The Midway Distributed Shared Memory System

Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213

Abstract

This paper describes the motivation, design and performance of Midway, a programming system for a distributed shared memory multicomputer (DSM) such as an ATM-based cluster, a CM-5, or a Paragon. Midway supports a new memory consistency model called entry consistency. Entry consistency guarantees that shared data becomes consistent at a processor when the processor acquires a synchronization object known to guard the data. Entry consistency is weaker than other models described in the literature, such as processor consistency and release consistency, but it makes possible higher performance implementations of the underlying consistency protocols. Midway programs are written in C, and the association between synchronization objects and data must be made with explicit annotations. As a result, pure entry consistent programs can require more annotations than programs written to other models. In addition to entry consistency, Midway also supports the stronger release consistent and processor consistent models at the granularity of individual data items. Consequently, the programmer can tradeoff potentially reduced performance for the additional programming complexity required to write an entry consistent parallel program.

This research was sponsored in part 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, by the Open Software Foundation (OSF), and by a grant from the Digital Equipment Corporation. Bershad was partially supported by a National Science Foundation Presidential Young Investigator Award. Sawdon was partially supported by a grant from the International Business Machines Corporation.
The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, OSF, DEC, IBM, the NSF, or the U.S. government.

1 Introduction

Midway is a distributed shared memory (DSM) programming system supporting multiple memory consistency models within a single parallel program. Midway is intended for use on medium-scale multicomputers (fewer than 100 nodes), such as an ATM-based cluster [Rider 89], a TMC CM-5, or an Intel Paragon. In addition to supporting processor consistency and release consistency, Midway supports a new memory consistency model called entry consistency. Entry consistency guarantees that shared data becomes consistent at a processor only when the processor acquires a synchronization object that guards the data. Furthermore, the only data that is guaranteed to be consistent is that guarded by the acquired synchronization object. This allows an implementation of entry consistency to reduce the frequency of global communication by exploiting synchronization patterns between processors. Midway's implementation of entry consistency requires that the relationship between data and synchronization objects (which is implicit in the structure of a parallel program) be made explicit to the compiler and the runtime system.

Midway supports multiple consistency models within a single program to ease the task of constructing a program that runs efficiently on a DSM system. A program running under Midway may contain data that is processor consistent, release consistent, or entry consistent. Furthermore, within a single run of a program, multiple consistency models may be active at the same time. This allows the programmer to begin with a processor consistent parallel program, and then selectively relax its consistency requirements for shared data by modifying the program to use one of the weaker models.

1.1 Motivation

A wide range of memory consistency models exists, and each offers a different guarantee about the strength and timeliness with which updates to shared memory take effect at processors distributed throughout a network. In order of strength, these models include sequential consistency [Lamport 79], processor consistency [Goodman & Woest 88], weak consistency [Dubois et al. 86], release consistency [Gharachorloo et al. 90] and entry consistency, which is described in this paper. In order, each model can increase a processor's tolerance for latency in the memory system by relaxing the rules that determine the behavior of operations which write to shared memory. Aggressive implementations of the weaker models are capable of delivering higher performance than those of stronger ones because they better tolerate network delays and limited bandwidth [Gharachorloo et al. 91, Zucker & Baer 92].

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 the 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.

These observations about explicit synchronization have led to a class of *weakly consistent* protocols [Dubois et al. 86, Scheurich & Dubois 87, Adve & Hill 89, 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.[1]

A weaker model offers fewer guarantees about memory consistency, but it ensures that a "well-behaved" program executes as though it were running on a sequentially consistent memory system. The definition of "well-behaved" varies according to the model. For example, in a processor consistent system, the programmer may not assume that all memory operations are *performed* in the same order at all processors. (A load or store is globally performed when it is performed with respect to all processors. A load is performed with respect to a processor when no write by that processor can change the value returned by the load. A store is performed with respect to a processor when a load by that processor will return the value of the store.) For weak consistency, the programmer may not assume that a processor's updates are performed at other processors until the updating processor issues a synchronization operation. For release consistency, only a processor's releasing synchronization operation guarantees that its previous updates will be performed at other processors, and only a processor's acquiring synchronization operation guarantees that other processors' updates have been performed at it. (A releasing synchronization operation signals to other processors that shared data is available, while an acquiring operation signals that shared data is needed.) For entry consistency, data is only consistent on an acquiring synchronization operation, and only the data known to be guarded by the acquired object is guaranteed to be consistent.

