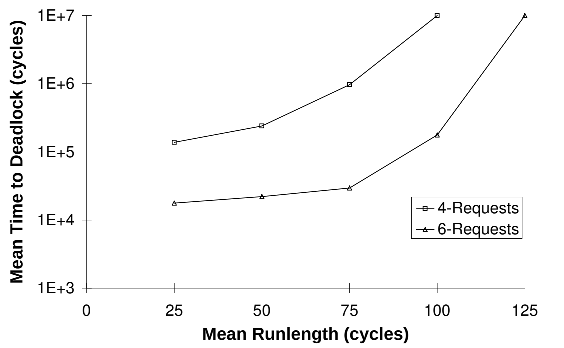



Figure 6-11: Deadlock-free interval as a function of run-length for 4 and 6 outstanding requests. This simulation uses 256 processors and Alewife parameters.

What is the primary conclusion that we can draw from the above experiments? Although deadlock can be made to occur for "high-stress" computational parameters, the previous sections indicate that it is relatively easy to depress the levels of deadlock via simple techniques such as increasing the local queue sizes, decreasing the run-length, or increasing the network routing freedom. In fact, a systems designer might use any or all of these techniques to make two-case delivery a viable alternative to other deadlock-avoidance techniques. Further, UDM-style interfaces have low enough overheads to enable rapid deadlock recovery; hence, a relatively high-frequency of deadlock could be tolerated without serious impact on performance. Such "micro-deadlock recovery" would involve the quick removal and buffering of a small number of messages from the network as a method for removing deadlock[10]. However, the next section shows that the deadlock-detection heuristic imposes a lower-bound on the latency involved in detecting deadlock, ultimately limiting the degree to which we can exploit quick deadlock detection and recovery.

### 6.3.5 Deadlock Detection and the Importance of Hysteresis

The deadlock detection algorithm that we have been using in the previous sections requires intimate knowledge of the contents of the queues on all nodes. This is not at all practical for a real implementation. Thus, as a final topic of exploration for DeadSIM, we would like to examine the efficacy of the two queue-level deadlock detection algorithms described in Section 4.3.3. The first of these (which we will call "OUTCLOGGED") triggers if the network output queue is *full* and *not moving* for a preset timeout period. The second of these (which we will call "IOCLOGGED") triggers if both the input *and* output queue are full and not moving for the timeout period. To explore these heuristics, DeadSIM implements a high-level monitoring facility that keeps track of the input and output queues for each simulated node. This is sufficient information to decide on which cycle (if any) a given heuristic would trigger.

To quantify the behavior of heuristic deadlock detection with respect to the presence of actual deadlocks, DeadSIM collects statistics on a parameter that we will call the *heuristic offset*. This value is the difference between the number of cycles into a DeadSIM simulation that the heuristic triggers and the number of cycles until an actual deadlock occurs. In practice, this means that for each simulation, we start with an empty network, then simulate until *both* a real deadlock and

[10] This is appropriate, given the memoryless nature of the deadlock process. See above discussion.

a heuristic detection event have occurred. We proceed to perform this multiple times, gathering statistics as we go.

Ideally, we would like the heuristic offset to be as small in absolute value as possible; this would mean that the heuristic was doing a really good job of detecting *actual* deadlocks. As we shall see, the deadlock detection heuristic exhibits an "ideal value" of the heuristic timeout: if the timeout is too small, then the heuristic triggers way too easily and the heuristic offset is large and negative; if the timeout is too large, then deadlock detection triggers only after deadlock has occurred – forcing the heuristic offset to be positive and roughly equal to the timeout value.

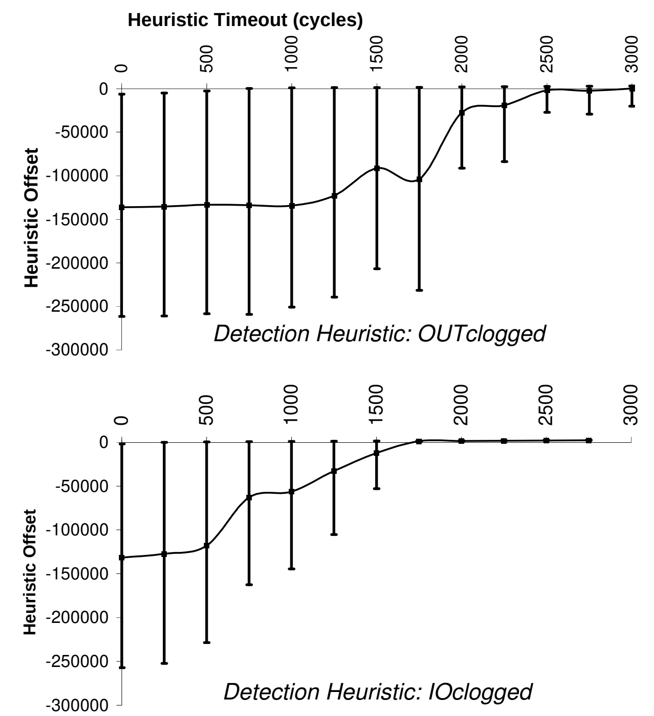



Figure 6-12: Plot of the heuristic offset as a function heuristic timeout for two deadlock-detection heuristics. These simulations are with 256 processors. The mean of the deadlock-free interval is 136,000 cycles.

Our results are exhibited by the two graphs in Figure 6-12. These graphs explore the average value of the heuristic offset for our two different heuristics as a function of the heuristic timeout value. The error-bars positive error bars report the maximum value of the heuristic offset, while the negative error bars report the magnitude of the standard-deviation in the heuristic offset[11]. For simulation parameters, we have made use of the Alewife parameters again (see previous sections), with 256 processors and a mean run-length of 30 cycles. This set of parameters yields a deadlock-free interval of approximately 136,000 cycles (*i.e.* we have chosen a high-stress set of parameters). Heuristic offsets are shown in cycles with error bars representing the standard-deviation of the values.

One conclusion is immediately obvious from these graphs: the hysteresis provided by the heuristic timeout is *extremely important* to the performance of the deadlock detection algorithm. For small timeout values, the heuristic offset is large, negative, and approximately equal to the deadlock-free interval. What this indicates is that mesh networks can exhibit long periods in which output queues are full and not moving. In this regime, the triggering of the detection is completely uncorrelated with the occurrence of actual deadlocks (as partially evidenced by the large deviation in the heuristic offset). This result implies that we should always wait out the presence of temporary queue blockages — in many cases they will go away. As a case in point, the J-machine, which triggered a network overflow interrupt as soon as either the input or output queue became full [73], introduced a lot of software overhead to avoid non-existent deadlocks.

As the heuristic timeout increases, the heuristic offset slowly reduces in *magnitude* until it be-

[11] Since these distributions have huge "tails", we are not reporting actual minimum values. These error-bars are there to indicate the magnitude of the deviation in heuristic offset.

comes positive. At this point, the timeout is long enough to avoid being triggered by spurious blockages and is triggered by actual deadlocks instead. In this regime, the triggering of the heuristic is directly correlated with the presence of deadlock, as evidenced by the relatively low deviation in the heuristic offset (essentially invisible here). Further, the heuristic offset varies almost directly with the heuristic timeout because the typical sequence of events is that deadlock occurs first, followed by the countdown of the congestion timeout. Because the standard-deviation and mean are small, this regime is the ideal operating point for our detection heuristic — it is the most accurate and directly correlated with deadlock. Note that we would like to pick the smallest value of the heuristic timeout which causes a crossover to the positive correlated regime, since this triggers the deadlock recovery process as soon as possible after deadlock has occurred.

Another conclusion that we can draw from Figure 6-12 is that the IOCLOGGED heuristic is more effective than the OUTCLOGGED heuristic for detecting deadlock: the IOCLOGGED heuristic becomes correlated with actual deadlock at a smaller value of the heuristic timeout. The reason for this behavior is simply that there are scenarios for which the output queue is clogged but the input queue still has space — the latter heuristic ignores these situations. Since the necessary timeout value is smaller, the heuristic can trigger closer to actual deadlock.

Finally, we should stress that the minimum timer value that we can tolerate and still achieve faithful deadlock detection represents a minimum granularity at which we can detect and recover from deadlock. Given the memoryless nature of deadlocks (discussed above), we would like make this detection time as short as possible so as to allow quick removal and buffering of a small number of messages. However, some residual detection latency will always exist; thus, a tradeoff exists between the accuracy of the detection heuristic and the granularity of deadlock recovery. This is clearly a topic for further study.

### 6.3.6 Future DeadSIM work

Our use of DeadSIM has done little more than scratched the surface with respect to characterizing protocol-level deadlocks in a multiprocessor. What is encouraging about the results that we have seen is that *actual* deadlocks are relatively infrequent for realistic systems parameters. When they occur, they tend to involve a small number of nodes, indicating that local recovery mechanisms (such as two-case delivery) are a good idea. In addition, we have seen that the local deadlock detection heuristics presented in Section 4.3.3 can be made to be quite accurate at detecting actual deadlock.

