Closing the Window of Vulnerability in Multiphase Memory Transactions: The Alewife Transaction Store

by

John David Kubiatowicz

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

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1993

Signature of Author . . . .
Department of Electrical Engineering and Computer Science
February 1, 1993

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

Accepted by . . . .
Campbell L. Searle
Chairman, Departmental Committee on Graduate Students

Closing the Window of Vulnerability in Multiphase Memory Transactions: The Alewife Transaction Store

by

John David Kubiatowicz

Submitted to the Department of Electrical Engineering and Computer Science
on February 1, 1993, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

Multiprocessor architects have begun to explore several mechanisms such as prefetching, context-switching and software-assisted dynamic cache-coherence, which transform single-phase memory transactions in conventional memory systems into multiphase operations. Multiphase operations introduce a *window of vulnerability* in which data can be invalidated before it is used. Losing data due to invalidations introduces damaging livelock situations. This thesis discusses the origins of the window of vulnerability and proposes an architectural framework that closes it. The framework employs fully-associative *transaction-buffers* and an algorithm called *thrashlock*. It has been implemented as one facet of the Alewife machine, a large-scale cache-coherent multiprocessor.

Keywords: multiprocessor, context-switching, cache-coherence, multi-phase memory transaction, transaction-buffer, victim caching, Alewife machine

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

Acknowledgments

Much of the act of writing academic papers and theses is an exercise in revisionist history. I hope that the motivations, interfaces, mechanisms, and justifications given in the following pages give the impression that the Alewife cache controller was designed from first principles. I, of course, know better ...

I would like to acknowledge Pizzeria Uno for being open until 12:30am most evenings. This permitted me to spend long hours designing the hardware described within. I have never designed a system as large (and some would stress *as complicated*) as the Alewife cache controller. Many mistakes and false starts are forever burned into my memory.

Many thanks to my advisor, Anant Agarwal, for giving me the chance to design my first machine. It has been a learning experience, to say the least.

Also, many thanks to the members of the Alewife team. Their opinions, sometimes voiced loudly, led me to develop better arguments in support of my design choices. Occasionally, they even led to the development of better solutions. A project as large as the Alewife machine is certainly a team effort. Without the support of simulators and other software systems written by them, I would not as close as I am to a working A-1000 cache controller. The next few months should be an exciting period.

Finally, special thanks to David Chaiken for groveling though early implementations of the cache controller. He has all but suspended his academic career for the last year and a half to write a high-level simulation of the A-1000. When all is said and done, we make a good team.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

One of the major thrusts of multiprocessor research has been the exploration of mechanisms that provide ease of programming, yet are amenable to cost-effective implementation. To this end, a substantial effort has been expended in providing efficient shared memory for systems with large numbers of processors. Many of the mechanisms that have been proposed for use with shared memory, such as rapid-context switching, software prefetch, fast message-handling, and software-assisted dynamic cache-coherence enhance different aspects of multiprocessor performance; thus, combining them into a single architectural framework is a desirable goal. Unfortunately, this combination of features introduces a *window of vulnerability* in which data requested by a processor can be lost before it is consumed, either through remote invalidation or through cache conflicts.

This thesis investigates several methods of closing the window of vulnerability, culminating in a unifying architectural framework, the *transaction store*. As described in Chapter 5, the complete framework has been implemented in the MIT Alewife machine [3]; however, other multiprocessor architects may choose to implement a subset of this framework. To this end, Chapter 4 discusses several partial solutions, each of which are appropriate to a different subset of mechanisms.

1.1 Multi-phase Memory Transactions

Many of the mechanisms associated with shared memory attempt to address a central problem: access to global memory may require a large number of cycles. 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.

In a traditional shared-memory multiprocessor, remote memory requests can be viewed as *split-phase* transactions, consisting of a request and a response. The time between request and response may be composed of a number of factors, including communication delay, protocol delay, and queueing delay. Since a simple single-threaded processor can typically make no forward progress until its requested data word arrives, it spins while waiting. When the data word arrives, the processor consumes the data immediately, possibly placing it in a local cache.

Rather than spinning, a processor might choose to do other useful work. To tolerate long access latencies, architects have proposed a number of mechanisms such as prefetching, weak ordering, multithreading, and software-enforced coherence. All are variations on a central theme: they allow processors to have multiple outstanding requests to the memory system. A processor launches a number of requests into the memory system and performs other work while awaiting responses. This capability reduces processor idle time and allows the system to increase its utilization of the network.

The ability to handle multiple outstanding requests 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 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

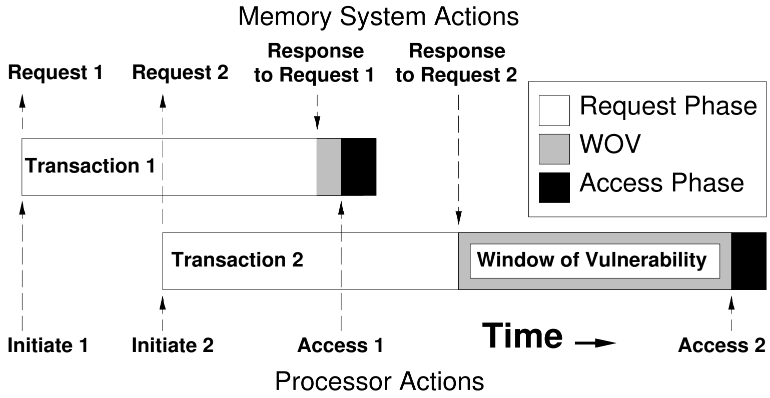

Figure 1-1: Multiple outstanding requests.

stores. This thesis explores the problems involved in closing the window of vulnerability in polled, context-switching processors. Signaling leads to related approaches; these are touched upon in Chapter 6.

Figure 1-1 illustrates the timing involved in overlapping access latency using a polling mechanism. The figure shows a time-line of events for two memory transactions that occur 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).

Since a processor continues working while it awaits responses from the memory system, it might not use returning data immediately. Such is the case in the scenario in Figure 1-1. 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. Note that a simple split-phase transaction can be seen as a degenerate multiphase transaction with zero cycles between response and access. 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.

1.2 High-Availability Interrupts

The window of vulnerability is also opened by another class of mechanisms. This class contains a number of mechanisms including fast I/O, interprocessor messages, synchronization primitives, and extensions of the memory system through software. When implementing such mechanisms, the successful completion of a spinning load or store to memory may depend on the execution of network interrupts. These asynchronous events must be able to fault an instruction which is in progress, thereby opening a window of vulnerability. The term *high-availability interrupt* is applied to such externally initiated pipeline interruptions.

1.3 The Framework

Closing the window of vulnerability involves ensuring forward progress for multiphase memory transactions. The consequences of lost data are more subtle and perilous than simple squandering of memory resources. The window of vulnerability allows scenarios in which processors repeatedly attempt to initiate transactions only to have them canceled during the window of vulnerability. In certain pathological cases, individual processors are prevented from making forward progress by cyclic *thrashing* situations. While such situations may be rare, they are as fatal as any other livelock or deadlock situation.

This thesis describes a framework that eliminates livelock problems associated with the window of vulnerability for systems with multiple outstanding requests and high-availability interrupts. The system keeps track of pending memory transactions in such a way that it can dynamically detect and eliminate pathological thrashing behavior. The 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.

Not all architects will need to implement the full gamut of mechanisms described in this thesis. For this reason, we describe the different subsets of the framework and the mechanisms that each subset will support. In order to motivate the architectural framework that we propose, Chapter 2 presents examples of shared memory mechanisms. Chapter 3 then shows how the window of vulnerability can impede a system's forward progress. Chapter 4 explores several components of the framework, each of which provides part of the solution for ensuring forward progress. Chapter 4 concludes with a hybrid architecture, called *ThrashLock*, that combines these components to implement all of the mechanisms. Chapter 5 describes how the issues discussed in earlier sections of this paper are reflected in the actual implementation of Alewife. As is shown in that chapter, the principle component of the implementation, the *transaction store*, is an important centralized resource and has many benefits over and above those which are developed by the earlier chapters. Chapter 6 discusses alternatives to the ThrashLock architecture which would be possible with more significant modifications to the processor pipeline. Finally, Chapter 7 concludes by examining the implications of the window of vulnerability on the design of shared memory systems.

Chapter 2

Hardware Mechanisms for Shared Memory Support

Three general classes of hardware support for efficient implementation of distributed shared memory 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. Atomic operations on critical system resources.

3. High-availability interrupts for response to high-priority asynchronous events, such as message arrival.

This section presents examples of some of the mechanisms that belong to these classes and makes a case for incorporating them into distributed shared memory machines.

The following section describes how each of these mechanisms leads to the same window of vulnerability problem. A given system might implement only a small subset of these mechanisms, in which case only a portion of our architectural framework would need to be implemented.

2.1 Coherent Caches with Multiple Outstanding Requests

Coherent caches are widely recognized as a promising approach to 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 [15, 23, 9].

In a cache-based system, memory and processor resources are wasted if no processing is done while waiting for memory transactions to complete. Such transactions include first-time data fetches and invalidations required to enforce coherence. 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 [7, 24], rapid context switching [28, 4], 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 [22, 16].

Allowing multiple outstanding transactions in a cache-based multiprocessor opens the window of vulnerability and leads to situations involving livelock.

2.2 Atomicity and Context Switching

In a system that supports multiple outstanding requests through context switching, the ability to perform complex atomic actions efficiently requires the occasional disabling of context switching. For example, we have observed that disabling is essential for performance in the presence of critical sections in a non-preemptive task scheduler. Furthermore, if a thread locks a critical system resource and then is forced to switch out, then performance suffers because many other tasks must wait for the context to

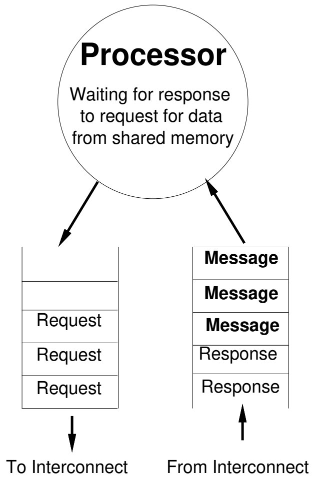

Figure 2-1: The need for high-availability interrupts.

release the lock. Thus, software on a context-switching machine should be able to disable context-switching temporarily. However, as explained in Section 4, this ability places a serious constraint on mechanisms that can be used to prevent livelock.

2.3 High-Availability Interrupts

The third class of mechanisms provides the ability to handle asynchronous, time-critical events under circumstances in which normal interrupts would be ignored. Such *high-availability* interrupts violate instruction atomicity by faulting loads or stores which are in progress. This class of interrupts allows migration of hardware functionality into software.

Figure 2-1 illustrates the need for high-availability interrupts in an architecture that supports fast message handling. In the figure, the processor is spinning while waiting to access a remote memory block. Several messages have entered the processor's input queue before the desired memory response. Consequently, the processor will not make forward progress unless a high-availability interrupt is invoked to process these messages.

