Lamport Clocks: Verifying a Directory Cache-Coherence Protocol

Manoj Plakal, Daniel J. Sorin, Anne E. Condon, Mark D. Hill
Computer Sciences Department,
University of Wisconsin - Madison,
1210 West Dayton Street, Madison, WI 53706, USA.
{plakal, sorin, condon, markhill}@cs.wisc.edu

Abstract

Modern shared-memory multiprocessors use complex memory system implementations that include a variety of non-trivial and interacting optimizations. More time is spent in verifying the correctness of such implementations than in designing the system. In particular, large-scale Distributed Shared Memory (DSM) systems usually rely on a directory cache-coherence protocol to provide the illusion of a sequentially consistent shared address space. Verifying that such a distributed protocol satisfies sequential consistency is a difficult task. Current formal protocol verification techniques [18] complement simulation, but are somewhat non-intuitive to system designers and verifiers, and they do not scale well to practical systems.

In this paper, we examine a new reasoning technique that is precise and (we find) intuitive. Our technique is based on Lamport's logical clocks, which were originally used in distributed systems. We make modest extensions to Lamport's logical clocking scheme to assign timestamps to relevant protocol events to construct a total ordering of such events. Such total orderings can be used to verify that the requirements of a particular memory consistency model have been satisfied.

We apply Lamport clocks to prove that a non-trivial directory protocol implements sequential consistency. To do this, we describe an SGI Origin 2000-like protocol [12] in detail, provide a timestamping scheme that totally orders all protocol events, and then prove sequential consistency (i.e., a load always returns the value of the "last" store to the same address in timestamp order).

1 Introduction

Modern high-performance multiprocessor systems are becoming increasingly complicated. System designers have proposed the use of a variety of complex and interacting optimizations to improve performance. This trend ignores the difficulty of verifying that the system still behaves correctly. Currently, industrial product groups spend far more time in verifying their system than in actually designing and optimizing the system.

A case in point is the design of large-scale cache-coherent shared-memory systems that are built using distributed-memory nodes with private caches that are connected by a general interconnection network. Such hardware Distributed Shared Memory (DSM, [19]) systems operate by sharing memory through a scalable *directory coherence protocol.* A directory protocol must present a consistent view of memory [1, 4] to the processing nodes, with sequential consistency (SC) [11] being a common requirement. The requirements of SC (quoting Lamport [11]) are:

`'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.'`

The directory itself is a data structure whose entries record, for every block of memory, the state (i.e., cache access permission) and the identities of the processors which have cached that block. The directory is often distributed along with the memory, as is the case in the protocol that we will verify. Nodes exchange messages with each other and with the directory to coordinate accesses to each block of memory.

Determining which messages are necessary requires delving into the subtleties of directory protocols. In our protocol, for example, a processor's message writing a block back to memory must be acknowledged so that we can distinguish the common case from the race condition where the directory has already given permission for the block to another processor and there is a "forwarding" message in flight.

Current protocol verification techniques based on model-checking and state-space search [18] do not seem to be intuitive to system designers, and they do not scale well to systems of a practical size. Alternatively, protocol optimizations are sometimes justified with imprecise informal arguments and the results of protocol simulations. We propose a new verification technique that is both precise (unlike informal arguments) and intuitive (unlike formal arguments). We have applied the technique to a non-trivial directory protocol that is similar, though not identical, to the protocol used in the SGI Origin 2000 [12].

Most memory consistency models, including SC, are defined in terms of a hypothetical ordering of all memory references. We propose to construct such an ordering by timestamping protocol events that occur in the system. Our timestamping scheme makes modest extensions to Lamport's logical clock scheme [10]. This scheme was used to maintain global time and implement mutual exclusion in a distributed system. We attach logical clocks, which are merely conceptual devices, to various parts of a multiprocessor system. These logical clocks are used to assign logical timestamps

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

SPAA 98 Puerto Vallarta Mexico
Copyright ACM 1998 0-89791-989-0/98/ 6...$5.00

to the *protocol transactions* (i.e., actions that cause processors to change their access permissions to blocks of data), and *memory operations* (loads (LDs), and stores (STs)) that occur while a protocol operates.

The timestamps are split into three positive integral components: global time, local time, and processor ID. Such 3-tuple timestamps can be totally ordered using the usual lexicographic ordering. Global timestamps order LD and ST operations relative to transactions "as intended by the designer." This is made precise in two *timestamping claims* later in our paper. One of these claims is that for every LD and ST operation on a given block, proper access is ensured by the *most recent* transaction on that block in Lamport time. (in contrast, in real time, a processor may perform a LD operation on a block after it has answered a request to relinquish the block [20]). Roughly, the other claim is that, in logical time, transactions are handled by processors in the order in which they are received by the block's directory. (In contrast, in real time, a processor may receive transaction-related messages "out of order").

Local timestamps are assigned to LD and ST operations in order to preserve program order in Lamport time among operations that have the same global timestamp. Local timestamps are not needed for transactions. They are used to enable an unbounded number of LD/ST operations between transactions. Processor ID, the third component of a Lamport timestamp, is used as an arbitrary tie-breaker between two operations with the same global and local timestamps, thus ensuring that all LD and ST operations are totally ordered.

