Lamport Clocks: Reasoning About Shared Memory Correctness

Daniel J. Sorin
Manoj Plakal
Mark D. Hill
Anne E. Condon

Technical Report #1367

March 1998

Lamport Clocks: Reasoning About Shared Memory Correctness[1]

Daniel J. Sorin, Manoj Plakal, Mark D. Hill and Anne E. Condon

Computer Sciences Department
University of Wisconsin - Madison
{sorin,plakal,markhill,condon}@cs.wisc.edu

Abstract

Modern shared memory implementations use many complex, interacting optimizations, forcing industrial product groups to spend much more effort in verification than in design. Current formal verification techniques are somewhat non-intuitive to system designers and verifiers, and these formal methods do not scale well to practical systems.

This paper seeks to give verifiers and designers a reasoning technique that is precise (unlike informal reasoning) and intuitive (unlike some formal models). To prove that a system obeys the desired consistency model, we would like a tool that allows us to create a total order of events. We modestly extend Lamport's logical clock work from distributed systems and apply it to shared memory systems. We use these so-called Lamport clocks to timestamp events and thereby create a total order. This total order can then be examined to see if it satisfies the desired consistency model. Lamport clocks are purely a reasoning tool, and they are never instantiated in hardware.

We demonstrate the value of Lamport clocks by showing that sequential consistency (SC) is obeyed by a variety of snooping bus-based coherence protocols, ranging from a simple cache-less system to a split-transaction out-of-order bus. We present timestamping schemes for all of the above systems and, in the case of the split-transaction bus, we use the timestamps to formally prove that the system satisfies SC.

1. This work is supported in part by Wright Laboratory Avionics Directorate, Air Force Material Command, USAF, under grant #F33615-94-1-1525 and ARPA order no. B550, National Science Foundation with grants MIP-9225097, MIPS-9625558, CCR 9257241, and CDA-9623632, a Wisconsin Romnes Fellowship, and donations from Sun Microsystems. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Wright Laboratory Avionics Directorate or the U.S. Government.

We have written a companion paper to appear in the *10th Annual Symposium on Parallel Algorithms and Architectures* [18]. That paper uses Lamport clocks to study a directory protocol, while this paper examines a complex shared bus protocol. Together the two papers cover the two major classes of coherence protocols. The complexity of both protocols, however, precludes presenting both in sufficient detail in one paper. .

1 Introduction

Many papers in the literature propose optimizations to improve computer system performance. These papers often ignore the difficult problem of verifying whether new, more-complex implementations are completely correct. Nevertheless, industrial product groups spend much more effort in verification than in design.

One area where the verification problem is acute is shared memory systems. Designers must implement a memory consistency model, such as sequential consistency (SC), correctly and with high performance. To be correct, an implementation of SC must insure 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 individual processor appear in this sequence in the order specified by its program.[12]

To achieve high performance, memory system implementors now exploit a rich pallet of optimizations, including snooping bus protocols, directory protocols, out-of-order buses, multiple buses, arbitrary interconnects, write buffers, invalidation queuing, superscalar execution, out-of-order execution, and speculative execution. Correctness is often demonstrated by intuitive arguments on individual optimizations and extensive simulations of the complete system.

Let's consider one example of an aggressive optimization. With a standard invalidation-based directory protocol (like that of the Stanford DASH [14]), a store request to the directory that finds many read-only copies outstanding causes an invalidation message to be sent to each copy and then acknowledged. Scheurich observes that processors can queue invalidations, send acknowledgments, and then perform invalidations, as long as the processors are "isolated" from the rest of the system [20]. Thus, it is possible for processor P1 to perform its store at a physical time pt_{store} before processor P2 does a load at physical time pt_{load} (i.e., $pt_{store} < pt_{load}$). P2 gets a "stale" value, and yet everything is "ok". While one can believe that Scheurich's optimization works in isolation, how can one be sure that it works when combined with some of the other optimizations listed above?

This paper seeks to make a modest step towards providing designers and verifiers with a more formal grounding for insuring that shared memory systems are correct. Lamport's definition of SC says an implementation must conform to an order "that exists". We seek to think about an implementation as dynamically constructing an order against which we can verify correctness. Our process works in three steps:

- The designer/verifier reasons about the system as concurrently performing events at some level of detail (e.g., loads, stores, get EXCLUSIVE coherence permissions, and receive invalidation messages).
- We give the designer/verifier rules to assign logical timestamps to all events that ensure that all causal interactions flow to larger logical time. We call our method *Lamport Clocks*, because it modestly extends Lamport's work from distributed systems [11]. Our method conceptually adds Lamport clocks to entities (e.g., processors, caches, directories, and buses) and conceptually adds Lamport timestamps to messages. *In no case are Lamport clocks or timestamps added to actual hardware.*
- The designer/verifier can then use the timestamps to prove that an implementation meets the requirements of a memory consistency model. With SC, for example, does a load always return the value of the last store (in Lamport time) to the same address? The proofs can vary from informal to reasonably formal to machine automated.

Recall that Scheurich's optimization allowed a store at physical time pt_{store} to occur before a load at physical time pt_{load} even though the load returns the old value and is thus logically before the store. With Lamport clocks, the load can get a logical timestamp lt_{load} from before the logical time of the invalidation (it has not been causally affected by the invalidation), while the store's logical timestamp lt_{store} will be larger than that of the invalidation, because it occurs causally after it. Thus, $lt_{load} < lt_{store}$ even though $pt_{store} < pt_{load}$. In logical time, the load correctly returns the old value because it occurs logically before the store. This example is illustrated in Table 1 and Table 2, where the arrow in Table 1 shows the causal relationship between the invalidate and the processing of this invalidate.

TABLE 1. Scheurich's example in physical time

	P1	P2
	invalidate	
physical	store	
time		load
		process invalidation

TABLE 2. Scheurich's example in logical time

	P1	P2
		load
logical	invalidate	
time		process invalidation
	store	

The rest of the paper is organized as follows. Section 2 proposes the notion of coherence epochs in a coherence protocol. We will use epochs as a tool for analyzing protocols, since protocol are responsible for maintaining certain invariants with respect to epochs. Section 3 explains the concept of Lamport clocks and how we can use Lamport clocks to delineate coherence epochs. Section 4 examines a simple bus system with a write-invalidate protocol. Section 5 analyzes a system with a complicated bus and a write-invalidate protocol. Section 6 analyzes a system with multiple interleaved buses. Section 7 shows how we can use Lamport clocks to show that a protocol is incorrect. Section 8 discusses how our technique compares with other verification techniques. Section 9 discusses follow-on work, including how our technique can be extended to directory protocols and how our technique can be applied to proving the correctness of protocols that obey consistency models other than SC. Finally, Section 10 summarizes our contributions.

2 Coherence Epochs

Memory coherence protocols in shared-memory multiprocessors ensure a global total order of memory operations on any single block of memory. In a typical write-invalidate protocol, the total order of operations on a block of memory is constructed as processors initiate transactions so as to obtain the proper coherence permissions for performing memory operations. If a processor will be reading a block, it makes a request for Read-Only access and receives the block in the SHARED state in its cache, after which it can proceed to execute loads on this block. If a processor needs to write to a block, it makes a request for Read-Write access and subsequently receives the block in the EXCLUSIVE state in its cache (possibly invalidating other SHARED copies or another EXCLUSIVE copy), after which it can execute loads and stores on this block. Let us define *memory operations* to be the loads and stores that are performed by a processor, and let us define *coherence transactions* to be events that cause processors to change their access permissions to blocks of data.

Most protocols insist that a processor wait until it actually has the data before it is allowed to execute a memory operation. This requirement led us to explore what it means to execute a memory operation, and we found it convenient to split the execution into what we refer to as "binding" and "performing". A memory operation can be *bound* to a transaction if that transaction gains us the appropriate coherence permission. Binding an operation is equivalent to reserving resources to actually read data from or write data to the cache (i.e., perform the instruction). An operation can then be *performed* once the data for the operation is present in the cache.

