Closing the Window of Vulnerability in Multiphase Memory Transactions*

John Kubiatowicz, David Chaiken, and Anant Agarwal
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

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 paper discusses the origins of the window of vulnerability and proposes an architectural framework that closes it. The framework is implemented in Alewife, a large-scale multiprocessor being built at MIT.

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.

This paper investigates such a unifying framework, and explores one consequence, the *window of vulnerability*. Although we have implemented the complete framework in the MIT Alewife machine [1], mechanisms can be mixed and matched; other multiprocessor designers may choose to implement a subset of this framework that suits their own needs.

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 stores. This paper explores the problems involved in closing the window of vulnerability in polled, context-switching processors. While signaling leads to related approaches, a detailed discussion of these is beyond the scope of this paper.

Figure 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

*Appeared in ASPLOS V, October 1992

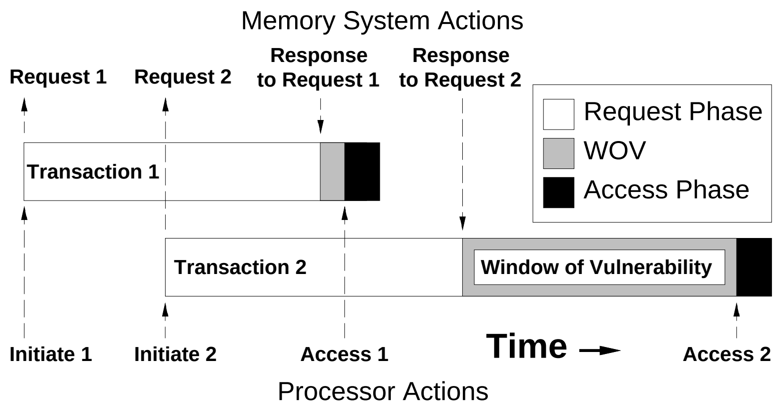

Figure 1: Multiple outstanding requests.

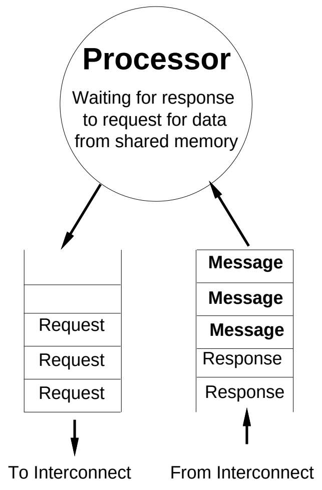

Figure 2: The need for high-availability interrupts.

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. 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.

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.

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.

Figure 2 illustrates this scenario with 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.

This paper 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 paper. 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, Section 2 presents examples of shared memory mechanisms. Section 3 then shows how the window of vulnerability can impede a system's forward progress. Section 4 explores several components of the framework, each of which provides part of the solution for ensuring forward progress. Section 4 concludes with a hybrid architecture that combines these components to implement all of the mechanisms. Section 5 describes how the issues discussed in earlier sections of this paper are reflected in the actual implementation of Alewife. The paper concludes by examining the implications of the window of vulnerability on the design of shared memory systems.

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.

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.

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 [2, 3, 4].

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 [5, 6], rapid context switching [7, 8], or weak ordering [9]. 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 [10, 11].

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

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 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.

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.

In Alewife, for example, high-availability interrupts are used to implement the LimitLESS coherence protocol [4], 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.

3 The Window of Vulnerability

To describe the window of vulnerability, we consider the memory system as a black-box that satisfies memory requests. While forward progress on the memory system side is important, it is beyond the scope of this paper. The window of vulnerability affects forward progress *after* the memory system has responded to a request. 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.

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 Primary and Secondary Transactions

The distinction between primary and secondary transactions, introduced next, incorporates non-binding prefetch into this definition of forward progress. When prefetching, a processor initiates a transaction by sending a request for data, then continues by executing its next instruction. Later, a load or store completes the transaction by accessing data (or spins/context-switches if the data is not yet available).

Thus, there are two distinct classes of transactions, *primary* and *secondary*. Primary transactions are associated with the instruction pointer of a processor thread and must complete before the thread can make forward progress. Secondary transactions, on the other hand, are associated with non-binding prefetch operations, and are not essential for the forward progress of a thread. They are, however, “upgraded” to primary status the moment a load or store attempts to access their data.