Sequential consistency is established in a sequence of lemmas, using the concept of *coherence epochs*. An epoch is an interval of logical time during which a node has read-only or read-write access to a block of data. The life of a block in logical time consists of a sequence 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 [23], we have proved the correctness of a bus protocol using the same proof structure; the proofs of the lemmas for the bus protocol are exactly as for the directory protocol of this paper, and only the proofs of the timestamping claims differ.

The rest of this paper is organized as follows. Section 2 is a specification of the directory protocol. In Section 3, we describe the details of the timestamping scheme and prove that the protocol obeys SC. Section 4 discusses related work in protocol verification. Section 5 summarizes our contributions and outlines future work that can be done with our verification technique.

2 Specification of a Directory-Based Coherence Protocol

2.1 Our Target Multiprocessor System

Our target multiprocessor system (shown in Figure) consists of a number of processing nodes and directory nodes connected by an interconnection network. Each processing node consists of a single processor, one or more levels of cache, and a network interface. Each directory node consists of a *directory* that is used to store protocol state information about a range of blocks of memory. These memory blocks are also allocated storage at the directory nodes. Blocks may be present in a processor's cache in one of three states: invalid, read-only or read-write.

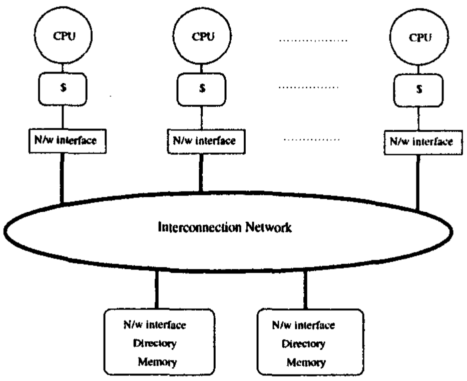


FIGURE 1. The Target Multiprocessor System

Both the caches and the directory are operated by finite-state-machine controllers which interact by exchanging messages in order to implement a coherence protocol. Notice that this system configuration subsumes the case where each directory node is co-located with a processing node and the directory controls access to the local memory owned by that node. We do not assume that the interconnect guarantees any kind of ordering of messages sent or received between nodes. We do assume that the network guarantees eventual and reliable delivery of all messages.

2.2 Preliminaries

Our directory protocol is inspired by Culler et al.'s description [4] of the SGI Origin 2000's protocol [12]. We would like to emphasize however, that our protocol differs in several respects from those in the above descriptions. Directory schemes vary in the manner in which they organize and allocate storage for the directory. We will assume that each memory block along with its directory entry is allocated storage in the local memory of some fixed "home node" for that block. For our purposes, a directory entry consists of a *block state* and a set CACHED of node IDs. This entry can encode one of the following 6 states:

- *Idle*: No node has a valid cached copy of this block. It is only valid at the home's memory.
- *Shared*: The block is currently cached at one or more nodes in the read-only state. CACHED contains a node's ID if and only if that node had requested a read-only cached copy of this block.
- *Exclusive*: The block is currently cached in the read-write state at exactly one node. CACHED contains this node's ID.
- *Busy-Shared*: The block had been in the Exclusive state and another node has requested the block in the read-only state. The directory is now in the process of transferring the block. CACHED contains the ID of the node requesting the block in the read-only state. Once in *Shared*, CACHED will re-include the ID of the original owner.
- *Busy-Exclusive*: This is similar to the *Busy-Shared* case above, with the only differences being that the new requesting node has requested a read-write copy of the block, and that once in *Exclusive*, CACHED will contain only the ID of the new requester.

- *Busy-Idle*: The block is in the process of going from the *Exclusive* state to the *Idle* state. More information about the need for this state can be obtained from the description of the Writeback request in Section 2.3.

When busy states behave similarly, we will use *Busy-Any* to refer to a block whose state is *Busy-Shared, Busy-Exclusive*, or *Busy-Idle*.

Our protocol is invalidation-based and allows either a single writer or one or more readers for each block of memory. We will use the following terms in our protocol description:

- The *requesting* node of a block is the node which issues a request (to the home node) for obtaining that block in the shared or exclusive state.
- The *owner* node of a block is either the home node (if no node has cached the block with read-write access) or the node with read-write access.

2.3 Protocol Specification

We will now informally describe how the protocol is used by a directory controller to handle requests sent to it by requesting nodes. Note that we have decoupled the generation of coherence requests from processor events. For instance, a *Get-Shared* request could be generated even before a processor suffers a read miss in its cache for that block. This may happen, for example, if a processing node is trying to tolerate memory latencies by prefetching blocks into its cache before it references them. The protocol supports the requests in Table 1.

For each request, there are several possible *transactions* which depend on the directory state. Including NACKed transactions (of which there are three), there are 14 distinct transactions (which are listed below) in total. Transactions 13 and 14 define transactions that correspond to a pair of requests, where one request is a Writeback and the other is a *Get-Shared, Get-Exclusive*, or *Upgrade* request. All other transactions involve only one request.

