Midway:

Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors

Brian N. Bershad   Matthew J. Zekauskas

September 1991

CMU-CS-91-170

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This research was sponsored by The Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing", ARPA Order No. 7330, issued by DARPA/CMO under Contract MDA972-90-C-0035.

Abstract

Distributed memory multiprocessing offers a cost-effective and scalable solution for a large class of scientific and numeric applications. Unfortunately, the performance of current distributed memory programming environments suffers because the frequency of communication between processors can exceed that required to ensure a correctly functioning program.

Midway is a shared memory parallel programming system which addresses the problem of excessive communication in a distributed memory multiprocessor. *Midway* programs are written using a conventional MIMD-style programming model executing within a single globally shared memory. Local memories on each processor cache recently used data to counter the effects of network latency.

Midway is based on a new model of memory consistency called *entry consistency*. Entry consistency exploits the relationship between synchronization objects and the data which they protect. Updates to shared data are communicated between processors only when not doing so would result in an inconsistent state given a properly synchronizing program.

Midway implements its consistency protocols in software, and has no dependencies on any specific hardware characteristic other than the ability to send messages between processors. A strictly software solution is attractive because it allows us to exploit application specific information at the lowest levels of the system, and because it ensures portability across a wide range of multiprocessor architectures.

1 Introduction

Today's multiprocessors can be characterized as either *shared memory machines* or *distributed memory machines*. Shared memory machines consist of a small number of tightly coupled processors that share a single central memory. Distributed memory machines consist of a (potentially) large number of loosely coupled processors, each with their own private memory but connected by means of a fast communication network. Shared memory machines are generally recognized as being convenient to program because hardware provides processors with a *consistent* view of global memory. Unfortunately, providing this consistency limits their scalability, and therefore the practical speedup they can deliver on large-scale problems. In contrast, distributed memory multiprocessors, such as Intel's Delta Touchstone, and multicomputers based on high-speed local area networks, such as Carnegie Mellon's Nectar [Arnould et al. 89], can be scaled to hundreds, or even thousands, of processors. Each processor in these systems has a large local memory, and executes autonomously from other processors. The latency of interprocessor communication on distributed memory multiprocessors ranges from a few hundred to a few hundred thousand processor cycles. The absence of a consistent shared memory in distributed memory multiprocessors has made these machines difficult to program, however.

Midway is a system for writing and running shared memory parallel programs on distributed memory multiprocessors. It combines the programmability of a shared memory multiprocessor with the scalability of a distributed memory system. *Midway* programs are written in conventional programming languages, such as C and C++. Concurrency within a program is expressed using *threads*, and controlled using *locks* and *barriers*. Using *Midway*, for example, a programmer can easily take a parallel program written for the Sequent Symmetry, a shared memory multiprocessor, and run it on the Delta Touchstone, a distributed memory multiprocessor.

Midway provides the programmer with a new model of memory consistency called *entry consistency*. Entry consistency exploits the relationship between synchronization events and the data for which those events occur to minimize the cost and frequency of interprocessor communication. Local memories on each processor cache recently used data and synchronization variables. Communication between processors (caches) occurs only at synchronization points, and only between processors for which the synchronization implies a direct causal relationship. Further, consistency can be maintained by transferring only inconsistent data between processors; data which is unchanged is never transmitted. Consequently, *Midway* provides an execution environment in which a parallel program's performance is ultimately limited by its internal synchronization patterns, and not the underlying memory consistency protocols.

In this paper we describe *Midway* and its use of entry consistency. In Section 2 we discuss the general problem of cache consistency for multiprocessors and present more details about entry consistency. In Section 3 we present the design of *Midway*, which is currently being implemented at CMU. In Section 4 we compare *Midway* to other distributed memory multiprocessing systems. In Sections 5 and 6 we describe some of the project's future directions and summarize.

2 Cache Consistency for Distributed Shared Memory Multiprocessors

In order to get acceptable performance out of a multiprocessor system, data must be placed close to the processors which are using it. Replication, together with placement, allows data which is undergoing read sharing to be efficiently accessed by a large number of processors, since several copies of the data can be kept close to several processors. This approach, called caching, gives rise to the *cache consistency problem*, which occurs whenever one processor writes a replicated shared data item — the replicas must somehow be made consistent with the most recent version of the data if not doing so causes one processor to observe stale data.

The timeliness with which cached data must be made consistent depends on the underlying memory model assumed by the programmer, and has a tremendous influence on the performance of the memory system. Programmers often assume that memory is *sequentially consistent*. This means that the "result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each processor appear in this sequence in the order specified by its program" [Lamport 79]. In a sequentially consistent system, one processor's update to a shared data value is reflected in every other processor's memory before the updating processor is able to issue another memory access. Unfortunately, sequentially consistent memory systems preclude many optimizations such as reordering, batching, or coalescing. These optimizations reduce the performance impact of having distributed memories with non-uniform access times [Dubois et al. 86].

Memory consistency requirements can be relaxed by taking advantage of the fact that most parallel programs already define their own higher-level consistency requirements. This is done by means of explicit synchronization operations such as lock acquisition and barrier entry. These operations impose an ordering on access to data within a program. In the absence of such operations, a multithreaded program is in effect relinquishing all control over the order and atomicity of memory operations to the underlying memory system. (Even in a sequentially consistent system, only individual memory accesses are guaranteed to execute atomically. Explicit synchronization is required for complex operations.)

These observations about explicit synchronization have led to a class of *weakly consistent* protocols [Dubois et al. 86, Scheurich & Dubois 87, Adve & Hill 90, Gharachorloo et al. 90]. Such protocols distinguish between normal shared accesses and synchronization accesses. The only accesses that must execute in a sequentially consistent order are those relating to synchronization; synchronous updates to shared data in other caches are unnecessary and can be done asynchronously, and sometimes not at all.[1]

2.1 Entry Consistency

We propose a new model of weakly consistent memory called *entry consistency*. Entry consistency takes advantage of the relationship between specific synchronization variables which protect critical sections and the shared data accessed within those critical sections. In an entry consistent system, a processor's view of memory becomes consistent only when it enters a critical section. The only shared memory that is guaranteed to become consistent is that which can be accessed within the critical section. The memory model provided by entry consistency matches that already used by many shared memory parallel programs—those which use critical sections to guard access to shared data, and for which unguarded access is not expected to return anything consistent. More importantly, the model provides no greater consistency than that required by these parallel programs. Consequently, it has the potential for a higher performance implementation than a memory model which delivers to a program "more" consistency than necessary.