Future work should include more interesting models for processor requests. In particular, we should include more "message-passing-like" communication. One possibility for artificial message-passing communication involves the concept of "Snakes" as used by Rich Lethin in his thesis [73]. These multi-hop messages express more of communication freedom present in message-passing algorithms. Also, further DeadSIM work should include more deterministic communication patterns and some notion of locality of memory access; this latter aspect is interesting because of the strong dependence of deadlock on number of nodes as illustrated by our current random simulation.

# Chapter 7

# All Good Things . . .

In this final chapter, we would like to draw some conclusions about the integration of message-passing and shared-memory communication, and, in particular, explore some of the lessons that we might take away from the Alewife prototype. The reach of the Alewife project as well as the scope of its implementation encompassed a large number of interrelated ideas. Further, many aspects of the Alewife system were designed from scratch (in particular, the integrated memory and communication subsystems). As a result, structures within the Alewife CMMU are tremendously more "organic" than they would be if the Alewife machine had been implemented by adapting pieces of a preexisting implementation. This makes evaluating the worth of individual hardware structures difficult, however, since such such structures often serve multiple functions. None-the-less, we can draw several high-level conclusions, as well as comment on specific mechanisms. We will end by mentioning some related work.

Alewife was a huge project, driven by a small number of people: one professor, one research associate, and 8 graduate students formed the initial group. Others came later, but much was accomplished in the first few years. As a group, we tackled multiprocessor algorithms, compilers, operating and runtime systems, synchronization mechanisms, communication models, and processor and memory hardware architecture. We implemented a complete system with custom chips, operating systems, and compilers that *works* and is *usable*: Ricardo Bianchini visited us from the university of Rochester one summer and was able to learn about Alewife, then port most of the SPLASH [101] benchmarks during his stay.

In this sense, Alewife was tremendously more successful than ever anticipated. Ultimately, of course, there was a cost. Some members of the research group (this author among them) spent a long time in graduate school as a result. However, much knowledge and experience was gained in the balance. If any time-related lesson might be gained from the Alewife implementation, it would be:

---

*Extreme care must be taken with respins of chips for large academic projects. Such respins are almost never justified, unless to fix project-stopping bugs.*

---

In the case of Alewife, the A-1001 CMMU took about a year and a half of tuning and redesign, only to exhibit noise problems. Respins of chips can be subject to second-system syndrome (the desire to include all features that were excluded from the original system). Further, a respin can only exacerbate the fact that academic hardware systems projects tend to run behind the technology curve. For the author, this was a hard-won lesson.

## 7.1 High-Level Lessons of the Alewife Machine

Respins aside, we might ask ourselves what lessons could be taken away from the Alewife multiprocessor architecture and implementation? Many lessons might be forthcoming, but we would like to focus in this chapter on those that are relevant to this thesis.

**Efficient Integration:** In the opening pages of this thesis, we stated that our goal was to explore the integration of shared memory and message passing at a number of levels, from communications models, to runtime systems, to hardware. One of the primary conclusions that can be drawn from the Alewife design and implementation is that:

---

*The integration of shared-memory and message-passing communication models is possible at a reasonable cost, and can be done with a level of efficiency that does not compromise either model.*

---

This is illustrated by the architecture of Chapter 5 and the microbenchmarks of Section 6.1. In Alewife, there were three principal contributers to this conclusion: (1) careful construction of the communication models and interfaces (in particular for message passing), (2) use of featherweight threads, and (3) use of service coupling to manage hardware resources. As a result, the message-passing overheads are as low or lower than many similar interfaces in machines such as the CM-5, J-machine, iWarp, and others — especially for interrupt driven messaging. Further, the shared-memory implementation has a fast local fill time and a low remote-to-local access time; examination of the timing for DRAM access in Alewife reveals that shared-memory operations are essentially as efficient as possible — a more specific implementation would be hard-pressed to produce better use of the DRAM resource.

Before demonstrating *how* to integrate communication mechanisms efficiently, we had to decide exactly *what* it was that we were integrating. Among other things, this lead to the User-Direct Messaging model, one of the key enabling factors for fast efficient messaging. Thus, we could state another major conclusion as:

---

*User-Direct Messaging is an attractive message-passing communication model since it provides low overhead access to the network and enables an efficient, low-cost implementation.*

---

The ability to provide direct user-level access to the network hardware in the common case, while virtualizing this interface and, in particular, the *user-level atomicity mechanism* leads to an extremely efficient implementation. The primary reason for this was that only the common cases needed to be handled directly in hardware, leading to simpler interfaces and less complicated protection mechanisms (*i.e.* ones that could be easily implemented in a standard RISC pipeline).

**Two-Case Delivery:** References to two-case delivery have been sprinkled throughout these pages. In Chapter 4, we discussed the fact that two-case delivery frees both the user and the operating system from concerns about deadlock. Further, as discussed in that chapter, two-case delivery is the only way to guarantee that a user does not deadlock the network while still providing a direct network interface that does not drop messages. It also greatly simplifies the implementation of network interfaces in a multiuser environment (Section 2.4.7). Combining this with the fact that two-case delivery simplifies the design of cache coherence protocols and is a natural adjunct to software assisted coherence protocols (*e.g.* LimitLESS), we conclude that:

---

*Two-case delivery is a universal exception philosophy that greatly simplifies deadlock avoidance and protection mechanisms, leading to fast, "RISC-like" communication subsystems. Further, second-case delivery is not common case behavior for a wide variety of communication scenarios.*

---

In Section 6.3.2, we explored two-case delivery from the standpoint of the frequency of entry into deadlock. In the Alewife machine, invocation of two-case delivery was non-existent for shared-memory programs and virtually non-existent for message-passing programs. Further, our simulator explorations showed that there were a number of machine parameters for which deadlock was not a major concern. Note that the user-level atomicity mechanism is an essential component of two-case delivery as applied to user-level message passing. Finally, in an integrated architecture such as Alewife, two-case delivery provides the only method for dealing with problems associated with using a single logical network channel to carry both shared-memory and message-passing communication traffic (see below, however).

We should note that the original Alewife network overflow handler was an extremely complicated piece of software, prone to subtle bugs. Many of these complexities resulted from two key aspects, however: (1) the lack of "transparency" as described in Section 2.4.4 forced relaunching of all message traffic, and introduced complexity in tracking queue-level resources to make this work; and (2) bugs in the original CMMU caused problems with buffering. A two-case delivery system for hardware with transparent access would be much easier to construct.

**Interaction Between Models:** In Sections 2.5 and 4.3.4, we explored some of the boundary conditions in the interaction between shared memory and message passing. Time and time again, we observed that the use of a single network channel to carry both shared-memory and message-passing communication traffic greatly complicated the interaction between models during atomic sections. In fact, the non-virtualized, operating-system's view of the network prohibits use of shared memory when network interrupts are disabled. During the course of writing the Alewife operating system, this proved to be a difficult rule to follow: given a machine that integrates both shared-memory and message-passing communication styles, use of shared memory within handlers is extremely desirable. In Section 4.3.5 we discussed how the user-level atomicity mechanism can be combined with two-case delivery to provide a way for users to access shared memory during atomic sections. Unfortunately, this almost guarantees that the combination of polling and shared memory will result in continuous second-case delivery. Thus, one important conclusion that could be drawn from previous chapters is:

---

*Machines that integrate shared-memory and message-passing communication models should provide two logical network channels, one for each type of message traffic.*

---

Note that this conclusion proposes a different use of logical channels than the one often employed, *i.e.* to remove deadlock, since we employ two-case delivery for that. Instead, this conclusion proposes the use of two independent network channels as a way of freeing message-passing and shared-memory communication models to work in concert (as opposed to in opposition).

Further, two-case delivery is best employed in an environment that can support “virtual buffering” [78]. With virtual buffering, data that is buffered for second-case delivery is placed into the virtual memory of the receiving application, rather than into real memory. Virtual buffering is made effectively “infinite” by reserving the option to page buffered message traffic to disk[1]. Reserving the option to page without introducing deadlock requires a second logical network; thus, in our conclusion about using two logical networks, we would propose to carry message traffic on one network and both cache-coherence messages and paging traffic (where paging traffic is a specialized type of message traffic) on the other.

**Service Coupling:** The whole notion of service coupling, *i.e.* stream-based handling of scheduling and processing, appears time and time again within the Alewife CMMU. As described in Section 5.2.4, service coupling was an extremely useful technique for integrating access to highly contended resources such as the DRAM. As a result, it was well suited for the implementation of integrated message-passing and shared-memory communication styles. The advantages of service coupling arise from the separation of scheduling, execution, and routing via a stream-based methodology. This separation permits operations to be initiated only after all of their prerequisites have been satisfied and sufficient resources are present for operations to complete in finite time. Further, the stream-based routing of results frees execution units to focus on extracting maximum utilization of resources. Thus:

---

*Service coupling provides maximum utilization for highly contended resources. It also permits guarantees of operation atomicity and finite completion time.*

---

In fact, modern superscalar processors support a service-coupling-like separation of scheduling from execution to maximize the utilization of functional units. As mentioned in Section 7.3 below, the FLASH architecture paper [64] has a diagram that seems to reflect service coupling; this appears to be a case of convergent evolution.