This leads to a natural separation of memory operations from the coherence transactions that provide them with the block to operate on. Memory operations need to be bound and performed in a manner consistent with their program order and the memory consistency model supported by the system. However, coherence transactions can be performed in any order whatsoever without affecting correctness. We view coherence transactions as defining *coherence epochs* on blocks. A coherence epoch for a block is a period of "time" during which a processor has certain coherence permissions for that block and hence can bind memory operations on that block. For example, if a processor executes a coherence transaction to get a block in the Read-Write state, then an EXCLUSIVE epoch for that block starts on that processor from the time when it can start binding memory operations on that block. The epoch will end with the last operation it binds for that block.

A coherence protocol can now be viewed as a mechanism for demarcating and *serializing* these epochs, thus ensuring that a valid copy of the block circulates among the processors and maintaining a global order of all the memory operations performed so far on the block. Put another way, a coherence protocol ensures a clean separation among epochs and that data gets correctly "handed over", like the baton in a relay race, from one epoch to the next. Of course, not all epochs need to be separated in this way since SHARED epochs can overlap.

The preceding discussion has glossed over concepts like "time", "next" epoch, and so on. Coherence epochs will be a well-defined notion only after we resolve the issue of ordering operations that occur on different processors of a multiprocessor system. Such a problem has already been tackled in distributed systems by Lamport. In the next section, we explain how we adapt Lamport's logical clocks to construct a scale of *logical* time over which we can define coherence epochs.

3 Lamport Clocks

The most intuitive approach to ordering events is to order them by the physical times at which they occur. However, physical time is generally not a useful tool for ordering, since it is often difficult or impossible to determine which events happened before other events on different processors. For example, imagine two processors with SHARED data performing loads on the same cycle. Moreover, ordering events in physical time is generally more restrictive than ordering them *logically.* Using physical time, for example, would preclude Scheurich's optimization. Logical ordering only restricts the order between events that are *causally related.* Two causally related events would be the sending of a message by one processor and the reception of that message at another processor.

While determining an order in physical time is often difficult, an order in logical time can be extracted if we know which events logically cause other events. If event A causes event B, then the logical time of event A should precede that of event B. To construct a total order of events in a multiprocessor system, we shall use an adaptation of the logical ordering scheme proposed by Lamport [11]. Lamport logical clocks are purely conceptual devices for reasoning about event ordering. References to logical clocks refer to the concept of a logical clock at a device and not to any physical hardware.

The rules for assigning logical timestamps provide us with a *partial* order of the events in the system, since causally unrelated events may be assigned the same logical timestamp at different processors. To obtain a *total* order, all ties are broken using an ID unique to each processor. Thus, the logical timestamps and processor IDs produce a total ordering of all the events in the system.

We have adapted Lamport's clocking scheme in a few ways so that we could apply it to our framework for reasoning about protocols. We need to assign timestamps to memory operations and coherence transactions. A coherence transaction is timestamped using a 2-tuple <global time, processor ID> where the global time is used to order coherence transactions and epochs on different processors. A memory operation is timestamped using a 3-tuple <global time, local time, processor ID>. The local time component is used to order the memory operations within an epoch while respecting program order. Our notation for Lamport time is similar to software release notation. For example, time 3.6.2 would refer to a global time of 3 and a local time of 6 at processor 2. Since global time has precedence over local time, time 3.6.2 occurs after time 2.20.1.

We can imagine that each processor has a global clock that is incremented upon every transaction that changes the state of any block in its cache. Bus transactions are timestamped with this global time and a local time of zero. A memory operation is assigned a global time equal to either the global time of the transaction to which it was bound or the global timestamp of the immediately preceding memory operation in the program order (if any), whichever is greater. The local timestamp of a memory operation is one greater than the local timestamp of the operation immediately before it in the program order or it is equal to one, if it is the first operation bound to a given transaction.

Lamport timestamps can be used to construct a partial order of the coherence transactions that occur in a multiprocessor system, and we can now use this order to determine the intervals of time over which coherence epochs are defined for memory blocks. Consider the example of a generic bus-based coherence protocol running on a split-transaction bus (shown in Figure 1 and Figure 2). Notice that only the loads and stores have Lamport timestamps that include processor ID, because we only have to create a total order for these operations. Also observe that the binds of the load and the store are within the epochs that are started when the global time is incremented to reflect the request for new access permission.

Physical time	P1	P2
1	Request SHARED permission and bind load	
2	Receive block from owner	
3		Request EXCLUSIVE permission and bind store
4	Queue invalidation	Receive block from owner
5		Perform store
6	Perform load, invalidate block	
7		Send block to P3

FIGURE 1. An example of Coherence Epochs (in Physical time)

Lamport time	P1	P2
1.0	Request SHARED permission and receive block from owner	
1.1.1	Bind load	
2.0	Invalidate block	Request EXCLUSIVE permission and receive block from owner
2.1.1		Bind store
3.0		Send block to P3

FIGURE 2. An example of Coherence Epochs (in Lamport time)

We see that the lifetime of each block (in Lamport time) can be divided into non-overlapping *coherence epochs*. A coherence epoch starts when a processor starts binding memory operations, and an epoch ends when it cannot bind any more operations. During each of its epochs, a block is present in either the SHARED or EXCLUSIVE state in a cache and this state does not change during the entire epoch. A processor may bind only loads on this block during a SHARED epoch, and it may bind both loads and stores during an EXCLUSIVE epoch. There might be periods of Lamport time when a block does not belong to any epoch: between the time when one epoch ends and another epoch begins, during which it is resident only in memory. Also note how operations are performed seemingly out-of-order in physical time while the logical view shows the load happening before the store.

In summary, Lamport timestamping allows us to order events occurring at different processors and delineate the coherence epochs for a block of memory in a shared-memory multiprocessor. The role of a cache coherence protocol can be defined in terms of maintaining invariants involving epochs. The Lamport framework allows protocols to be specified and verified in an easier and more intuitive manner, and we will show that it also exposes optimizations in protocol implementations that may not have been evident earlier.

The next sections of this paper are examples of how to reason about protocol correctness by using coherence epochs that are bounded in Lamport time. We will start with a simple shared bus protocol in Section 4, and then we will build up to more realistic bus protocols in Sections 5 and 6.

4 Simple Bus Protocol

Consider a simple bus, where each processor has a cache, all data is cacheable, and the bus supports a write-invalidate coherence protocol with the following transactions: Get EXCLUSIVE (GX), Get SHARED (GS), Upgrade from SHARED to EXCLUSIVE (UPG), Writeback data to memory (WB), and Put SHARED (PUTS, an eviction of SHARED data). A processor can change its cache access permission to a block by issuing the appropriate transaction on the bus (in Section 5, the PUTS transaction will not go on the bus, but we will keep this protocol simple). Table 3 shows the behavior of the five transactions. Note that the term *owner* refers to a processor that has EXCLUSIVE access.

TABLE 3. Protocol Transactions

Transaction	Access Precondition for requester	Initial State of other cache(s)	Provider of data to requester	Final state of other cache(s)	Final state of requester
GX	INVALID	INVALID	memory	INVALID	EXCLUSIVE
		SHARED	memory	INVALID	EXCLUSIVE
		EXCLUSIVE	old owner	INVALID	EXCLUSIVE
GS	INVALID	INVALID	memory	INVALID	SHARED
		SHARED	memory	SHARED	SHARED
		EXCLUSIVE	old owner	SHARED	SHARED
UPG	SHARED	INVALID		INVALID	EXCLUSIVE
		SHARED		INVALID	EXCLUSIVE
WB	EXCLUSIVE	INVALID		INVALID	INVALID
PUTS	SHARED	INVALID		INVALID	INVALID
		SHARED		SHARED	INVALID

To keep our bus simple for now, assume that all data transfers and access permission changes related to a transaction occur before the bus accepts the next transaction. Assume that the bus and each cache have a Lamport clock, as shown in Figure 3.