Entry consistency can be defined concretely in terms of a parallel program's behavior. A critical section is bounded by a pair of synchronization accesses to a synchronization variable s. An *acquire* access occurs at the beginning of a critical section, and is used to gain access to a set of shared memory locations. A *release* access occurs at the end of the critical section, and is used to signal that access is available. The

[1]In practice, synchronization accesses need only be *processor consistent* [Goodman & Woest 88], that is, writes issued from a single processor must be performed in the order issued at all processors, but writes from different processors need not be observed in the same order everywhere. The distinction between sequentially consistent and processor consistent synchronization is small, however it is easier to build a processor consistent system.

acquire and release accesses bound critical sections into which threads must gain *entry*. The synchronization variable (s) controlling access to the critical section is said to *guard* the shared data (D_s) which can be manipulated within the critical section. A memory system is entry consistent, then, if

- an *acquire* access of s is not allowed to perform with respect to processor p_i until all updates to D_s have been performed with respect to p_i. An update to a memory location is said to *perform with respect to processor* p_i at a point in time when a subsequent read of that location by p_i returns the value written by the update [Dubois et al. 86, Scheurich & Dubois 87].

This implies that no processor will read D_s older than that most recently written, provided that all accesses to data in D_s are performed within critical sections guarded by s. The *owner* of s is the processor which last acquired s, and therefore may perform updates to D_s. A synchronization variable can be owned by only one processor at a time. If a processor owns a given synchronization variable, then threads on that processor may enter and exit the associated critical sections without having to communicate updates of shared memory locations to other processors. On the other hand, if a processor which does not own s attempts to acquire it, then a message must be sent to the owning processor in order to acquire ownership of s and to transfer any updated values to D_s which are more recent than those in the acquiring processor's cache.

Data in an entry consistent system can be read-replicated by replicating synchronization variables as well as data, provided synchronization accesses are specified as exclusive or non-exclusive. Synchronization variables continue to be owned by a single processor, but may be replicated if they are accessed in non-exclusive mode. A processor must perform an exclusive access to a synchronization variable s in order to update any data in D_s. Reading D_s, though, requires only non-exclusive access through s. This gives rise to two further conditions for entry consistency:

- Before an exclusive mode access to a synchronization variable s by processor p_i is allowed to perform with respect to p_i, no other processor may hold s in non-exclusive mode.

- After an exclusive mode access to s has been performed, any processor's next non-exclusive mode access to s may not be performed until it is performed with respect to the owner of s.

These conditions ensure that exclusive-mode accesses are indeed exclusive. The final condition implies that a processor may perform a sequence of non-exclusive accesses to s without having to communicate with s's owner each time. Communication is only required when the non-exclusive accesses are interrupted by an exclusive access by another processor.

2.2 Some Benefits and Assumptions of Entry Consistency

The primary benefit of entry consistency is that interprocessor communication need only occur across exclusive mode acquire accesses. Consequently, the frequency of interprocess communication is a function of the program being run, not the cache consistency protocol. No communication is required for repeated accesses and releases of the same synchronization variable on the same processor, or for read accesses to mostly read-only data — common patterns in parallel programs [Eggers 89, Bennett et al. 90a]. The messages required to transmit new data values can be delayed until the synchronization variable guarding those values is actually needed by another processor. Finally, the messages that are necessary to sequentially order accesses to synchronization variables can also be used to transmit updates to the guarded data.

Entry consistency makes a number of assumptions about the behavior of parallel programs and the runtime environment. First, as an instance of a weakly consistent protocol, entry consistency requires that synchronization accesses can be distinguished from other accesses. Second, entry consistency requires that exclusive synchronization accesses can be distinguished from non-exclusive accesses. Third, entry consistency requires an association between shared data and its guarding synchronization variable. Fourth, entry consistency requires that it be possible to determine the shared data which has been updated with respect to a processor that is acquiring a synchronization variable. In Section 3 we describe an implementation strategy which satisfies these assumptions.

2.3 A Family of Consistency Protocols

Entry consistency is one of a family of memory consistency models. Figure 1 illustrates how entry consistency relates to other members of the consistency family with respect to strictness. With sequential consistency, all shared accesses must be ordered the same on all processors. Processor consistency [Goodman 89] allows writes from different processors to be observed in different orders, although writes from a single processor must be performed in the order that they occurred. Explicit synchronization operations must be used for accesses that should be globally ordered. The primary benefit of processor consistency is that it allows a processor's reads to bypass its writes. Weak consistency [Dubois et al. 86] treats shared data accesses separately from synchronization accesses, but requires that all previous data accesses be performed before a synchronization access is allowed to perform. This allows loads and stores between synchronization accesses to be reordered freely.

most strict

sequential consistency	order of every shared access important
processor consistency	writes from different processors not ordered explicit synchronization for event ordering
weak consistency	synchronization access vs. data access
release consistency	synchronization: acquire vs. release
entry consistency	data tied to synchronization variable synchronization: exclusive vs. non-exclusive

less strict

Figure 1: Spectrum of Consistency

Release consistency [Gharachorloo et al. 90] is most closely related to entry consistency. Release consistency ensures that *all* previous shared data updates are consistent (performed) before a release of a synchronization variable is observed by *any* processor. The conditions for release consistency are:

1. Before an access to a shared data item is allowed to perform with respect to any other processor, all previous acquire accesses must be performed, and

2. Before a release access is allowed to perform with respect to another processor, all previous accesses to shared data must be performed, and

3. Synchronization accesses are processor consistent with respect to one another.

Like entry consistency, release consistency distinguishes between acquire and release accesses. Unlike entry consistency, release consistency does not associate the data guarded by the synchronization variable with the synchronization variable, and does not distinguish between exclusive and non-exclusive mode accesses.