[1] Excessive paging is prevented by the scheduler.

## 7.2 How Do the Lessons of Alewife Apply Today?

As discussed at the beginning of Chapter 6, the Alewife multiprocessor represents technology that was state of the art two or three processor generations ago. Thus, the obvious question arises: can any of the mechanisms and implementation techniques be applied to future architectures?

**Fast Message Interfaces:** In answering this question, we must ask ourselves which aspects of the Alewife implementation would not longer apply. Clearly, modern processors with off-chip, first-level caches are essentially unheard off (with the possible exception of PA-RISC processors from Hewlett Packard). This has two consequences: (1) it complicates the process of integrating external network interfaces; and (2) it makes "colored" loads and stores less useful. The important thing to note about both of these issues is that the Alewife implementation took advantage of the opportunities presented by the Sparcle interfaces. This was convenient, but by no means necessary to implement Alewife mechanism in a system that includes them from the outset.

In fact, let us explore the issue of fast network interfaces and their relationship to the processor. The fact that modern processors have on-chip caches has spurred a complete industry of research into network interface techniques whose sole aim is to correct for the fact that these processors are bad at performing uncached accesses to I/O devices[2]. This is a reactionary approach at best, since *communication is as fundamental as computation*. Basing a research program on correcting for deficiencies in current processors seems to be doomed to obscurity, since processors are changing very rapidly (evidence the current revolution in out-of-order and stream-based execution).

Consequently, as stressed at many points throughout this thesis, the primary view that we have taken is that communication interfaces are part of the ISA; thus, network interfaces should be integrated directly into processor pipelines. Our *modus operandi* was that we tackled fundamental problems (such as atomicity) in ways that were simple enough to be implemented directly within the processor. It is the contention of this author that the User-Direct Messaging interface (complete with the user-level atomicity mechanism and two-case delivery) has this property. To put this more succinctly:

> *Modern processors should include User-Direct Messaging interfaces (including user-level atomicity) in their instruction sets.*

Of course, there is a clear aspect of the messaging interface that belongs in the memory system (*i.e.* DMA) instead of in the processor. However, the output descriptor array and message input windows are aspects of the interface that could be easily integrated onto the processor, with output "packets" pipelined to the output pins for handling by the memory system; in fact, if these windows were made the same size as cache-lines, many mechanisms for moving data for the memory system could potentially be used for handling interfaces[3].

[2] The CNI interface [86] is one particularly clever example of correcting for this deficiency; in some sense, User-Direct Messaging degenerates to a CNI-style delivery during second-case delivery (*i.e.* buffered delivery).

[3] Note that providing a "storeback descriptor array" for describing DMA operations would make the `ipicst` instruction atomic (in the sense of `ipilaunch`), more directly mirroring the semantics of the `receive` operation.

Some would say that this type of integration has been proposed in the past and has failed to "catch on". However, the pressures for parallel execution have slowly built over time. Even companies such as Intel produce processors that directly support cache-coherence for two and four processor multiprocessors. Enterprise applications, with intensive database components, seem to be leading the drive for increased parallelism. Further, complex distributed systems, handling data from vast reaches of the globe through the Internet, have introduced needs for parallelism, communication, and complexity management that were unheard of just a few years ago. Visionaries such as Bill Dally who advocated the tight integration of processors and communication over 10 years ago were, perhaps, ahead of their time.

**Featherweight Threads and Fast Interrupts:** An important counterpart to the User-Direct Messaging model was the use of featherweight threads. This simple threading technique permitted interrupt handlers to run rapidly and with a complete set of registers. Rapid interrupt handling benefits many aspects of a system, not just message handlers: it permits the reasonable tradeoff between hardware and software, without encountering the "performance cliff" usually associated with software exception handlers.

Although the original use of register windows was to avoid saving and restoring registers across procedure boundaries, global compiler techniques (live register analysis, *etc.*) can be used to greatly lessen the cost of register maintenance across these boundaries. However, similar analysis does not really benefit fast interrupt handling, since the forwarding of live register information to interrupt handlers would be more expensive than simply saving and restoring all registers. Thus, the "caching" aspects of featherweight threading really requires at least two independent register sets.

What does this mean for modern architectures? The UltraSPARC [111] supports register windows of the same type as Sparcle; featherweight threading could be utilized directly on UltraSPARC. Further, a recent and intriguing development in the design of dynamic pipelines is the notion of simultaneous multithreading [109]. Simultaneous multithreading (SMT) was proposed as a technique for increasing the utilization of functional units in a superscalar pipeline; it is rumored to be present in the early stages of new processor designs (such as the Alpha). What is intriguing about SMT is that it provides a user view of the processor that includes multiple independent threads (complete with register sets). Use of featherweight threading techniques on SMT processors would lead to the possibility of very fast interrupts. It would also lead to an interesting tradeoff between functional-unit parallelism and fast interrupt handling. Thus:

---

*Featherweight threading, which has the potential to vastly reduce the overhead of fast atomic interrupts, is very much appropriate for modern processors.*

---

**Local Unshared-Memory Space:** One aspect of the Alewife architecture that we have mentioned in a number of places is the presence of a local unshared-memory space. This simple

This would simplifying the processor/memory interface a bit by removing the potential need for multiple `ipicst` instructions to be issued on a given message (see [77]).

concept is surprisingly important and not normally present in modern system architectures.

---

*A local, unshared-memory space is extremely useful because it enables software exception handers to make forward progress under all circumstances, regardless of the state of the system (and, in particular of the global, shared-memory space).*

---

Interestingly enough, many existing (modern) processors would have trouble supporting a local memory space in the sense of Alewife's, since they do not distinguish multiple classes of addresses (*e.g.* shared *vs.* unshared). This is a problem because most of these chips are highly integrated and include a cache controller for the external cache on chip: not surprisingly, such cache controllers do not demote cached accesses to uncached accesses in situations for which dirty lines cannot be retired to the memory system (see the discussion in Section 4.3.7). This, in turn, can prevent network overflow recovery through two-case delivery, since there would be situations in which the network handler could not run because its instructions need to displace dirty global memory lines[4]. It is the presence of a custom cache controller and transaction-buffer constraints in the CMMU that makes the Alewife local unshared-memory space generically useful for all types of exceptions (especially two-case delivery). Note that having this type of memory access is tremendously powerful and easily applied to modern microprocessors; the demotion of cached-accesses to uncached accesses simply needs to be supported by the on-chip cache controller.

**Transaction Buffer:** Finally, the transaction buffer was the single most useful structure in the Alewife CMMU: it helped to remove the window of vulnerability livelock; it helped to combine multiple outstanding requests to the same memory line; it provided a flexible flush queue to memory that enabled the implementation of the local, unshared-memory space; it was an integral part of DMA coherence; it provided a data staging area that efficiently supported the direct pipelining of data from the memory system or network to the cache, while still supporting the return of data destined for a context that was not actively executing. The list goes on. In fact, the ability to track outstanding memory transactions has proven useful in a number of systems outside of Alewife. Many modern processors have similar structures (often called load or store buffers) to permit the retiring of memory instructions from the execution pipeline before they have completed.

---

*Processor-side, transaction-buffer structures are incredibly useful because they maintain important information about the execution state of outstanding memory operations without blocking pipeline execution.*

---

It is interesting to note that the presence of transaction-buffer-like structures in modern processors can go a long way toward lessoning the problems associated with the window of vulnerability. By permitting the immediate committing of a requested load or store, we could prevent systematic thrashing in a way that was not available in Alewife's form of polling context switching.

[4]This could be solved by adding another level of caching (over and above the one supported by the on-chip cache controller), but that is probably undesirable from a memory latency standpoint.

## 7.3 Related Work

The communications mechanisms in Alewife build on the results of a number of researchers and have spurred an equal amount of research. In this section, we would like to mention and examine some of these other research projects.

### 7.3.1 Hardware Integration of Communication Models

Recent architectures demonstrate emerging agreement that it is important to integrate message-passing and shared-memory communication in the same hardware platform [3, 45, 93, 99, 40]. This current interest began, perhaps, with an Alewife paper on the advantages of integration [58]. However, other machines, such as the BB&N-Butterfly [17] supported integrated shared-memory and message-passing communication (via bulk DMA) long before this. What *is* new, however, is the implementation of extremely efficient *fine-grained* shared-memory and message-passing mechanisms together in a way that does not compromise either mechanism — allowing tradeoffs between communication models to reflect high-level properties of the models rather than of the implementation. This was seen in Section 6.2.3 which summarized some of the results of [28].

The Stanford FLASH multiprocessor [64] takes the "specialized coprocessor" approach to providing a plethora of mechanisms. As a later generation project, FLASH tackled a number of multiuser and reliability issues that were not explored by Alewife (although multiuser issues were addressed in FUGU [77, 78], a related project). By replacing the equivalent of the Alewife CMMU with a programmable memory controller, the FLASH team gained flexibility to explore a number of different shared-memory models and message-passing communication styles. There were two immediate consequences to this approach, however: First, by allowing "anything to be implemented", FLASH's focus was a bit diffuse, spanning a large space of possible protocols. In contrast, Alewife's focus was toward synthesizing essential mechanism. Among other things, the Alewife messaging interface is much simpler as a consequence of its implementation in hardware. Also, one essential difference between the FLASH and Alewife messaging interfaces is that FLASH implements global DMA coherence whereas Alewife implements local coherence.