Programs with good behavior do not assume a stronger consistency guarantee from the memory system than is actually provided. Each model's definition of good behavior places demands on the programmer to ensure that a program's access to shared data conforms to that model's consistency rules. For example, with entry consistency, a processor must not access a shared item until it has performed a synchronization operation on the item's associated synchronization object. These rules provide the memory system with in-

[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.

formation to allow a well-behaved program to execute as though it were running on sequentially consistent memory system. Unfortunately, the rules can add an additional dimension of complexity to the already difficult task of writing new parallel programs and porting old ones. The additional programming complexity can result in higher performance, though, because it provides greater control over communication costs.

1.2 Multiple models

Midway allows a programmer to navigate through a subset of the consistency models, selecting one, or several, to achieve an acceptable tradeoff between performance and programmability. A program written for Midway can use entry consistency, release consistency, or processor consistency. For all of these models, local memories on each processor cache recently used data and synchronization objects. With entry consistency, communication between processors occurs only when a processor acquires a synchronization object. Only the data guarded by the synchronization object is guaranteed to become consistent at the time of the acquire. Consequently, Midway provides an execution environment where a parallel program's performance is ultimately limited only by its internal synchronization patterns.

Although entry consistency enables the use of low-overhead consistency mechanisms, writing an entry consistent program requires more work than writing one to a stronger model. For example, every synchronization object must be identified; every use of such an object must be explicit; every shared data item must be associated with a synchronization object; and synchronization accesses should be qualified as read-only or read-write for best performance.

To make these restrictions less onerous, Midway provides a graceful migration path away from more strongly consistent models to entry consistency. A programmer begins with a processor consistent parallel program or algorithm and sets Midway's consistency model "dial" to processor consistency. The runtime system can be used to collect reference patterns for shared data so that strongly consistent code which accesses heavily shared data can be reorganized to use a weaker consistency model. With this, a programmer can quickly get an application running on the DSM, although the application may not run very quickly.

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. It also ensures portability across a wide range of multicomputer architectures. The system described in this paper is operational on a cluster of MIPS R3000-based DECstations running CMU's Mach 3.0 operating system [Accetta et al. 86] over both ethernet and an ATM network.

1.3 Related work

Memory consistency models for DSM systems have been implemented in both hardware and software. Earlier hardware-based systems used snooping protocols where each processor monitored a shared bus to implement processor consistency. The Stanford DASH [Lenoski et al. 92] multiprocessor supports release consistency in hardware using a directory-based protocol over a dedicated low-latency interconnect.

Most software systems intended for parallel programming have implemented these same consistency models using conventional virtual memory management hardware and local area networks. Li's Ivy system [Li 86] described the first implementation of such a *page-based* DSM and was followed by several other systems [Fleisch 87, Forin et al. 89]. Munin [Carter et al. 91] is a software system which uses release consistency to support automatic data caching over a local area network. Munin is unique among weak consistency systems, in that it implements multiple consistency *protocols* which can be used on a type-specific basis. Munin uses hints from the programmer to determine the access patterns to shared data items, and then selects the best consistency protocol for each. Munin differs from Midway in that it offers multiple implementations of a single consistency model (release consistency), whereas Midway supports multiple consistency models within a single program.

Lazy release consistency [Keleher et al. 92] is a technique for implementing release consistency through causal broadcast. It has been shown through simulation to greatly reduce the number of messages required by systems such as Munin. Our work with entry consistency can be considered as an extreme variant of lazy release consistency in that Midway's explicit association between synchronization objects and data offers the runtime system additional information about causality.