As a result of these differences, two ordering restrictions are present in release consistency, but are not present in entry consistency. First, release consistency requires that all previous accesses by a processor for synchronization variables be performed with respect to all other processors before that processor's shared access is observed by any other processor. This includes accesses to synchronization variables which do not guard the data being accessed. In contrast, entry consistency only requires that the acquire access for the synchronization variable which guards the data being accessed be performed. An acquire for a synchronization variable could be pending for the entire time another critical section is executed. This allows the programmer and the scheduler to place multiple threads of control on each processor, and to overlap computation with communication by context switching whenever a thread has to wait for a remote acquire to be performed. Stated another way, entry consistency only requires that synchronization accesses be *thread consistent*; a thread's acquire and release accesses must be performed in the order that they were issued by the thread. In contrast, release consistency requires that synchronization operations be processor consistent.

Second, release consistency requires that all updates to *any* shared data must be performed before a release is performed. This includes data not guarded by the just released synchronization variable. Since these remote accesses occur before the synchronization variable is released, the length of time required to execute a critical section is increased. In contrast, entry consistency only requires accesses guarded by the synchronization variable being released be performed remotely, and then only when the variable is next acquired by a remote processor. As a result, only relevant data is transferred, and the transfer occurs entirely *outside* the critical section.

Figure 2 illustrates the difference between the two consistency models for the simple case of a processor updating data guarded by a lock which is held in another processor's cache. In the figure, time proceeds downwards, and diagonal arrows indicate messages to remote memories. There are two key points to note from the figure. First, with entry consistency, updates arrive at the acquiring processor with the lock; there are no "cache misses" for shared data within the critical section. Second, with entry consistency, the lock can be released without having to communicate with remote memories, whereas with release consistency, the lock must remain held until the shared accesses have been performed in remote memories. Both of these characteristics serve to reduce the amount of communication because of shared accesses, and to reduce the length of time to execute a critical section.

3 *Midway*—Programming Support for Entry Consistency

Midway is a programming and runtime environment which provides entry consistency semantics for shared memory parallel programs running on distributed memory machines. *Midway* consists of three main components: a set of simple language extensions in the form of keywords and function calls that are used to annotate a parallel program, a runtime system which implements entry consistency, and a compiler which generates code that maintains reference information for shared data. A sample *Midway* environment is illustrated in Figure 3. Parallel programs, written in C, C++, or ML, are processed by the *Midway* compiler and then linked in with the runtime library. The runtime library provides a portable, machine-independent interface to entry consistent shared memory across a wide range of processor and interconnect architectures. For example, a *Midway* program could be written and debugged on the Sequent, a true shared

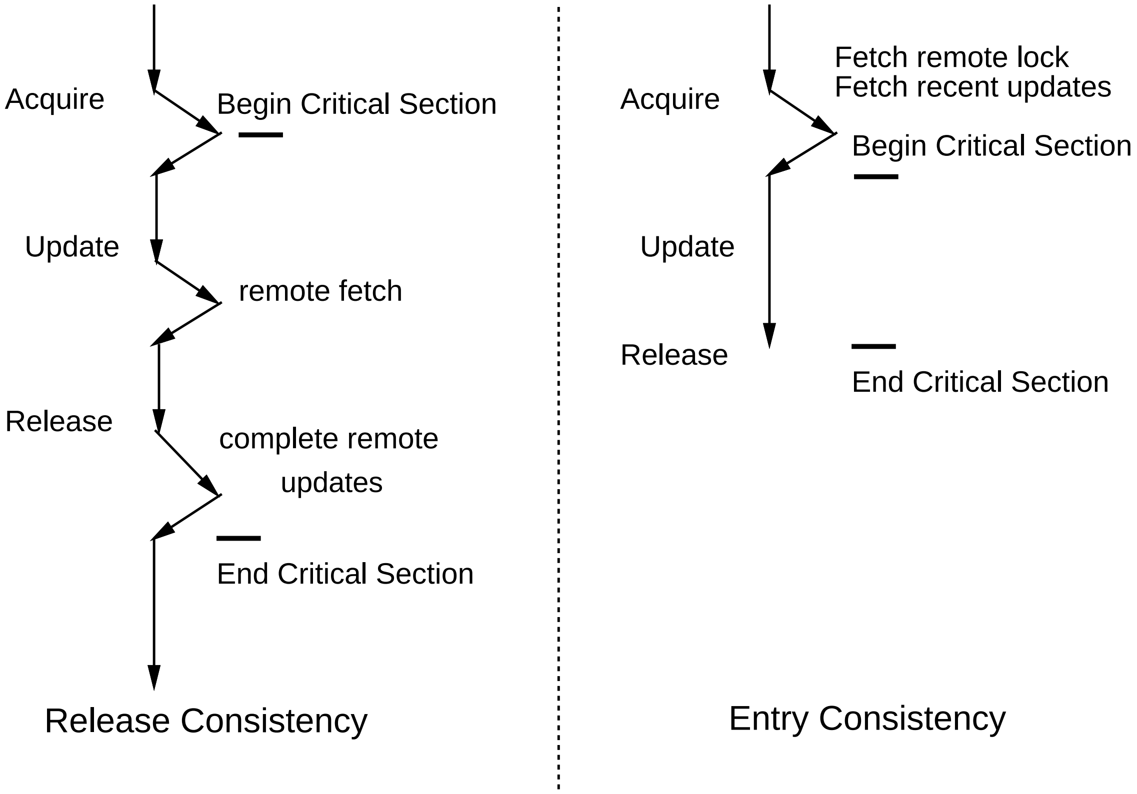

Figure 2: Behavior of Two Weakly Consistent Memory Models

memory multiprocessor, and then compiled and relinked for execution on a Delta Touchstone distributed memory multiprocessor. The remainder of this section describes the structure and behavior of the runtime environment and compiler. Our initial implementation runs on a network of MIPS-based workstations connected by ethernet. Subsequent ports will handle faster networks and more specialized distributed memory multiprocessors. The performance of the system will be described in a future paper.