The second consequence of providing a programmable memory controller is that it much harder to optimize use of critical resources such as DRAM. In FLASH, the inherent latency problems with programmable memory controllers are ameliorated to some extent by using long cache lines and overlapping the acquisition of data from DRAM with the software processing of coherence requests. None-the-less, this approach is unable to take advantage of direct pipelining of data from the DRAM into the processor cache, since it must wait for software decisions to be made before forwarding data upward in the memory hierarchy. This is another cost of flexibility: coherence operations take many more cycles than they would if hard-coded.

It is interesting to note that the scheduling architecture adopted by FLASH [64] bears resemblance to the service-coupling methodology of Section 5.2.4: requests are queued up at a hardware scheduler (called the *inbox*), then forwarded as a "stream" of requests directly to the *protocol processor*. Results are finally passed to the *outbox* for post distribution. In fact, Figure 4.2 of [64] has a topological similarity to our Figure 5-18 on page 180. Although it was not discussed outside of MIT, the Alewife microarchitecture was completed before this paper was published; hence, we can only conclude that this is a case of convergent evolution.

The Typhoon [93] and M-machine [40] architectures have approached the integration of message-passing and shared-memory communication by combining minimal "virtual-memory" style fine-grained data mapping with fast message interfaces to permit software-managed cache coherence. Typhoon includes a coprocessor for handling cache-coherence requests, while the M-machine reserves a hardware thread for special message handling. Unfortunately, the problem with both of these approaches is that they tend to compromise shared-memory performance, leading to a suppressed desire on the part of programmers to perform actual communication. Note that the protocol processor in the FLASH multiprocessor is highly tuned for coherence-style operations, whereas this is not true of either Typhoon or the M-machine. Although it remains to be seen, it is this author's feeling that exacerbating the memory bottleneck by placing software in latency-critical paths is undesirable from a performance standpoint.

The Cray T3D and T3E [99] integrates message passing and hardware support for a shared address space. Message passing in the T3E is flexible and includes extensive support for DMA. However, the T3E does not provide cache coherence.

### 7.3.2 Hardware Supported Shared-Memory

Alewife's combination of hardware for common case operations and software for "uncommon" sharing patterns is still unique to this author's knowledge. Other cache-coherent shared-memory machines provide either full hardware support for shared memory or implement coherence entirely in software.

DASH [71] is a cache-coherent multiprocessor that uses a full-mapped, directory-based cache coherence protocol implemented entirely in hardware. From the standpoint of timing, the DASH project was a sibling project to Alewife, although the DASH implementation methodology (FPGAs) produced working prototypes much quicker than Alewife. The DASH multiprocessor supports release consistency, instead of sequential consistency (and, in fact, was one of the driving forces behind the creation of the release-consistent model [41]). For latency tolerance (or amelioration), DASH includes prefetching and a mechanism for depositing data directly in another processor's cache. As discussed in Chapter 4, deadlock issues in DASH were handled through a combination of multiple virtual channels and negative acknowledgments. Recently, the SGI Origin [67] has appeared as a commercial offspring of DASH. This machine has taken fast, distributed, hardware cache-coherence to a new level of performance.

The KSR1 and DDM [43] provide a shared address space through cache-only memory. These machines also allow prefetching. Issues of deadlock are handled in specialized ways in both of these machines: the KSR1 takes advantage of its ring-based network topology, while the DDM exploits its hierarchy of buses. The Sequent NUMA-Q machine takes the FLASH approach of providing a programmable memory controller, although this controller is not as well tuned as the FLASH controller resulting in long remote access times.

A few architectures incorporate multiple contexts, pioneered by the HEP [102], switching on every instruction. These machines, including Monsoon [88] and Tera [6], do not have caches and rely on a large number of contexts to hide remote memory latency. In contrast, Alewife's block multithreading technique switches only on synchronization faults and cache misses to remote memory, permitting good single-thread performance and requiring less aggressive hardware multithreading support. A number of architectures, including HEP, Tera, Monsoon, and the J-machine, also provide support for fine-grain synchronization in the form of full/empty bits or tags.

### 7.3.3 Message-Passing Communication

The trend in message interfaces has been to reduce end-to-end overhead by providing user access to the interface hardware. The User-Direct Messaging model and interfaces build on previous work in messaging models and mechanisms.

**Messaging Models:** The User-Direct Messaging (UDM) model is similar to Active Messages [113] and related to Remote Queues (RQ) [16] as an efficient building-block for messaging within a protection domain. User-Direct Messaging differs from Active Messages [113] in that it includes explicit control over message delivery for efficiency; thus, for instance, users may choose to receive messages via polling or via interrupts.

The relationship between User-Direct Messaging and RQ is not surprising given that this author participated in both models. RQ provides a polling-based view of a network interface with support for system interrupts in critical situations. User-Direct Messaging, on the other hand, offers a more general view in which the application freely shifts between polling and user-interrupt modes. The RQ implementation on Alewife used a software version of user-controlled atomicity and the RQ paper outlined a hardware design in progress. The refined details of that hardware interface appeared in Section 2.4.2. In particular, the formalized notion of atomicity as well as the resulting user-level atomicity mechanism were only mentioned in the RQ paper, whereas they are integral parts of the UDM model. Like RQ, UDM depends on buffering to avoid deadlock rather than on explicit request and reply networks.

The Active Message Applications Programming Interface [81] is a complete programming system for messaging communication. This API could be efficiently implemented on top of User-Direct Messaging. UDM is thus lighter weight and more general. Interestingly enough, *virtual queueing*, as discussed in Section 4.3.6, is an explicit feature of this API; in this specification, virtual queues are referred to as *communication endpoints* and contain queuing resources for reception of both inter-domain and intra-domain messages

The Polling Watchdog [82] integrates polling and interrupts for performance improvement. The resulting programming model is interrupt-based in that application code may receive an interrupt at any point; the application cannot rely on atomicity implicit in a polling model. A polling watchdog uses a timeout timer on message handling to accelerate message handling if polling proves sluggish. The user-level atomicity mechanism for UDM includes an identical timer but uses it only to let the operating system clear the network. A polling watchdog mode could be implemented in Alewife, if so desired.

**Direct Network Interfaces:** Several machines have provided direct network interfaces. These include the CM-5, the J-machine, iWarp, the *T interface, Alewife, and Wisconsin's CNI [68, 35, 13, 89, 3, 86]. These interfaces feature low latency by allowing the processor direct access to the network queue. Direct NIs can be inefficient unless placed close to the processor. Anticipating continued system integration, we place our NI on the processor-cache bus or closer (see discussing above, in Section 7.2).

Both the J-machine [87] and the CM-5 export hardware message-passing interfaces directly to the user. These interfaces differ from the Alewife interface in several ways. First, in Alewife, messages are normally delivered via an interrupt and dispatched in software, while in the J-machine, messages are queued and dispatched in sequence by the hardware. On the CM-5, message delivery through interrupts is expensive enough that polling is normally used to access the network. Second, neither the J-machine, nor the CM-5 allow network messages to be transferred through DMA. Third, the J-machine does not provide an atomic message send like Alewife does; this omission complicates the sharing of a single network interface between user code and interrupt handlers.

The CNI work [86] shows how to partly compensate for a more distant NI by exploiting standard cache-coherence schemes. This assumes that all messages are copied directly into memory buffers by the network hardware, then extracted by the processor. Among other things, this uses memory bandwidth that would not be taken in first-case delivery with UDM and furthermore supports only a polling interface (and hence no notion of atomicity), rather than permitting the user to request interrupts on message arrival. To some extent (ignoring the issue of interrupts), the UDM network interface degenerates into a CNI-like interface during second-case delivery, although without the same efficiency; memory-based buffering is considered to be an exception situation in Alewife. Note, that in CNI, hardware places messages directly into buffers, making the process of virtualizing the network interface for multiple users much more difficult.

Direct interface designs have mostly ignored issues of multiprogramming and demand paging. In [78], we discuss the use of User-Direct Messaging in the context of a multiuser system (FUGU). The CM-5 provides restricted multiprogramming by strict gang scheduling and by context-switching the network with the processors. The *T NI [89] would have included GID checks and a timeout on message handling for protection as in FUGU. The M-machine [40] receives messages with a trusted handler that has the ability to quickly forward the message body to a user thread.

**Memory-Based Interfaces:** Memory-based interfaces in multicomputers [12, 18, 97, 99, 104] and workstations [36, 37, 108, 112] provide easy protection for multiprogramming if the NI also demultiplexes messages into per-process buffers. Automatic hardware buffering also deals well with sinking bursts of messages and provides the lowest overhead (by avoiding the processors) when messages are not handled immediately.