1.4 The rest of this paper

In Section 2 we describe entry consistency. In Section 3 we describe Midway's programming interface in the context of multiple consistency models. In Section 4 we describe the important aspects of Midway's implementation and show that the infrastructure required by entry consistency can be adapted to provide each of the stronger consistency models. In Section 5 we discuss performance. In Section 6 we present our conclusions.

2 Entry consistency

Entry consistency takes advantage of the relationship between specific synchronization objects that protect critical sections and the shared data accessed within those critical sections. A critical section is a region of code that accesses data which may have been written by another processor. A synchronization object controls a processor's access to the code and data in the critical section. Examples of critical sections are code sequences guarded by a mutex, or phased by a barrier. In an entry consistent system, a processor's view of shared memory becomes consistent with the most recent updates *only* when it enters a critical section.

The entry consistent model matches that already used by many shared memory parallel programs, namely, the use of critical sections to guard access to shared data for which the results of an unguarded access is undefined.

2.1 Performing store operations

A consistency model does not define whether store operations are performed at a processor using an invalidation-based or an update-based protocol. With an invalidation-based protocol, an operation is performed at a remote processor by invalidating an entry in that processor's local cache. The processor's next access to the invalidated entry results in a cache miss and a round-trip network message to fetch the missed value. With an update-based protocol, an operation is performed at a remote processor when the stored value is deposited in that processor's cache. This allows the next access to the item to always be satisfied locally. The advantage of an invalidation-based protocol is that consistency messages can be smaller because they contain only addresses, not data. The advantage of the update-based protocol is that it greatly reduces the likelihood of a cache miss.

Midway's implementation of entry consistency uses an update-based protocol. In relatively high latency networks, where interprocessor communication is on the order of thousands of processor cycles, the effect of a cache miss on processor performance can be substantial. For example, assuming a RISC processor with a 10 *ns* cycle time, the latency of resolving a cache miss over an ATM network with a 100 μsec round-trip time is on the order of 10,000 instructions. Consequently, it is critical to use an update-based protocol to minimize the chance that a processor experiences a cache miss.

An advantage of entry consistency with an update-based protocol is that interprocessor communication is only necessary during the acquisition of synchronization objects. By updating only at synchronization points, and only between the synchronizing processors, new values for data guarded by a synchronization object may be coalesced and delivered to a processor all at once. By ensuring that updates are performed with respect to a processor when it enters a critical section, unexpected delays in a critical section as a result of cache misses cannot occur. Moreover, no communication is required for repeated accesses and releases of the same synchronization object on the same processor — common patterns in parallel programs [Eggers 89, Bennett et al. 90].

2.2 Caching synchronization objects

Entry consistency facilitates strategies which permit synchronization objects to be cached on the processor(s) where they were most recently used. For a synchronization object s, we define the *owner* as the processor that last acquired s. Only the owner of s may perform updates to the data guarded by s. The processor that owns a synchronization object may enter and exit the associated critical sections without having to communicate updates of shared memory to other processors. A processor becomes an owner of s by sending a message to the current owner. The current owner ensures that all updates to the data guarded by s are then performed at the new owner.

An unfortunate aspect of single ownership is that no more than one processor at a time can access a given shared location even if the location is only being read. To guarantee consistency, a processor must hold the appropriate synchronization object. However, that synchronization object, if used in the classical sense (such as a semaphore), only permits mutually exclusive access to the data. Consequently, straightforward use of synchronization objects to ensure consistency can limit concurrency.

We address this problem by defining two modes of access to synchronization objects: *exclusive* and *non-exclusive*. Synchronization objects continue to be owned by a single processor, but may be replicated if they are held only in non-exclusive mode. A processor must perform an exclusive access to a synchronization object *s* in order to update any data guarded by *s*. By definition, that processor becomes the owner of *s*. Reading data guarded by *s*, though, only requires non-exclusive access to *s*.