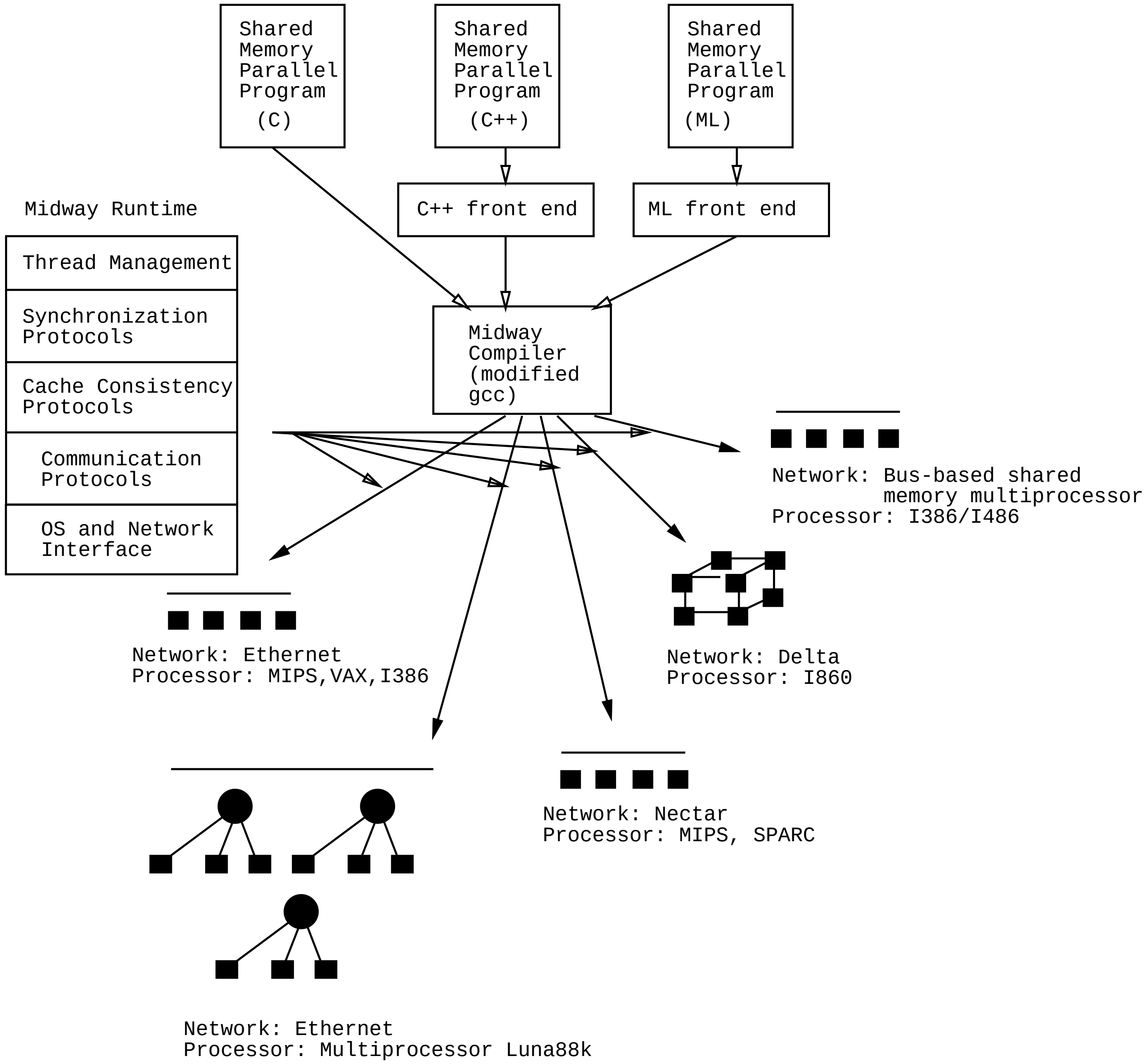

Figure 3: Midway Programming and Execution Environment

3.1 Programming Language Support for *Midway*

Midway's concurrency primitives are based on the Mach C-Threads interface [Cooper & Draves 88]. A *Midway* program written in C looks just like many other parallel C programs in that it uses thread management directives such as *fork* and *join*, and synchronization primitives such as *lock* and *unlock*. The choice of C as the base language was made because C has become a "portable assembly language" into which a large number of higher-level languages such as C++ [Stroustrup 86], Modula-3 [Cardelli et al. 88], ML [Tarditi et al. 90] and Scheme [Bartlett 89] can be compiled. Once the C version of *Midway* is stable, we will extend the environment to support C++, an object-oriented language with a good track record for parallel programming [Bershad et al. 88], and ML, a "mostly functional" programming language with a growing user and application base.

A *Midway* program must include some extra information so that the compiler and runtime system can enforce entry consistency. Specifically,

- All synchronization variables must be explicitly declared as *synchronization* data types. The only allowable synchronization constructs are those based on these variables.
- All shared data must be explicitly declared as instances of shared datatypes using a new keyword `shared` which is understood by the compiler. The C `typedef` construct can be used to define shared versions of non-shared datatypes.
- All shared data must be explicitly associated with at least one synchronization variable which guards access to the data. This association is dynamic, and is made by calls to the runtime system.

Programs which include this extra information and which execute according to the following constraints will observe consistent shared memory:

- Any thread's access to shared data must be dynamically enclosed within an acquire and release of a synchronization variable guarding the data.
- A thread must acquire a synchronization variable in exclusive mode if it intends to modify any of the data guarded by that synchronization variable prior to the next release.

In contrast, programs which rely on implicit synchronization or on updates to unguarded shared data being reflected to other processors may observe inconsistencies in the memory.

We do not anticipate that the annotation rules and usage constraints will present a major burden for the parallel programmer. Thread management libraries already provide primitives for synchronization; it is rare for programmers to "roll their own." Except for certain asynchronous algorithms, access to shared data must be synchronized, and there exists already an implicit relationship within a program between specific synchronization variables and specific shared data. *Midway* requires only that this implicit relationship be made explicit.[2]

[2] A primary motivation for supporting ML is that ML has a rich type and module system which will allow us to automatically extract the relationship between synchronization objects and shared mutable data [Cooper & Morrisett 90]. A distributed memory parallel ML program will need no additional annotations beyond those already required for it to typecheck correctly.