Memory-based application interfaces provide low overhead when access to the network hardware is relatively expensive (true for most current systems) and when latency is not an issue. Increased integration of computer systems and the mainstreaming of parallel processing challenges both of these assumptions: on-chip network interfaces can have low overhead and parallel programs frequently require coordinated scheduling for predictable, low latencies [9]. Alewife provides low latency for applications where latency matters while including low-cost and reasonably efficient buffering as a fallback mode.

### 7.3.4 Two-Case Delivery and Deadlock

The notion of two-case delivery has appeared in various guises over the years. Perhaps the clearest example of this was with the J-machine [87], which provided overflow interrupts on *both* the input and output queues. The resulting system was studied extensively by Rich Lethin in [73]. Interestingly enough, the fact that the overflow interrupts were generated *immediately* after a corresponding queue became full was a source of performance problems in this machine. The Alewife deadlock detection algorithm adds a crucial hysteresis as we saw in Section 6.3, where we explored the frequency of overflow detection as a function of the congestion timeout.

One of the results noted by Lethin is that a shared-memory style of communication (request/reply) tends to lead to overall shorter queue sizes (given infinite queues) then so-called "Snakes" (message-passing styles that hop from one node to another in a random pattern). Certainly, we saw in Section 6.3 that shared-memory programs are less likely to invoke second-case delivery than message-passing programs; in that sense, these results agree. However, the results of [73] do not take into account hysteresis, instead characterizing the frequency with which queues are full. We believe that this is often the wrong question (at least for machines that support better detection heuristics than "queue full"), since temporary blockages are a natural aspect of wormhole-routed networks.

In exploring the frequency of *actual* deadlock in Section 6.3, we were asking the question: what is the minimum frequency with which a second-case delivery mechanism must be invoked. This gives a lower-bound on the amount of buffering that must be performed to avoid deadlocking the machine. Although we took a stab at this, much more work is required.

Greater exploration of User-Direct Messaging and two-case delivery in the context of a multiuser FUGU system is explored in [78, 79]. There, the use of loose gang scheduling is shown to prevent the arbitrary buildup of buffered messages as a consequence of scheduler mismatches with multiple simultaneous processes.

### 7.3.5 Tradeoffs Between Communication Models

Last, but not least, a number of researchers have explored the effects of integrated shared-memory and message-passing communication models on performance. As mentioned in Section 6.2.3, it is only when the these mechanisms have roughly equivalent performance that any tradeoff is possible. In Alewife, the communication mechanisms are close enough one another in performance that fundamental properties of these communication mechanisms come into play, rather than implementation artifacts. This is indicated by graphs such as Figure 6-4 in which the performance of applications based on message passing and and those based on shared memory cross each other as a function of network bandwidth.

Very few (if any) existing machines provide the level of parity between communication mechanisms that Alewife provides. Thus, studies of the inherent cost of different communication models on physical hardware are scarce. None-the-less, the Alewife experiments in Section 6.2.3 (and in [28]) were strongly influenced by studies from Wisconsin, Stanford, and Maryland. Our comparison of communication mechanisms is similar to Chandra, Larus and Rogers [96], although we have available a larger set of mechanisms and we generalize to a range of system parameters. This generalization is similar to the study of latency, occupancy, and bandwidth by Holt et.

al [48], which focuses exclusively upon shared-memory mechanisms. Although the Alewife machine provides an excellent starting point for the comparison of a large number of communication mechanisms, our results are greatly enhanced by our use of emulation, an approach inspired by the work at Wisconsin [107].

Chandra, Larus and Rogers compare four applications on a simulation of a message-passing machine similar to a CM-5 multiprocessor against a simulation of a hypothetical machine also similar to a CM-5, but extended by shared-memory hardware [96]. Their results are a good point of comparison for our emulation results, since both Alewife and the CM-5 are SPARC-based architectures with very similar parameters. They found that message passing performed approximately a factor of two better than shared memory while simulating a network latency of 100 cycles.

Our results agree qualitatively with studies from Stanford. Holt et al. found latency to be critical to shared-memory performance, as we did. They also found that node-to-network bandwidth was not critical in modern multiprocessors. Our study shows, however, that bandwidth across the *bisection* of the machine may become a critical cost in supporting shared memory on modern machines. Such costs will make message passing and specialized user-level protocols [39] increasingly important as processor speeds increase.

Woo et al. [117] compared bulk transfer with shared memory on simulations of the FLASH multiprocessor [64] running the SPLASH [101] suite. They found bulk transfer performance to be disappointing due to the high cost of initiating transfer and the difficulty in finding computation to overlap with the transfer. Although, Alewife's DMA mechanism is cheaper to initiate than theirs, we also found bulk transfer to have performance problems. Our problems arose from the irregularity of our application suite, which caused high scatter/gather copying costs and limited data transfer size. It should be noted, however, that Alewife DMA mechanisms were found to greatly enhance applications (in particular the operating system) with less irregular communication patterns or with large blocks of data.

Klaiber and Levy [56] study the performance of programs which accesses shared-memory or message-passing runtime libraries. These libraries generated traces for shared-memory and message-passing simulators, to generate statistics on message traffic. However, their programs were not fined tuned for any particular architecture, and hence not fair to either. Our programs are highly optimized for the mechanisms and performance parameters of a machine supporting both communication methods. Further, their machine independent libraries tend to introduce unnecessary overheads for both methods, leading to additional loss of comparison accuracy. Finally they report only message traffic, not execution time numbers. They do not show how message traffic impacts runtime.

# Appendix A

# Active Message Scheduling

In this appendix, we present a fragment of the interrupt handler code for a user-level active message. The fragment that we present here (in Figure A-1) is the so-called "fast path", *i.e.* the code that runs if no exceptional cases are taken. A breakdown of the costs for this code were presented in Table 6-4 (page 201), where this was referred to as the "hard" scheduler.

This code exhibits two important properties: (1) it provides featherweight threading of contexts on Sparcle; and (2) it makes use of the hardware features provided by the user-level atomicity mechanism. Our goal in the next few paragraphs is to point out aspects of the code that support these two properties. The scheduling philosophy that is supported by this code is simplistic, but powerful enough to perform many interesting tasks. There are two "levels" of operation. Background tasks are considered "level 0" and are at lower priority than user-level active message handlers which are at "level 1". The implicit assumption here is that there is some unspecified scheduler that handles level 0 tasks. Our goals are to (1) disable access to level 0 tasks when we have one or more operating level 1 tasks; and (2) perform featherweight scheduling of level 1 tasks. Note that we assume that these level 1 tasks have no guaranteed priorities amongst themselves other than that they will operate at higher priority than background.

To understand this code, a brief word about registers in the Sparcle processor is appropriate. The Sparcle processor has eight overlapping register windows which Alewife uses as four, non-overlapping contexts. Thus, windows are used in pairs. The two windows associated with a context are called the *user frame* and *trap frame* respectively. Because of the overlapping nature of windows, these two contexts share 8 registers. By convention, these registers are associated with the user frame. The end result of this is that each context has 24 "user" registers and 8 "trap" registers. To help in understanding the code of Figure A-1, registers have been given symbolic names with well-defined prefixes. Thus, user registers start with "`uf`", registers that are exclusively in the trap frame start with "`tf`", registers that are trap frame names for user registers start with "`th`".

In addition to context-specific registers, there are 8 global registers that are not associated with any particular context. These three registers, called `alloc_wim`, `background_bits`, and `background_psr`, are the key to fast thread scheduling. These registers are considered to be "supervisor-only", even though Sparcle does not support protected registers[1]. The `alloc_wim` register contains information about which of the four Sparcle contexts are currently in use; the