How does a processor bind a memory operation in this system? First, it looks in its cache to determine if a transaction is required. If the data is already present in the cache with the correct permissions, then the processor can immediately bind the operation. If a transaction is required, the processor issues it on the bus. The processor now re-tries the memory operation and succeeds in binding it, because the data is now in the cache and has the correct permissions. Subsequent memory operations that hit in the cache can continue to be bound.

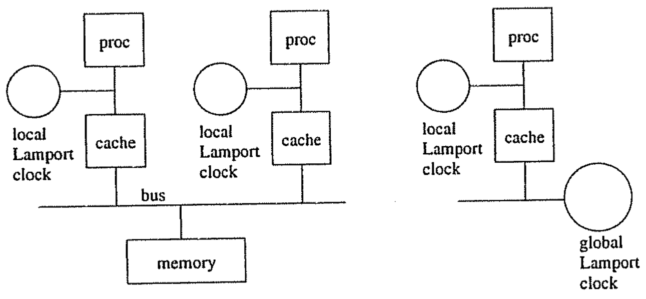

FIGURE 3. Simple Bus with Caches

SC is maintained through several protocol policies, and it is instructive to use Lamport clocks to examine these policies. Each processor binds loads and stores in program order. A cache with an EXCLUSIVE block can assign timestamps to loads and stores (without consulting the bus), since the coherence protocol guarantees that no other processor can access the block at this Lamport time. A processor wishing to access the block would have to use the bus and, therefore, could only start an epoch that began later in Lamport time. A cache with a SHARED block can hand out timestamps to loads. If other caches have the same block SHARED, they can hand out nearly identical timestamps without coordinating, because the loads do not affect each other.

Using Lamport clocks to order events reveals how epochs are delineated in this system. When a cache places a GX on the bus, it begins an epoch in Lamport time in which it can freely timestamp memory operations to the block. The epoch is terminated at the Lamport time given to the transaction that causes the cache to change its permission from EXCLUSIVE (external GX or GS). Similarly, an GS begins a SHARED epoch, and the epoch ends, in Lamport time, at the timestamp given to a local PUTS or an external UPG or GX.

Snooping bus protocols with caches reveal some of the utility of Lamport clocks. Consider the pseudocode in Table 4, where all processors initially have block B in the SHARED state and A is equal to 7. It is not obvious when reasoning with physical time that we can implement Scheurich's optimization, which allows processors to buffer the invalidation of B caused by the upgrade performed by P1. Using Lamport time, however, it is fairly straightforward to prove that we can make this optimization. For this example, let us assume that we are only interested in allowing P3 to buffer upgrades. In this case, we would like to allow P3 to load B at physical time 4, even though the upgrade has already been placed on the bus by P1 at physical time 2.

Table 5 shows the Lamport order extracted by allowing buffering. Note that LD = Load and ST = Store. The order maintains sequential consistency, and it permits P3 to perform a load at physical time 4 that we might not have thought permissible before reasoning with Lamport time. Thus, Lamport time is less restrictive than physical time.

TABLE 4. Can invalidates be buffered?

physical time	P1	P2	P3
1	LD r1=B /* gets 7 */	LD r1=B /* gets 7 */	
2	UPG B		
3	ST B=9		
4			LD r1=B /* can we do this and get 7? */
5	WB B		

TABLE 5. Buffering Invalidates

Lamport time	P1	P2	P3
2.3.2		LD r1=B	
2.5.1	LD r1=B		
2.9.3			LD r1=B /* gets 7 */
3.0	UPG B		
3.1.1	ST B=9		
4.0	WB B		

We have shown that we *can* buffer the invalidate, but how long can we wait before having to process it? Once again, Lamport clocks help us to reason about this problem. A processor can buffer invalidates indefinitely while it is performing local operations. It only needs to process the invalidates before it performs any coherence transaction (i.e., global operation), because a global operation forces the processor to synchronize with the rest of the system. In our example, P3 cannot globally timestamp any operation until it has processed the invalidation, but it could bind as many local operations (i.e., loads) as desired before handling the invalidate.

5 Complicated Bus Protocol

We will now discuss a protocol that is similar to several current bus protocols. The utility of Lamport clocks will be best shown in the analysis of this more complicated example.

5.1 Informal Description

To improve the performance of shared bus SMPs, designers incorporate more complicated bus protocols. A common feature of more complicated buses is that a transaction does not have to be completed before the next transaction can begin. In simple, circuit-switched bus protocols, coherence transactions are *serialized* (i.e., if processor P1 has requested a block in the EXCLUSIVE state and memory has not responded yet, then P1 will not release the bus until it receives the block from memory). Until P1's transaction has completed, a circuit-switched bus disallows other transactions from this processor as well as transactions from other processors. Split-transaction buses, on the other hand, allow systems to pipeline requests and responses. However, split-transaction buses are more difficult to prove correct, because transactions are not always atomic. Many split-transaction buses, such as Sun Microsystem's Gigaplane™ [23], also permit data to be returned out of order with respect to the requests for it.

In a split-transaction protocol, a transaction is composed of an action and zero or more reactions, where the action is a request and the reaction consists of the responses of all processors and memory modules to the action. For example, if a processor needs a block in the EXCLUSIVE state, it arbitrates for the bus, makes a request on the bus, and then it releases the bus. If memory or another processor has to respond, it will eventually send a reply on the bus, thus completing the transaction. Bus utilization is improved by allowing multiple transactions to proceed in parallel. Table 6 defines the actions and reactions for the protocol that we shall use in this section. All actions except PUTS use the bus. Notice that the actions in this protocol correspond to the transactions in the simple protocol, because the simple protocol waited for the reactions to complete before initiating the next transaction.

The split-transaction nature of the protocol may require transfer of coherence permissions from processor to processor even before the data has arrived in response to an original coherence request. For example, suppose processor P1 makes a request for block B in the EXCLUSIVE state. Before memory has a chance to respond, processor P2 can make a request for the same block B in the EXCLUSIVE state. Now how do we handle this? One alternative would be to disallow it [13]. This prevents P2 from making a request for B until memory has responded to P1's request. Another possible alternative would be to let memory (or possible a dedicated bus controller) keep track of outstanding transactions. This agent could then send blocks to requesting processors. A third alternative would be for ownership to transfer immediately upon requests [23]. In this case, processor P1 becomes the owner of the block immediately after it makes its request. It then sees P2's request and records this fact so that it can send the block to P2 when it receives the block from memory. The protocol would need to impose certain constraints to avoid livelock and ensure forward progress. We pursue this last approach in our proposed protocol.

TABLE 6. Protocol actions and reactions

Actions		Reactions	
Code	**Description**	**Code**	**Description**
GX	Get EXCLUSIVE	INV	Invalidate
GS	Get SHARED	DWG	Downgrade (EXCLUSIVE to SHARED)
WB	Writeback	SEND	Send data
PUTS	Put SHARED		
UPG	Upgrade (SHARED to EXCLUSIVE)		

Our protocol will transfer permissions immediately upon requests, even though the data may not get transferred for some amount of time after the transfer of ownership. This suggests the concept of maintaining two separate states for each block -- one state (the address state) is maintained at the bus interface while the other state (the data state) is maintained at the processor's cache. The address state (A-state) changes immediately on coherence actions while the data state (D-state) may lag behind while waiting for the reaction(s).