An exclusive-mode access to a synchronization object *s* requires that no other processor holds *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* is performed with respect to the owner of *s*. This enables a processor to perform a sequence of non-exclusive accesses to *s* without having to communicate with *s*'s owner each time.

2.3 Programming to entry consistency

Entry consistency makes several 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 be distinguished from other accesses. Second, entry consistency requires an association between shared data and its guarding synchronization object. Third, to enable concurrent read-sharing, entry consistency requires that exclusive synchronization accesses are distinguished from non-exclusive accesses. Finally, entry consistency requires that updates are performed with respect to an acquiring processor. The last constraint affects Midway's implementation, while the first three affect its programming interface. Specifically,

- All synchronization objects should be explicitly declared as instances of one of Midway's synchronization data types, which include locks and barriers.

- All shared data must be explicitly labeled with the keyword **shared**, which is understood by the compiler.

- All shared data must be explicitly associated with at least one synchronization object. This is made by calls to the runtime system, is dynamic, and may change during the execution of a program.

Programs that include the necessary labeling information, and precede all accesses to shared data with an access to the appropriate synchronization object will observe a sequentially consistent shared memory.

3 Other models in Midway

A parallel program's consistency requirements can be buried within its algorithms and sharing patterns. Midway's implementation of entry consistency requires that they be made explicit. This may be difficult because it can require a complete understanding of a program or algorithm, and can be a major barrier to porting someone else's code.

From a performance standpoint, a complete transformation into entry consistency may not be necessary. In many parallel programs, most communication is of a few primary data structures. While a large number of secondary data structures may be used, they are shared, or at least modified, with low frequency. There would be only a marginal performance impact when using a stronger consistency model to manage these infrequently modified items. For example, some programs maintain a set of flags which change infrequently, such as when new data is available, or when an algorithm has terminated. This kind of data may be most easily managed as processor consistent. There also exist programs that can tolerate minor inconsistencies in their results, and underspecify their synchronization. This is done, for example by *locus*, *mp3d* and *pthor* from the Splash application suite [Singh et al. 92]. These programs can be converted to entry consistency by, for example, binding all data to a barrier, but this would oversynchronize the processors. Instead, managing the data with its initially assumed consistency model may be the best solution.

Because entry consistency may be hard to use and may not always offer a performance advantage, a Midway program may also contain data which is release or processor consistent. Entry consistent data is associated with a synchronization object. Data not associated with a synchronization object, but with a "flush interval" is maintained according to processor consistency. The flush interval controls the rate at which updates are propagated (in issue order) to other processors. An item that is neither associated with a synchronization object nor a flush interval is assumed to be release consistent. A processor's updates to release consistent data are performed at remote processors only when a release from the updating processor is necessary to satisfy another processor's acquire.

4 Implementation

The implementation of Midway consists of three main components: a set of keywords and function calls used to annotate a parallel program, a compiler which generates code to maintain reference information for shared data, and a runtime system to implement several consistency models.

4.1 Compiler and language support for Midway

Midway's concurrency primitives are based on the Mach C-Threads interface [Cooper & Draves 88]. A Midway program is written in C, and looks like many other parallel C programs that use thread management directives such as *fork* and *join*, and synchronization primitives such as *lock* and *unlock*. Shared data can be allocated either dynamically or statically, but must be tagged as shared during storage allocation. All references to shared data, however, do not need a shared qualifier, so procedures can take pointers to data which is either shared or unshared.

Midway requires a small amount of compile-time support to implement its consistency protocols. Whenever the compiler generates code to store a new value into a shared data item, it also generates code that marks the item as "dirty" in an auxiliary data structure. Other information necessary to implement entry consistency, such as the association between synchronization objects and guarded data, is specified at runtime with procedure calls into Midway's runtime system.