[1]There is precedent for persistent supervisor registers, however. See the PowerPC [91] and the MC88100 [83].

```
ACT_MESS_ENTRY()   ; This is the entry point for the active message interrupt.
        rdpsr       %tf_old_psr                          ; Save PSR of interrupted context
        andcc       0b11111111, %alloc_wim               ; Is there a free context?
        bne,a       checklevel                           ; Yes, continue
        wrwim       0b11111111, %alloc_wim               ; Going to non-allocated context

    ⇒ Here we would have exception code to free up a context by unloading a thread.

    checklevel:
        xnorcc      %alloc_wim, %background_wim, %g0     ; Check– At background level?
        beq,a       newcontext                           ; Yes, save return CTX
        move        %tf_old_psr, %background_psr
    newcontext:
        nextf       %tf_old_psr, %g0, %th_old_psr        ; Goto new context
        andn        %alloc_wim, %tf_wim_bits, %alloc_wim ; Allocate context
        ldio        %desc_length, %g0                    ; Check desc_length (F/E) and pri-invert (B/I)
        restore     %g0, %g0, %g0                        ; Goto user frame.
        cbeandi,a   dispatch                             ; Empty & Idle => go for it
        setatom     (ATOM|TIMER|DISPOSE)                 ; Set atomicity mode (no extend)
        cbempty,a   dispatch                             ; Empty => priority-invert only
        setatom     (ATOM|TIMER|DISPOSE|EXTEND)          ; Set atomicity mode (with extend)

    ⇒ Here we would have exception code to save the output_descriptor queue.

    dispatch:
        ldio        %input_window+1, %l0                 ; Get address of message handler
        wrwim       %background_bits, %alloc_wim         ; Enable level-1 tasks (handlers)
        jmpl        %l0, %g0                             ; Go call handler.
        wrpsr       %uf_scratch, SUPER_BIT               ; Flip into user mode.
    user_return:                                         ; User code returns here!
        disatom     TIMER                                ; Trap if extend or message not freed
        ta          SYS_MESS_RETURN                      ; Syscall into kernel (no return).
-----------------------------------------------------------------------------------------------
SYS_MESS_RETURN()   ; System-call entry point taken on return from handler.
        setatom     0                                    ; Clear atomicity mechanism
        or          %alloc_wim, %tf_wim_bits, %alloc_wim ; Deallocate Context
        xnorcc      %alloc_wim, %background_bits, %g0    ; Check – out of level-1 tasks?
        beq         exit_to_level_0                      ; Yes, return to level 0
        rdpsr       %th_scratch                          ; Restore PSR for next time around.
    next_level_1:
        bpos        exit_to_level_1                      ; No unloaded level-1 tasks
        wrwim       %alloc_wim, %background_bits         ; Remove context from wim
        subcc       %alloc_wim, %tf_wim_bits, %g0        ; Is this the only free context?
        beq         exit_to_level_1A:                    ; Yes. Do not reload now.
        subcc       %tf_tpc, %tf_check_return, %g0       ; Check – proper return?

    ⇒ Here we would have exception code to reload some level-1 thread that was previously unloaded.

    exit_to_level_1:
        subcc       %tf_tpc, %tf_check_return, %g0       ; Check – proper return?
    exit_to_level_1A:
        bne         weird_return                         ; Error – User didn't return normally!
        disint      CSWITCH                              ; Disable cswitching (for next time)
        prev        %g0, %g0, %g0                        ; Go to another level-1 task
        wrpsr       %tf_old_psr                          ; Restore PSR of that task
        jmpl        %tf_tpc, %g0                         ; Perform return from
        rett        %tf_tnpc, %g0                        ;    trap sequence.
    exit_to_level_0:
        subcc       %tf_tpc, %tf_check_return, %g0       ; Check – proper return?
        wrpsr       %background_psr                      ; Restore PSR/return to background task
        wrwim       %alloc_wim, %g0                      ; Background wim.
        bne         weird_return                         ; Error – User didn't return normally!
        disint      CSWITCH                              ; Disable cswitching (for next time)
        jmpl        %tf_tpc, %g0                         ; Perform return from
        rett        %tf_tnpc, %g0                        ;    trap sequence.
```

Figure A-1: Scheduler code for user-level active message interrupts on the A-1001 CMMU. This code makes use of both featherweight threads and the hardware user-level atomicity mechanism.

lowest eight bits of this register are in the same format as the `WIM` register, with adjacent pairs of bits representing the windows of a context. A context is currently in use if its adjacent bits are zeros and free if they are ones. The `background_bits` register points at contexts that are currently in use for background (level 0) tasks. Finally, the `background_psr` register contains a copy of the processor status register `PSR` for the background task that was interrupted to begin execution of the first level 1 task. This value contains, among other things, the window pointer value for the interrupted context; hence, restoring this value (with a `wrpsr` instruction) will return to the original context.

This code contains two key entry points: `ACT_MESS_ENTRY()` and `SYS_MESS_RETURN()`. Each of these begins execution in the trap frame of the user context that was executing at the time that execution began. The first of these is entered by the active message interrupt, and the second is entered by a special "Trap" instruction, shown at the end of `ACT_MESS_ENTRY()`, for returning to system level. Some understanding of SPARC instruction set is appropriate for understanding this code (see [105]); however, note that all control-flow instructions (branches, jumps, etc) have one delay slot. The high-level code flow during the execution of an active message handler is that we begin execution at `ACT_MESS_ENTRY()`. The `jmpl` instruction at line 20 calls the user handler code. When the user returns from the handler, we begin with execution of line 22. Then at line 23, we invoke the software trap instruction that causes us to return to system level and begin execution at line 24. Finally, the handler exits with lines 40/41 or lines 47/48.

## A.1 Featherweight Threading

Given the above description, we have two distinct sets of code that perform featherweight threading. First we perform allocation: at lines 2–4, we check to see if there are any free contexts and prepare the wim so that a `nextf` instruction will take us to a free context (by inverting the `alloc_wim` register, we enable only contexts that are free). Line 5 represents an unspecified amount of exception code that is responsible for freeing up a context if we cannot find one. Note that, if we are changing from level 0 to level 1, the code at lines 6–8 saves the `background_psr` for later use. Note that we assume that the thread descriptors and stack frames are already preallocated and loaded into appropriate registers. This is the reason that our dispatch code (lines 18–21) does not bother setting up user state other than loading the handler address from the message.

Second we perform deallocation: lines 25–32 perform a series of checks. If we are returning to level 0, we branch off and restore the proper `WIM`. Alternatively, if we are not returning to level 0, then we may have to do one of two things. If we have unloaded level 1 tasks (because we previously ran out of contexts), then we must decide whether to reload or not[2]. The featherweight heuristic says that we want to reload such unloaded tasks *unless* we have just freed up the only free context. The reason for this is that we get the desired featherweight behavior in that a set of short handlers that do not exit their atomic sections will execute very rapidly (except for the first, which might have to unload a context).

[2] This is indicated by the unload code by setting the sign bit of `background_bits`, triggering a negative condition in the comparison of line 26 if there are unloaded tasks.

## A.2 User-Level Atomicity Mechanism

The code of Figure A-1 employs and enables use of the hardware user-level atomicity in several ways. First, since every active message is scheduled as a complete thread, the transition from handler atomic section to full thread can be accomplished by a message handler (if desired) with a single-cycle `disatom` instruction. Further, the atomicity hardware is responsible for starting and stopping the timer and tracking whether or not the user remembers to free the input message; hence this dispatch code is not responsible for these functions.

Instead, the dispatch code is responsible for initializing the mechanism properly. We see this in lines 11–16 (line 12 is there as a delay slot of the `ldio`). The `ldio` instruction at line 11 sets the *full/empty* and *busy/idle* coprocessor condition codes to indicate two pieces of information: *full* means that the output_descriptor array contains a partial message description that must be saved and *busy* indicates that a priority inversion has been detected. The fast path, if neither of these are true, is checked at line 13 (this is a coprocessor branch on empty and idle) instruction; in this case, we initialize the atomicity mechanism by setting three of the atomicity control bits: `atomicity_assert`, `timer_force`, and `dispose_pending`. The slightly slower path, taken if only a priority inversion is present, sets the `atomicity_extend` bit as well. We have omitted the more complicated unloading of the output_descriptor array (represented by exception code at line 17).

On return from the user handler, line 22 is responsible for triggering exception conditions if they still exist, i.e. causing an `atomicity_extend` or `dispose_failure` trap. Note that we do this by attempting to shut off the `timer_force` bit (which we know is on), not the atomicity bit (hence avoiding the danger of a race condition if the user did not decide to exit their atomic sections). Lines 33, 35, or 42 perform a check against a reserved trap frame register (`tf_check_return`) to make sure that this was the return path taken by the user. Finally, line 24 disables the atomicity mechanism entirely (for exit).

Some of the odd ordering of instructions seen here is to fill delay slots for various instructions (*e.g.* for `wrpsr` and `wrwim` with respect to things they affect).

# Bibliography


[1] Sarita V. Adve and Mark D. Hill. Weak Ordering - A New Definition. In *Proceedings 17th Annual International Symposium on Computer Architecture*, pages 2–14, New York, June 1990.

[2] Sarita V. Adve, Mark D. Hill, Barton P. Miller, and Robert H.B. Netzer. Detecting Data Races on Weak Memory Systems. In *Proceedings of the 18th Anual Symposium on Computer Architecture*, May 1991.

[3] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie, and Donald Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 2–13, June 1995.

[4] Anant Agarwal, John Kubiatowicz, David Kranz, Beng-Hong Lim, Donald Yeung, Godfrey D'Souza, and Mike Parkin. Sparcle: An Evolutionary Processor Design for Multiprocessors. *IEEE Micro*, 13(3):48–61, June 1993.

[5] Anant Agarwal, Beng-Hong Lim, David A. Kranz, and John Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *Proceedings 17th Annual International Symposium on Computer Architecture*, pages 104–114, Seattle, WA, June 1990.

[6] Gail Alverson, Robert Alverson, and David Callahan. Exploiting Heterogeneous Parallelism on a Multithreaded Multiprocessor. In *Workshop on Multithreaded Computers, Proceedings of Supercomputing '91*. ACM Sigraph & IEEE, November 1991.

[7] Jennifer M. Anderson and Monica S. Lam. Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. In *Proceedings of SIGPLAN '93 Conference on Programming Languages Design and Implementation*. ACM, June 1993.

[8] Kazuhiro Aoyama and Andrew A. Chien. The Cost of Adaptivity and Virtual Lanes in a Wormhole Router. *Journal of VLSI Design*, 2(4):315–333, 1993.