The address tags at each processor maintain one of three states for each block: A_X (EXCLUSIVE), A_S (SHARED), or A_I (INVALID). In addition, the memory node has an A-state for each block. For the memory to be A_X means that all other nodes have the block A_I, and if the memory is A_I then one node has the block in A_X.

Similarly, the data tags indicate D_X, D_S, or D_I. Furthermore, the data tags record information regarding pending transactions for a block (e.g., an invalidation may have been received for a block before the block has actually arrived at the cache from the bus). The D-state of the memory node is not defined. The D-states, although sometimes useful as a reasoning tool, can be ignored in the proof of the protocol. In Section A.1, we will specify the protocol by providing tables indicating how the address states are changed by a processor's actions and by reactions received from the bus. A queue is used to buffer messages from the bus (including a processor's own messages) before they are processed by the cache controller. The system is depicted in Figure 4.

We contend that loads and stores can be bound as soon as we obtain the appropriate permission for the A-state. The permission requirements are show in Table 7. Once permission for a block B has been obtained, the values associated with the words of B are deterministic. This does not imply that these values have already been stored to B or even calculated yet - it simply means that, from that moment until P gets the data, the value of B *when it arrives at P* can only take on one value. Once P has permission for B, it can bind a load or store (whose performance may have to wait for the data to arrive) to a word of B and continue issuing subsequent instructions.

FIGURE 4. Cache coherence system

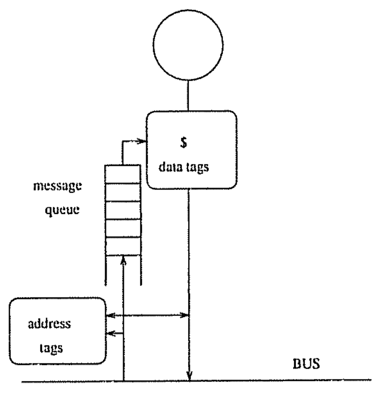

TABLE 7. Conditions for binding loads and stores

Action	Access Precondition	What to do if precondition is false
LD	A_S or A_X	GS
ST	A_X	GX

Transactions other than PUTS on a given block are serialized by the bus. This sequence is referred to as the *transaction serialization order.* For each node N, a sequence of t bus transactions (i.e., excluding PUTS) on block B defines a unique sequence $A_1, A_2, ..., A_t$ of associated A-states for N, given some initial A-state value at N. If A_i is not equal to A_{i-1} for some $i \geq 1$, we say that the ith transaction in the sequence "potentially affects" node N. The following claim relates the order that potential A-state changes are "considered" by a processor to the order in which transactions appear on the bus. A processor "considers" a potential A-state change in response to the corresponding transaction by checking its A-state and possibly changing it. Considerations are atomic in the sense that a processor completely considers an A-state change before considering the next.

Claim 1: The sequence of potential A-state changes on block B at a node are considered in the order implied by the serialization of the corresponding transactions on the bus.

In addition to considering A-state changes due to bus transactions, a processor also changes A-state in response to a PUTS. The changes in A-state on a given block due to PUTS transactions and the potential changes in A-state due to bus transactions are totally ordered at a processor. Corresponding to a sequence of t' transactions (including PUTS) on a block at a processor, there is a unique sequence $A'_1, A'_2, ..., A'_{t'}$ of A-states of that block at the processor, given some initial A-state value. If A'_i is not equal to A'_{i-1}, then the ith transaction "affects" N, and the transaction "implies that N's A-state for block B change from A'_{i-1} to A'_i".

For example, suppose that two nodes N_1 and N_2 are operating on block B. Let N_1 make a GS request on the bus before (in real time) N_2 makes a GX request on the bus. If N_1 does not do a PUTS before N_2 does the GX, then the sequence of A-states for B at N_1 is A_S, A_I. The GX both potentially affects as well as affects N_1, and the GX implies that N_1's A-state changes from

A_S to A_I. However, if N_1 does a PUTS before the GX, then the sequence is A_S, A_I, A_I. The GX potentially affects N_1 but it does not affect it.

Each transaction except PUTS implies an "upgrade" of A-state (i.e., change from state A_I to A_S, from A_I to A_X, or from A_S to A_X) at exactly one node, where a node is either a processor node or the memory node. Also, each transaction implies a "downgrade" of A-state (i.e., change from A_X to A_S, from A_X to A_I, or from A_S to A_I) at zero or more nodes.

5.2 Lamport timestamping scheme

Imagine that each processor and the memory have global clocks that are updated in real time. The clock is updated upon every action that affects some block in p_i's cache. The clocks are used to associate global timestamps with LD/ST operations and with transactions (thus defining epochs).

Suppose that the t^{th} transaction T affects node N. If T is not a PUTS, at the moment that the A-state changes, N adjusts its global clock to equal t, and it assigns a timestamp of *t.0* to T.

Let OP be the k^{th} LD/ST operation in the program order of processor p_i. First we consider the global component of OP's timestamp. Suppose that OP is bound to the t^{th} transaction on the bus. Then, the global timestamp for OP is defined to be equal to the maximum of {t, global timestamp of the (k-1)st LD/ST in p_i's program order (if any)}. The local timestamp of OP is defined to be 1 if OP is the first LD/ST operation of p_i in program order with global timestamp t. Otherwise, it is equal to 1 + local timestamp of the (k-1)st LD/ST operation of p_i. The local time of PUTS transactions is assigned identically to LD/ST operations. The global time of a PUTS transaction is the same as the global time of the last (in real time) considered bus transaction. Thus, a processor creates a total order of local events (LD, ST, PUTS), and we think of a PUTS as occurring in this order like LD/ST operations.

If G is the set of transactions and L is the set of LD/ST operations by all processors, then the timestamping scheme defines a total ordering on (G $\cup$ L) that is consistent with the local program order of each processor and also with the times at which transactions go out on the bus.

5.3 Outline of Proof of Protocol

Appendix A contains a proof of the protocol described in this section. It begins with a formal specification of the protocol, and it then states two facts about processor behavior requirements. Two timestamping claims are also introduced, and they make precise the fact that global timestamps order memory operations relative to transactions "as intended by the designer". These claims allow us establish sequential consistency in a sequence of lemmas using the concept of coherence epochs. The life of a block in logical time consists of a set of such epochs. One lemma shows that, in Lamport time, operations lie within appropriate epochs. That is, each LD lies within either a read-only or a read-write epoch, and each ST operation lies within a read-write epoch. Another lemma shows that the "correct" value of a block is passed from one node to another between epochs. The proofs of these lemmas build in a modular fashion upon the timestamping claims, thereby localizing arguments based on specification details. In other work [18], we have proved the correctness of a directory protocol using the same proof structure; the proofs of the lemmas for the directory protocol are exactly as for the directory protocol of this paper, and only the proofs of the timestamping claims differ.

5.4 Reasoning with Lamport Clocks

Consider a processor wishing to execute two successive stores to different blocks, where the first one is not in its cache but the second one is and it is already EXCLUSIVE. Can this protocol allow the processor to bind the second store and subsequent instructions before receiving the data for the block of the first store? This optimization would improve performance by overlapping miss latency with useful work, if we can show with Lamport clocks and coherence epochs that it does not violate SC. Once the processor issues the GX for the first block on the bus, the coherence epoch ends for any other processors that have the block. Even though the data for the block will not arrive until later, the EXCLUSIVE epoch has begun for the processor. Therefore, the processor can still bind the first store since no other processor will have access to the block until the writing processor gets the data, writes the block, and relinquishes it to another processor. Therefore, the second store can also be bound since the second block is already available in EXCLUSIVE state and the store is not waiting for any previous instructions to be bound. This optimization which appears so obvious when reasoning in Lamport time is not intuitive when analyzing the system in physical time. In addition to reasoning about specific optimizations, Lamport clocks can be used to formulate a formal proof that an entire complicated memory system protocol obeys SC, as shown in Appendix A.