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 synchronous 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. In contrast, a processor that saves its pipeline state when context-switching (thereby saving its faulting instruction) would retry only the faulted data access. Each context in such a processor would have at most one primary transaction at a time.

Unless otherwise noted, this paper 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 An Example of a Livelock Scenario

Four distinct types of thrashing can occur during the window of vulnerability. One of these, *invalidation* thrashing, arises from protocol invalidation for highly contended memory lines and is described in detail in this section. The remaining three result from replacement in a direct-mapped cache. In *intercontext* thrashing, different contexts on the same processor can invalidate each other's data. *High-availability interrupt* thrashing occurs when interrupt handlers replace a context's data in the cache. The last, *instruction-data* thrashing, appears for processors that context-switch by polling and which must refetch load or store instructions before checking for the arrival of data.

This section describes invalidation thrashing in order to demonstrate a typical livelock scenario. Figure 3 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 4 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; 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.

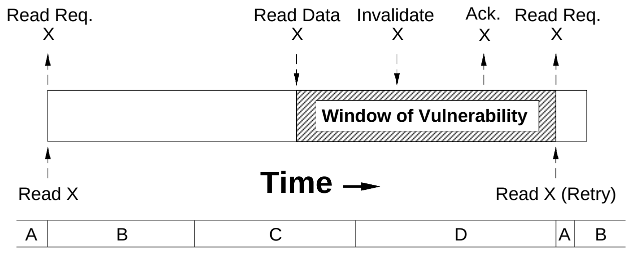

Figure 3: Time-line illustration of invalidation thrashing. The shaded area is the window of vulnerability.

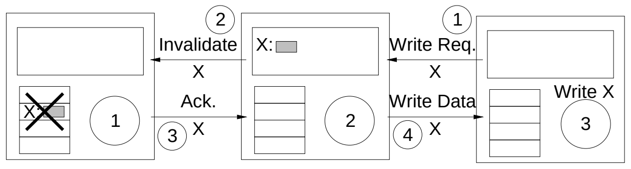

Figure 4: Diagram of cache coherence invalidation.

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).

3.3 Severity of the Window of Vulnerability

This section substantiates our claim that the window of vulnerability poses a significant problem in shared memory architectures. 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. Figure 5 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

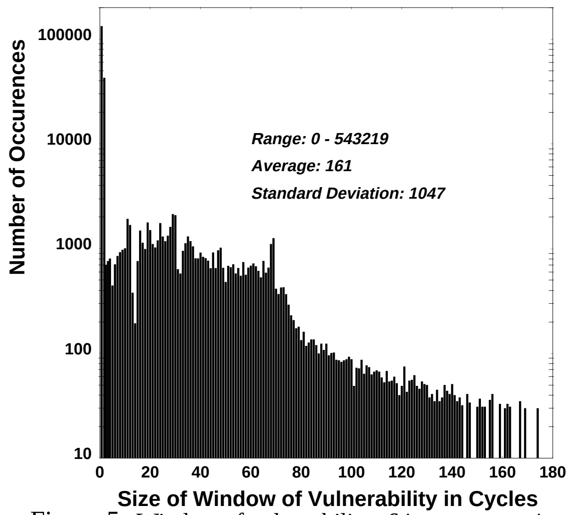

Figure 5: Window of vulnerability: 64 processors, 4 contexts.

plot the data on a logarithmic scale and to eliminate events having a frequency of less than 30 occurrences. Due to a few extremely long context run-lengths, the tail of this particular graph actually runs out to 543,219 cycles! The high standard deviation provides another measure of the importance of the graph's tail.

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 5 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*.

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

4 Closing the Window of Vulnerability

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 Section 2. This is shown in Table 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 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*.

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 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.

As described, the above scheme does not quite eliminate all intercontext thrashing, because one context can unlock the data requested by another context. We call this premature lock release.

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 2: Properties of window of vulnerability closure techniques.

This scheme 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.

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.