An alternative to relying on the compiler to generate code which marks items as dirty is to use the virtual memory system to trap writes to shared data. This is the approach taken with page-based systems such as Ivy and Munin. Although this approach allows programs to run with an unmodified compiler, it has several drawbacks that can limit its performance. First, virtual memory systems, and their underlying MMU hardware, do not have particularly fast fault handling times [Appel & Li 91], and those times are getting relatively slower, not faster [Anderson et al. 91]. Second, page-based strategies can incur a large number of write-faults in the presence of false sharing. This happens when unrelated data items on the same page are written by different processors. Third, faults which occur during a critical section increase the amount of time to execute the critical section, thereby increasing contention. Similarly, faults which occur during a barrier sequence result in processors finishing at staggered times, even though the computation may statically appear load-balanced.

4.2 Synchronization management

Distributed synchronization management enables processors to acquire synchronization objects not presently held in their local memories. Two types of synchronization objects are supported: locks and barriers. Locks are acquired in either exclusive or non-exclusive mode by locating the lock's owner using a distributed queueing algorithm [Forin et al. 89].

Barriers permit SIMD-style processing by synchronizing multiple processors across sequential phases of a computation. A processor delays at a barrier until all other processors reach that same barrier. Shared data accessed within a barrier must be made consistent only at the point 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).

Midway associates a manager processor with each barrier synchronization object. Processors "cross" the barrier by sending a message to the manager and waiting for a reply. The crossing message contains the barrier name and all updates to shared data associated with the barrier that were performed by the crossing processor. The manager coalesces the updated values it receives from all processors, then releases the processors by sending the coalesced updates back to each processor. Midway also supports a terminating barrier that can be used to coalesce the final results of a program at a single processor. Upon crossing a terminating barrier, the data is coalesced at the manager, but is not flushed back to the participating processors.

4.3 Cache management

Distributed cache management ensures that a processor never enters a critical section without having received all updates to the shared data guarded by that synchronization object. While this condition could be satisfied by transferring all shared data guarded by the synchronization object, 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 uses Lamport's happens-before relationship [Lamport 78] to impose a partial ordering on updates to shared data with respect to synchronization accesses.

Each processor p_i maintains a monotonically increasing counter c_i which serves as its local clock. Whenever p_i sends a message, for example to synchronize, to p_j, it increments c_i and includes c_i in the message. Upon receipt of the message, p_j sets its clock c_j to $max(c_j, c_i)$. Each synchronization object s has an associated timestamp t_s which is set to the value of c_j whenever its ownership transfers to another processor p_j. Each shared data value v guarded by s has an associated timestamp t_v that is logically set to the local clock value whenever v is updated. When processor p_i requests s from p_j, the request contains p_i's last value of t_s, $t_{s.i}$, which is the "time" that p_i last observed 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.

Timestamps are arranged in memory so that the runtime system can quickly convert from a shared item's address to its timestamp. Midway avoids computing a timestamp for each update, delaying until the timestamp is needed by the synchronization protocol. On store, the local timestamp field is set to zero to indicate that the associated data item has been modified. When a synchronization object s is requested from a processor p_i, all data guarded by s whose timestamp is zero will have their timestamps set to c_i. When a shared data item is allocated, the granularity of the timestamp (in effect, the cache line size) can be selected by the programmer according to the expected access patterns to the item. For example, a large contiguous object may be backed by many timestamps to improve the granularity of sharing and update information. Timestamp granularity can be as fine as a single byte.

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 object guarding the item. When next acquiring the guarding synchronization object s, the discarding processor indicates that it has not held s for a "very long time."

4.4 Supporting stronger consistency models

Much of Midway's infrastructure for entry consistency can be leveraged to support processor consistency and release consistency. Supporting these other models requires that the compiler and runtime detect writes to shared data, perform updates at other processors in the order required by the model, and recover from cache misses which occur when a processor accesses a shared data item not present in its local cache. For this, Midway overloads the timestamp mechanism described earlier. The compiler emits the same code for a store to a shared address, but, at runtime, the computed timestamp is treated differently.