TABLE 1. Protocol requests

Request	Current Cache Permission	Desired Cache Permission
Get-Shared	invalid	read-only
Get-Exclusive	invalid	read-write
Upgrade	read-only	read-write
Writeback	read-write	invalid

Transactions:

Get-Shared: The requester sends a *Get-Shared* request to the home. What happens next depends on the state of the block in the home directory:

1. *Idle*: The home clears CACHED and adds the requesting node's ID to CACHED. It then sends the block to the requester and sets the state to *Shared*. The requester loads the block in the read-only state into its cache[1].
2. *Shared*: The home adds the requesting node's ID to CACHED and sends the block to the requester. The requester loads the block in the read-only state into its cache.
3. *Exclusive*: The home changes the state to *Busy-Shared*, removes the current owner's ID from CACHED and adds the requesting node's ID to CACHED. It then forwards the request (along with the identity of the requester) to the current owner of the block. The owner sends the block directly to the requester, downgrades the status of the block in its cache to read-only and sends an update message (with the block) to the home. The home then stores the block to local memory, adds the former owner's ID to CACHED and changes the state to *Shared*. The requester loads the block into its cache in the read-only state.
4. *Busy-Any*: The home sends the requester a negative acknowledgment (NACK), indicating that the requester should try again later.

Get-Exclusive: Again, the cases depend on the directory state:

5. *Idle*: The directory changes the state to *Exclusive*, clears CACHED and adds the requesting node ID to CACHED. It then sends the block to the requester, which loads the block in the read-write state in its cache.
6. *Shared*: All cached copies must be invalidated. The home makes a list of the nodes corresponding to the node IDs in CACHED and then clears CACHED. It then changes the state to *Exclusive* and adds the requesting node's ID to CACHED. The home sends invalidations (containing the identity of the requester) to the nodes in the list it constructed. It then sends the number of invalidations, along with the block, to the requester. Each of the sharers invalidates its copy of the block and sends an acknowledgment to the requester. The requester waits until it receives all acknowledgments before loading the block in the read-write state into its cache.
7. *Exclusive*: The home sets the directory state to *Busy-Exclusive*, removes the current owner's ID from CACHED and adds the requesting node's ID to CACHED. It then forwards the request (along with the identity of the requester) to the owner. The owner invalidates its copy of the block, sends an acknowledgment with the block to the requester and sends an update message to the home. The home then changes the state to *Exclusive*. The requester loads the block in the read-write state into its cache.
8. *Busy-Any*: The request is NACKed.

Upgrade: As before, the cases depend on the directory state, but we now have to tackle a number of race conditions:

- *Idle*: This is impossible. This situation, and three other situations which will be encountered later, will be shown to be impossible in Appendix B.

9. *Shared*: This is handled just like the *Shared* case for the *Get-Exclusive* request, the only difference being that the home does not need to send the block in its reply to the requester. The requester then changes the state of the block in its cache from read-only to read-write.
10. *Exclusive*: This means that another node's *Get-Exclusive* or *Upgrade* request must have beaten this *Upgrade* request to the home and the home must have sent an invalidation to the current requester. The home NACKs the request, forcing the requester to re-try with a *Get-Exclusive* request.

[1] Here and later, our intended meaning is that the requester waits until the block arrives, after which it loads the block into its cache in the appropriate state.

11. *Busy-Any*: The request is NACKed.

Writeback: The owner sends a *Writeback* request to the home along with the block. One expects that the directory will be in state *Exclusive* with CACHED pointing to the requester. Some of the subtleties of directory protocols, however, are revealed by the other cases that race conditions make possible:

- *Idle*: Impossible. See Appendix B.
- *Shared*: Impossible. See Appendix B.

12. *Exclusive*: The home stores the block to memory, changes the state to *Idle* and sends an acknowledgment to the (former) owner. The owner then changes the state of the block in its cache to invalid.
13. *Busy-Shared*: We have a race condition here. The requester's ID is not present in CACHED (proved in Appendix B). Instead, another node's ID is present. This means that this other requester has made a *Get-Shared* request to the home and the home has forwarded the request to the current owner (present requester). The forwarded request and the write-back have managed to pass each other in the network. Our protocol resolves this race condition by combining the two requests. When the home receives the write-back, it changes the state from *Busy-Shared* to *Shared.* It also sends the block returned in the write-back request to the new requester, as well as a special "busy" write-back acknowledgment to the former owner which tells it to ignore the forwarded request. The owner waits for an acknowledgment from home, buffering any forwarded requests it receives. When it receives a "busy" acknowledgment, it sets the state of the block in its cache to invalid and discards the buffered forwarded request (if any) or remembers to ignore the first forwarded request it receives (and only after it receives such a request can it generate a request of its own).
14. *Busy-Exclusive*: Similar to the *Busy-Shared* case, but with two race conditions distinguished by which node's ID is present in CACHED:

    (a) The requester's ID is not present. This case is similar to the race condition in the *Busy-Shared* case above.