3.2 Synchronization Management

Midway's runtime system performs two separate functions: distributed synchronization management and distributed cache management. Distributed synchronization management enables processors to acquire synchronization variables which are not presently held in their cache. Distributed cache management ensures that a processor never enters a critical section without having first received all updates to shared data guarded by the synchronization variable controlling access to that section. Two types of synchronization variables, locks and barriers, are supported.

Locks

Locks are acquired in either non-exclusive or exclusive modes. *Midway* uses a distributed queueing algorithm similar to the ones described in [Forin et al. 89, Lee & Ramachandran 90] to locate the owner of a synchronization variable. Each processor maintains a "best guess" as to the location of each synchronization variable's owner. When a lock's owner changes from p_i to p_j, both processors update their best guess to p_j, guaranteeing that a chain of best guesses eventually terminates at the owner. To acquire a lock s, a processor p_i sends a request for s to the processor named by its best guess, p_j. If the best guess is correct, the request is either granted or queued. If the guess is incorrect, p_j may itself have an outstanding request for the lock pending on another processor. If so, the request is queued at p_j, and p_j becomes responsible for satisfying the request after its own request has been satisfied. If p_j is not already waiting for the lock, it forwards p_i's request to its best guess and the algorithm repeats itself. Whenever ownership of s transfers from p_i to p_j, the queue of outstanding requests for s is transferred as well.

To acquire a lock s in non-exclusive mode, any other processor, p_j, holding the lock is found using the best guess strategy. If p_j is the owner, the request is queued until s is not held in exclusive mode. Then processor p_j grants s to p_i, marks s as non-exclusive, and inserts p_i into the *invalidate set* for s. The invalidate set is used to manage exclusive mode requests and is described below. If p_j is not the owner, but holds s in non-exclusive mode, then p_j grants s to p_i and inserts p_i into its version of s's invalidate set. At p_j's convenience, it notifies the owner of s that p_j has joined the invalidate set for s. Once a processor has acquired s in non-exclusive mode, it may perform subsequent non-exclusive mode accesses without interprocessor communication until an intervening exclusive mode access occurs.

To acquire s in exclusive mode, the owner, p_j, must first be located using the distributed queuing algorithm. If the lock is held by p_j in exclusive mode, then the request is queued. If the lock is held by p_j in non-exclusive mode, then an invalidate message is sent to each processor in p_j's version of s's invalidate set (each processor that holds s in non-exclusive mode) and the request is queued at p_j until the last reader finishes (thereby overlapping the invalidation with the lock transfer). Processor p_i's version of s's invalidate set is sent along with each invalidation message. If p_k, which holds s in non-exclusive mode, gets an invalidate message for s, it checks to see that its version of s's invalidate set is a subset of the invalidate set contained in the invalidate message. If not, it forwards copies of the invalidate message to the processors that were missing from the message's invalidate set. Processor p_k then replies to the owner with its invalidate set. Once the new owner has received a reply from each processor in the transitive closure of the invalidate set of the previous owner of s, the thread that had initiated the exclusive mode request can proceed.

Barriers

Barriers permit pseudo SIMD-style computation by synchronizing multiple threads across sequential phases of a computation. When a thread reaches a barrier, it delays until all other threads reach that same barrier. Shared data accessed within a barrier must be made consistent only at the points where the barrier computation proceeds from one phase to the next. Within a phase, there are no consistency guarantees for data updated during that phase (unless other synchronization primitives are used), so threads may assume only that data from previous phases has reached a consistent state.

Midway associates a manager processor with each barrier synchronization variable. Processors "cross" the barrier by sending a message to the manager and waiting for a reply. The message contains the barrier name and all updates to shared data accessed by the sending processor during the current phase. The manager coalesces the updated values it receives from all processors participating in the barrier computation. The manager releases the processors by sending the coalesced updates back to each participating processor. Figure 4 illustrates a two processor barrier, with the manager running on a third processor. Processor p_1 has modified variable d_1 and p_2 has modified d_2. The manager combines these updates and feeds them back to the two participating processors, which can then reintegrate the updates into their caches.

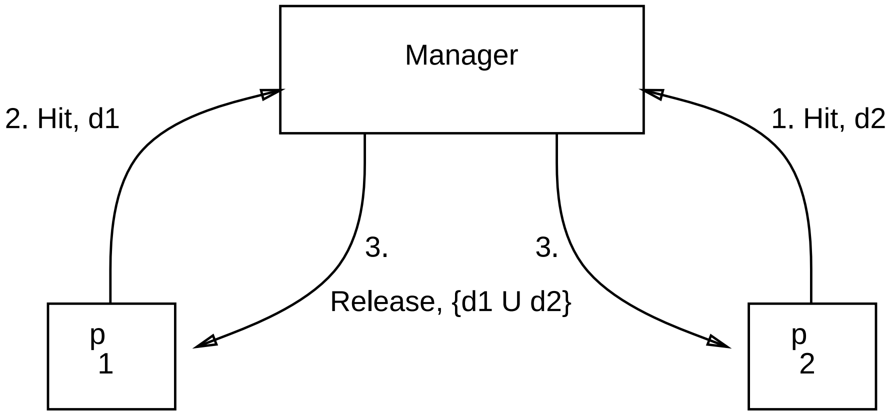

Figure 4: A Two Processor Barrier

3.3 Cache Management

Access to a synchronization variable and its guarded data must be managed so that updates are performed before the synchronization access is allowed to do so. While this condition could be satisfied by transferring all shared data guarded by a synchronization variable, entry consistency requires only that updated data more recent than that contained in an acquiring processor's cache be transferred. To determine which updates are more recent than others, *Midway* imposes a partial order on updates to shared data with respect to synchronization accesses.

Two events A and B can be related by a *happens-before* relation if and only if there exists an intervening synchronization operation between them [Lamport 78]. If A and B occur on the same processor, then that intervening operation is the synchronizing clock tick which conceptually occurs between every issued instruction. If, on the other hand, A and B occur on different processors, p_i and p_j respectively, then A happens before B if and only if A happens before a message is sent from p_i which causes a message to arrive at p_j before event B happens. If there is no such message, then A and B cannot be ordered according to the happens-before relation and are said to occur simultaneously. *Midway* uses the happens-before relationship to impose a partial ordering on shared data updates as follows:

1. Each processor p_i maintains a monotonically non-increasing counter c_i that serves as its local clock.
2. Whenever p_i sends a message to p_j, p_i increments c_i and includes c_i in the message. Upon receipt of the message, p_j sets its c_j to $max(c_j, c_i)$.
3. Each synchronization variable s has an associated timestamp t_s that is set to the value of c_j whenever ownership transfers to another processor p_j.
4. Each shared data value v has an associated timestamp t_v that is set to the value of t_s, where s guards v, when v is updated.