In Alewife, for example, high-availability interrupts are used to implement the LimitLESS coherence protocol [9], a fast user and system-level messaging facility, and network deadlock recovery. LimitLESS interrupts must be able to occur under most circumstances, because they can affect forward progress in the machine, both by deadlocking the protocol and by blocking the network. Since the message passing interface relies on software for queueing, network queueing interrupts must be able to run under most circumstances. The network overflow interrupt relieves potential deadlock situations by redirecting input packets into local memory and relaunching them when the situation has abated.

Chapter 3

The Window of Vulnerability

This chapter examines the window of vulnerability in more detail. It first defines several terms, then introduces four distinct types of thrashing which can arise. Finally, it explores the severity of this problem through simulation.

3.1 Preliminaries

As shown in Figure 3-1, 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. Let us assume, for the sake of discussion, that *all requests which are made to the memory-system are eventually satisfied.* Note that this stipulation is difficult to guarantee in practice and is an interesting topic in its own right; it is, however, outside the scope of the current discussion.

3.1.1 Forward Progress

Consequently, 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.

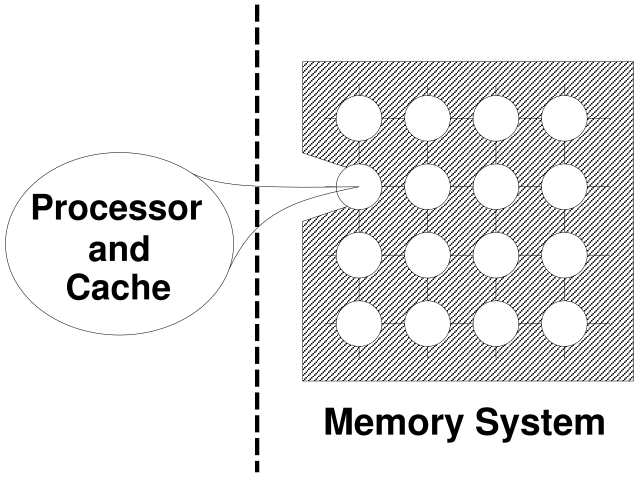

Figure 3-1: Treat the memory system (complete with network, coherence protocol, and other nodes) as a "black box".

To be more precise, a processor 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 can be said to make forward progress if the instruction pointer is advanced beyond it.

3.1.2 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.

3.1.3 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[1]. In contrast, a processor that saves its pipeline state when context-switching (thereby saving its faulting instruction) would retry only the

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

faulted data access. Each context in such a processor would have at most one primary transaction at a time.

Unless otherwise noted, this thesis 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 of Section 4.3, but they are required for the thrashlock mechanism of Section 4.5.

3.2 Four Different Livelock Scenarios

This section introduces the four distinct types of livelock or data thrashing which can occur in the processor's cache system. 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 ether'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. Chapter 4 will discuss methods of eliminating these livelock situations.

3.2.1 Invalidation Thrashing

Figure 3-2 illustrates the interaction between the window of vulnerability and cache coherence that leads to livelock. The figure gives the currently enabled context in the bar shown under the time-line. The scenario may be interpreted as follows: First, context A of the processor attempts to access memory block X (Read X). Since the data word is not currently in the processor's cache, the memory system issues a request (Read Req. X) and causes the processor to switch contexts. When the response to the request (Read Data X) returns to the processing node, context C is active. The shaded region indicates the window of vulnerability between the memory system response and the instant that context A is reenabled. During the window, the memory system causes block X to be invalidated from the processor's cache.

Figure 3-3 shows the multi-node scenario that causes this invalidation. There are three processing nodes in the figure: node 1 is the node associated with the time-line in Figure 3-2; node 2 is the home node for block X; and node 3 is the node that causes the interference. Some time after the home node has serviced the request, node 3 issues a write request for block X to node 2. In response, node 2 transmits an invalidation message to node 1, waits for an acknowledgment message, and eventually transmits write permission to node 3. As a result, node 1 must repeat its read request when it reenables context A at the end of the time-line in Figure 3-2.

There is no reason to expect that node 3 will actually complete the write to block X before node 1 repeats its read request! If this is the case, it is possible for node 2 to invalidate block X in node 3 before the write is finished. Given an unfortunate coincidence in timing, this vicious cycle of *invalidation* or *internode* thrashing can continue forever. Our simulations indicate that this thrashing is an infrequent event, but it does happen at some point during the execution of most programs. Without a solution to the thrashing scenario, the system would livelock (effectively causing the machine to crash).

3.2.2 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 3-4 uses a time-line to illustrate this thrashing scenario. Contexts A and C try to access blocks X and Y, respectively. Since blocks X and Y are congruent with respect to the cache mapping (see Figure 3-5), when the data for block Y arrives, it knocks the data for block X out of the cache. Each context prevents the other from making forward progress by replacing cached

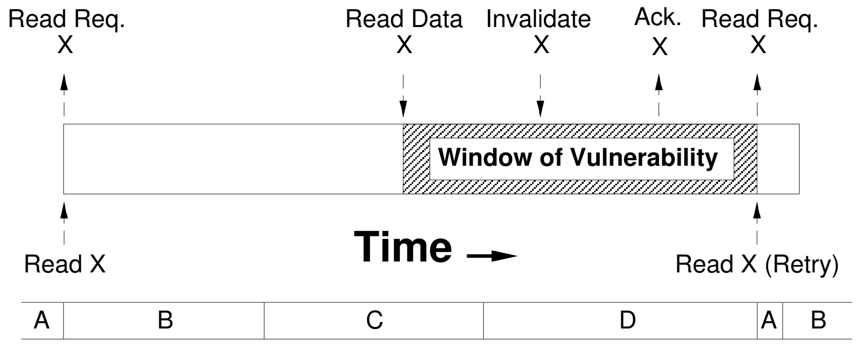

Figure 3-2: Time-line illustration of invalidation thrashing. The shaded area is the window of vulnerability.

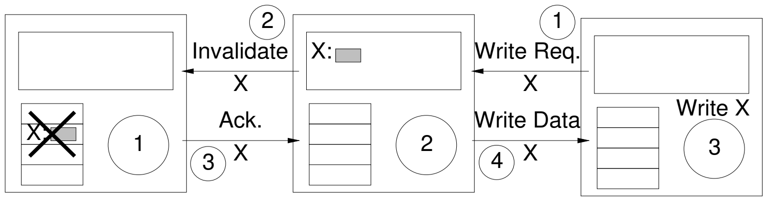

Figure 3-3: Diagram of cache coherence invalidation.

data during the window of vulnerability. As a consequence of this *replacement* or *intercontext* thrashing, a context-switching processor can livelock itself.

3.2.3 High-Availability Interrupt Thrashing

Figure 3-6 demonstrates a special case of replacement thrashing that is caused by high-availability interrupt handlers, rather than by a context-switching processor. 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

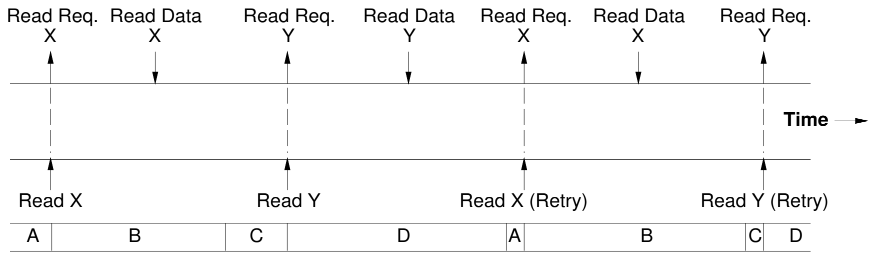

Figure 3-4: Time-line illustration of replacement thrashing.

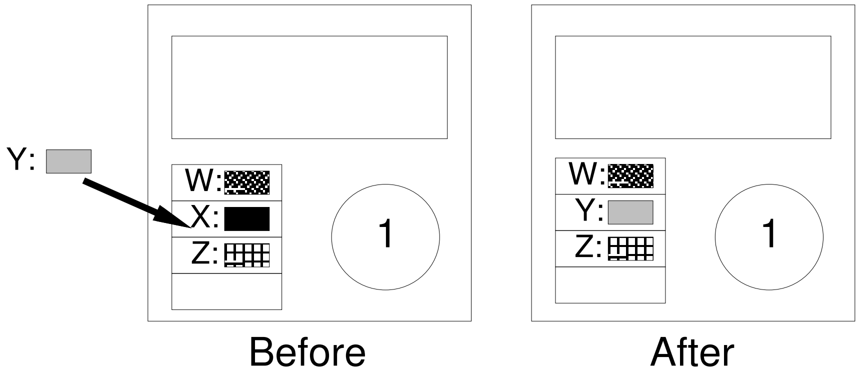

Figure 3-5: Diagram of cache replacement.

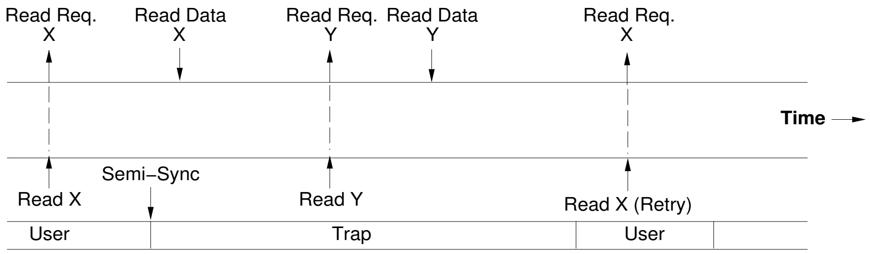

Figure 3-6: Time-line illustration of high-availability interrupt thrashing.

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.

Note that all interrupt code can cause this type of thrashing. High-availability interrupt thrashing is particularly important because it can occur even when context-switching has been disabled. This will be revisited in Chapter 4.

3.2.4 Instruction-Data Thrashing

As discussed in Section 3.1.3, instructions may need to be refetched whenever the processor reenables a context. Figure 3-7 shows a replacement scenario caused by a cache conflict between a load instruction and its data. Here, both the instruction and its data are congruent in the cache. Initial fetching of the instruction proceeds normally. However, when the processor subsequently requests data (Read Data D), it contexts switches, later returning to find that the data has displaced the instruc-

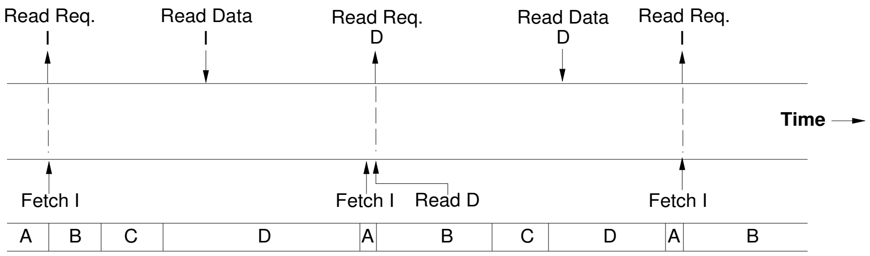

Figure 3-7: Time-line illustration of instruction-data thrashing.

tion. The re-requested instruction, in turn, displaces the data, and a thrashing cycle commences.

Note that instruction-data thrashing is independent of timing, since a single context is competing with itself for resources. 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 deadlock. This is in marked contrast to uniprocessor instruction-data thrashing, which does not cause deadlock but rather degrades performance.

3.2.5 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).