    (b) The requester's ID is present in CACHED. This means that the requester had originally made a *Get-Exclusive* request to the home which caused the former owner to send the block to the requester and send an update message to the home. Subsequently, the requester's writeback beat the update message to the home. The home writes the block sent by the requester into memory, clears CACHED, sends an acknowledgment to the requester and changes its state to *Busy-Idle*. When the update message finally arrives, the home goes to the *Idle* state. The requester then sets the state of the block in its cache to invalid.

- *Busy-Idle*: Impossible. See Appendix B.

2.4 Processor Behavior Requirements

We also need to specify the behavior of a requester/owner with regard to the requests they can generate and the responses they need to provide to external requests :

- We assume that a node maintains at most one outstanding request for each block. Multiple requests for different blocks are allowed.
- NACKed requests need to be re-tried. The new request needs to take into account the *current* state of the block and the type of access to be performed. A re-tried request is equivalent to a new network transaction and does not continue to use the resources of the original transaction (which are freed).
- Invalidations and forwarded requests for a block should be buffered until the current outstanding *Get-Shared, Get-Exclusive, Upgrade,* or *Writeback* transaction for that block, if any, has been completed. For example, a node may have requested a read-only copy of a block, and it may receive an invalidation before it receives the reply to its request.
- Consider a LD/ST operation to block B, call it OP, of some fixed processor p_i. If permission to perform OP was obtained via transaction T, we say that OP is *bound* to transaction T. To ensure forward progress, we require that if transaction T is issued in order to obtain permission to bind OP, then upon completion of T (assuming it is not NACKed), OP is bound to T, even if an invalidation arrived in the meantime.
- We assume that in the protocol, if OP_1 appears before OP_2 in p_i's program order, then the real time at which OP_1 is bound is less than or equal to the real time at which OP_2 is bound. There is a discussion in Appendix A about when this real time requirement can be relaxed.
- The following two facts give 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 (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.

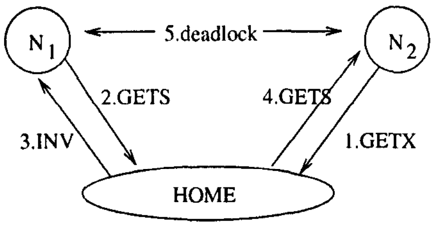
Fact 2: Suppose that as a result of transaction T_2, p_i sends away block B. Let T be the most recent transaction at p_i prior to T_2 (in real time) that caused p_i to receive block B. Then, the value of word w of block B sent by p_i in response to T_2 is the last ST to word w of block B in p_i's program 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 complete all bound LD operations on block B before invalidating that block. Also, in order that Facts 1 and 2 apply to the case that T or T_2, respectively, is a Get-Shared at a processor other than p_i, we say that in this case processor p_i sends the value of block B to itself as well as to the other processor who issued the Get-Shared request. Also, when a processor does an Upgrade, we consider that it receives a value from itself. Thus, corresponding to every transaction of p_i to which an operation is bound, a value is received by p_i (possibly from itself).

2.5 Extending the Protocol to allow Silent Eviction

Most protocols allow a node to silently evict a read-only block from its cache without notifying the Home. The protocol that we have described in the previous subsections does not include such silent evictions, which we shall refer to as *Put-Shared*. The use of *Put-Shared* combined with buffering of invalidation messages leads to a rather subtle race condition. Consider the scenario depicted in Figure 2. This involves a node N_1 which initially had a block in the read-only state in its cache, evicted it silently and then proceeded to issue a *Get-Shared* request for the same block. Meanwhile, another node N_2 has issued a *Get-Exclusive* request for the same block and this has beaten the *Get-Shared* request to the Home which proceeds to send an invalidation to N_1.

FIGURE 2. Deadlock produced by allowing Put-Shared



The invalidation is buffered and not responded to before a response to N_1's *Get-Shared* request is received. Meanwhile, the Home forwards the *Get-Shared* request to N_2 which buffers the request and does not respond to it until it receives an acknowledgment from N_1. So now, we have deadlock with N_1 and N_2 each waiting for a message from the other before they can proceed.

The basic problem is that of a node which had a block in the read-only state, silently evicted it and then re-requested it in the read-only state. If an invalidation message now arrives at the node, does the invalidation apply to the previous "incarnation" of the block (i.e., before it was silently evicted) or does it apply to the block that will be sent in response to the outstanding request?

There are two possible ways in which we can solve this problem. These methods differ in the way they process invalidations received for blocks for which a node has an outstanding request. The node can either buffer the invalidation (as we do) or apply it immediately (as in the SGI Origin 2000 and DASH). These alternatives are described in [4].

One solution is to still allow messages to be buffered until outstanding requests are completed (or NACKed). The deadlock is broken by N_2 which can recognize this situation when it occurs i.e., when it receives a forwarded request from the very node from which it is to receive an acknowledgment. In this case, it can treat the forwarded request as an implicit acknowledgment and proceed to bind its stores. N_2 can then send the data to N_1 directly, telling it to ignore any invalidation that has been buffered by N_1. N_2 also sends an update message to the home, as in the normal operation of the protocol. In case no invalidation has been received yet, N_1 has to remember to drop the first invalidation that it sees for that block, and furthermore it cannot generate any new request for that block until it receives this invalidation.