6 Multiple Interleaved Bus Protocol

System designers have implemented coherence protocols on systems with multiple buses. The Sun Ultra™ Enterprise™ 10000 [1], for example, uses four buses. Figure 5 illustrates a system with two split-transaction out-of-order buses. In a system with k buses, bus access is interleaved by address such that traffic involving address A uses the bus with number A modulo k. A multiple bus system could provide an increase in bus bandwidth, but it is not intuitively obvious what restrictions must be applied to such a protocol to ensure that it obeys SC.

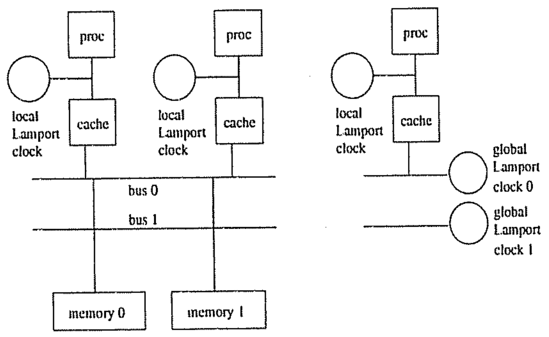

FIGURE 5. Multiple Interleaved Buses

How do we assign Lamport timestamps in a multiple bus system? Since the buses are independent and synchronous, we can simply add a term to the global part of the timestamps. The timestamping is similar to the scheme used for the complicated bus protocol, but it uses the bus ID to arbitrarily order simultaneous events on different buses. For example, two events at global time 3 on buses 0 and 1 would be arbitrarily ordered to have global times 3.0 and 3.1, respectively. But how do we timestamp local events that do not depend on using a particular bus, such as cache hits? Since we know what bus we would have to use if we missed in the cache, one option is to use the global time of the last operation we did on that bus. The effect of this policy is the same as if each processor had a separate cache for each bus and a separate Lamport clock at each cache.

With this timestamping scheme in place, we can just analyze a multiple bus protocol in the same way that we analyzed the single bus protocol. Coherence epochs are still defined over the global part of the timestamps (unless they end on a PUTS), but global time now consists of 2 terms. For example, a coherence epoch could be bounded by the global times 3.4 and 4.4. In Lamport time, we can think of this system as doing k sequential operations per bus clock. Since transactions involving the same block must use the same bus, there is no reason why we cannot arbitrarily order (in Lamport time) the physically concurrent events on different buses. Transactions on the same bus are still ordered in the same manner as in the single bus protocol. The proofs of optimizations and the complete protocol are now nearly identical to those of the single bus protocol.

7 Proof of Incorrectness of a Protocol

Let us now show how we can use Lamport clocks to demonstrate that a protocol does not obey SC. Consider a shared bus protocol with first-in-first-out (FIFO) write buffers at each processor, and assume that the processors are executing the code shown in Table 8. A is initially equal to 7.

What value of A should P2 get from its load? For the system to obey SC, it must return the value of the most recent store in Lamport time, and that value is 9. The write-back on P1 is logically before the GS on P2 and, therefore, logically before the load on P2.

But what value will P2 actually get? If the store from P1 is still in its write buffer when P2 performs its load, then P2 will get a value of 7. This protocol has a race condition, since the value of the load by P2 depends on whether the store from P1 has passed through its write buffer. Referring back to Section A.1, we can see that the protocol in this section violates Fact 2.

If we continue to reason with Lamport clocks, one solution to the inconsistency problem becomes obvious. We see that the store must logically precede the writeback. Therefore, the system would obey SC if processors flushed their write buffers before issuing writebacks. We assume that the processor must have EXCLUSIVE access to a block to which it is writing before it sends the write into the write-buffer.

TABLE 8. Write Buffer Example

physical time	P1	P2
1	LD r1=A /* gets 7 */	
2	UPG A	
3	ST A=9	
4	WB A	
5		GS A
6		LD r1=A /* gets ? */

8 Related Work[1]

Most of the related work in coherence protocol verification is based on formal methods [19] that use state-space search of finite-state machines, and theorem-proving techniques. These are rigor-

1. Note to editors and referees: This section is an exact copy of the Related Work section in our companion paper in the 10th Annual Symposium on Parallel Algorithms and Architectures (see footnote on first page).

ous methods that can capture subtle errors but they are currently limited to small systems because of the state space explosion for large, complicated systems. For example, the SGI Origin 2000 coherence protocol is verified for a 4-cluster system with one cache block in [7], the correctness of the Stanford FLASH coherence protocol is verified for small test programs and small configurations in [17], and the SPARC Relaxed Memory Order (RMO) memory consistency model is verified for small test programs in [16]. Formal verification software, such as Murϕ [5], is a useful tool in the verification process, and techniques for handling the state space explosion [19] may enable verification software to tackle larger systems. In contrast, though, our approach can precisely verify the operation of a protocol in a system consisting of any number of nodes and memory blocks.

A formal approach devised by Shen and Arvind uses term rewriting to specify and prove the correctness of coherence protocols [22]. Their technique involves showing that a system with caches and a system without caches can simulate each other. This approach lends itself to highly succinct formal proofs, yet it may be an intellectual challenge for system designers. Lamport clocks may be easier to grasp, while not lacking expressive power. It is not clear whether or how the two techniques complement each other. Term rewriting relies on an ordering of rewrite rules (each of which corresponds to an event) and, as such, may benefit from the Lamport clock technique which can order events in logical time.

There is another body of work that delves into memory consistency models that are more aggressive than sequential consistency [2, 3, 4, 6, 8, 9, 10, 21]. Handling more aggressive models leads to formalisms that are more powerful but more complex than we require (e.g., they must handle non-atomic stores). Furthermore, much of this work seeks to characterize when programs will appear sequentially consistent even when running on the more aggressive hardware, an issue that is moot for us.

Informal intuitive reasoning is more tractable and easier to understand than formal analysis, but it becomes less convincing as it becomes more informal. Moreover, the flaws in memory system designs are generally the subtle types of flaws that would be missed by high-level intuitive reasoning. Informal reasoning is often combined with extensive simulation in an effort to explore the state space for bugs in the protocol, but simulation is expensive and cannot be guaranteed to uncover every obscure bug in a protocol. Reasoning with Lamport clocks is attractive because it provides a "semi-formal" methodology that incorporates much of the thoroughness of formal analysis and much of the intuitive appeal of informal reasoning. Lamport clocks also offer the opportunity to analyze specific parts of the protocol to prove the validity of an optimization, whereas other verification techniques often require complete analysis of the system before any optimization can be validated. Lamport clocks have also been used in other research, including a paper by Neiger and Toueg [15] that uses the clocks to determine what knowledge is available to each processor in a parallel algorithm.

9 Follow-on Work

We now briefly discuss several issues that, due to space limitations, cannot not be fully explored in this paper.

Lamport clocks are well-suited towards directory protocols, because the explicit messages that are sent between processors correspond to the messages in Lamport's original scheme. In other work [18], we designed a directory protocol that is representative of several current protocols and, just like with the bus protocols discussed in this paper, we developed a timestamping scheme for the protocol. Once we had a timestamping scheme and we could create a total order of the loads and stores, the analysis and formal proof of the system were similar to those of the bus protocols.

We can also allow the memory consistency model to restrict the orders of I/O references that are allowed - a protocol that strictly obeys SC should require that I/O operations follow SC in the same way that memory references do. Let us assume that our system has memory-mapped I/O where I/O operations are treated as uncached memory operations. In order to create a Lamport order for all references (memory and I/O), we must be able to timestamp I/O operations. Currently, though, the interaction of I/O with the memory consistency model in modern multiprocessors is not well defined.