When processor p_i requests s from p_j, the request contains p_i's last value of t_s, t_{s_i}, indicating the "time" that p_i last held s. For each shared value v guarded by s, if $t_v > t_{s_i}$, then p_i's cache has a stale version of v, and p_j must transfer the new value of v with s. Processor p_i performs the update to its own copy of v before allowing the acquire access of s to complete.

This protocol ensures that those shared data items which are inconsistent are made consistent before the inconsistency could be observed. Further, a cached data value that has not been updated since an acquiring processor last held a synchronization variable is not needlessly sent between processors. Because all updates are transmitted at critical section entry time, critical section path lengths are not inflated due to cache misses or invalidates.

Midway does not assume that processors have infinite caches. At any point, a processor may discard a shared data item as long as that processor does not presently own the synchronization variable guarding the shared value. When next acquiring the guarding synchronization variable s, the discarding processor must only indicate that it has not held s for a "very long time" by using a zero timestamp value. This ensures that the current holder will provide any values that have been discarded.

3.4 Memory Management

Midway's consistency machinery relies on timestamps maintained by software to keep track of which shared data items have been modified. Associated with every shared data item is at least one timestamp (t_v). When a shared data value is allocated, the granularity of the timestamp (in effect, the *cache line size*) can be selected according to the expected access patterns of the item. A large contiguous object may be backed by many timestamps.

Timestamps are arranged in memory so that the runtime system can quickly convert from a shared item's address to its timestamp. On writing a shared data item, the address of the timestamp line is determined. The shared data item's timestamp field is then set to zero to indicate that it has been *modified*. When a synchronization variable s is requested of p_i, all data guarded by s whose timestamp is zero will have their timestamp set to c_i. In this way, *Midway* avoids computing a timestamp for each update, delaying that action until the timestamp is needed by the synchronization protocol.

3.5 Compiler Support for *Midway*

Midway requires a small amount of compile-time support. An update to a shared data item must also update the timestamp associated with that item. *Midway* uses a modified version of `gcc`, a public-domain C compiler, to inline timestamp updates on writes to shared data. Other information necessary to implement entry consistency, such as the association between synchronization variables and guarded data, is declared at runtime by means of procedure calls into *Midway*'s runtime system.

4 Related Work

This section reviews several other projects that support shared memory parallel programming on distributed memory multiprocessors.

4.1 Non-Replicated Object-Based Approaches

The cache consistency problem can be solved by never allowing a data item to reside in more than one cache at a time. Instead, modifiable data migrates on-demand from one processor to another. Several software-based systems that rely on this policy have been built [Jul et al. 88, Chase et al. 89, Almes et al. 85]. While these systems avoid the problems of caching (maintaining consistency), they also fail to reap the benefits (reduced communication costs, higher performance). Programs may cache data explicitly by creating purely local copies, but this approach denies, rather than addresses, the problems of cache consistency. Cheriton [Cheriton 88] argues that such *problem-oriented shared memory* permits an application to choose a consistency mechanism that is most appropriate for the problem being solved. This approach, though, fails to serve the needs of parallel programs that have similar, well-defined consistency requirements, forcing each to implement its own solution.

4.2 Page-Based Approaches

Page-based distributed shared memory systems maintain sequential consistency through the use of conventional memory management hardware together with sophisticated virtual memory techniques. Data that has been read-replicated on many processors can only become inconsistent as a result of one processor writing the data. Using the protection mechanisms provided by memory management system, a replicated page can be marked read-only so that any thread attempting to write the data will fault. The fault-handler detects that the faulted page is replicated, invalidates all replicas, makes the page writable and resumes the faulting thread. When another processor attempts to access the (now invalid) page, it will fault and its fault-handler will copy the page from another processor's memory. The key advantage of page-based systems is that they allow any program, written in any language, to run correctly on a distributed shared memory system. Li's Ivy system [Li 86] described the first implementation of page-based distributed shared memory, and there have been several subsequent implementations [Fleisch 87, Forin et al. 89].

Page-based systems have several drawbacks that limit their performance. First, when different data items on the same page are referenced by different processors, the invalidation protocol causes that page to migrate between processors. This is the same interference phenomenon that occurs with hardware-based invalidation protocols, and is why such protocols rely on relatively small cache lines. Second, in page-based systems processors interact whenever an inconsistency *occurs*, rather than when one could be detected. Finally, cache misses and invalidates that occur during a critical section serve to inflate the amount of time it takes to get through the critical section, possibly increasing contention. Virtual memory systems, and their underlying MMU hardware, do not have particularly fast fault handling times [Appel & Li 91]. Additionally, handling an invalidation or miss during a critical section with a remote message adds several milliseconds to what might otherwise be a low-latency critical section.

4.3 A Replicated Object-Based Approach

Clouds [Ramachandran & Khalidi 88] is an object-oriented distributed memory system that extends Li's page-based approach in three ways. First, the unit of coherence is the *segment* which corresponds to an object, rather than the page (which corresponds to no programming abstraction). Second, the memory mapping hardware is integrated with the distributed memory system so that object faults, fetches and invalidates can be handled without operating system involvement. Finally, Clouds permits the memory system to be influenced by synchronization operations [Ramachandran et al. 88], so that a series of operations can be performed on a segment before activating the consistency protocols in a manner similar to protocols that implement release consistency.

4.4 Hardware Support for Release Consistency

Stanford's DASH multiprocessor [Lenoski et al. 90] provides release consistency, guaranteeing that memory modified within a critical section becomes consistent before the end of the critical section. As mentioned earlier, the guarantees of release consistency are stronger than necessary: updates are communicated to processors as they occur, rather than as they are needed.