Data items maintained according to processor or release, but not entry, consistency, initially have the high bit in their associated timestamp set. On a store to the line, if the high bit of the timestamp field is set, then the store will be performed at all other processors independent of any *particular* synchronization operation. The modified address is recorded in a per-processor queue of pending updates and the high bit of the timestamp is cleared to ensure the item is not queued again. This strategy for queue updates allows us to use the same compiler-emitted code sequence when updating both entry consistent and non-entry consistent data. Once the non-entry consistent data has been queued, subsequent stores continue to clear the timestamp field just as if the data were entry consistent.

Associated with each cache line is a *copyset* that defines the processors holding a copy of the line in their local memories. A pending update to a line need only be performed at the processors in that line's copyset. For release consistent data, pending updates are flushed by a processor any time the processor's releasing synchronization operation is performed at another acquiring processor. Processor consistent data is flushed at these points, as well as at the periodic interval specified by the programmer. In either case, when an update is flushed by a processor, the high bit in the item's timestamp is set to catch that processor's next store to the item at that processor.

Under entry consistency, all data bound to a synchronization object is prefetched when the object is acquired. With processor consistency and release consistency, there is no associated synchronization object and no way to know which data to prefetch or when to prefetch it. Consequently, on a processor's first reference to an item, that processor will not be in the item's copyset and will not have received any prior updates. We detect this condition with a *copyset fault*, which is implemented with virtual memory page faults.

Each processor marks virtual memory pages that contain cache lines for which the processor is not in the copyset as *no-access*. Access to such a page causes a page fault, and the faulting processor fetches the faulted page from the page's *home* processor. Every page has a home, based on the page's virtual address. Before the home node returns the page's data, the faulting node is added to the page's copyset and all other members are informed of the change. Any processor but the home processor may remove itself from the copyset of a page by notifying the page's home processor.

With this strategy, all cache lines within a virtual page are part of the same copyset. A page fault occurs only when a processor adds itself to a cache line's copyset; otherwise runtime write detection to shared data is done with compiler-emitted code, rather than with faults. Thus multiple processors may write to a page at the same time.

For all three consistency models, Midway uses update, rather than invalidate, to perform writes to other processors. This is done for two reasons. First, it guarantees that processors never experience a cache miss except on a copyset fault. Second, it enables the use of the same update machinery used for entry consistency.

5 Performance

In this section we look at the behavior of a simple parallel program under Midway using both entry consistency and release consistency. For our measurements, we use two metrics which we present as a function of the number of processors on which an application runs: execution time and message count. Our measurements show that a program written to entry consistency requires substantially fewer messages than one to the stronger models. This translates into improved execution time.

5.1 The hardware platforms

Our implementation of Midway runs on MIPS R3000-based DECstation 5000/120s and 5000/200s on top of two networks: a 10Mb/sec ethernet and a 155 Mb/sec ATM network. The operating system is Mach 3.0 with CMU's Unix server [Golub et al. 90]. The DECstation 5000/120 uses a 20Mhz R3000 with a 12.5Mhz Turbochannel (TC) bus interface. The 5000/200 uses a 25Mhz R3000 with a 25Mhz Turbochannel interface. Presently, we have only four 155Mb/sec ATM network interfaces, and we use these to connect the faster and more network capable 5000/200s through the central switch.

We present results on three configurations: *slow-ether*, which uses the slower DS5000/120s connected by ethernet; *fast-ether*, which uses the faster DS5000/200s connected by ethernet, and *fast-atm*, which is like fast-ether, except that we use the ATM network instead. The slow-ether configuration allows us to evaluate performance on more than four processors, while the fast-ATM network lets us look at the system's behavior on a faster network with faster processors. We include the fast-ether numbers to provide a common point of comparison between slow-ether and fast-ATM.