The other solution is not to buffer invalidation messages until any outstanding requests complete, but to apply them immediately, thus treating them as NACKs. The requesting node will also have to remember to drop the reply to its original request, and then make a new request. This is the solution adopted in the SGI Origin 2000 and the DASH (as confirmed in [13]). Forward progress considerations are met by the use of higher-level mechanisms that detect a possible lack of progress and take over with corrective measures.

We have decided to adopt the first approach in our protocol. The additions to the protocol are (1) a *Put-Shared* action, (2) the deadlock detection done by a node (as described above) that requests Exclusive permission for a block from the Home and (3) acknowledgment of all invalidations received for a block that is invalid in the cache. The *Put-Shared* action can be performed by any node that has a read-only copy of a block. After performing this action, the block's state in the cache changes to invalid. Note that we call this an action rather than a transaction (such as *Get-Shared*, *Get-Exclusive*). These terms will be explained further in the next section where we provide a formal proof of correctness of the protocol which includes these additions.

3 A Timestamping Scheme and a Proof of Correctness

3.1 Notation and Basic Properties of the Protocol

In this section, we define some notation used to reason about the protocol. First, we classify all coherence activity that occurs in our system as being related to either *transactions* or *actions*. Transactions are "global" events initiated by a *Get-Shared*, *Get-Exclusive*, *Upgrade* or *Writeback* request for a block sent by a node, and involve the Home node of that block and perhaps one or more other nodes. Actions are local events that are private to a node and which other nodes do not need to know about. Currently, the *Put-Shared* action is the only example of an action in our protocol.

Next, we define the notion of a per-block Address-state, or A-state, of a node. The A-state of a block at a node is used to capture the Home node's view of the state of the block at that node after the node has performed, or participated in, a sequence of (non-NACKED) transactions. In particular, the A-state of a block at a node will reflect the change in coherence status implied by a message sent from the Home (and possibly other nodes). This change could have been brought about in response to a request made by that node, or through an invalidation or forwarded request sent by the Home. A node's A-state for a block B is defined to be one of A_I, A_S, or A_X (the intended meanings are "invalid", "shared" and "exclusive" respectively). The A-state is set to A_I when the node receives an invalidation or a forwarded *Get-Exclusive*, or an acknowledgment for its own *Writeback* request. The A-state is set to A_S when the node receives a downgrade, or a response to its own *Get-Shared* request. Finally, the A-state is set to A_X when the node receives a response to its own *Upgrade* or *Get-Exclusive* request, along with all associated invalidation acknowledgments. As a special case, when the protocol performs the deadlock detection described in Section 2.5, we define the A-state of the node receiving the invalidation to change from A_S to A_I and then to the A-state appropriate to its original request, when it receives a block from its former owner. The directory entry for a block also has an A-state which is one of A_I, A_S or A_X (when the busy bit is not set), according as the directory entry state is *Exclusive*, *Shared* or *Idle* respectively. This allows us to refer uniformly to the A-state of a node, where a node could refer to either a processor or a directory.

Note that *actions* do not change the A-state. So if a processor's A-state for a block is A_S, it remains A_S even after the processor performs a *Put-Shared* and the block is invalid in the cache. Hence, the A-state is not just a synonym for the processor's cache state. It is important to realize that the A-state is a conceptual device that is used to reason about the protocol. In a hardware implementation of this protocol, the cache controller would use the actual cache state to determine future actions, and not the A-state.

Transactions on a given block are serialized by the block's directory. Hence, we can speak about a sequence of transactions on the same block where the ordering is implied by their serialization at the directory. For each node N, a sequence of t transactions on block B (where the order among transactions is seen at the Home) 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 i^{th} transaction in the sequence "affects" N and that the transaction "implies that N's A-state for block B change from $A_{(i-1)}$ to $A_{(i)}$". For example, if nodes N_1 and N_2 start with an initial state of A_I, and the sequence of transactions at the Home is N_1's *Get-Shared*, N_2's *Get-Exclusive* and N_2's *Writeback*. Then the sequence of A-states for N_1 and N_2 is A_I, A_S, A_I, A_I and A_I, A_I, A_X, A_I respectively. The *Get-Exclusive* affects both nodes as well as the directory node, while the *Writeback* affects N_2 and the directory. In the special case that a node is the directory, we say that it is also affected by all transactions resulting from *Get-Shared* requests, even though no change in the A-state at the directory may be implied by such a transaction.

Each transaction 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. 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. In the special case that node N is the directory, we say that N's A-state "downgrades" as a result of every *Get-Shared* transaction, even though its A-state may not be changed by the transaction. On each transaction, exactly one node upgrades and zero or more nodes downgrade.

The definitions of "affects" and "implies" in the previous two paragraphs depend only on the sequence of transactions on block B at B's directory. In Claim 2 below, we show that the protocol specification ensures that, at every node, the actual sequence of changes to the A-state for block B occurs in the order implied by the serialization of the transactions at B's directory, even though messages on successive transactions may be received out of order by a node.