The DASH hardware provides very fast remote memory and synchronization accesses (on the order of 80 processor cycles) over an extremely high-bandwidth interconnect. At this level of performance, the added flexibility provided by entry consistency is unlikely to yield a significant performance improvement. On systems with higher access latencies (thousands of cycles), we expect that the relaxed implementation enabled by entry consistency will have an effect.

Finally, it's worth noting that it would be difficult to implement entry consistency strictly in hardware because of the need to associate arbitrary data structures and specific synchronization variables. One area where hardware support might be useful is in the maintenance of dirty bit information. Hardware caches already keep track of which cache lines have been dirtied. A natural extension to this function would be to have the hardware keep track of not only which data was dirty, but the local Lamport time at which it was last written. In this way, the additional overhead of computing and maintaining dirty bit information for each update to a shared location could be eliminated by (or at least shifted to) the hardware. We intend to explore this possibility as the system develops.

4.5 Software Supported Release Consistency

Munin [Bennett et al. 90b] is a software system for running shared memory parallel C++ programs on distributed memory multiprocessors. Munin supports automatic data caching through the use of six type-specific consistency protocols. Munin's runtime system uses hints from the programmer to determine the access patterns of shared data items, and to select the best consistency protocol for each data object. Munin implements release consistency with *delayed update queues*. Writes to shared data within a critical section are detected by means of a page fault (as with page based approaches). The page is duplicated, possibly after obtaining the most recent data from another processor, and the faulting instruction is restarted. Further writes to shared data within that page proceed without faulting. At lock release, modified data is detected by comparing dirtied pages with the duplicates of the original page. This modified data is then flushed to other processors with either an invalidation or an update (depending on the object's protocol).

Although *Midway* and Munin have similar philosophies (delay expensive network communication), they differ in three main ways: the consistency model they support, the number of consistency protocols used to implement the model, and implementation strategy.

The differences related to the consistency model affect what must be transferred and when. On release, Munin must transfer all updates since the last release. This is true even if the Munin programmer associates data with the synchronization variable and can be a problem when nested locks are used, or when multiple threads on one processor hold different locks. Munin must communicate during a release (while still holding the lock) to bring shared data values up to date. In contrast, no communication is done in *Midway* until another processor requests the lock being released, and only data associated with the requested lock is transferred.

An important difference between the two systems is in the number of data sharing types that each defines. Munin allows the programmer to choose from a suite of six sharing modes, thereby controlling the communication protocol by which consistency is being provided. The proper choice of mode can reduce access latency because the mode provides information about the expected access pattern for an object. If the wrong mode is chosen, though, the overhead could be greater than necessary, and may even fail to provide consistency. *Midway* takes a "less is more" approach here, supporting only one shared datatype, making it hard for the programmer to choose the wrong one. Further, in an environment of slow networks, fast processors and expensive messages, it is unclear whether multiple protocols offer much of an advantage for programs in which the frequency of communication can be directly tied to the synchronization patterns of the program.

Other differences between the systems relate to implementation and not philosophy. Munin detects updates by protecting pages that contain shared data. A shared memory access takes a page fault and the page is copied, possibly after contacting other processors to get the latest data for that page. *Midway* sets dirty bits when shared data is written; page fault overhead is avoided and no communication with other processors is necessary. Munin's approach increases the time spent in the critical section due to page fault, copy, and remote access overhead. *Midway* still has the remote access overhead, but it is incurred outside the critical section. Writes to shared data in *Midway* do have an overhead (a few cycles on the MIPS processor, for example), but that overhead is low relative to the time to handle a page fault.

5 Future Work

Midway will be both a system for writing parallel programs, and a laboratory with which to perform further research in distributed memory parallel programming and systems design. Among the research problems which we intend to address with *Midway* are:

- *Distributed Memory Scheduling.* In the initial system, whenever a thread tries to acquire a synchronization variable s held by another processor's cache, s and any updates to shared data guarded by the s are moved to the acquiring thread's processor. If there is high interprocessor contention for a synchronization variable, then it can be more efficient to transfer the thread to the data and execute the critical section on one processor.

- *Distributed Thread Scheduling.* A desirable property of any multiprocessor is that it be *work conserving,* that is, no processor idles while there is work to do. We expect it will be difficult to guarantee a work conserving distributed shared memory system while keeping the overhead of scheduling low. Nevertheless, we intend to explore hierarchical scheduling algorithms together with hints about processor utilization (similar to the hints used to locate synchronization variables) to keep processors busy.

- *Detection of Access Anomalies.* A side-benefit of *Midway*'s explicit annotation is that it enables certain types of parallel programming errors to be caught easily at compile-time or run-time. For example, it is possible to determine that a shared data value is not guarded by any synchronization variable, or that a shared variable is accessed *anomalously*, that is, by two or more processors without an intervening synchronization operation on the variable's guarding synchronization variable. In this way, the frequency and severity of programming errors can be reduced.

- *Heterogeneity.* By passing the type information of shared data items from the *Midway* pre-compiler onto the runtime system, shared data can be transferred between unlike processor architectures using the data marshalling techniques found in heterogeneous RPC systems [Bershad et al. 87].

- *Fault Tolerance.* An important problem in a loosely-coupled distributed memory system is that processors may fail during a long running computation. Without recovery mechanisms, the failure of a single processor can cause an entire computation to fail. *Midway* will provide an ideal distributed memory environment in which to explore failure recovery mechanisms; the interaction between processors is both minimized and completely contained within cache consistency messages.

- *Operating System Support for Entry Consistency.* We expect that part of the consistency protocol could be made the responsibility of the operating system, thereby reducing communication latency. In particular, the message forwarding component of the distributed queueing algorithm could be implemented transparently in the operating system, much as today's systems transparently forward IP packets.

- *Hardware Support for Entry Consistency.* We hope to investigate the scope and expense of hardware mechanisms that assist in the implementation of entry consistency. For example, keeping track of which shared data items have been written on a given processor could easily be done in hardware (e.g. fine-grained dirty bits), rather than having to execute extra code during updates to shared data.

6 Summary

Midway is intended to be a portable software environment which enables efficient shared memory parallel programming on a range of distributed memory multiprocessors. It provides a weak form of memory consistency which uses fine-grained information about synchronization behavior to reduce the cost of maintaining shared memory. In this way, the amount of communication in a parallel program is governed by the program's synchronization patterns, and not the underlying consistency protocols.