5.2 Matrix multiply

We present the results of a simple matrix multiply application running on several processors to provide an integrity check for Midway, to show the interaction between processor speed and network speed, and to compare the behavior of a single program under two different consistency models. *MM-ec* is a matrix multiply program which multiplies two 512×512 floating point matrices on from one to eight processors using entry consistency. A master processor writes the two input matrices A and B, and then spawns slave processes on the other machines. The master and each slave machine acquires a non-exclusive (read) lock on all of A, and a non-exclusive lock on a portion of B. Each processor then computes its portion of the result matrix C, and crosses a final barrier. It returns that portion of C that it has written back to the master.

The main advantage of using entry consistency for matrix multiply is that the initial message required to satisfy each slave processor's non-exclusive lock acquisition transfers the input matrices. Similarly, when a slave terminates, the message it sends indicating that it has crossed a barrier also transfers the results back to the master. In effect, the consistency model in combination with a synchronization protocol natural to the problem provides optimal clustering of data, both in terms of number of bytes transferred and number of messages generated.

To assess the importance of this clustering, we have written a version of matrix multiply to use release, rather than entry, consistency (*MM-rc*). Table 1 shows the elapsed time, speedup, time spent computing and transferring data, amount of data transferred, and the total number of messages for *MM-ec* and *MM-rc*. Since fast-ATM represents the best possible configuration, we show the results of *MM-rc* only for it. In the single-processor case, we compiled the program to run on a uniprocessor, eliminating synchronization and timestamp management overhead.

Elapsed time is the interval beginning after all processors have started and ending when the master processor terminates. This includes the time to move the input and output data sets between processors. The data transfer times shown are from the perspective of the master processor (which sources and sinks all data). The input and output data sizes are only a function of the number of processors (problem partitioning), and not the processor, network or consistency model. The number of messages is a function of the number of processors and the consistency model. The compute time is the time spent actually working on the matrices and, as shown, scales with the number

	# Procs.	Elapsed (secs)	Speedup	Input Transfer (secs)	Compute Results (secs)	Output Transfer (secs)	Data Transferred (mbytes)	# Msgs
MM-ec: slow-ether	1	282	1	0	0	0	0	0
	2	148.4	1.90	12.92	132.4	3.05	2.14	24
	4	84.9	3.32	12.08	66.3	6.51	4.81	72
	6	65.0	4.34	13.72	44.4	6.92	7.13	120
	8	58.6	4.81	17.56	33.2	7.87	9.36	168
MM-ec: fast-ether	1	164	1	0	0	0	0	0
	2	92.8	1.77	8.70	81.4	2.66	2.14	24
	4	53.3	3.08	8.50	40.6	4.17	4.81	72
MM-ec: fast-ATM	1	164	1	0	0	0	0	0
	2	83.5	1.96	1.53	81.7	0.31	2.14	24
	4	43.3	3.79	1.86	40.9	0.50	4.81	72
MM-rc: fast-ATM	1	164	1	0	0	0	0	0
	2	86.8	1.89	3.14	82.0	1.61	2.17	1802
	4	48.4	3.39	6.06	41.1	1.28	4.97	5106

Table 1: *Breakdown of the performance of matrix multiply using both entry consistency and release consistency.*

of processors.

Looking only at the entry consistent configurations, speedup is slightly less than linear because the communication overheads do not decrease as processors are added. The fact that speedup at 4 processors for fast-ether is worse than for slow-ether shows the impact of increasing processor performance without increasing network performance. A single DECstation 5000/200 is roughly twice as fast at matrix multiply as a DECstation 5000/120. Communication overhead on both systems is roughly the same, therefore there is less room for improvement when running on a network of 5000/200s. In contrast, speedup is much closer to linear on the fast-ATM network where communication overhead is less than 6% of total execution time (as opposed to almost 25% on the fast-ether configuration).