As hardware optimizations have led to better performance, they have also caused memory system designers to develop other memory consistency models that enable the programmer to understand the behavior of such systems. Lamport clocks can be applied to other consistency models, but verifying systems that observe other consistency models requires that we devise new rules for what value a load can return. We determine a Lamport timestamping scheme to produce a total order of memory operations and show that the order satisfies the requirements of the memory consistency model.

Another major issue in protocol design is ensuring the avoidance of deadlock and livelock. These situations are generally caused by either request-reply circular dependencies or finite buffering resources. Lamport clocks can be used to detect the possibility of deadlock or livelock. If no processor cannot timestamp its oldest load or store, the system is deadlocked. Livelock is a bit subtler to detect, since it requires that no processor can timestamp its oldest load or store for an indefinite amount of time.

10 Conclusions

As memory systems have become increasingly complex, it is becoming more and more difficult to verify that they implement the desired memory consistency models. We have developed a new technique that uses the notion of Lamport clocks to reason about the correctness of shared memory systems. Our technique assigns timestamps to events of interest in a shared memory system and uses the timestamps to arrange these events in a total order. This total order can then be exploited to prove the correctness of the implementation. Lamport ordering also enables us to clearly delineate the bounds of a *coherence epoch* for a block of memory in a shared memory system. Coherence epochs allow us to define the role of coherence protocols, and they give a concrete foundation for arguing about the correctness of various optimizations. We have used the Lamport clock technique to verify the implementation of sequential consistency in a number of shared memory systems.

Acknowledgments

This paper started out as an idea proposed by Mark Hill and developed into a class project for CS 757: Advanced Computer Architecture II (Multiprocessor architecture), offered by Jim Goodman in Spring'97 at the CS Dept., UW-Madison. We would like to thank Jim Goodman for his comments and encouragement.

References

[1] The Ultra Enterprise 10000 Server. http://www.sun.com/servers/datacenter/whitepapers/E10000.ps.

[2] Sarita V. Adve and Mark D. Hill. Weak Ordering—A New Definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, Seattle, Washington, May 28–31, 1990.

[3] Hagit Attiya and Roy Friedman. A Correctness Condition for High-performance Multiprocessors. In *Proceedings of the 24th Annual ACM Symposium on the Theory of Computing*, pages 679–690, May 1992.

[4] William W. Collier. *Reasoning About Parallel Architectures.* Prentice-Hall, Inc., 1992.

[5] D. Dill, A. Drexler, A. Hu, and C. Yang. Protocol Verification as a Hardware Design Aid. In *Proceedings of the IEEE International Conference on Computer Design : VLSI in Computers and Processors*, pages 522–525, 1992.

[6] Michel Dubois, Christoph Scheurich, and Faye Briggs. Memory Access Buffering in Multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Atchitecture*, pages 434–442, June

1986.

[7] Asgeir Th. Eiriksson and Ken L. McMillan. Using Formal Verification/Analysis Methods on the Critical Path in Systems Design: A Case Study. In *Proceedings of the Computer Aided Verification Conference*, Liege, Belgium, 1995. appears as LNCS 939, Springer Verlag.

[8] Kourosh Gharachorloo, Sarita V. Adve, Anoop Gupta, John L. Hennessy, and Mark D. Hill. Specifying System Requirements for Memory Consistency Models. Technical Report 1199, University of Wisconsin – Madison, December 1993.

[9] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory Consistency and Event Ordering in Scalable Shared-memory Multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[10] Phillip B. Gibbons, Michael Merritt, and Kourosh Gharachorloo. Proving Sequential Consistency of High-Performance Shared Memories. In *Symposium on Parallel Algorithms and Architectures*, pages 292–303, July 1991.

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

[12] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):241–248, September 1979.

[13] James P. Laudon and Daniel Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th International Symposium on Computer Architecture*, Denver, CO, June 1997.

[14] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, May 1990.

[15] Gil Neiger and Sam Toueg. Simulating Synchronized Clocks and Common Knowledge in Distributed Systems. *Journal of the Association for Computing Machinery*, 40(2):334–367, April 1993.

[16] Seungjoon Park and David L. Dill. An Executable Specification, Analyzer and Verifier for RMO (Relaxed Memory Order). In *Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 34–41, Santa Barbara, California, July 17–19, 1995.

[17] Seungjoon Park and David L. Dill. Verification of FLASH Cache Coherence Protocol by Aggregation of Distributed Transactions. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 288–296, Padua, Italy, June 24–26, 1996.

[18] Manoj Plakal, Daniel J. Sorin, Anne E. Condon, and Mark D. Hill. Lamport Clocks: Verifying a Directory Cache-Coherence Protocol. *10th Annual Symposium on Parallel Algorithms and Architectures*, June 1998.

[19] Fong Pong and Michel Dubois. Verification Techniques for Cache Coherence Protocols. *ACM Computing Surveys*, 29(1):82–126, March 1997.

[20] Christoph Scheurich. Access Ordering and Coherence in Shared Memory Multiprocessors. Ph.D. Dissertation CENG 89-19, University of Southern California, May 1989.

[21] Dennis Shasha and Marc Snir. Efficient and Correct Execution of Parallel Programs that Share Memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, April 1988.

[22] Xiaowei Shen and Arvind. Specification of Memory Models and Design of Provably Correct Cache Coherence Protocols. Group Memo 398, Massachusetts Institute of Technology, June 1997.

[23] A. Singhal, D. Broniarczyk, F. Cerauskis, J. Price, L. Yuan, C. Cheng, D. Doblar, S. Fosth, N. Agarwal, K. Harvey, E. Hagersten, and B. Liencres. Gigaplane: A High Performance Bus for Large SMPs. *Hot Interconnects IV*, pages 41–52, 1996.

Appendix A: Proof of Split-Transaction Protocol

A.1 Formal Protocol Specification

Table 9 illustrates the operation of the protocol with respect to changes in the A-state. The table is to be read from left to right as a sequence of events in chronological order. These events are triggered by a coherence request made by the "local" processor to a given block B. The first three columns indicate the local processor's initial state, the action it wishes to perform and the new local state immediately after the request is made on the bus. Block B can be in state A_I, A_S, or A_X, and this state is updated immediately upon a processor's request for a new access state.

The next 3 columns describe the response of all other remote processors to the request made by the local processor. The responses of the remote processors depend on their current address state. Any actions that require waiting (such as an invalidate for a block whose data has not yet arrived from memory) are placed into the message queue. Notice that a PUTS action does not generate any responses since it is not issued on the bus.

The last 2 columns of Table 9 indicate the response of memory to this action. Remote processors have priority over memory in responding to a coherence request. Any remote processor that is in the EXCLUSIVE (A_X) state will assert a common "owned" line on the bus to indicate this fact to memory. Memory reacts to the request after all remote processors have had a chance to react to it. Memory will provide data for a block only if no remote processor asserted the "owned" line. Memory has a queue to buffer messages from the bus. Table 11 describes how memory processes this queue and eventually sends data.

What we have not defined yet are the reactions performed by processors after they queue messages. The reactions of processors are defined in Table 10, and these reactions illustrate the behavior of the processors with respect to the D-states. Observe that Table 10 defines the behavior of a processor with respect to the D-state *for one action* for that block. In other words, there could be more than one simultaneous D-state for a given block. For example, P1 could get A_X access to block B, get invalidated by a remote GX (down to A_I), and then gets A_S access to B all before the data comes back for either request. In this case, there are two D-states for B. Another important point to mention about this table is the case where P1 has a request for data outstanding (say from a GX) and an invalidate waiting in its queue. The table shows an initial and final state of D_I, but P1 really has D_X permission until it decides to process the invalidate, and this could be an indefinite amount of time. This situation is not reflected in the table due to the limited benefit of doubling its size to cover these cases.