3.3 Severity of the Window of Vulnerability

This section supports the claim that the window of vulnerability poses a significant problem in shared memory architectures. To evaluate our proposed architecture, the Alewife research group constructed a cycle-by-cycle simulation of the processor, network, and cache controller (now called the communications and memory management unit). This simulation environment, called ASIM[2] permits parallel programs that are written in C or LISP (and soon FORTRAN) 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.

As an example of one of the statistics, the Alewife simulator calculates the time between the instant that a data block becomes valid in a cache due to a response from memory and 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 3-8 shows typical measurements of the window of vulnerability. The graph is a histogram of window 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

[2] 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.

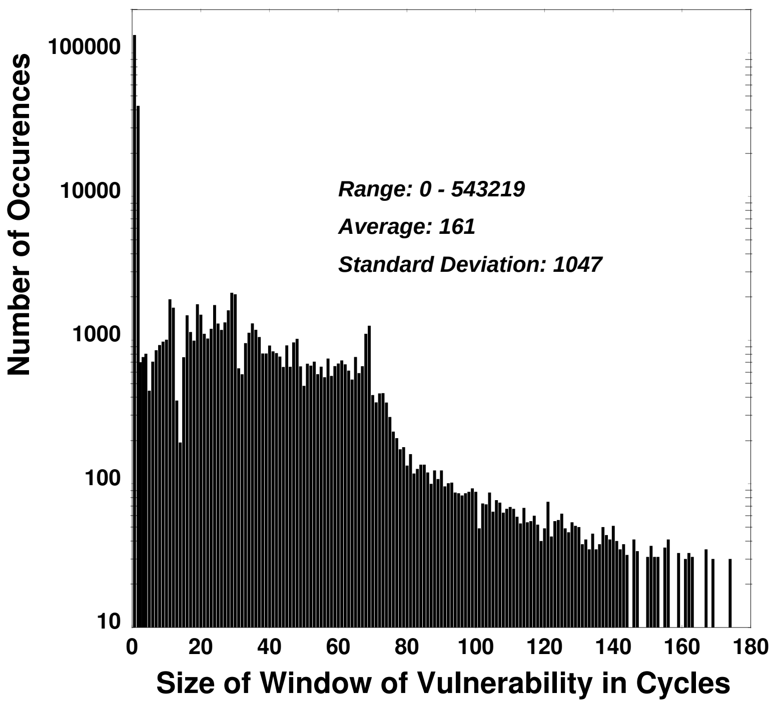

Figure 3-8: Window of vulnerability: 64 processors, 4 contexts.

the importance of the graph's tail.

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 certain critical sections of the task scheduler that disable context switching, as described in Section 2. 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 3-8 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 of our argument, 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*.

Chapter 4

Closing the Window

This section discusses a range of solutions for eliminating the livelock associated with the window of vulnerability. Three self-contained solutions are discussed, namely *associative locking*, *thrashwait*, and *associative thrashlock*. Each is appropriate for a different combination of the mechanisms of Chapter 2. As summarized in Table 4.1, 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 4.1 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 4.1: Window of Vulnerability closure techniques. *Multi* represents coherent caches and multiple requests. *Disable* represents disabling of context switching. *HAI* represents high-availability interrupts.

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 4.2: Properties of window of vulnerability closure techniques with respect to the complete set of features.

Locking involves freezing external protocol actions during the window of vulnerability by deferring invalidations. *Thrashwait* is a heuristic that dynamically detects thrashing situations and 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 4.2 summarizes the deficiencies of each of these mechanisms with respect to supporting the complete set of mechanisms. Associative thrashlock is a hybrid technique, and is discussed in Section 4.5. Note that only associative thrashlock permits the full set of mechanisms.

4.1 Locking (Partial Solution)

One approach to closing the window involves locking transactions during their window of vulnerability. For the moment, we will assume that returning data (responses) are placed in the cache; later, we consider the addition of an extra set of buffers for memory transactions.

Locking involves two state bits for each line in the cache. To prevent intercontext and high-availability interrupt thrashing, the system needs a *lock* bit to signal that a cache line is locked and cannot be replaced. When the line is accessed, the lock bit associated with the line is cleared. To prevent invalidation thrashing, we need a *deferred invalidate* bit; invalidations to locked lines are deferred by setting this bit. Deferred invalidation is performed (and acknowledged) when the requesting context returns and clears the lock bit.

4.1.1 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. Also, the transaction-in-progress state allows a processing node to consolidate accesses from different contexts to the same memory block.

We refer to this scheme as *touchwait*, because data blocks are held until the requesting context returns to "touch" it. Touchwait eliminates the livelock scenarios of the previous section, because the cache retains data blocks until the requesting context returns to access them.

4.1.2 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, for other schemes, the cost is less significant; more details will be given in Chapter 5.

4.1.3 Deadlock Problems

Unfortunately, the locking mechanism can lead to four distinct types of deadlock, illustrated in Figure 4-1. This figure contains four different *waits-for graphs* [6], which represent dependencies between transactions. In these graphs, the large italic letters

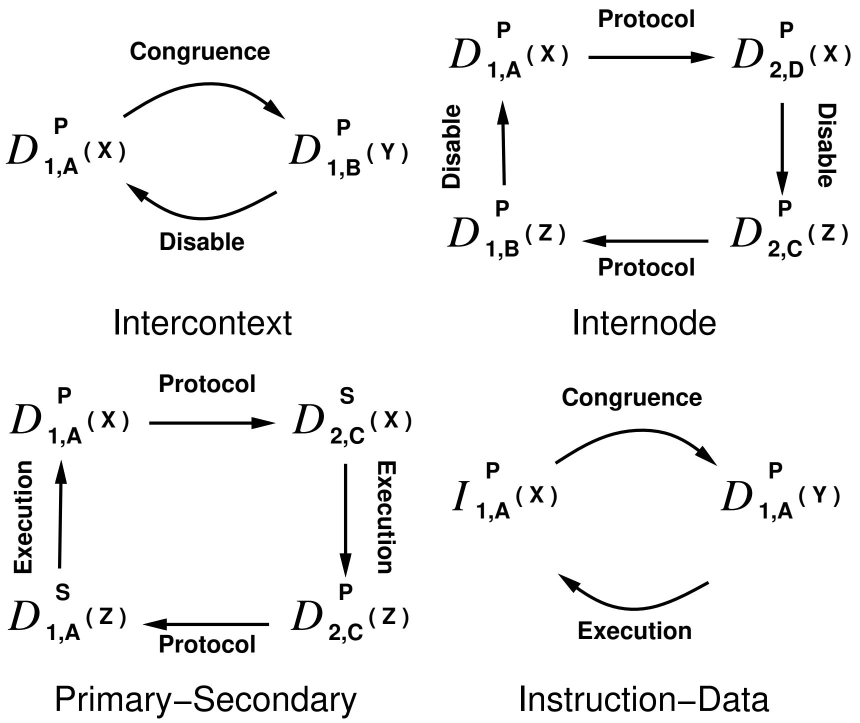

Figure 4-1: Deadlocks that result from pure locking. ($X \equiv Y$, $X \neq Z$)

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.

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.

4.2 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 (see Section 4.1.2). The use of a transaction buffer architecture has been presented in several milieux, such as lockup-free caching [18], *victim caching* [17], and the *remote-access cache* of the DASH multiprocessor [23].

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 4-1, because it eliminates all of the *congruence* arcs of Figure 4-1.

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 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 4.1, associative locking is sufficient for systems which do not permit context-switching to be disabled.

4.3 Thrashwait

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 follows[1]:

```
DO_GLOBAL_PROCESSOR_REQUEST(Address, Context)
if (data is ready for Address) ⇒ cache hit
  then clear tried_once[Context]
       clear thrash_wait
       return READY
if (Transaction-in-progress[Address]) ⇒ still waiting for transaction
  then if (thrash_wait or context-switching disabled)
         then return WAIT
         else return SWITCH
if (tried_once[Context]) ⇒ detected thrashing!
  then send RREQ or WREQ
       set thrash_wait
       return WAIT
⇒ normal cache miss
send RREQ or WREQ
set tried_once[Context]
if (context-switching disabled)
  then return WAIT
  else return SWITCH
```

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.

[1]Adapted from Chaiken [10]. The pseudocode notation is borrowed from [11]

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, because the transaction has completed.

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.

4.3.1 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.

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 4-2: Elimination of instruction-data thrashing through Thrashwait. At the point marked with (†), Thrashwait is invoked since Data_TO[0] is set.

As shown in Figure 4-2, 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.)

4.3.2 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 4-2 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.

4.3.3 Freedom From Deadlock

In this section, we prove that the thrashwait algorithm does not suffer from any of the deadlocks illustrated in Figure 4-1. 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.

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 4.1.3: *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.

4.3.4 Thrashwait and High-Availability Interrupts

Despite its success in detecting thrashing in systems without high-availability interrupts, thrashwait 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. 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.

4.4 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 4.2, 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.

4.5 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[2]. Thus, associative thrashlock has the same transaction dependency graph as thrashwait without high-availability interrupts (as in Section 4.3.3). 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 4.3.1). 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 instruction accesses incoherent. Since invalidations are never generated for instructions, the effect of the lock is nullified (no protocol arcs).

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

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 that even though the issues involved in closing the window of vulnerability are complicated, the end product is relatively straightforward. The next chapter examines an actual implementation of associative thrashlock in the Alewife communications and memory-management unit.

Chapter 5

Implementation of the Framework

The Alewife machine employs associative thrashlock to close the window of vulnerability. This chapter presents details about the implementation of the transaction store, including slight alterations in the thrashlock algorithm for programmability, the state of the transaction buffers, pipeline concerns, and transaction-buffer allocation. It also discusses a few added benefits of the transaction-buffer architecture, namely independence from network ordering, victim-caching, and non-binding prefetch.

5.1 Alewife and the A-1000 CMMU

Alewife is a large-scale multiprocessor with distributed shared memory. As shown in Figure 5-1, an Alewife processing node consists of a 33 MHz Sparcle processor, 64K bytes of direct-mapped cache, a 4Mbyte portion of globally-shared main memory, and a floating-point coprocessor. The Sparcle processor is a modified SPARC processor [19, 2], utilizing register-windows for rapid context-switching and block multithreading [4]. The current implementation provides four distinct hardware contexts. Both the cache and floating-point units are SPARC compatible. The nodes communicate via messages through a cost-effective direct network with a mesh topology. A single-chip communications and memory management unit (CMMU) on each node holds the cache tags and transaction buffers (described below), implements a variant of the cache coherence protocol described in [9], and provides a direct message-passing

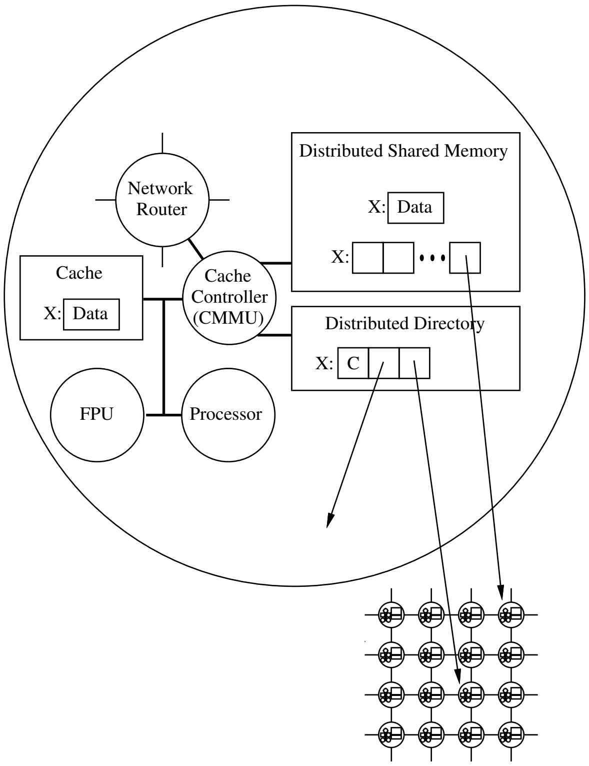

Figure 5-1: An Alewife Processing Node.

interface to the underlying network [20]. A block-diagram of this chip is shown in Figure 5-2.