Claim 1: For each transaction T, a message is sent to every processor affected by T. Also, if node N upgrades as a result of T, exactly those nodes that are affected by transaction T (other than N) send a message to N.

Proof: Claim 1 can be proved true for all transactions T by induction on serialization order of the transactions at the block's directory.

Claim 2: The sequence of A-state changes on block B at a node occurs in real time in the order implied by the serialization of transactions on block B at its directory.

Proof: A case-by-case proof of Claim 2 can be found in Appendix A.

3.2 Timestamping in a Directory Protocol

Imagine that each processor has a global clock that is updated in real time. In addition, each directory entry has a global clock. The clocks are used to associate global timestamps with LD and ST operations and with transactions (thus defining *coherence epochs*). Distinct nodes may assign distinct timestamps to the same transaction. We only use global clocks for transactions (i.e., to delineate epochs); local time will be used to distinguish LD/ST operations within the same epoch. Note that we do not timestamp the *Put-Shared* action.

Let us first consider the timestamping of transactions. All of the following applies to a fixed block B. Suppose that a transaction T implies a downgrade at node N. At the moment that its A-state changes, N increments its global clock by 1 and assigns the updated time to that transaction. Suppose that a transaction T implies an upgrade at node N. At the moment that N's A-state changes, N updates its clock to equal

1 + max{N's current clock time, timestamps assigned to T by all nodes other than N that are affected by T},

and assigns the updated time to transaction T. By Claim 1, exactly those nodes other than N that are affected by transaction T send a message to N. The above definition of timestamp is well-defined because N does not upgrade its A-state until it has received a message from all other nodes that are affected by transaction T. We can think of each affected node as sending its timestamp of T along with its message to N. Thus at the moment that N upgrades its A-state, it has all of the information needed to timestamp transaction T.

Claim 3: For a transaction T on block B,

(a) The timestamps of the downgrades associated with T are less than or equal to the timestamp of the upgrade associated with T.

(b) The timestamp of the upgrade associated with T is less than the timestamp of the upgrade associated with any transaction T' on block B occurring after T in the serialization order at the directory, so long as one of T or T' is a Get-Exclusive or Writeback

Proof: Claim 3 can be proved true for all transactions T by induction on the serialization order of the transactions at the block's directory. The proof of Claim 3(b) relies on Claim 2 and the fact that the Lamport order of transactions (as defined by their global timestamps) is the same as their order in real time at the directory

Now, we need to assign timestamps to LD and ST operations. If LDs and STs were always performed in program order immediately after binding, one could simply timestamp an operation by the current time of the processor's global clock at the moment the operation is performed. Our definition is more general, and applies also to cases where a processor may perform operations out of order.

The global time stamp of an operation OP (a LD or ST) at p_i is set to be equal to

max{p_i's timestamp of the transaction to which the LD/ST is bound, global timestamp of last LD or ST at p_i in program order}

The local timestamp of OP is defined to be 1 if OP is the first operation in program order with global timestamp t and is otherwise equal to one plus the local timestamp of the most recent operation in the program order.

We now consider an example which illustrate the timestamping scheme. Consider first a scenario containing 2 nodes (N_1 and N_2) and 2 blocks of memory (A and B). N_1 has block A in the read-only state, while N_2 wants to obtain block A in the read-write state. N_1 also is performing stores to block B. Table 2 shows the scenario in physical time, while Table 3 shows the scenario in Lamport time where events have been ordered by their timestamps. We assume that the global clocks of both processors are initially set to 1.

TABLE 2. 2 nodes, 2 blocks, physical time

Time	N_1	N_2
1	send *Get-Exclusive* for A	store to B
2		bind load from A
3		receive invalidate for A, send ack

TABLE 2. 2 nodes, 2 blocks, physical time

Time	N_1	N_2
4	receive ack for A	perform bound load, invalidate from cache
5	store to A	

TABLE 3. 2 nodes, 2 blocks, Lamport time

Timestamp	N_1	N_2
1.10.2		store to B
1.11.2		load from A
2		invalidate A, send ack
3	receive ack for A	
3.1.1	store to A	

Note that, in this example, the Lamport ordering places N_2's load from A before N_1's store to A even though they may occur out-of-order in an aggressive implementation of our protocol, which buffers the invalidation to apply it much later while sending the acknowledgment immediately [20].

Claim 4: 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: The proof of Claim 4 uses the fact that binding of operations is done in program order in real time (4th bullet of Section 2.4). These real-time properties 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 A.

3.3 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. A *coherence epoch* is simply a Lamport time interval [t1,t2) during which a node has access to a block. All LDs and STs that have global timestamp t where $t_1 \leq t < t_2$ are contained in epoch $[t_1,t_2)$. 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. In the example from the previous section, the shared epoch of A at N_2 ended at global time 2 while A's exclusive epoch at N_1 started at global time 3. We build up to the proof of sequential consistency using the two timestamping claims of Section 3.2.