By point of comparison, the Munin system yielded an 8-fold speedup for an 8 processor matrix multiply using release consistency running over a 10 Mb/sec ethernet [Carter et al. 91]. The processors used there, however were substantially slower than those used in our fast-ether configuration. A single Munin processor could compute the product of two 400x400 *integer* matrices in a little over 700 seconds, almost five times slower than a DS5000/200 running on a larger (512$\times$512) and harder (floating point) input set. From this we conclude that uniprocessor performance has reached the point where ethernet is no longer a viable network for parallel processing, even for well-structured parallel applications such as matrix multiply.

Moving to the release consistent runs, we can see the impact of Midway's implicit prefetch that comes during lock acquisition for entry consistency. Instead of transferring the entire input and output matrices in single messages, as is done with entry consistency, the release consistent implementation misses frequently as can be seen by the number of messages sent. Most of the messages correspond to the transfer of a page of data. Specifically, each message for the entry consistent run corresponds to a synchronization request, which also occurs during the release consistent run. The additional messages for the release consistent run are for data transfer. The elapsed time difference between the release and entry consistent runs is due to the overhead of having to send more messages, even though the total amount of data transferred is nearly unchanged.

6 Conclusions

There exist a range of memory consistency models that provide different kinds of behavior both in terms of semantics and performance. Strongly consistent memory systems simplify porting and reasoning about programs written for shared memory multiprocessors but are limited in their ability to conceal latency. Entry consistency takes into account both synchronization behavior and the relationship between synchronization objects and data. This allows the runtime system to hide the network overhead of memory references by folding all memory updates into synchronization operations.

References

[Accetta et al. 86] Accetta, M. J., Baron, R. V., Bolosky, W., Golub, D. B., Rashid, R. F., Tevanian, Jr., A., and Young, M. W. Mach: A New Kernel Foundation for Unix Development. In *Proceedings of the Summer 1986 USENIX Conference*, pages 93–113, July 1986.

[Adve & Hill 89] Adve, S. V. and Hill, M. D. Weak Ordering – A New Definition. In *Proceedings of the 16th Annual Symposium on Computer Architecture*, pages 2–14, May 1989.

[Anderson et al. 91] Anderson, T., Levy, H., Bershad, B., and Lazowska, E. The Interaction of Architecture and Operating System Design. In *Proceedings of the Fourth Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 108–121, April 1991.

[Appel & Li 91] Appel, W. and Li, K. Virtual Memory Primitives for User Programs. In *Proceedings of the Fourth Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 96–107, April 1991.

[Bennett et al. 90] 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 125–134, May 1990.

[Carter et al. 91] Carter, J. B., Bennett, J. K., and Zwaenepoel, W. Implementation and Performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[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.

[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 *Proceedings of the Summer 1986 USENIX Conference*, 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.

[Gharachorloo et al. 91] Gharachorloo, K., Gupta, A., and Hennessy, J. Performance Evaluation of Memory Consistency Models for Shared Memory Multiprocessors. In *Proceedings of the Fourth Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 245–259, April 1991.

[Golub et al. 90] Golub, D., Dean, R., Forin, A., and Rashid, R. Unix as an Application Program. In *Proceedings of the Summer 1990 USENIX Conference*, pages 87–95, June 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.

[Keleher et al. 92] Keleher, P., Cox, A., and Zwaenepoel, W. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, pages 13–21, May 1992.

[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):690–691, September 1979.

[Lenoski et al. 92] Lenoski, D., Laudon, J., Gharachorloo, K., Weber, W., Gupta, A., Hennessy, J., Horowitz, M., and Lam, M. The Stanford DASH Multiprocessor. *IEEE Computer Magazine*, 25(3):63–79, March 1992.

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

[Rider 89] Rider, M. Protocols for ATM Access Networks. *IEEE Network*, January 1989.

[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.

[Singh et al. 92] Singh, J., Weber, W., and Gupta, A. SPLASH: Stanford Parallel Applications for Shared Memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[Zucker & Baer 92] Zucker, R. N. and Baer, J.-L. A Performance Study of Memory Consistency Models. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, pages 2–12, May 1992.