This diagram, which is not to scale, illustrates the major components of the CMMU, namely, *cache management*, *memory management*, *transaction store*, *remote transaction machine*, and *network interface*. In addition, the cache tags-file consists of two banks of static RAM and comprises roughly one third of the chip area.

The cache management block is responsible for handling cache fills and invalidations, as well as full/empty bit synchronization [14, 26]; it is responsible for processing all control (non-data) messages for the processor-side of the LimitLESS cache-coherence protocol [9]. The memory management block handles the memory-side of the LimitLESS protocol, as well as memory requests from the local processor and DMA requests from the Network Interface; in addition, it handles DRAM refresh and error-correction (ECC). The network interface is responsible for Alewife's message-passing facilities as described in [21]; it provides efficient support for two distinct classes of message traffic: *Remote Procedure Invocation* involving short messages with

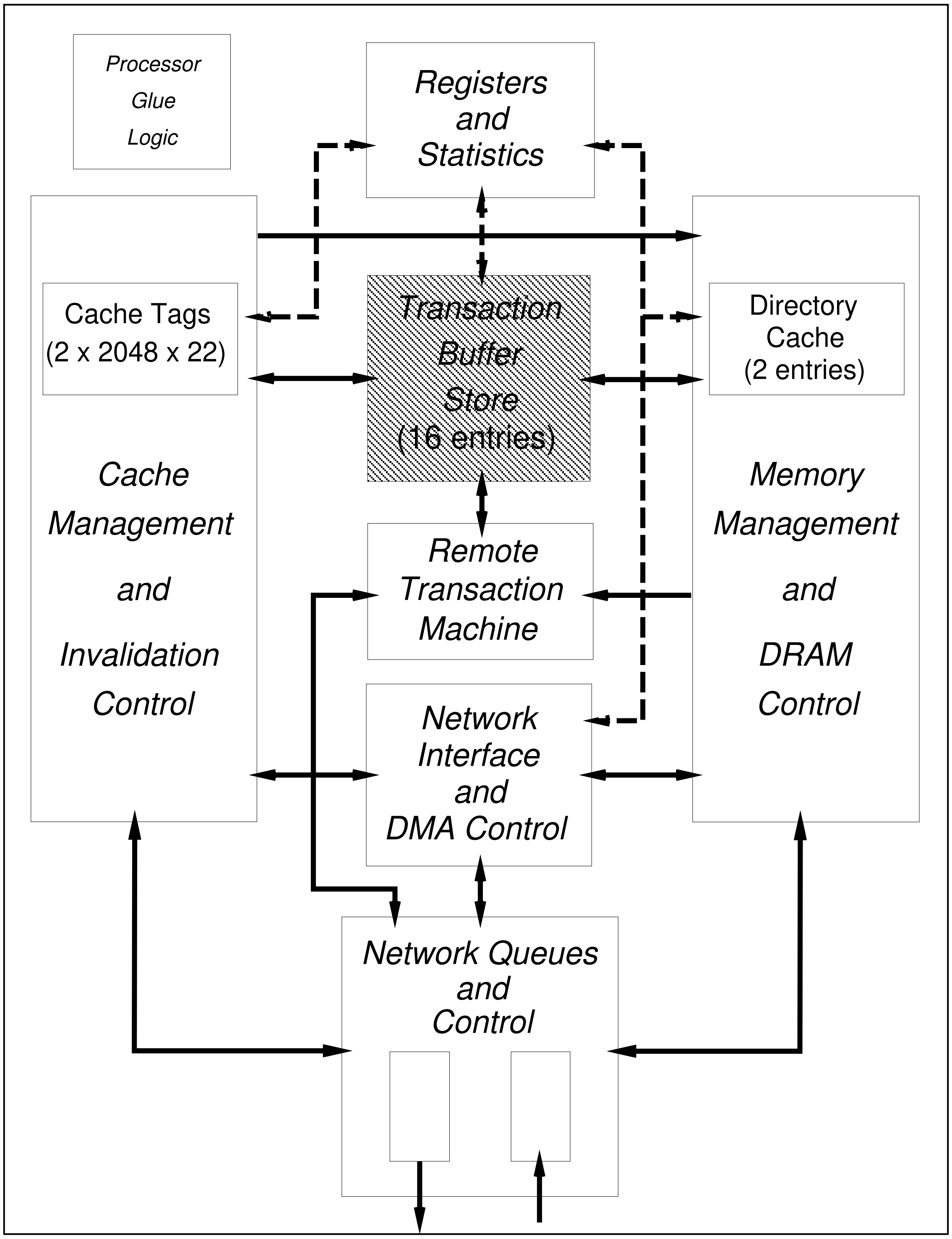

Figure 5-2: Internals of the A-1000 Communications and Memory Management Unit

values that are derived from processor registers, and *Block Data Transport*, involving the transfer of large blocks of data directly from memory via direct-memory access (DMA). The remote transaction machine handles protocol transitions for returning data (both from the network and the local memory), and supervises the transport of data between the network and the transaction store. It is also responsible for garbage-collecting reclaimable buffers when resources are low.

The *transaction store* provides a fully associative set of 16 *transaction buffers* to implement a variant of the associative thrashlock mechanism. Each of the transaction buffers can track one outstanding memory transaction and contains storage for an address, data, and state bits. A new transaction buffer is allocated for each memory request. At the beginning of a transaction (during the request phase), the data portions of the buffer are empty. Later, when a response returns from memory, the data portions of the buffer are used to hold the response until it is requested again by the processor or invalidated.

Chapter 4 mentioned that architectures with mechanisms akin to transaction buffers have appeared in several of machines for a number of reasons. This is not accidental. The explicit tracking of outstanding memory transactions has implications beyond closing the window of vulnerability. It permits reconstruction of an appropriate packet order in the face of network reordering. It also allows forward-progress guarantees to be made about the machine during periods of scarce or non-existent resources. And as multiprocessor architectures move beyond a single point of failure and into fault-tolerance, transaction buffer architectures will likely emerge again, as a vehicle for detecting and correcting transmission failures. Details of the transaction store of the A-1000 CMMU forms the primary topic of the remaining sections of this chapter.

Most of the A-1000 CMMU is implemented in a hardware description language called LES, for Logic Expression Synthesis. This language is included in LSI Logic's MDE and C-MDE[1] tool sets for the design of applications-specific integrated circuits (ASICs). At this point in time, the state of the art in hardware synthesis is not

[1] Both LES, MDE, and C-MDE are trademarks of LSI Logic Corporation.

yet capable of matching the speed and density of other design techniques (such as the explicit entry of circuits via schematic CAD tools). For the A-1000, this cost was particularly evident in the transaction store, which could have benefitted significantly from custom cell design. However, the ease of design afforded by hardware synthesis far outweighed the cost in cycle-time and area.

The resulting chip, which is 12mm×12mm, will be fabricated by LSI Logic with their 300K, 0.6μ hybrid gate-array process. The design is in its final stages of testing.

5.2 Transaction Store

The transaction store lies at the heart of the Alewife CMMU. This centralized module keeps track of all outstanding data transactions, both local and remote. In doing this, it combines several related functions:

- Window of vulnerability closure. The A-1000 employs a modified form of the thrashlock mechanism of Chapter 4.
- 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. This has an important advantage over a FIFO replacement queue: since the A-1000 recovers from *network overflow*[2] by interrupting to software, it is imperative that replacements to unshared variables can bypass replacements to shared variables; the latter may require network resources.
- Small, fully-associative cache. Under normal circumstances, the "access" phase of a transaction includes transferring a complete cache line from a transaction

[2]By their very nature, cache-coherence protocols introduce a dependence between the input and output queues of a memory controller: they process read and write requests by returning data. This leads to a possibility for *protocol deadlock*, since it introduces a circular dependence between the network queues of two or more nodes. *Network overflow* is a condition in which the output queue to the network has been clogged for a "long" period of time and is a good indicator that something is amiss [21]

buffer to the cache. It is also possible, however, to perform "uncached" reads and writes which access data directly in the transaction store rather than filling the cache; afterwards, the corresponding transaction buffers remain in the transaction store for future access. Such lines remain under complete control of the cache-coherence protocol.

- Repository for prefetched data. This is a simple extension of the previous item: non-binding prefetch operations return data to the transaction store.

The transaction store thus provides an important centralized resource. To permit this single interface to be used for all data movement between processor, network, and memory, the transaction store is fully bypassed and contains provisions for pipelining of data directly through transaction buffers.

As shown in Figure 5-3, the transaction store is is composed of two major blocks, the *transaction state* and the *transaction data*. Together, these modules comprise sixteen transaction buffers. The transaction state 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 from Chapter 4, and a *transaction monitor* for managing the pool of buffers. The transaction data module 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.