References

[Adve & Hill 90] Adve, S. V. and Hill, M. D. Weak Ordering – A New Definition. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 2–14, May 1990.

[Almes et al. 85] Almes, G. T., Black, A. P., Lazowska, E. D., and Noe, J. D. The Eden System: A Technical Review. *IEEE Transactions on Software Engineering*, SE-11(1):43–59, January 1985.

[Appel & Li 91] Appel, A. W. and Li, K. Virtual Memory Primitives for User Programs. In *Proceedings of the 4th ACM Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, April 1991.

[Arnould et al. 89] Arnould, E., Bitz, F., Cooper, E., Kung, H., Sansom, R., and Steenkiste, P. The Design of Nectar: A Network Backplane for Heterogeneous Multicomputers. In *Proceedings of the 3rd ACM Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989.

[Bartlett 89] Bartlett, J. Scheme $->$ C, a Portable Scheme-to-C Compiler. Technical Report 89/1, Digital Equipment Corporation Western Research Laboratory, Palo Alto, California, January 1989.

[Bennett et al. 90a] Bennett, J. K., Carter, J. B., and Zwaenepoel, W. Adaptive Software Cache Management for Distributed Shared Memory Architectures. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 125–135, May 1990.

[Bennett et al. 90b] Bennett, J. K., Carter, J. B., and Zwaenepoel, W. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. In *Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 168–176, March 1990.

[Bershad et al. 87] Bershad, B. N., Ching, D. T., Lazowska, E. D., Sanislo, J., and Schwartz, M. A Remote Procedure Call Facility for Interconnecting Heterogeneous Computer Systems. *IEEE Transactions on Software Engineering*, SE-13(8):880–894, August 1987.

[Bershad et al. 88] Bershad, B. N., Lazowska, E. D., and Levy, H. M. PRESTO: A System for Object-Oriented Parallel Programming. *Software: Practice and Experience*, 18(8):713–732, August 1988.

[Cardelli et al. 88] Cardelli, L., Donahue, J., Glassman, L., Jordan, M., Kalsow, B., and Nelson, G. Modula-3 Report. Technical Report #31, Digital Equipment Corporation's Systems Research Center, Palo Alto, California, August 1988.

[Chase et al. 89] Chase, J. S., Amador, F. G., Lazowska, E. D., Levy, H. M., and Littlefield, R. J. The Amber System: Parallel Programming on a Network of Multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 147–158, December 1989.

[Cheriton 88] Cheriton, D. Problem-Oriented Shared Memory: A Decentralized Approach to Distributed System Design. In *Proceedings of the Sixth International Conference on Distributed Computing Systems*, pages 162–169, June 1988.

[Cooper & Draves 88] Cooper, E. C. and Draves, R. P. C Threads. Technical Report CMU-CS-88-54, Department of Computer Science, Carnegie-Mellon University, February 1988.

[Cooper & Morrisett 90] Cooper, E. and Morrisett, J. Adding Threads To ML. Technical Report CMU-CS-90-186, Department of Computer Science, Carnegie-Mellon University, December 1990.

[Dubois et al. 86] Dubois, M., Scheurich, C., and Briggs, F. Memory Access Buffering in Multiprocessors. In *Proceedings of the 13th Annual Symposium on Computer Architecture*, pages 434–442, June 1986.

[Eggers 89] Eggers, S. J. *Simulation Analysis of Data Sharing in Shared Memory Multiprocessors*. PhD dissertation, University of California, Berkeley, March 1989.

[Fleisch 87] Fleisch, B. D. Shared Memory in a Loosely Coupled Distributed System. In *Proceedings of the SIGCOMM87 Workshop on Frontiers in Computer Communications Technology*, August 1987.

[Forin et al. 89] Forin, A., Barrera, J., Young, M., and Rashid, R. Design, Implementation and Performance Evaluation of a Distributed Shared Memory Server for Mach. In *1988 Winter Usenix*, January 1989.

[Gharachorloo et al. 90] Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., and Hennessy, J. Memory Consistency and Event Ordering in Scalable Shared Memory Multiprocessors. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 15–26, May 1990.

[Goodman & Woest 88] Goodman, J. and Woest, P. The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor. In *Proceedings of the 15th Annual Symposium on Computer Architecture*, pages 422–431, Honolulu, Hawaii, June 1988.

[Goodman 89] Goodman, J. R. Cache Consistency and Sequential Consistency. Technical Report 61, SCI Committee, March 1989.

[Jul et al. 88] Jul, E., Levy, H., Hutchinson, N., and Black, A. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[Lamport 78] Lamport, L. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[Lamport 79] Lamport, L. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):241–248, September 1979.

[Lee & Ramachandran 90] Lee, J. and Ramachandran, U. Synchronization with Multiprocessor Caches. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 27–37, May 1990.

[Lenoski et al. 90] Lenoski, D., Laudon, J., Gharachorloo, K., Gupta, A., and Hennessy, J. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 148–159, May 1990.

[Li 86] Li, K. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD dissertation, Department of Computer Science, Yale University, September 1986.

[Ramachandran & Khalidi 88] Ramachandran, U. and Khalidi, M. An Implementation of Distributed Shared Memory. Technical Report GIT-ICS-88/50, School of Information and Computer Science, Georgia Institute of Technology, December 1988.

[Ramachandran et al. 88] Ramachandran, U., Ahamad, M., and Khalidi, M. Unifying Synchronization and Data Transfer in Maintaining Coherence in Distributed Shared Memory. Technical Report GIT-ICS-88/23, School of Information and Computer Science, Georgia Institute of Technology, June 1988.

[Scheurich & Dubois 87] Scheurich, C. and Dubois, M. Correct Memory Operation of Cache-Based Multiprocessors. In *Proceedings of the 14th Annual Symposium on Computer Architecture*, pages 234–243, June 1987.

[Stroustrup 86] Stroustrup, B. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 1986.

[Tarditi et al. 90] Tarditi, D., Acharya, A., and Lee, P. No Assembly Required: Compiling Standard ML to C. Technical Report CMU-CS-90-187, Department of Computer Science, Carnegie-Mellon University, November 1990.