Problems Unfortunately, the locking mechanism can lead to four distinct types of deadlock, illustrated in Figure 6. This figure contains four different *waits-for graphs* [12], which represent dependencies between transactions. In these graphs, the large italic letters represent transactions: "*D*" for data transactions and "*I*" for instruction transactions. The superscripts – either "P" or "S" – represent primary or secondary transactions, respectively. The subscripts form a pair consisting of processor number (as an arabic number) and context number (as a letter). The address is given in parentheses; in these examples, X and Y are congruent in the cache ($X \equiv Y$), while X and Z are not equal ($X \neq Z$).

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

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

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

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

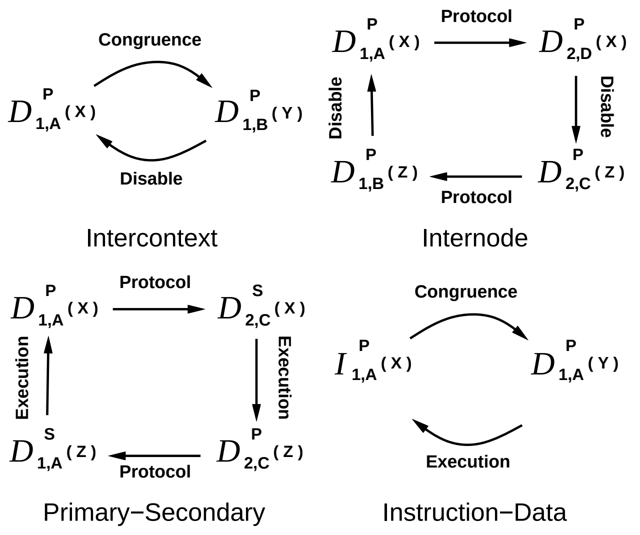

Figure 6: Deadlocks that result from pure locking. ($X \equiv Y$, $X \neq Z$)

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). The use of a transaction buffer architecture has been presented in several milieux, such as lockup-free caching [13], *victim caching* [14], and the *remote-access cache* of the DASH multiprocessor [3].

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 6, because it eliminates all of the *congruence* arcs of Figure 6.

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 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. In order 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. 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. When the following criteria are true, the memory system detects a thrashing situation:

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.

Multiple Primary Transactions Systems requiring two primary transactions can be accommodated by providing two tried-once bits, one for instructions and the other for data. To see why a single bit is not sufficient, consider an instruction-data thrashing situation with a single tried-once bit. 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.

The presence of two separate tried-once bits solves this problem. Instruction and data accesses are handled independently, according to the above algorithm. In fact, this two-bit solution can be generalized to a system with an arbitrary number of primary transactions. The only requirement for multiple transactions is that each primary transaction must have a unique tried-once bit that can be associated with it each time the context returns to begin reexecution. (This can become somewhat complex in the face of deep pipelining or multiple-issue architectures.)

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

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

Freedom From Deadlock In this section, we prove that the thrashwait algorithm does not suffer from any of the deadlocks illustrated in Figure 6. 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: *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.

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. Thus, associative thrashlock has the same transaction dependency graph as thrashwait without high-availability interrupts (as in Section 4.3). 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). When thrashing is detected, the appropriate lock is invoked. This locking scheme reintroduces a deadlock loop similar to the primary-secondary problem shown above. Fortunately, in this case the loop is rather unnatural: it corresponds to two processors, each trying to fetch as an instruction a word that is 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.

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 rather simple. The next section discusses our experiences building a system based on associative thrashlock.

5 Implementation of the Framework

The Alewife machine employs the associative thrashlock framework to close the window of vulnerability. This section overviews some of the key parameters of this implementation. For additional details, see [15].

Alewife is a large-scale multiprocessor with distributed shared memory. 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 [16], utilizing register-windows for rapid context-switching [8]. Our 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 communication 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 [4], and provides a direct message-passing interface to the underlying network.

The CMMU's *transaction store* is the heart of Alewife's implementation of associative thrashlock. The transaction store is a fully associative set of 16 *transaction buffers*. Since the Alewife machine context-switches via synchronous traps, each of the four hardware contexts can have up to two primary transactions (see Section 3.1). As required by the thrashlock scheme, the cache-controller provides eight tried-once bits (two per context) and two lock bits.

The transaction store is used as a small, fully-associative cache. All contexts access the transaction store by address, and any context may access a transaction buffer with a matching address. Figure 7 illustrates the processor-side connections to the transaction store. Each of the 16 *transaction buffers* contains an address, state bits, and space for a complete memory-line. The transaction buffers record the state of all outstanding memory transactions.