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 store 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 minimal 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). The

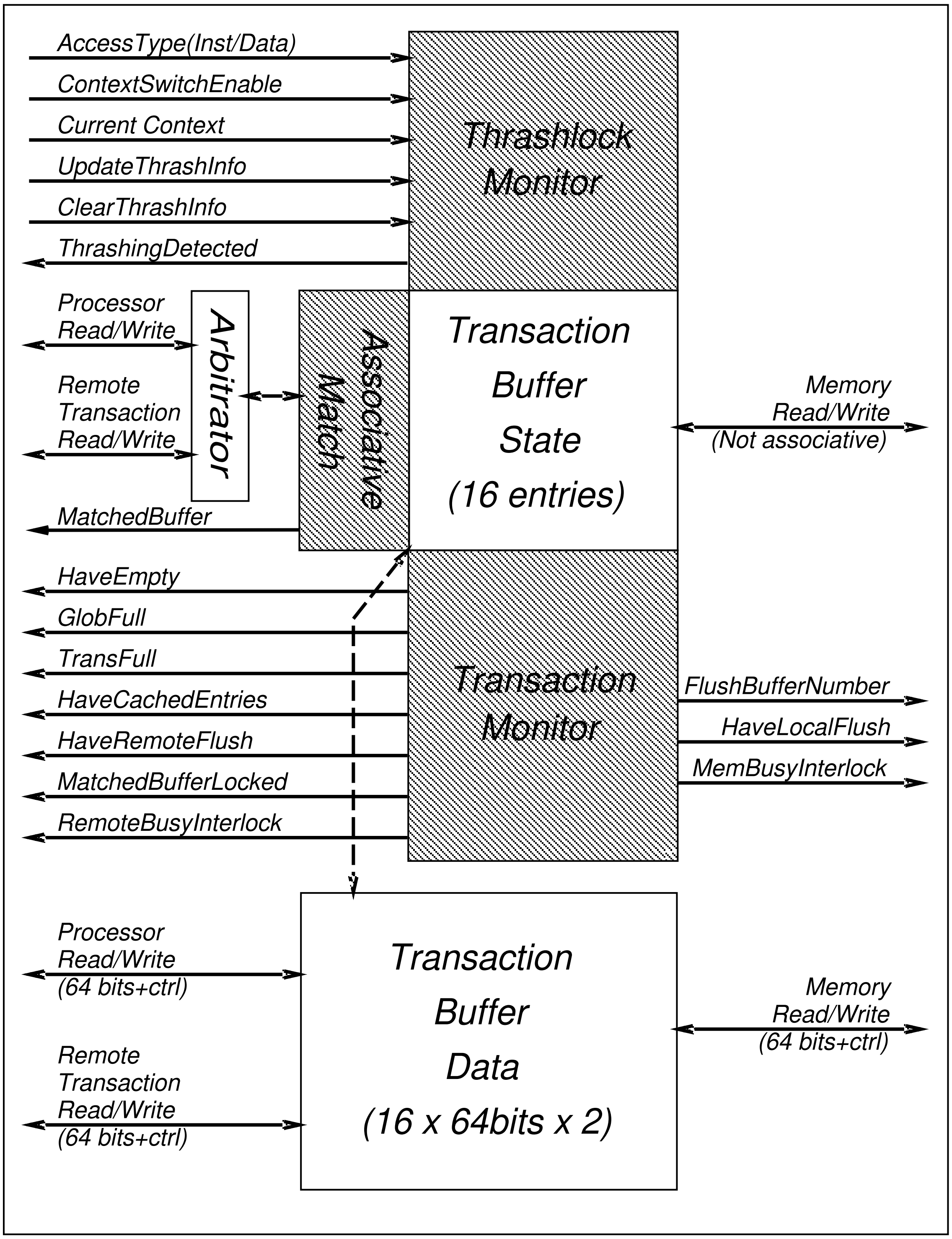

Figure 5-3: The transaction store.

ability to trade buffer usage between primary transactions and software prefetching is discussed in Section 5.6.

We will now proceed to examine each of the major blocks of Figure 5-3. Section 5.3 discusses the transaction state bits and ways in which they can be combined to form valid transaction buffer states. This includes a discussion of the way in which the transaction store allows the Alewife coherence protocol to be insensitive to reordering in the network. Then, Section 5.4 discusses the associative matching operation which can be performed on these buffers. This operation takes an address as input, and returns a non-empty transaction buffer with a matching address. Section 5.5 discusses the thrashlock monitor and introduces *tracking vectors* as a mechanism for implementing associative thrashlock with general buffer allocation. Each of these vectors associate a primary transaction with the transaction buffer that currently holds its state. This section proceeds with issues of thrashlock implementation, including the reordering of requests by the processor pipeline. Finally, Section 5.6 discusses the transaction monitor and its relationship to buffer allocation, garbage-collection, and flush processing.

5.3 Transaction State

Figure 5-4 illustrates the state of a transaction buffer. Several of its components were discussed in Chapter 4. The *address* field is 28 bits[3]. 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* are for synchronization on the four data words. Finally, the eight state bits are divided into 6 different fields. Some of these fields encode stable states indicating that transactions are in progress or that data is valid. Others encode transient states which are entered for short periods of time during the processing of a request. Briefly, these six fields are as follows:

[3]Sparcle addresses are 32 bits and memory-lines are 16 bytes long.

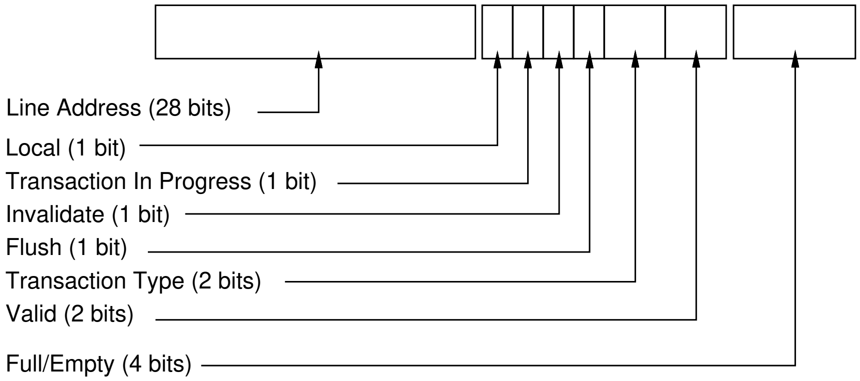

Figure 5-4: The state of a transaction buffer.

- The *Local*(LOC) bit caches information on whether the address is local (either shared or unshared) or remote. This bit is computed once, when the buffer is allocated.

- The *Transaction-In-Progress*(TIP) bit indicates that this transaction buffer represents a transaction which is in the request phase, *i.e.* one for which 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 4. Thus, it is used to delay invalidations to locked buffers until after the requesting contexts return for their data. As shown in Section 5.3.4, 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, `READ_ONLY` (00), `READ_WRITE` (10), and `READ_WRITE_DIRTY` (11). These will be abbreviated `RO`, `RW`, and `RWD` in the following. 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), then all three states are legal. `RWD` indicates that the buffer contains data which is dirty with respect to memory. A final state, `BUSY` (01), is only valid with an empty buffer; it indicates that a transaction has been terminated by a `BUSY` message from memory[4].

- The *Valid*(VAL) field contains a two-bit Gray code which indicates the degree to which buffer data is valid: `INVALID`(00), `ARRIVING`(01), `HALF_VALID`(11), and `VALID`(10). 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. Recall that transaction data is divided into two 64-bit pieces. `ARRIVING` indicates that protocol transitions have occurred, but no data has yet been placed in the transaction buffer. `HALF_VALID` indicates that the first 64-bit chunk of data has been written into the transaction buffer. These two states permit pipelining of data through a transaction buffer. Coupled with data bypassing, the transaction store can be used efficiently as an intermediate storage area for *all* data transactions, including local cache fills.

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`. Also, note that the `BUSY` transaction type is only valid in the Empty state.

Before exploring Table 5.1 in more detail, let us examine a simple transaction of the type introduced in the first few chapters. 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.

[4] This message is a negative acknowledgment on the request. It indicates that the memory was unable to satisfy the request. See [10] for more information.

Buffer State					Description	Note
TIP	F	INV	TT	Val		
0	0	0	—	no	Empty and available for allocation.	A, G
			RO	yes	Read-Only data present	A
			RW	yes	Read-Write data present (clean)	A
			RWD	yes	Read-Write data present (dirty)	A
0	0	1	RO	yes	Read-Only data/pending invalidate	B
			RW	yes	Read-Write data (clean)/pending invalidate	B
			RWD	yes	Read-Write data (dirty)/pending invalidate	B
0	1	0	RWD	yes	Flush dirty data (no protocol action).	C, H
0	1	1	RO	no	Send Acknowledgment.	C
			RW	no	Send Acknowledgment.	C
			RWD	no	Flush F/E bits and write permission.	C, H
			RWD	yes	Flush dirty data and write permission.	C
1	0	0	RO	no	Read transaction.	D
			RW	no	Write transaction.	D
			RW	yes	Write transaction/read data valid.	D
1	0	1	RO	no	Read trans/premature read INV.	E
			RW	no	Write trans/premature write INV.	E
			RW	yes	Write trans/read data valid/prem write.	E
1	1	0	RO	no	Read trans/flush on arrival	F
			RW	no	Write trans/flush on arrival	F
1	1	1	RO	no	Read trans/flush on arrival/prem read INV.	E
			RW	no	Write trans/flush on arrival/prem write INV.	E

Note	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 buffer state bits. Missing states are illegal.

When data returns from memory, the *remote transaction machine* gives the address from the returning packet to the associative matching facilities 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 subsections explore different classes of states and their uses. These include states for caching data items, states for tracking transactions in progress, transient states, and states which permit insensitivity to network reordering.

5.3.1 The Transaction Store as a Cache

States in Table 5.1 that are marked with note **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 Chapters 1–4, 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 store, 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 Store to act much like a small second-level cache. Consequently, non-binding prefetches simply return their data to the transaction store[5]. Further, cache-lines can be victim cached [17] in the transaction store. This means that the last few lines that have been replaced from the primary cache are placed into the transaction buffer and can be transferred back to the primary cache without a long access latency should they be re-requested by the processor. Finally, special "uncached" loads and stores can access data directly in the transaction store without first transferring it to the primary cache.

Use of the transaction store as a generic cache has two consequences. First, it

[5] 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 next section.

requires some form of garbage-collection or replacement algorithm, since buffers which are in the cached state can accumulate, leaving no buffers for primary transactions. The A-1000 employs the following solution to this problem: all buffers that are in the cached state and not part of a primary transaction are considered *reclaimable* and may be flushed at any time. Garbage collection is scheduled by the transaction monitor (Section 5.6) and performed by the remote transaction machine.

Second, the issue of duplicate data in the cache and transaction buffer must be addressed. The A-1000 uses as a policy that the only duplication which is allowed 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[6]. As soon as the data is written by the processor, the read-only copy will be either invalidated or overwritten.

5.3.2 Transaction in Progress States

States in Table 5.1 which have their TIP bits set indicate transactions which are in the request phase. These are transactions for which requests have been sent but data has not yet been received. The presence of these 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 with note **D** are the ones which are entered when the transaction is first initiated. Others can be entered upon the execution of `flush` instructions or by reordering in the network.

5.3.3 Transient States

States in Table 5.1 which are marked with notes **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.

[6]This situation is allowed because the memory-side of the coherence protocol explicitly checks to see if the requester of write permission is also one of the readers. In this case, it doesn't invalidate the requester's copy before granting write permission.

Section 5.6 discusses the mechanism behind this scheduling. The one exception to immediate scheduling is that buffers in states marked with **B** can be protected by buffer locks (in the sense of Chapter 4). Such buffers remain unmolested in the transaction store until their locks are released, at which point they become scheduled for processing. Buffer locks are discussed in Section 5.5.

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 mentioned earlier, the transaction store 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[7]. The use of transient states in this way has two advantages over employing a more conventional FIFO queue. First, memory can process flushed cache-lines out of order during network overflow recovery. In particular, since network blockage 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

[7]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.