TABLE 9. Action Table

Init Local Proc State	**Local Proc Action**	**New Local Proc State**	**Remote Proc State**	**Remote Proc Action**	**New Remote Proc State**	**Did Any Proc Assert Owned?**	**Memory Action**
A_I	GX and queue own GX	A_X	A_I			Y	queue GX
		A_X				N	queue GX
		A_X	A_S	queue INV	A_I	Y	queue GX
		A_X				N	queue GX
		A_X	A_X	queue INV, assert owned	A_I	Y	queue GX
	GS and queue own GS	A_S	A_I			Y	queue GS
		A_S				N	queue GS
		A_S	A_S			Y	queue GS
		A_S				N	queue GS
		A_S	A_X	queue DWG, assert owned	A_S	Y	queue GS
A_S	UPG and queue own UPG	A_X	A_I			N	
		A_X	A_S	queue INV	A_I	N	

TABLE 9. Action Table

Init Local Proc State	Local Proc Action	New Local Proc State	Remote Proc State	Remote Proc Action	New Remote Proc State	Did Any Proc Assert Owned?	Memory Action
	PUTS	A_I	any				
A_X	WB and queue own WB	A_I	A_I			N	queue WB

TABLE 10. Reaction Table

INITIAL STATE			ACTION	FINAL STATE	
Initial State	Pending Transactions	Head of Protocol Message Queue	Processor Response	Final State	New Pending Transactions
D_I	*nothing*	own GX		D_I	data reply
	nothing	own GS		D_I	data reply
	data reply	*empty*		D_I	data reply
	data reply	INV		D_I	data reply, INV
	data reply, INV	EXCLUSIVE data reply	SEND	D_I	
	data reply, INV	SHARED data reply		D_I	
	data reply	DWG		D_I	data reply, DWG
	data reply, DWG	EXCLUSIVE data reply	SEND	D_S	
	data reply	SHARED data reply	load into cache	D_S	
	data reply	EXCLUSIVE data reply	load into cache,	D_X	
D_S	*nothing*	*empty*		D_S	
		INV		D_I	
		own UPG		D_X	
D_X	*nothing*	*empty*		D_X	
		INV	SEND	D_I	
		DWG	SEND	D_S	
		own WB	SEND	D_I	

TABLE 11. Memory Actions/Reactions

INITIAL STATE				FINAL STATE	
Outstanding WB	**Outstanding GX/GS**	**Head of Queue**	**Action**	**Outstanding WB**	**Outstanding GX/GS**
N		GX	SEND	N	
N		GS	SEND	N	
N		WB		Y	
Y		GX		Y	GX
Y		GS		Y	GS
Y	GX	data reply	SEND, commit data	N	
Y	GS	data reply	SEND, commit data	N	
Y		data reply	commit data	N	

In the case that ACTION is a GS at a processor other than p_i, causing a downgrade at p_i, we say that when a processor p_i downgrades, it sends the value of block B to itself and memory, as well as to the other processor who issued the GS request. Also, when a processor does an UPG, we consider that it receives a value from itself. Thus, corresponding to every upgrade action (GX, GS, UPG) of p_i, a value is received by p_i (possibly from itself).

The following two facts state processor responsibilities. Fact 1 says that a processor must ensure that a load returns the value of a store it just did (if any) or the value it obtained for the block otherwise. Fact 2 says that, when a processor sends a block away, it must send the values of recent processor stores to that block (if any) or the values it received.

Fact 1: Let LD-OP be a LD from word w of block B at p_i that is bound to transaction T. Let ST-OP be the last ST to word w of block B by p_i (if any) prior to LD-OP in p_i's program order.

(a) If ST-OP is also bound to transaction T, then the value loaded by LD-OP equals the result of ST-OP.

(b) Otherwise, the value loaded by LD-OP equals the value of word w of block B received by p_i in response to transaction T.

Fact 2: Suppose that as a result of transaction T_2, p_i sends away block B. Let T be the most recent transaction at p_i prior to T_2 (in real time) that caused p_i to receive block B. Then, the value of word w of block B sent by p_i in response to T_2 is the last ST to word w of block B in p_i's program order that is bound to T, if any. If no ST to word w of block B is bound to T, then the value of word w of block B sent by p_i is the value received by p_i in response to transaction T.

Note: As long as p_i sends the correct value for each word w of block B, then it is not required to perform all bound LD operations on block B before invalidating that block.

A.2 Timestamping Claims

We now make assertions about the timestamping in the following claims.

Claim 2: For a transaction T on block B,

(a) If T is not a PUTS, the timestamps of the downgrades associated with T are equal to the timestamp of the upgrade associated with T.

(b) If T is not a PUTS, the timestamp of the upgrade associated with T is less than the timestamp of the upgrade associated with any transaction on block B occurring after T in the transaction serialization order.

(c) If T is a PUTS, then let T_0 be the last transaction at that node, prior to T in the transaction serialization order, that caused the block's A-state to change to A_S. Let T_1 be any transaction for block B that occurs later than T_0 in the transaction serialization order. Then, the timestamp of T is less than the timestamp of T_1.

Note: The proof of Claim 2(b) relies on Claim 1 and the fact that the Lamport order of transactions is the same as their order in real time.

Claim 3: Every LD/ST operation on block B at processor p_i is bound to the most recent (in Lamport time at p_i) transaction on block B that affects p_i.

The proof of Claim 3 uses the fact that binding of operations is done in program order in real time. This real-time property of the protocol can be relaxed somewhat while maintaining the correctness of this claim. This issue is discussed and the claim is proved in Appendix B.

A.3 Proof of Sequential Consistency

By construction, the Lamport ordering of LDs and STs within any processor is consistent with program order. Therefore, to prove sequential consistency, it is sufficient to show that the value of every load equals the value of the most recent store.

We frame the proof of sequential consistency in terms of coherence epochs. An epoch is simply a time interval [t1,t2) during which a node has access to a block. A SHARED or EXCLUSIVE epoch for block B at node N starts at time t_1 if a transaction with timestamp t_1 (at N) implies that N's A-state for block B changes to A_S or A_X respectively. The epoch ends at time t_2, where t_2 is N's timestamp of the next transaction on block B that implies a change in A-state at N. We build up to the proof of sequential consistency using the two timestamping claims of Section A.2.

The proof is constructed from three lemmas whose proofs can be found in Appendix B. Lemma 1 shows that two processors cannot have read-write permission to the same block at the same (Lamport) time, nor can any processors have read-only permission if any processor has read-write permission. Lemma 2 states that processors do LDs and STs within appropriate epochs. Finally, Lemma 3 shows that the "correct" block value is passed among processors and the memory between epochs.

Lemma 1: EXCLUSIVE epochs for block B do not overlap with either EXCLUSIVE or SHARED epochs for block B in Lamport time.

Lemma 2:

(a) Every LD/ST operation on block B at p_i is contained in some epoch for block B at p_i and is bound to the transaction that caused that epoch to start.

(b) Furthermore, every ST operation on block B at p_i is contained in some exclusive epoch for block B at p_i and is bound to the transaction that caused that epoch to start.

Lemma 3: If block B is received by node N at the start of epoch $[t_1,t_2)$, then each word w of block B equals the most recent store to word w prior to t_1 or the initial value in memory, if there is no store to word w prior to global time t_1.

The proof of the Main Theorem shows how sequential consistency follows from the lemmas.

Main Theorem: The value of every load equals the value of the most recent store or the initial value, if there has been no prior store.

Proof: Consider a LD at processor p_i. Let the LD be bound to transaction T_1 which has timestamp t_1 at processor p_i. There are two cases.

The first case is that the most recent ST has global timestamp at least t_1. In this case, from Lemmas 1 and 2, this ST is also at processor p_i and is bound to transaction T_1. Therefore, by Fact 1 (a), the value of the LD equals the value of the most recent ST.