The transaction store is completely integrated with the cache-

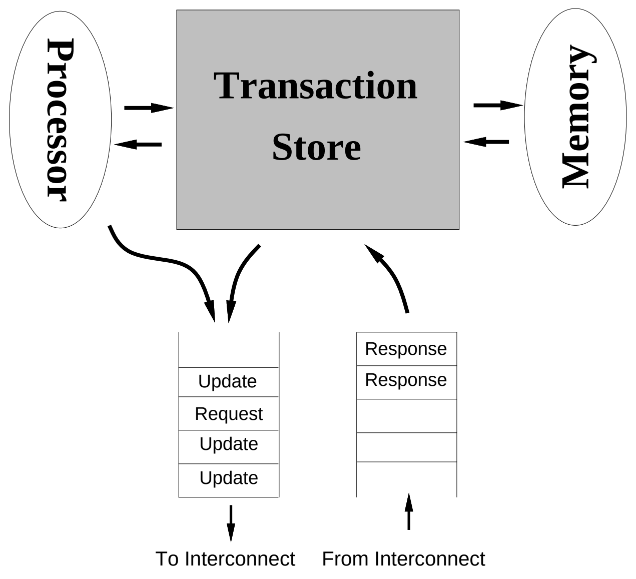

Figure 7: The transaction store.

coherence protocol; indeed, it is much like a multiprocessor victim cache[14]. Data may be transferred between transaction buffers and the cache or the processor may access transaction buffers directly. In addition, special instructions permit the processor to initiate non-binding prefetches. The transaction store has independent data paths to the processor, to memory, and to the network. A single module of associative match circuitry is shared by the processor, network and memory.

In addition to implementing the associative thrashlock framework, the transaction store has two additional benefits. First, since the transaction store explicitly records the state of outstanding transactions, it allows the Alewife cache-coherence protocol to be independent of network ordering. Relaxing the constraint of in-order delivery is desirable because it permits systems to be built with networks that employ adaptive routing to avoid hot-spots or bad connections.

Second, since context switching on Sparcle is a polling mechanism, contexts may retry memory accesses multiple times before the requested data word becomes available. The transaction store prevents redundant requests that could result from multiple retries by recording the state of all outstanding memory transactions. The same mechanism allows the requests from different contexts to the same cache line to be consolidated.

Both a register-transfer level implementation and a high-level simulation of Alewife's associative thrashlock framework are operational. Final transistor-level verification of the CMMU is currently in progress. This chip will be fabricated with a 1μ standard-cell process by LSI Logic, Inc. All of the other components of the Alewife system, including the Sparcle processor, the I/O board, and the node board have been fabricated. Sparcle and the I/O board are fully functional, while the node board awaits a finished controller for final testing.

6 Conclusion

This paper 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 this paper, then Table 1 may be used to decide which of the other two solutions are sufficient.

What is the appropriate amount of hardware required to close the window of vulnerability? It is possible to imagine 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 could 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 introduces additional hardware complexity. 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 provides a solution to the window of vulnerability problem in a polled system. 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. The fundamental architectural trade-off pits hardware expense and complexity against exceptional events that are uncommon, but potentially fatal.

7 Acknowledgments

We would like to acknowledge Pizzeria Uno for being open until 12:30am most evenings. This permitted one of us (John Kubiatowicz) to spend long hours designing the actual hardware described within. The members of the Alewife team made possible the simulation system used to study the window of vulnerability. This research is funded by NSF grant # MIP-9012773 and DARPA contract # N00014-87-K-0825.

References

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

[2] 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.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] Anant Agarwal, Beng-Hong Lim, David A. Kranz, and John Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *Proceedings 17th Annual International Symposium on Computer Architecture*, pages 104–114, Seattle, WA, June 1990.

[9] Sarita V. Adve and Mark D. Hill. Weak Ordering - A New Definition. In *Proceedings 17th Annual International Symposium on Computer Architecture*, pages 2–14, New York, June 1990.

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

[11] Kirk Johnson. The impact of communication locality on large-scale multiprocessor performance. In *19th International Symposium on Computer Architecture*, pages 392–402, May 1992.

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

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

[14] 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.

[15] 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.

[16] 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.