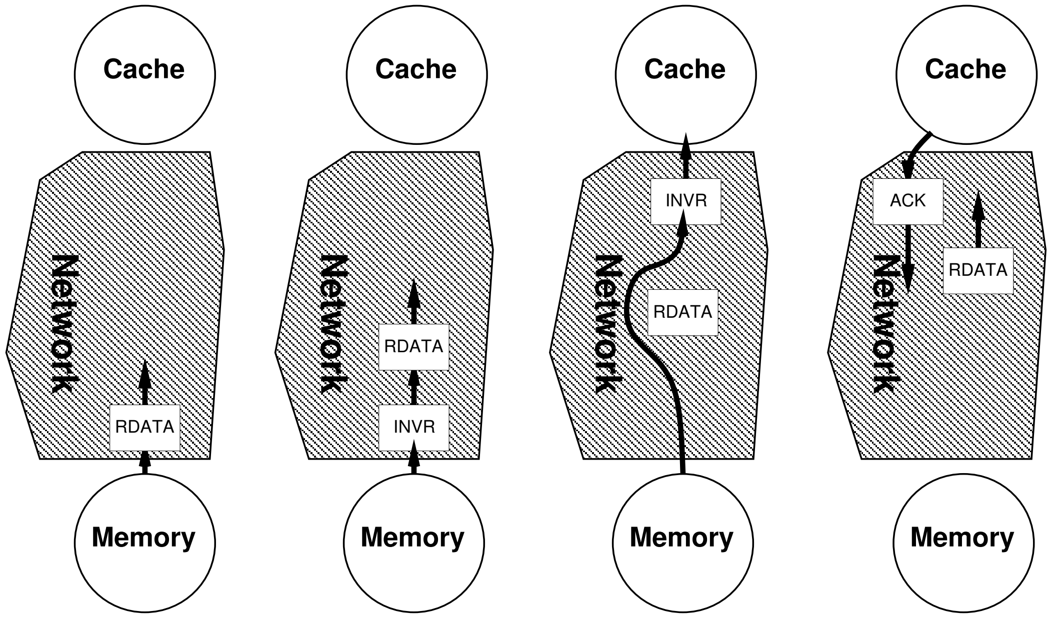

Figure 5-5: The effect of network reordering on an uncompensated protocol.

bits. The garbage collector discards buffers in this way.

5.3.4 Insensitivity to Network Reordering

One advantage to maintaining explicit state for all outstanding memory transactions is that the coherence protocol can be insensitive to reordering in the network. What this statement means is that the Alewife coherence protocol does not rely upon the preservation of order for messages sent from a specific source node and received by a specific destination node[8]. Such independence has a number of advantages. It permits the use of adaptive or fault-tolerant networks. 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 also affords some flexibility in choosing the algorithm which is used for network overflow recovery[21].

A simple example, shown in Figure 5-5, illustrates the danger of reordering in

[8]Machines which use a general interconnect, such as Alewife and DASH[23] support networks with bidirectional channels and nondeterministic packet scheduling. Ordering between disparate source/destination pairs has never been assumed for protocols on such networks.

[9]Protocol invalidations require the examination of the cache tags-file, while returning data requires only the transaction buffer. Consequently, in the A-1000, invalidations are handled by the cache management machine while data is handled by the remote transaction machine.

the network. 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 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.

The Alewife transaction store plays a fundamental role in handling reordering in the network. 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 A-1000 carefully restricts the number and type of protocol messages which 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. The transaction store assists by providing state at the cache side of the protocol.

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

[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.

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.

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 5-5 and can lead to incoherence if not dealt with properly. Memory will be incorrectly notified that an invalidation has occurred.

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. States entered as a result of premature invalidation are marked with an **E** in Table 5.1.

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].

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

[12]If the buffer is locked (See Section 5.5.2), then returning data will be placed into the buffer and the TIP bit will be cleared. This places the buffer in a transient state (of type B). 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.

5.4 Associative Matching Port

The associative matching port of the transaction store supports a parallel search for buffers. It takes an address as input and returns a transaction buffer with a matching address. This port is arbitrated for each cycle by the cache management hardware and the remote transaction machine[14]. 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[15]. 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 5.3.4.

[14]Note that the satisfaction of a data request by local memory is partially handled by the remote transaction machine. The remote transaction machine takes an address from the memory machine and uses the associative matching port to locate the appropriate transaction buffer. It also modifies the buffer state by lowering TIP and setting the Valid field to `ARRIVING`, just before passing the buffer identifier back to memory. Memory then proceeds to fill it with data.

[15]An associative matching option which is provided by the A-1000 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 [21] which coexist with the protocol but which are exclusively interested in dirty data.

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[16].

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). These two match signals are then combined with signals from other buffers 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 store. It may select one of two buffer destinations: either the next empty buffer or the current matched buffer. The transaction monitor, described in Section 5.6 manages empty buffers.

5.5 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 5-3, the thrashlock monitor takes as input the current context number $(0-3)$, the current access 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 THRASHINGDETECTED signal, which is used to freeze the processor pipeline. Further, two control signals, CLEARTHRASHINFO and UPDATETHRASHINFO are used by the cache management hardware to alter the state of the Thrashlock monitor. This interface will be described later, after the introduction of *tracking vectors*.

[16]The result of such a match is sent with requests to local memory.

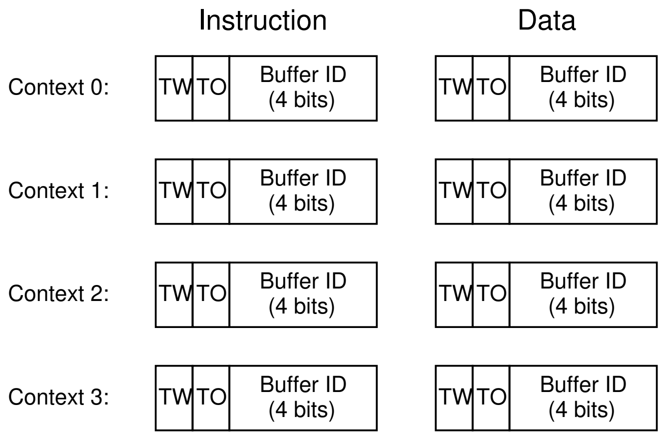

Figure 5-6: Tracking vectors for implementing the thrashlock mechanism.

5.5.1 Tracking Vectors

Figure 5-6 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. This is discussed in Section 5.5.5.

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 4.3, 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 (these happen in parallel in the A-1000):

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 number in the appropriate tracking vector. Also, set its TO bit.

5. Finally, if thrashing was detected on a previous incarnation of this primary transaction, set the TW bit.

To permit thrash detection across interrupts, tracking vectors *must not* be altered by accesses to local, unshared memory locations.

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.

5.5.2 Buffer Locks

The Thrashlock mechanism requires as many buffer locks as simultaneous primary transactions. Recall from Section 4.5 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 A-1000 generates a 16-bit vector of locks, called the *No_Invalidate* vector. A bit in this vector is set if the corresponding transaction buffer is pointed at by a live tracking vector whose TW bit is asserted.

The No_Invalidate 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 No_Invalidate 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 5-4, 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 5.6, is responsible for scheduling transient states in accordance with current buffer locks.

Finally, a brief discussion of the number of buffer locks is in order. As discussed in Chapter 4, 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 which they interrupted.

2. *All* accesses to global memory must be from a context which differs from interrupted context.

3. *All* accesses to global memory in trap handlers must be restricted 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 4.5, which simply disallowed global accesses in the trap handler (automatically satisfying conditions (2) and (3) above). The additional freedom afforded by conditions (2) and (3) is particularly advantageous for operating-systems variables which are part of global structures but are accessed only by the local processor.

5.5.3 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 *No_Reclaim* vector, which is derived from the set of tracking buffers: a bit in this vector is set if the corresponding transaction buffer is pointed at by a live tracking vector. The No_Reclaim 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. Premature lock release was introduced in another guise in Section 4.1.2. 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` messages: we simply mark them as empty, with the special transaction type of `BUSY`. Such transactions are not considered to be thrashed. This is advantageous because `BUSY` messages are generated during periods in which the memory is sending or waiting for invalidations. Such periods can last for a number of cycles. Consequently, if thrashing is detected on busied transactions (and context-switching disabled), it is possible for the cache system and memory system to get locked into a cycle of sending requests and `BUSY` messages[17]. It is much better to conserve memory and network bandwidth by making a single request and context-switching.

5.5.4 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 A-1000 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, which is defined as follows:

[17]Note that the presence of `BUSY` messages is intimately linked to forward progress at the memory side of the protocol. As discussed at the beginning of Chapter 3, this thesis *assumes* that all requests to memory are eventually satisfied.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\text{ThrashDetected} =} \\
1 & \text{Context\_Switching\_Disabled} \lor \\
2 & \text{Thrash\_Wait}_{Data} \lor \text{Thrash\_Wait}_{Inst} \lor \\
3 & (\text{Tried\_Once} \land \neg\text{MatchedTXB} \land \text{TT}(\text{Pointer}) \neq \texttt{BUSY}) \lor \\
4 & (\text{Tried\_Once} \land \text{MatchedTXB} \land \text{ID}(\text{MatchedTXB}) \neq \text{Pointer})
\end{array}
$$

Where MatchedTXB signals a matched transaction buffer with TIP=1 or Valid≠0, ID(MatchedTXB) is the number of this buffer, Pointer and Tried_Once are fields from the active tracking vector, TT(Pointer) is the transaction-type of the buffer pointed at by Pointer, and $\text{Thrash\_Wait}_{Data}$ and $\text{Thrash\_Wait}_{Inst}$ are from the current context.

The ThrashDetected signal is used only when data is not available. By way of explanation, line (1) asserts that the disabling of context-switching causes all accesses to behave as if they were thrashing. Line (2) illustrates the persistent thrash-detection state discussed earlier. Line (3) contains the simple thrashwait detection mechanism, with a simple modification to prevent the detection of thrashing on busied transactions. Finally, line (4) illustrates the detection of premature lock release.

Two control signals which can be used to manipulate the thrashlock monitor are the ThrashDetected and ClearThrashInfo signals. To define their behavior, we introduce two additional symbols. Let Thrash_Wait (with no subscript) be either $\text{Thrash\_Wait}_{Data}$ or $\text{Thrash\_Wait}_{Inst}$ depending on whether the current access is for data or for instructions respectively. Further, let WriteTXB denote the transaction buffer which the processor is about to modify this cycle. WriteTXB is either an empty buffer or the current associatively matched buffer. Then, UpdateThrashInfo and ClearThrashInfo have the following behavior *after the next clock edge*:

$$
\begin{array}{lrl}
\text{UpdateThrashInfo} \Rightarrow & \text{Thrash\_Wait} & \leftarrow \text{ThrashDetected} \\
 & \text{Tried\_Once} & \leftarrow 1 \\
 & \text{Pointer} & \leftarrow \text{WriteTXB} \\
 & & \\
\text{ClearThrashInfo} \Rightarrow & \text{Thrash\_Wait} & \leftarrow 0 \\
 & \text{Tried\_Once} & \leftarrow 0
\end{array}
$$

Finally, an updated version of the processor-access pseudo-code of Section 4.3 can now be constructed. Let TIP(MatchedTXB) be the TIP bit from the matched transaction buffer. Then, a processor access is as follows:

```
Do_Global_Processor_Request(Address, Context, AccessType)
if (cache-hit for Address of type AccessType)
  then assert ClearThrashInfo
       return READY
if (¬MatchedTXB) ⇒ No matching buffer.
  then send RREQ or WREQ
       allocate and initialize new transaction buffer
       assert UpdateThrashInfo
       if (ThrashDetected) ⇒ thrashing signaled.
         then return WAIT
         else  return SWITCH
if (TIP(MatchedTXB)=1) ⇒ Currently have active transaction.
  then if (ThrashDetected)
         then assert UpdateThrashInfo
              return WAIT
         else  return SWITCH
⇒ Data waiting in transaction buffer.
fill cache from buffer
clear transaction buffer
assert ClearThrashInfo
return READY
```

5.5.5 Pipeline Reordering

Tracking vectors are an extension of the processor pipeline, because they maintain the memory access state of ongoing instructions. Unfortunately, complications arise because they are implemented *outside* this pipeline. In particular, the pipeline reorders memory accesses with respect to the stream of memory requests from the processor; the fetch of a load or store instruction appears several cycles before the data access that it invokes. Multiple instruction fetches can happen in the interim. This is illustrated by Figure 5-7, which shows a segment of a program that contains a load instruction followed by three single-cycle arithmetic instructions. Until the load commits, the two primary accesses are the fetch of the load instruction and the fetch of its data. These are shaded in the diagram.