The second case is that the most recent ST has global timestamp less than t_1. In this case, by Lemma 2, no ST prior to this LD is bound to transaction T_1. Therefore, by Fact 1 (b), the value of the LD equals the value received by p_i in response to transaction T_1. By Lemma 3, this value equals the value of the most recent ST or the initial value if there has been no prior store.

Appendix B: Proofs of Claim 3 and the Lemmas

Claim 3: Every LD/ST operation on block B at processor p_i is bound to the most recent (in Lamport time at p_i) transaction on block B that affects p_i.

Proof: Let OP_2 be a LD or ST operation on block B with global timestamp t_2. Since OP_2's timestamp is t_2, OP_2 cannot be bound to a transaction with timestamp greater than t_2. Let T_1 be the transaction on block B with the largest timestamp, say t_1, at p_i such that $t_1 \leq t_2$. We need to show that OP_2 is not bound to a transaction occurring earlier than T_1; hence OP_2 must be bound to T_1.

Let OP_1 be the earliest LD/ST operation (not necessarily to block B) in p_i's program order with the global timestamp t_2. Note that OP_1 may equal OP_2. Also, since OP_1 is the first OP with global timestamp t_2, OP_1 must be bound to the transaction with timestamp t_2 at p_i. The order in which changes in A-state at a processor are written in real time is the same as the Lamport ordering of the corresponding transactions at that processor. Hence, the value of the A-state for block B at the real time that OP_1 is bound must be the value implied by a transaction on block B occurring no earlier than T_1. Since OP_2 is bound in real time no later than OP_1 is bound, it cannot be bound to a transaction occurring earlier than T_1, as required.

Comment: the proof of Claim 3 uses two facts about the protocol relating real time to Lamport time: (a) the order in which changes in A-state at a processor are written in real time is the same as the Lamport ordering of the corresponding transactions at that processor, and (b) binding occurs sequentially in real time. However, the protocol can be relaxed while maintaining the correctness of Claim 3. For example, suppose that the A-states are updated periodically (using queues to order pending updates) and that during an update of transactions with timestamps in the range $[t_1,t_2)$, the binding process is suspended. The order in which the A-states are updated need not agree with the order of the corresponding actions, as long as at the end of the update period, the A-state value of each block equals that implied by the most recent transaction prior to that with timestamp t_2. Once the A-states are up to date, binding of LD/STs can be resumed. Binds of the next contiguous group of LD/ST operations on blocks for which the A-state is set appropriately can be performed out of order, thus relaxing the real time ordering assumption for binds, as long as potential changes in A-state are being queued until the binding process is again suspended.

Lemma 1: EXCLUSIVE epochs for block B do not overlap with either EXCLUSIVE or SHARED epochs for block B in Lamport time.

Proof: Let $[t_1,t_2)$ be an EXCLUSIVE epoch for block B at node N. Let transaction T_1 cause the epoch to begin. We claim that no node has an epoch for block B that overlaps with $[t_1,t_2)$.

We first argue that no epoch for block B that starts prior to time t_1 overlaps with $[t_1,t_2)$. By Claim 2 (b), the start of such an epoch E would have to result from a transaction occurring before T_1 in the serialization order. Therefore, the end of epoch E would have to result from some transaction T_0 on block B occurring no later than T_1 (possibly $T_0 = T_1$). If T_0 is not a PUTS, then Claim 2 (a) ensures that the end of epoch E must be less than or equal to the timestamp of T_0 at a unique node, say N_2, that upgrades its A-state as a result of T_0. Also, by Claim 2 (b), the timestamp of T_0 by N_2 must be less than the timestamp of T_1 by N. If T_0 is a PUTS, then by Claim 2 (c), the timestamp of the PUTS (which is the end of the epoch) is less than the timestamp of T_1. Hence, in any case, E ends in Lamport time before $[t_1,t_2)$ starts.

Clearly, the only epoch starting at time t_1 is at node N, since N is the only processor whose A-state is not A_I after transaction T_1. To complete the proof, we note that the next transaction, say T_2, on block B after T_1 must be assigned timestamp t_2 by N. If node N_2 upgrades its A-state as a result of T_2, Claim 2 (a) ensures that N_2's timestamp of T_2 must be greater than t_2. Therefore, by Claim 2 (b), if an epoch E starts as a result of transaction T_2 or some transaction later than T_2, E must start at a time greater than t_2, as required.

Lemma 2: (a) Every LD/ST operation on block B at p_i is contained in some epoch for block B at p_i and is bound to the transaction that caused that epoch to start. (b) Furthermore, every ST operation on block B at p_i is contained in some EXCLUSIVE epoch for block B at p_i and is bound to the transaction that caused that epoch to start.

Proof: Let OP be a LD/ST on block B with global timestamp t_2. By Claim 3, OP is bound to the most recent transaction at p_i no later than t_2, say T_1, that affects block B of p_i. Let t_1 be p_i's timestamp of T_1. Part (a) of Lemma 2 then follows for the following reasons: Since OP is bound to T_1, T_1 must imply that p_i's A-state for block B changes to A_S or A_X and so an epoch for block B at p_i starts at time t_1. Moreover, since T_1 is the most recent transaction no later than t_2 that affects block B of p_i, the epoch starting at t_1 must end at some time later than t_2. Therefore, OP is contained in some epoch for block B at p_i and is bound to the transaction that caused that epoch to start. Part (b) follows from the further observation that if OP is a ST then T_1 must cause an EXCLUSIVE epoch to start at p_i.

Lemma 3: If block B is received by node N at the start of epoch $[t_1,t_2)$, then each word w of block B equals the most recent store to word w prior to t_1 or the initial value in the memory, if there is no store to word w prior to global time t_1.

Proof: We prove the claim for all nodes by induction on epoch starting time t_1. The basis case is the first action that causes block B to be sent. In this case the block is sent from the memory and equals the initial value of the block in the memory.

Suppose that the claim is true for all epochs with starting time less than t_1, and suppose that block B is sent from node N_0 to node N_1 in response to transaction T_1, which has timestamp t_1 at N_1. First, suppose that N_0 is not equal to N_1. Let transaction T_0 be the most recent action on block B prior to T_1 in serialization order. Since N_0 sends block B in response to T_1, T_0 must be cause an EXCLUSIVE epoch to start at N_0 and therefore affects N_0. Let T_0 have timestamp t_0 at N_0. From Claim 2, N_0's EXCLUSIVE epoch for block B starting at time t_0 must end prior to time t_1. Moreover, since T_0 and T_1 are consecutive transactions on block B in serialization order, there is no epoch at any processor between the time that N_0's epoch ends and N_1's epoch begins at time t_1.

We consider two cases. The first case is that the last ST to word w of block B prior to time t_1 is actually prior to t_0. Therefore, no STs to word w of block B are bound to T_1. By Fact 2, the value W_0 of word w of block B sent by N_0 is the value received by N_0 in response to T_0. By the induction hypothesis, W_0 equals the value of the most recent store to word w of block B prior to time t_0 or the initial value of word w in the memory, if no prior store. Therefore, the value sent by N_0 equals the value of the most recent store or the initial value in the memory, if no prior store.

The second case is that the last ST to word w of block B prior to time t_1 occurs after time t_0. By Claim 3 and Lemma 2 (b), such STs must be done by node N_0. By Fact 2, in this case the value of word w of block B sent by N_0 in response to T_1 is the last ST to word w of block B in p_i's program order that is bound to T_0. Moreover, the last ST bound to T_0 has global timestamp less than t_1. Therefore, the value sent by N_0 equals the value of the most recent store to word w of block B.

This completes the proof of Lemma 3 in the case that, in response to T_1, block B is sent by a node other than p_i.

The situation in which $N_0 = N_1$, (i.e., in response to T_1, the value of block B is sent from p_i to itself) is similar, but only the first case above can arise.