[9] Remzi H. Arpaci, David E. Culler, Arvind Krishnamurthy, Steve G. Steinberg, and Katherine Yelick. Empirical Evaluation of the CRAY-T3D: A Compiler Perspective. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 320–331, June 1995.

[10] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Adaptive Software Cache Management for Distributed Shared Memory Architectures. In *Proceedings 17th Annual International Symposium on Computer Architecture*, New York, June 1990. IEEE.

[11] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Publishing Company, Reading, MA, 1987.

[12] Matthias A. Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, Edward W. Felten, and Jonathan Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings 21st Annual International Symposium on Computer Architecture (ISCA'94)*, pages 142–153, April 1994.

[13] S. Borkar, R. Cohn, G. Cox, T. Gross, H.T. Kung, M. Lam, M. Levine, B. Moore, W. Moore, C. Peterson, J. Susman, J. Sutton, J. Urbanski, and J. Webb. Supporting Systolic and Memory Communication in iWarp. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 70–81, June 1990.

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

[15] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. PROTEUS: A high-performance parallel-architecture simulator. Technical Report MIT/LCS/TR-516, Massachusetts Institute of Technology, September 1991.

[16] Eric Brewer, Fred Chong, Lok Liu, Shamik Sharma, and John Kubiatowicz. Remote Queues: Exposing Message Queues for Optimization and Atomicity. In *Proceedings of the Symposium on Parallel Algorithms and Architectures (SPAA'95)*. ACM, 1995.

[17] *Inside the Butterfly Plus*. BB&N Advanced Computers Inc., 1987.

[18] Greg Buzzard, David Jacobson, Milon Mackey, Scott Marovich, and John Wilkes. An Implementation of the Hamlyn Sender-Managed Interface Architecture. In *Proceedings of the Second Symposium on Operating System Design and Implementation*, pages 245–259, 1996.

[19] David Callahan and Ken Kennedy. Compiling Programs for Distributed-Memory Multiprocessors. *Journal of Supercomputing*, 2(151-169), October 1988.

[20] David Callahan, Ken Kennedy, and Allan Porterfield. Software Prefetching. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 40–52. ACM, April 1991.

[21] David Chaiken. Smart Memory Systems. *CMG Transactions*, pages 23–32, Winter 1993.

[22] David Chaiken, Craig Fields, Kiyoshi Kurihara, and Anant Agarwal. Directory-Based Cache-Coherence in Large-Scale Multiprocessors. *IEEE Computer*, 23(6):41–58, June 1990.

[23] 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.

[24] David L. Chaiken. *Mechanisms and Interfaces for Software-Extended Coherent Shared Memory*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 1994. MIT/LCS TR-644.

[25] David Lars Chaiken. Cache Coherence Protocols for Large-Scale Multiprocessors. Technical Report MIT-LCS-TM-489, Massachusetts Institute of Technology, September 1990.

[26] Andrew A. Chien. A Cost and Speed Model for k-ary n-cube Wormhole Routers. In *Hot Interconnects*, 1993.

[27] Fredric T. Chong. *Parallel Communication Mechanisms for Sparse, Irregular Applications*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, December 1996.

[28] Fredric T. Chong, Rajeev Barua, Fredrik Dahlgren, John Kubiatowicz, and Anant Agarwal. The Sensitivity of Communicaton Mechanisms to Bandwidth and Latency. In *Proceedings of the Fourth Annual Symposium on High-Performance Computer Architecture*, February 1998.

[29] Douglas W. Clark. Large-Scale Hardware Simulation: Modeling and Verification Strategies. In *Proceedings of the 25th Anniversary Symposium, Carnegie Mellon University*, Pittsburgh, PA, September 1990. Carnegie Mellon University.

[30] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[31] William J. Dally. Virtual-channel Flow Control. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):194–205, March 1992.

[32] William J. Dally and Hiromichi Aoki. Deadlock-Free Adaptive Routing in Multicomputer Networks Using Virtual Channels. *IEEE Transactions on Parallel and Distributed Systems*, 4(4):466–475, April 1993.

[33] William J. Dally and Charles L. Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Transactions on Computing*, C-36(5):547–553, May 1987.

[34] W. J. Dally et al. Architecture of a Message-Driven Processor. In *Proceedings of the 14th Annual Symposium on Computer Architecture*, pages 189–196, Washington, D.C., June 1987. IEEE.

[35] 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.

[36] Chris Dalton, Greg Watson, David Banks, Costas Calamvokis, Aled Edwards, and John Lumley. Afterburner. *IEEE Network*, pages 36–43, July 1993.

[37] Peter Druschel, Larry L. Peterson, and Bruce S. Davie. Experiences with a High-Speed Network Adaptor: A Software Perspective. In *Proceedings of the Conference on Communication Architectures, Protocols and Applications*, pages 2–13, 1994.

[38] Michel Dubois, Christoph Scheurich, and Faye A. Briggs. Synchronization, coherence, and event ordering in multiprocessors. *IEEE Computer*, 21(2):9–21, February 1988.

[39] Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, Ioannis Schoinas, Mark D. Hill James R. Larus, Anne Rogers, and David A. Wood. Application-Specific Protocols for User-Level Shared Memory. In *Supercomputing 94*, 1994.

[40] Marco Fillo, Stephen W. Keckler, W.J. Dally, Nicholas P. Carter, Andrew Chang, Yevgeny Gurevich, and Whay S. Lee. The M-Machine Multicomputer. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 146–156, Ann Arbor, MI, November 1995. IEEE Computer Society.

[41] 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.

[42] Kourosh Gharachorloo, Sarita V. Adve, Anoop Gupta, John L. Hennessey, and Mark D. Hill. Programming for Different Memory Consistency Models. *Journal of Parallel and Distributed Computing*, pages 399–407, August 1992.

[43] Erik Hagersten, Anders Landin, and Seif Haridi. DDM — A Cache-Only Memory Architecture. *IEEE Computer*, 25(9):44–54, September 1992.

[44] R.H. Halstead and T. Fujita. MASA: A Multithreaded Processor Architecture for Parallel Symbolic Computing. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 443–451, New York, June 1988. IEEE.

[45] John Heinlein, Kourosh Gharachorloo, Scott Dresser, and Anoop Gupta. Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 38–50. ACM, October 1994.

[46] 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.

[47] 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.

[48] C. Holt, M. Heinrich, J. P. Singh, E. Rothberg, and J. Hennessy. The effects of latency, occupancy and bandwidth on the performance of cache-coherent multprocessors. Technical report, Stanford University, Stanford, California, January 1995.

[49] David V. James, Anthony T. Laundrie, Stein Gjessing, and Gurindar S. Sohi. Distributed-Directory Scheme: Scalable Coherent Interface. *IEEE Computer*, pages 74–77, June 1990.

[50] Kirk Johnson. The impact of communication locality on large-scale multiprocessor performance. In *19th International Symposium on Computer Architecture*, pages 392–402, May 1992.

[51] Kirk L. Johnson. *High-Performance All-Software Distributed Shared Memory*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, December 1995.

[52] Kirk L. Johnson, M. Frans Kaashoek, and Deborah A. Wallach. CRL: High-performance all-software distributed shared memory. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[53] N.P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings, International Symposium on Computer Architecture '90*, pages 364–373, June 1990.

[54] Stafanos Kaxiras. Kiloprocessor Extensions to SCI. In *Proceedings of the 10th International Parallel Processing Symposium*, April 1996.

[55] Stefanos Kaxiras and James R. Goodman. The GLW Cache Coherence Protocol Extensions for Widely Shared Data. In *Proceedings of the 10th ACM International Conference on Supercomputing*, May 1996.

[56] A. Klaiber and H. Levy. A Comparison of Message Passing and Shared Memory for Data-Parallel Programs. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, April 1994.

[57] Kathleen Knobe, Joan Lukas, and Guy Steele Jr. Data Optimization: Allocation of Arrays to Reduce Communication on SIMD Machines. *Journal of Parallel and Distributed Computing*, 8(2):102–118, August 1990.

[58] David Kranz, Kirk Johnson, Anant Agarwal, John Kubiatowicz, and Beng-Hong Lim. Integrating Message-Passing and Shared-Memory; Early Experience. In *Principles and Practice of Parallel Programming (PPoPP) 1993*, pages 54–63, San Diego, CA, May 1993. ACM. Also as MIT/LCS TM-478, January 1993.

[59] David Kroft. Lockup-Free Instruction Fetch/Prefetch Cache Organization. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, pages 81–87, June 1981.

[60] John Kubiatowicz and Anant Agarwal. Anatomy of a Message in the Alewife Multiprocessor. In *Proceedings of the International Supercomputing Conference (ISC) 1993*, Tokyo, Japan, July 1993. IEEE. Also as MIT/LCS TM-498, December 1992.

[61] John Kubiatowicz, David Chaiken, Anant Agarwal, Arthur Altman, Jonathan Babb, David Kranz, Beng-Hong Lim, Ken Mackenzie, John Piscitello, and Donald Yeung. The Alewife CMMU: Addressing the Multiprocessor Communications Gap. In *HOTCHIPS*, August 1994.