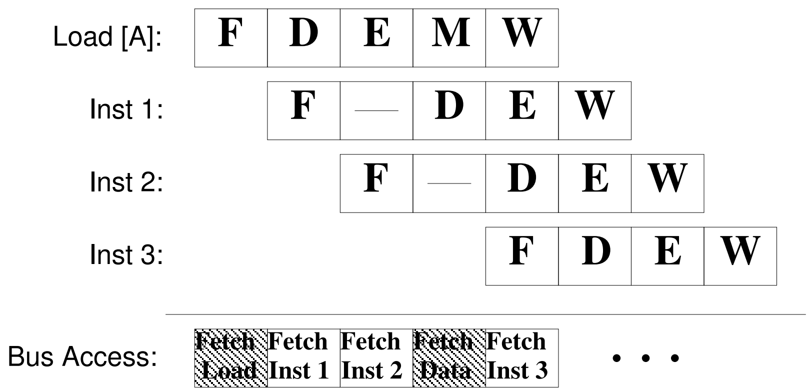

Figure 5-7: Pipeline reordering of memory accesses. Primary bus accesses are shaded. Pipeline stages are **F**etch, **D**ecode, **E**xecute, **M**emory, and **W**riteback.

Since `Inst 1` and `Inst 2` are semantically after the `Load` instruction, two conditions must be true:

- Attempts to fetch `Inst 1` and `Inst 2` should not alter the state of the instruction tracking vector until the `Load` instruction completes, or at least must not interfere with the thrashing protection provided by it.

- Attempts to fetch `Inst 1` and `Inst 2` must not prevent the data access from completing. This must be true even if context-switches are signaled on these fetches.

The second condition is handled automatically by the pipeline: once the `Load` instruction has been fetched successfully, the data access will be attempted even if one or both of the following instruction fetches are faulted.

The first requirement, however, is not as easy to satisfy. Ideally, the instruction tracking vector would be dedicated to the instruction fetch of the `Load` instruction until this has committed, at which point it would transfer its allegiance to the fetch of the next instruction (*Inst 1*). An implementation which was integrated directly with the pipeline would probably employ multiple shadow tracking vectors arranged in a stack which reduced to two well-defined vectors when the pipeline was interrupted (much like the program-counter stack).

Since pipeline modifications to Sparcle were not possible, however, some other technique had to be employed. Consequently, the Alewife machine employs two heuristics which yield a similar result, from the standpoint of forward progress, as if all accesses were exactly identified, but which may occasionally hold the pipeline for longer than necessary or cause an extra context switch[18]. These are:

1. Never overwrite live tracking buffers that point to transactions in progress (*i.e.* TIP set).

2. Never thrashwait requests whose associated tracking buffers are live and point to transactions in progress.

Whenever an instruction fetch to shared memory is initiated, these conditions ensure that the next few instruction fetches will be ignored by the thrashlock monitor: in the A-1000 it is not possible for requests to be satisfied (*i.e.* TIP lowered) in less than five cycles. Therefore, in situations in which the `Load` instruction must be fetched from a remote node, the instruction tracking vector will remain dedicated to this fetch.

The second condition forces a slight modification of line (3) of the definition of ThrashDetected. This condition is necessary to prevent detection of thrashing when an instruction fetch legitimately starts a new transaction and flags a context switch. Since this context switch cannot be recognized until the execute state, the next few instructions will be fetched anyway. Consequently, if one of these next few instructions also misses in the cache and starts a new transaction, the fact that the Tried_Once bit is set (from the first instruction miss) causes the unmodified version of line (3) to be flagged, freezing the pipeline unnecessarily. We prevent this incorrect detection of thrashing by recognizing that the instruction tracking vector is live and points to a buffer with TIP = 1.

5.6 Transaction Monitor

The transaction monitor, shown in Figure 5-3, is responsible for the allocation and management of transaction buffers. It takes state information from each of the 16

[18] The way in which the thrash-wait bits are used can also hold the pipeline unnecessarily, since setting a data thrash-wait bit will freeze the pipeline during the fetching of `Inst 1` and `Inst 2`.

buffers and combines it with the *No_Reclaim* and *No_Invalidate* vectors (see Section 5.5) to perform a number of supervisory tasks. These are listed below:

5.6.1 Buffer Allocation Constraints

Since transaction buffers are a limited resource in the A-1000, 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 No_Reclaim vector. Then, two allocation invariants which are maintained by the cache management machine can be defined as follows:

$\textsc{TransactionCount} \leq 12 \qquad (< \max \textsc{GlobalCount})$

Where TRANSACTIONCOUNT is the total number of buffers for which:

$(\textsc{GlobalBuffer} \wedge \mathrm{TIP} = 1) \vee$

$(\mathrm{Flush} = 0 \wedge \mathrm{INV} = 0 \wedge \mathrm{Valid} \neq 0) \vee$

$\neg\, \textsc{Reclaimable}$

$\textsc{GlobalCount} \leq 14 \qquad (< \text{Total number of buffers})$

Where GLOBALCOUNT is the total number of buffers for which:

$(\textsc{GlobalBuffer} \wedge \mathrm{TIP} = 1) \vee$

$(\textsc{GlobalBuffer} \wedge \mathrm{INV} = 1) \vee$

$(\textsc{GlobalBuffer} \wedge \mathrm{Flush} = 0 \wedge \mathrm{Valid} \neq 0) \vee$

$\neg\, \textsc{Reclaimable}$

Note that these invariants are different from the constraints on the number of buffers with identical addresses which were discussed in Section 5.4. These allocation invariants are maintained by the cache system since:

1. The cache management machine never allocates transaction buffers which violate them.

2. Both counts are monotonically decreasing with respect to state transitions other than allocation[19].

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, 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:

$$\begin{aligned}\textsc{TransFull} &\Rightarrow (\textsc{TransactionCount} \geq 12)\\ \textsc{GlobFull} &\Rightarrow (\textsc{GlobalCount} \geq 14)\end{aligned}$$

Thus, 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 first checks

[19]Actually, the TransactionCount limit may be violated briefly by one transition: unshared transaction in progress → 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.

to make sure that *both* TRANSFULL and GLOBFULL are deasserted (since the new buffer would fall into both categories). If not, then it asserts the processor hold line until the new transaction can be launched or until a high-availability interrupt must be processed.

5.6.2 Allocation of Empty Buffers

Those buffers that are in the empty state and not protected by the No_Reclaim 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.

5.6.3 Identification of Buffers for Garbage Collection

As mentioned in Section 5.3, use of the transaction store 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 No_Reclaim 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[20] 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.

[20]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.

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.

5.6.4 Scheduling of Transient States

As discussed in Section 5.3, some transaction buffer states are transient, namely those with note **C** in Table 5.1 and those marked with note **B** which are not locked. The transaction monitor combines the No_Invalidate vector, described in Section 5.5.2, 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 5-3) 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[21]. 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.

[21] The memory machine always verifies that it has sufficient resources to finish a request *before* starting to process it.

5.7 Transaction Data

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. The memory bus and internal network queues are also 64 bits. One of the goals which impacted the design of the transaction store 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 A-1000, 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 A-1000 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 5.3. Logic to modify and examine the Valid bits is integrated directly into the data access ports of the transaction store.

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[22].

[22] The Gray code makes this somewhat cheaper to implement.

Chapter 6

Closing the Window of Vulnerability: The Signaling Approach

Chapters 4 and 5 describe a method for closing the window of vulnerability which may be employed with standard RISC pipelines. This direction was appropriate for the Alewife machine, since the Sparcle processor was the result of a collaborative effort between MIT and LSI-logic which modified an existing SPARC processor *without* altering the pipeline[19]. The model of context-switching which is appropriate for such "vanilla" architectures is that of *polling*, *i.e.* retrying memory requests until they are satisfied.

Other approaches to closing the window are possible, however. This chapter starts by examining a simple extension of the mechanisms of the previous chapter, then continues with more pipeline-intrusive *signaling* alternatives.

6.1 Retaining Invalidated Read Data

One simple modification to the mechanisms of Chapter 5 eliminates the window of vulnerability for read operations: when the data for a read request is invalidated before the read has committed, acknowledge the invalidation but hold on to the data. At the time that the requesting context returns, pass a single word (or atomic double-word) of data to the processor, then discard the data. Giving a single item of invalidated data to the processor does not violate cache consistency, since the load *could* have committed just *before* the invalidation.

Further, holding of data in this way is straightforward to implement in the framework of Chapter 5: when data from a primary transaction is invalidated during the window of vulnerability, the transaction buffer which contained it is protected by the No_Reclaim vector. Consequently, this data is still available, even though the buffer is technically "empty". Tracking vectors are necessary to detect thrashing as before and to allow a single access to what otherwise appears to be an empty transaction buffer. Note that care must be taken with this approach to prevent contexts other than the requesting context from accessing this data, since that could cause a violation of the supported memory-model[1].

For a context-switching processor, it is not entirely clear which approach to the handling of invalidated read data is desirable, that presented in the earlier chapters (re-requesting of invalidated data) or that presented here (using of stale data). While this distinction is not important from a correctness standpoint, it can have performance implications, especially for synchronization variables. Data which is retained after invalidation is guaranteed to be stale with respect to memory. For programs which use a test and test-and-set strategy to acquire spin locks, use of stale data can have the unfortunate consequence of raising the frequency with which write operations are attempted for variables which are already locked. Further, for machines that use Full/Empty bits for synchronization, the acquisition of stale data can cause the consumer to incorrectly conclude that a value is not yet ready when in reality it was produced during the execution of another context. Depending on the cost of the backoff strategy which is invoked after synchronization failure, the use of stale data may have a non-trivial performance impact. More study is called for here.

For instruction accesses, this solution is probably superior to the one given in the previous chapters. It eliminates all concerns of deadlock which are raised by multiple

[1] For a discussion of memory-model issues, see [8, 12, 13, 1].

primary transactions with thrashlock (Section 4.5).

6.2 Signaling with Block Multithreading

With more intrusive pipeline modifications, other solutions become possible. Rather than polling, a system could eliminate the window of vulnerability by consuming data as soon as it returns. This is the *signaling* model of context-switching. The context which requested a given piece of data can be notified or *signaled* to resume execution as soon as its data is available. At that point, the context needs to progress past the deferred load or store. In this section, we will examine signaling in the context of block-multithreaded processors, that is processors that context-switch only on cache misses or synchronization faults. We will call this *basic* signaling. Next section will consider processors which interleave the execution of multiple threads.