Lemma 1 shows that two processors cannot have "conflicting" permission to the same block at the same (Lamport) time. 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 directory between epochs. Proofs of the lemmas can be found in Appendix A.

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 the directory, 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 time stamp 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 time stamp 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. QED.

4 Related Work

Most of the related work in coherence protocol verification is based on formal methods [18] that use state-space search of finite-state machines, and theorem-proving techniques. These are rigorous 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 [6], the memory subsystem of the Sun S3.mp cache-coherent multiprocessor system is verified for one cache block in [17], the correctness of the Stanford FLASH coherence protocol is verified for small test programs and small configurations in [16], and the SPARC Relaxed Memory Order (RMO) memory consistency model is verified for small test programs in [15]. In contrast, 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. We find Lamport clocks 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 [1, 2, 3, 5, 7, 8, 9, 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. In other work [23], we show that Lamport clocks also offer the opportunity to analyze, formally or semi-formally, 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 [14] that uses the clocks to determine what knowledge is available to each processor in a distributed algorithm.

5 Conclusions and Future Work

Shared-memory systems are becoming increasingly complex and the need of the hour is for better verification tools that are intuitive, precise and scalable. We propose a verification framework based on Lamport's logical clock scheme that creates a total order of relevant protocol events. This order is a constructive realization of the ordering hypothesized in the definitions of various memory consistency models. We can then construct proofs that show that the requirements of a particular memory consistency model are met in this total order. The notion of coherence epochs arises naturally from such a logical ordering of events, and this notion clarifies the operation of the protocol as well as its proof of correctness. We have presented our technique and then successfully applied this technique to the proof of a non-trivial directory cache-coherence protocol. We expect the technique to apply equally well to any other directory protocol, or a bus-based protocol (as shown in [23]).

Future work with Lamport clocks will extend the range of systems to which our analysis can be applied, and we plan on devising a generic proof that can be easily tailored to new systems. The new systems that will be analyzed may include: clusters of SMPs, systems with consistent I/O, and systems that obey consistency models other than sequential consistency. We also believe that Lamport clocks are a useful tool for reasoning about the possibilities of deadlock, livelock, and starvation in a directory protocol, and we intend to explore this area of research.

6 References

[1] 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.

[2] 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.

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

[4] David Culler, Jaswinder Pal Singh, and Anoop Gupta. *Draft of Parallel Computer Architecture: A Hardware/Software Approach,* chapter 8: Directory-based Cache Coherence. Morgan Kaufmann, 1997.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

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

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

[12] 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.

[13] Daniel Lenoski. Personal communication, March 1998.

[14] 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.

[15] 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.

[16] 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.

[17] Fong Pong, Michael Browne, Andreas Nowatzyk, and Michel Dubois. Design Verification of the S3.mp Cache-Coherent Shared-Memory System. *IEEE Transactions on Computers*, 47(1):135–140, January 1998.

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

[19] J. Protic, M. Tomasevic, and V. Milutinovic. Distributed Shared Memory: Concepts and Systems. *IEEE Parallel and Distributed Technology*, pages 63–79, 1996.

[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] Daniel J. Sorin, Manoj Plakal, Mark D. Hill, and Anne E. Condon. Lamport Clocks: Reasoning About Shared-Memory Correctness. Technical Report CS-TR-1367, University of Wisconsin-Madison, March 1998.

Appendix A: Proofs of Claim 2, Claim 4, and the Lemmas

Claim 2: The sequence of A-state changes on block B at a node occurs in real time in the order implied by the serialization of transactions on block B at its directory.

Proof: Claim 2 is easily seen to be true for block B's directory entry since the directory processes transactions in order. Now, suppose that T_1 and T_2 are two transactions affecting block B of p_i, where T_1 occurs before T_2 in the transaction serialization order at the directory, and that T_2 is the first transaction after T_1 on block B that affects p_i. From Claim 1, a message is sent to p_i both as a result of T_1 and T_2 (although these may not arrive at p_i in order). We need to show that the change in A-state resulting from T_1 at p_i occurs before the change in A-state resulting from T_2. We consider two cases.