[62] John D. Kubiatowicz. Closing the Window of Vulnerability in Multiphase Memory Transactions: The Alewife Transaction Store. Technical Report TR-594, MIT, November 1993.

[63] Kiyoshi Kurihara, David Chaiken, and Anant Agarwal. Latency Tolerance through Multithreading in Large-Scale Multiprocessors. In *Proceedings International Symposium on Shared Memory Multiprocessing*, Japan, April 1991. IPS Press.

[64] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA) 1994*, Chicago, IL, April 1994. IEEE.

[65] *l64811 SPARC Integer Unit Technical Manual*. LSI Logic Corporation, Milpitas, CA 95035, 1989.

[66] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9), September 1979.

[67] James Laudon and Daniel Lenoski. The SG Origin: A ccNUMA Highly Scalable Server. In *Procedings of the 24th Annual International Symposium on Computer Architecture*, June 1997.

[68] Charles E. Leiserson, Aahil S. Abuhamdeh, and David C. Douglas et al. The Network Architecture of the Connection Machine CM-5. In *The Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*. ACM, 1992.

[69] 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.

[70] 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.

[71] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The dash prototype: Logic overhead and performance. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):41–61, Jan 1993.

[72] Daniel E. Lenoski and Wolf-Dietrich Weber. *Scalable Shared-Memory Multiprocessing*. Morgan Kaufmann Publishers, 1995.

[73] Richard Anton Lethin. *Message-Driven Dynamics*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, March 1997.

[74] J. Li and M. Chen. Compiling communication-efficient programs for massively parallel machines. *IEEE Transactions on Parallel and Distributed Systems*, 2:361–376, July 1991.

[75] Beng-Hong Lim. *Reactive Synchronization Algorithms for Multiprocessors*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 1995. MIT/LCS TR-664.

[76] Beng-Hong Lim and Ricardo Bianchini. Limits on the Performance Benefits of Multithreading and Prefetching. In *Proceedings of the International Conference on the Measurement and Modeling of Computer Systems*, May 1996.

[77] Ken Mackenzie, John Kubiatowicz, Anant Agarwal, and M. Frans Kaashoek. FUGU: Implementing Protection and Virtual Memory in a Multiuser, Multimodel Multiprocessor. Technical Memo MIT/LCS/TM-503, October 1994.

[78] Kenneth Mackenzie, John Kubiatowicz, Matthew Frank, Walter Lee, Victor Lee, Anant Agarwal, and M. Frans Kaashoek. Exploiting Two-Case Delivery for Fast Protected Messaging. In *Proceedings of the Fourth Annual Symposium on High-Performance Computer Architecture*, February 1998.

[79] Kenneth M. Mackenzie. *The* FUGU *Scalable Workstation: Architecture and Performance*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, February 1998.

[80] N. K. Madsen. Divergence preserving discrete surface integral methods for Maxwell's curl equations using non-orthogonal unstructured grids. Technical Report 92.04, RIACS, February 1992.

[81] A. Mainwaring and D. Culler. Active Messages: Organization and Applications Programming Interface (API V2.0). University of California at Berkeley, Network of Workstations Project White Paper, September 1995.

[82] Olivier Maquelin, Guang R. Gao, Herbert H. J. Hum, Kevin Theobald, and Xin-Min Tian. Polling Watchdog: Combining Polling and Interrupts for Efficient Message Handling. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 179–188, May 1996.

[83] *MC88100 RISC Microprocessor User's Manual, Second Edition*. Motorola, 1990.

[84] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. COMP TR90-114, Rice University, Houston, Texas, May 1990.

[85] Todd Mowry and Anoop Gupta. Tolerating Latency Through Software-Controlled Prefetching in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87–106, June 1991.

[86] Shubhendu S. Mukherjee, Babak Falsafi, Mark D. Hill, and David A. Wood. Coherent Network Interfaces for Fine-Grain Communication. In *Proceedings of the 23rd International Symposium on Computer Architecture*, pages 247–258, May 1996.

[87] M.D. Noakes, D.A.Wallach, and W.J. Dally. The J-Machine Multicomputer: An Architectural Evaluation. In *In Proceedings of the 20th Annual International Symposium on Computer Architecture 1993*, pages 224–235, San Diego, CA, May 1993. ACM.

[88] G. M. Papadopoulos and D.E. Culler. Monsoon: An Explicit Token-Store Architecture. In *Proceedings 17th Annual International Symposium on Computer Architecture*, pages 82–91, New York, June 1990. IEEE.

[89] Gregory M. Papadopoulos, G. Andy Boughton, Robert Greiner, and Michael J. Beckerle. *T: Integrated Building Blocks for Parallel Computing. In *Supercomputing '93*, pages 624–635. IEEE, November 1993.

[90] Timothy M. Pinkston and Sugath Warnakulasuriya. On Deadlocks in Interconnection Networks. In *Procedings of the 24th Annual International Symposium on Computer Architecture*, June 1997.

[91] *PowerPC Microprocessor Family: The Programming Environments*. IBM Microelectronics and Motorola, 1994.

[92] *R10000 Microprocessor User's Manual, Ver 2.0*. MIPS Technologies/Silicon Graphics, 1996.

[93] Steve K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA) 1994*, Chicago, IL, April 1994. IEEE.

[94] Anne Rogers and Keshav Pingali. Process Decomposition through Locality of Reference. In *SIGPLAN '89, Conference on Programming Language Design and Implementation*, June 1989.

[95] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. Complete Computer Simulation: The SimOS Approach. In *IEEE Parallel and Distributed Technology*, Fall 1995.

[96] Satish Chandra and James Lars and Anne Rogers. Where is Time Spent in Message-Passing and Shared-Memory Programs. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 61–73, San Jose, California, 1994.

[97] Klaus E. Schauser and Chris J. Scheiman. Experience with Active Messages on the Meiko CS-2. In *Proceedings of the 9th International Symposium on Parallel Processing*, 1995.

[98] *Scalable Coherent Interface. IEEE P1596 SCI standard.*, 1989.

[99] Steven L. Scott. Synchronization and Communication in the T3E Multiprocessor. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 26–36, October 1996.

[100] 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.

[101] J.P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical Report CSL-TR-92-526, Stanford University, June 1992.

[102] B.J. Smith. Architecture and Applications of the HEP Multiprocessor Computer System. *Society of Photo-optical Instrumentation Engineers*, 298:241–248, 1981.

[103] J. E. Smith, G. E. Dermer, B. D. Vanderwarn, S. D. Klinger, C. M. Rozewski, D. L. Fowler, K. R. Scidmore, and J. P. Laudon. The ZS-1 Central Processor. In *Proceedings of the 2nd Annual Conference on Architectural Support for Programming Languages and Operatings Systems*, October 1987.

[104] Marc Snir and Peter Hochschild. The Communication Software and Parallel Environment of the IBM SP-2. Technical Report IBM-RC-19812, IBM, IBM Research Center, Yorktown Heights, NY, January 1995.

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

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

[107] Steven Reinhart and James Larus and David Wood. A Comparison of Message Passing and Shared Memory for Data-Parallel Programs. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, April 1994.

[108] Chandramohan A. Thekkath, Henry M. Levy, and Edward D. Lazowska. Efficient Support for Multicomputing on ATM Networks. UW-CSE 93-04-03, University of Washington, Seattle, WA, April 1993.

[109] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levey. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, June 1995.

[110] Ivan Tving. *Multiprocessor Interconnection using SCI*. PhD thesis, Technical University of Denmark, Department of Computer Science, August 1994.

[111] *UltraSPARC Programmer Reference Manual*. Sun Microsystems, 1995.

[112] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*. ACM, December 1995.

[113] 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.

[114] Deborah A. Wallach, Wilson C. Hsieh, Kirk L. Johnson, M. Frans Kaashoek, and William E. Weihl. Optimistic Active Messages: A Mechanism for Scheduling Communication with Computation. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 217–226, August 1995.

[115] Wolf-Dietrich Weber and Anoop Gupta. Exploring the Benefits of Multiple Hardware Contexts in a Multiprocessor Architecture: Preliminary Results. In *Proceedings 16th Annual International Symposium on Computer Architecture*, pages 273–280, New York, June 1989.

[116] Colin Whitby-Strevens. The Transputer. In *Proceedings 12th Annual International Symposium on Computer Architecture*, pages 292–300, New York, June 1985. IEEE.

[117] Steven Cameron Woo, Jaswinder Pal Singh, and John L. Hennessy. The Performance Advantages of Integrating Block Data Transfer in Cache-Coherent Multiprocessors. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 219–229, San Jose, California, 1994.

[118] W. A. Wulf. Evaluation of the WM Architecture. In *Proceedings of the 19th Annual Internatioinal Symposium on Computer Architecture*, May 1992.

[119] Donald Yeung. *Multigrain Shared Memory Systems*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, February 1998.

[120] Donald Yeung, John Kubiatowicz, and Anant Agarwal. MGS: A Multigrain Shared Memory System. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA'96)*, May 1996.

[121] H. Zima, H.-J. Bast, and M. Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6(1), 1988.

# Index