In a block-multithreaded processor such as Sparcle, only one thread of control is active at any one time. Thus, to permit signaling, loads or stores which are in inactive contexts must be committed in some fashion which is independent of this main thread. Consider, for a moment, a single primary (data) transaction. Signaling completely eliminates the problems which were discussed in the previous chapters:

- All loads and stores commit with a single network round-trip.
- No deadlocks are introduced by the disabling of context-switching, since data is never locked.
- High-availability interrupts do not open the window of vulnerability; loads and stores to global memory commit, even if the processor is currently executing an interrupt handler.

Consequently, signaling is attractive; it does not require thrash detection or locking.

The implementation of basic signalling requires modifications to the processor pipeline. Four mechanisms are required:

- Context management hardware which permits contexts to be put to sleep when performing a "context-switch" and awoken when their data returns. Note that this context management effects normal execution of the thread and is different from the mechanisms which commit load and store operations for inactive contexts[2].

- An extra write port in the register file, to permit returning read data to be placed directly in an appropriate register. Alternatively, an extra cycle or two can be stolen from the normal write port. A variant of "tracking vectors" are also necessary to associate primary transactions with outstanding addresses and destination registers.

- Write-buffering which can merge data from the processor with returning data from memory[3]

- Context-dependent fetch-buffers in which to place the results of global instruction fetches.

The hardware management of contexts is more complicated with signaling than with polling. The polling model of context-switching employs synchronous traps to signal context switches. This is appropriate, since faulted instructions are completely retried. Further, when a context is not the active context, its only state is in the register file (and possibly in a context-dependent status word and program counter). The signaling model, however, eliminates the retrying of instructions. Instead, contexts are allowed to advanced beyond the access on which they are waiting then disabled. As soon as data returns, the waiting access commits and the context is reenabled. Later, this context begins execution at the instruction *after* the one which just committed. This suggests a number of implementation possibilities:

[2]In the Sparcle processor, the context-switch trap code uses a special instruction, **`nextf`**, to advance the register window pointer to the next active context. The behavior of this instruction is governed by a special software mask which allows each individual context to be enabled or disabled. In a signaling system, a similar mask might be examined by hardware during context-switch operations and modified directly during the dispatching of requests and arrival of responses.

[3]Note that the decision to context-switch or not to context-switch after a write item is entered into the write-buffer depends on the memory-model which is supported. The simplest form of sequential consistency would require that a context be disabled until an outstanding write has committed.

- Complete duplication of pipeline state for each context. This is probably not a good solution, since a large amount of state may be present in various stages of the pipeline. In fact, this eliminates the advantage afforded by block multi-threading.

- Duplication of fetch buffers. This is a better solution, since at least one fetch buffer per context is necessary to permit the fetching of global instructions. Further, if extra fetch bandwidth exists, instructions from disabled contexts can be fetched in preparation for a future context switch. When a context-switch is forced on a data access, all instructions must be drained from the pipeline.

- No duplication of pipeline state other program counter(s) and status words. Here, context-switches allow the current access to advance beyond the access in question, empty the processor pipeline of instructions following this, then switch to the next context. The key is that some form of external instruction buffering for global instructions is necessary.

The second option has intriguing possibilities for reducing the cycle-penalty of a context switch, even in processors with multiple instruction issue. The last option is the simplest and behaves much like a simple RISC pipeline for polled context-switching, except for the fact that it is faulted at the instruction *after* the one which is pending.

One complication arises when signaling is combined with memory operations which have trapping behavior. For instance, certain Sparcle load and store instructions trap if the Full/Empty bit of their data has a specified value. Examples include `ldt` (load and trap on empty) and `stt` (store and trap on full). Should returning data item need to signal a fault of the corresponding load or store, this will require additional complexity to cause a trap to a disabled context (and for an instruction which has has already passed the execute stage). The simplest solution to this problem is to separate the data access from the trapping behavior. Trapping loads can be replaced by a two-instruction sequence which uses a non-trapping load followed by a test-and-trap instruction. For trapping stores, an Alpha-style load-locked/store-conditional sequence [5] is probably desirable, with a Full/Empty bit check sandwiched between two non-trapping data operations.

6.3 Signaling With Interleaved Multithreading

Signaling is much easier to implement for processors which support multiple concurrent (or interleaved) hardware threads, such as HEP [27] or Monsoon [25]. This ease stems from the fact that no extra mechanism is needed to allow accesses to commit immediately on data arrival. Such pipelines are designed around a signaling philosophy. Simply reenabling a disabled context when its data arrives is sufficient, because the thread begins to execute concurrently with other threads. However, since it may be a number of cycles before this execution commences, data must be temporarily buffered and invalidations deferred until the action commences[4]. Such processors would not suffer from the complications of faults as just discussed, since the appropriate thread of control would be reenabled to perform the actual read or write operation. This operation could commit or fault as appropriate.

Unfortunately, processors which support interleaved multithreading have disadvantages. The concurrency comes at the cost of additional hardware complexity. Further, in the examples given above, interleaved multithreading equates to poor single-thread performance, since instructions from individual threads are issued no more than once every few cycles (every 8 cycles for the examples given). This is done to simplify the task of handling pipeline hazards from multiple simultaneous threads.

Poor single-thread performance is not an intrinsic problem with interleaved multithreading, however. A more general pipeline scheduler could start on each cycle with a set of runnable threads. This set of threads could be altered by both the cache hardware and the operating system software. Instructions can be issued from this set, contingent upon availability of pipeline resources, avoidance of hazards, and other

[4]This is, in essence, a lock on the data. Since threads are executing concurrently, deadlock is not a problem. However, some care must be taken with context-switch disable mechanisms, so that they allow contexts to commit data actions which are in progress although disabled.

constraints. Assuming that sufficient bypassing has been included in the pipeline, such a processor should be able to maintain a high single-thread instruction issue rate, in addition to performing multithreading.

Chapter 7

Conclusion or "To Poll or Not To Poll"

This thesis has discussed the livelock and deadlock problems associated with the window of vulnerability and specified an architectural framework that solves those problems. A combination of multiphase memory transactions and the mechanisms associated with shared memory may be implemented using the associative thrashlock approach. If a system only needs to support a subset of the mechanisms described in Chapter 2, then Table 4.1 may be used to decide which of the other solutions are sufficient.

What is the appropriate amount of hardware required to close the window of vulnerability? As shown in Chapter 6, it is possible to construct architectures that take completely different approaches to solving the problems associated with multiphase memory transactions. For example, the Alewife architecture forces contexts to *poll* until they complete their outstanding transactions. Alternatively, a system can eliminate the window of vulnerability inherent in a polling model by *signaling* or reenabling a context immediately when its memory access completes. Such is the case in dataflow or message-passing architectures. Polling has a smaller hardware cost and optimizes for the common case when average remote access latency is shorter than polling frequency. This is true precisely when the window of vulnerability is long (Section 3.3). Signaling is less sensitive to remote access latency, but requires more than a "vanilla" RISC pipeline. If memory operation have trapping semantics, then signalling introduces additional complexity to handle delayed traps to disabled contexts. System parameters or philosophy determine whether polling, signaling, or a hybrid approach is most appropriate.

A multiprocessor could also avoid the window of vulnerability by eschewing the use of caches. In a system without caches, all memory requests could be serviced by distributed modules. By serializing transactions, memory modules would ensure both coherence and forward progress. However, such a system would have to provide extremely high bandwidth between processing nodes and memory modules in order to achieve high performance.

The associative thrashlock framework presented in this thesis provides a solution to the window of vulnerability problem for a polled, context-switching processor. The framework allows the use of caches to reduce the bandwidth required from the interconnect, and permits processors to store just enough information to recreate the pipeline state of a context when necessary. Instead of closing the window of vulnerability by brute force, the Alewife architecture dynamically detects the situations that can lead to deadlock and livelock. Only when these relatively rare situations arise does the system close the window. This fundamental architectural trade-off pits hardware expense and complexity against exceptional events that are uncommon, but potentially fatal. Further, this framework is more than just a mechanism for eliminating livelock. The explicit tracking of transactions permits the coherence protocol to be insensitive to network reordering, and provides an essential component in allowing network-overflow recovery in software. It also promises to be important for fault-tolerant shared memory.

Consequently, even if (or when) context-switching processors move beyond polling into signalling, the transaction store will remain as an important facet of shared-memory caching systems.

Bibliography

[1] Sarita V. Adve and Mark D. Hill. Weak Ordering - A New Definition. In *Proceedings 17th Annual International Symposium on Computer Architecture*, New York, June 1990.

[2] Anant Agarwal, Johnathan Babb, David Chaiken, Godfrey D'Souza, Kirk Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Gino Maa, Ken MacKenzie, Dan Nussbaum, Mike Parkin, and Donald Yeung. Sparcle: Today's Micro for Tomorrow's Multiprocessor. In *HOTCHIPS*, August 1992.

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

[4] 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, New York, June 1990.

[5] *Alpha Architecture Reference Manual.* Digital Press, 1992.

[6] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Publishing Company, Reading, MA, 1987.

[7] 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.

[8] Lucien M. Censier and Paul Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, December 1978.

[9] 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.

[10] David Lars Chaiken. Cache Coherence Protocols for Large-Scale Multiprocessors. Technical Report MIT-LCS-TM-489, Massachusetts Institute of Technology, September 1990.

[11] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[12] Michel Dubois, Christoph Scheurich, and Faye A. Briggs. Synchronization, coherence, and event ordering in multiprocessors. *IEEE Computer*, pages 9–21, February 1988.

[13] 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.

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

[15] 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.

[16] Kirk Johnson. The impact of communication locality on large-scale multiprocessor performance. In *19th International Symposium on Computer Architecture*, May 1992.

[17] 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.

[18] David Kroft. Lockup-Free Instruction Fetch/Prefetch Cache Organization. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, pages 81–87, June 1981.

[19] John Kubiatowicz. The Sparcle Processor: A Modified SPARC for Multiprocessing. ALEWIFE Memo No. 3, Laboratory for Computer Science, Massachusetts Institute of Technology. Revised February 1993, March 1990.

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

[21] John Kubiatowicz and Anant Agarwal. Anatomy of a Message in the Alewife Multiprocessor. Submitted to 1993 International Supercomputing Conference, December 1992.

[22] Kiyoshi Kurihara, David Chaiken, and Anant Agarwal. Latency Tolerance through Multithreading in Large-Scale Multiprocessors. In *Proceedings International Symposium on Shared Memory Multiprocessing*, April 1991.

[23] 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 49–58, New York, June 1990.

[24] 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.

[25] G. M. Papadopoulos and D.E. Culler. Monsoon: An Explicit Token-Store Architecture. In *Proceedings 17th Annual International Symposium on Computer Architecture*, New York, June 1990. IEEE.

[26] B.J. Smith. A Pipelined, Shared Resource MIMD Computer. In *Proceedings of the 1978 International Conference on Parallel Processing*, pages 6–8, 1978.

[27] B.J. Smith. Architecture and Applications of the HEP Multiprocessor Computer System. *SPIE*, 298:241–248, 1981.

[28] 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.