- Case 1: *p_i requests transaction T_1*. If p_i receives messages relating to transaction T_2 before p_i has changed its A-state corresponding to T_1 (transactions 3,6,7,9 from Section 2.3), then p_i buffers such messages until all processing of transaction T_1 has been completed (refer to the 3rd bullet in Section 2.4). Otherwise (transactions 1,2,5,9,12,13,14), there is no that way p_i's A-state could change due to T_2 before finishing T_1.
- Case 2: *p_i does not request transaction T_1*. First, suppose that T_1 implies that p_i's A-state changes from A_X to A_S or A_I. Therefore, T_1 must result from a *Get-Shared* or *Get-Exclusive* request from a processor, p_j, other than p_i. In these cases (transactions 3 and 7 from Section 2.3), the directory enters the busy state and remains in that state until it receives a response from p_j, at which point p_i's state has been changed to A_S or A_I as appropriate. Therefore, the change in A-state at p_i implied by T_1 occurs before the directory leaves the busy state. T_2 is not NACKed, and so the directory does not send a message to p_i regarding transaction T_2 until after leaving the busy state for T_1. Furthermore, p_i does not change its A-state as a result of T_2 until it receives a message from the directory regarding T_2. Therefore, the change in A-state corresponding to T_2 occurs after the change in A-state corresponding to T_1.
  The only other possible case is that T_1 implies that p_i's state changes from A_S to A_I. Hence, T_1 must result from a *Get-Exclusive* or *Upgrade* request from a processor, p_j, other than p_i. In this case, the only way that T_2 can affect p_i is if T_2 is requested by p_i. T_2 could be a *Get-Shared*, a *Get-Exclusive* or an *Upgrade*, since the actual state of the block B in p_i's cache could be either read-only, or invalid due to a *Put-Shared* action which does not affect the A-state. If T_2 is a *Get-Shared* or *Get-Exclusive* then, by the definition of A-state in Section 3.1, the A-state at p_i changes from A_S to A_I (due to T_1) and then immediately to A_S or A_X (due to T_2), as appropriate. If T_2 is an *Upgrade*, it is NACKed by the directory (due to T_1, transaction 10 from Section 2.3). Since T_2 affects p_i, it cannot be a NACKed request, and therefore it must be requested by p_i after p_i has changed its state to A_I. QED.

Claim 4: 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 time stamp 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. By the fact that the Lamport order at p_i equals the real-time order of changes of A-state 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. QED.

Comment: the proof of Claim 4 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 4. 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 transactions, 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 block 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 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 3 (b), 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 $T_0 = T_1$). Claim 3 (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 3 (b) again, the timestamp of T_0 by N_2 must be less than the timestamp of T_1 by N. Hence 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 3 (a) ensures that N_2's timestamp of T_2 must be greater than t_2. Hence, by Claim 3 (b), if an epoch E starts as a result of transaction T_2 or a transaction later than T_2, E must start at a time greater than t_2, as required. QED.

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 4, 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. QED.

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 directory, 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 directory and equals the initial value of the block in the directory.

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 3, 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 directory, 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 directory, 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 4 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 time stamp 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. QED.

Appendix B: Impossible Transactions

Upgrade with Directory being Idle: Assume that p_i is the processor performing the *Upgrade* on block B and that it obtained read-only access with transaction T. Some other processor must have performed a *Get-Exclusive* or *Upgrade* and then a *Writeback* before p_i's *Upgrade* reached the directory. Let transaction T' be the first *Get-Exclusive* or *Upgrade* transaction on block B after T in the serialization order, and assume that it occurs at processor p_j. T' (via transactions 6 or 9 for *Get-Exclusive* or *Upgrade*, respectively) ensures that p_j must wait for an acknowledgment from p_i before obtaining read-write access. In turn, p_i cannot send an acknowledgment until its *Upgrade* is processed by the directory. Until then, p_j cannot do a *Writeback*, and thus the Directory cannot be *Idle*.

Writeback: Assume that p_i is the processor performing the *Writeback* on block B and that it obtained read-write access with transaction T.

- *Directory is Idle*: Some other processor must have performed a *Get-Exclusive* and then a *Writeback* before p_i's *Writeback* reached the directory. Let transaction T' be the first *Get-Exclusive* transaction on block B after T in the serialization order and assume that it occurs at processor p_j. However, T' (transaction 7) ensures that the directory will go into *Busy-Exclusive* until it receives a message from p_i. Hence, p_j cannot obtain read-write access before p_i's *Writeback* has been processed by the directory, because p_j cannot receive a reply from the Home until p_i's *Writeback* request is received and processed by the directory.
- *Directory is Shared*: For the directory to be *Shared*, some other processor must have performed a *Get-Shared* before p_i's *Writeback* reached the directory. Let transaction T' be the first *Get-Shared* transaction on block B after T in the serialization order and assume it occurs at processor p_j. However, T' (transaction 3) ensures that the directory will go into *Busy-Shared* until it receives a message from p_i. Therefore, p_i's *Writeback* cannot see a *Shared* directory.
- *Directory is Busy-Shared*: Some other processor must have performed a *Get-Shared* before p_i's *Writeback* reached the directory. Let transaction T' be the first *Get-Shared* transaction on block B after T in the serialization order and assume it occurs at processor p_j. T' (transaction 3) ensures that the directory will go into *Busy-Shared* until it receives a message from pi. Once the directory enters *Busy-Shared*, CACHED only contains p_j's ID. Therefore, p_i's ID cannot be in CACHED.
- *Directory is Busy-Idle*: Some other processor, p_j must have performed a *Get-Exclusive*, received the block from p_i, and performed a *Writeback* that beat p_i's update message to the directory. At this point, any processor that makes a *Get-Shared*, *Get-Exclusive*, or *Upgrade* request for B will get NACKed (transactions 4, 8, and 11). Only p_i can change the state out of *Busy-Idle*, and this will happen when its update message arrives at the directory. No *Writeback* can occur while in *Busy-Idle* because no processor has read-write access (p_i is already in the invalid state once it has sent the block to p_j).