Specifying and Verifying a Broadcast and a Multicast Snooping Cache Coherence Protocol

Daniel J. Sorin, *Student Member*, *IEEE*, Manoj Plakal, *Student Member*, *IEEE*, Anne E. Condon, Mark D. Hill, *Fellow*, *IEEE*, Milo M.K. Martin, *Student Member*, *IEEE*, and David A. Wood, *Member*, *IEEE*

Abstract—In this paper, we develop a specification methodology that documents and specifies a cache coherence protocol in eight tables: the states, events, actions, and transitions of the cache and memory controllers. We then use this methodology to specify a detailed, modern three-state broadcast snooping protocol with an unordered data network and an ordered address network that allows arbitrary skew. We also present a detailed specification of a new protocol called Multicast Snooping [6] and, in doing so, we better illustrate the utility of the table-based specification methodology. Finally, we demonstrate a technique for verification of the Multicast Snooping protocol, through the sketch of a manual proof that the specification satisfies a sequentially consistent memory model.

Index Terms—Cache coherence, protocol specification, protocol verification, memory consistency, multicast snooping.

1 Introduction

A cache coherence protocol is a scheme for coordinating access to shared blocks of memory. Processors and memories exchange messages to share data and to determine which processors have read-only or read-write access to data blocks that are in their caches. A processor's access to a cache block is determined by the state of that block in its cache, and this state is generally one of the five MOESI (Modified, Owned, Exclusive, Shared, Invalid) states [32]. Processors issue requests, such as Get Exclusive or Get-Shared, to gain access to blocks. They can also lose access to blocks, either by choice (e.g., a cache replacement) or when another processor's request steals a block away. Many invalidate protocols maintain the invariant that there can either be one writer and no readers or no writer and any number of readers.

What is protocol specification? Cache coherence protocols for shared memory multiprocessors are implemented via the actions of numerous system components and the interactions between them. These components include cache controllers, directory controllers, and networks, among others. The specification of a cache coherence protocol must detail the actions of each of these components for every combination of state it could be in and event that could happen. For example, it must specify the actions performed by a cache controller that has Exclusive access to a cache block when a Get-Shared request for that block arrives from another node, and it must specify the new state that the cache controller enters.

- *D.J. Sorin, M. Plakal, M.D. Hill, M.M.K. Martin, and D.A. Wood are with the Computer Sciences Department, University of Wisconsin, 1210 West Dayton Street, Madison, WI 53706. E-mail: {sorin, plakal, markhill, milo, david}@cs.wisc.edu.*
- *A.E. Condon is with the Department of Computer Science, University of British Columbia, 201-2366 Main Mall, Vancouver, BC V6T1Z4. E-mail: condon@cs.ubc.ca.*

Manuscript received 2 Mar. 2000; revised 8 Mar. 2001; accepted 21 Nov. 2001.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 111633.

What is protocol verification? Verification of a cache coherence protocol involves proving that a protocol specification obeys a desired memory consistency model, such as sequential consistency (SC) [21]. To verify that a protocol satisfies a consistency model requires proving that it obeys certain invariants about what value a load from memory can return. For example, to satisfy SC, the loads and stores from the different processors must appear to the programmer to be in some total order where 1) the value of a load equals the value of the most recent store to the same address in the total order, and 2) the total order respects the program order at each of the processors.

Why is verification difficult? At a high level, protocols can be represented as in Fig. 1, which illustrates the specification of a cache controller for a three state (Modified, Shared, Invalid) protocol. There are a handful of states, with atomic transitions between them.

Since cache coherence protocols are simply finite state machines, it would appear at first glance that it would be easy to specify and verify a common three state (MSI) broadcast snooping protocol. Unfortunately, at the level of detail required for an actual implementation, even seemingly straightforward protocols have numerous transient states and possible race conditions that complicate the tasks of specification and verification. While older protocols only permitted one outstanding miss and required that a request and its responses were atomic, current protocols allow transactions to be split and allow multiple outstanding requests. Thus, other requests and responses can be interleaved between a request and its response. This additional concurrency enables higher performance, but it increases the complexity by often introducing transient states. For example, a single cache controller in a "simple" MSI protocol that we will specify in Section 2.1 has 11 states (8 of which are transient), 13 possible events, and 21 actions that it may perform. The other system components are similarly complicated, and the interactions of all of these

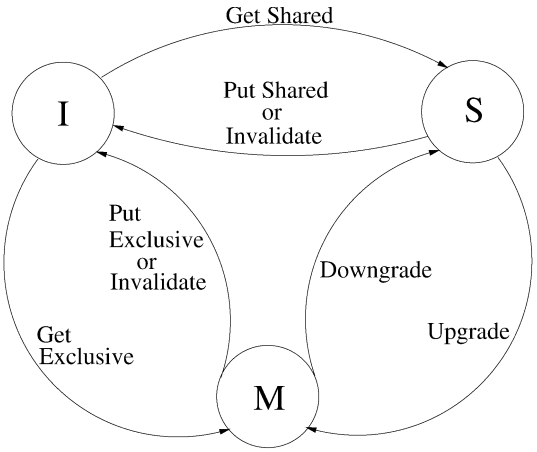

Fig. 1. High-level specification for cache controller.

components are difficult to specify and verify. Moreover, the number of states in the system is roughly proportional to the number of states in each coherence controller to the power of the number of controllers. This state space explosion makes the use of tools such as model checkers prohibitive when the number of processors is large, even when proving that simple invariants are maintained.

Why is verification important? Rigorous verification is important, since the complexity of a detailed, implementable protocol makes it difficult to design without any errors. Many protocol errors can be uncovered by simulation. Simulation with random testing has been shown to be effective at finding certain classes of bugs, such as lost protocol messages and some deadlock conditions [33]. However, simulation tends not to be effective at uncovering subtle bugs, especially those related to the consistency model. Subtle consistency bugs often occur only under unusual combinations of circumstances, and it is unlikely that undirected (or random) simulation will drive the protocol to these situations. Thus, complete and perhaps more formal verification techniques are needed to expose these subtle bugs.

What kind of specification is required for verification? Complete verification of a cache coherence protocol should be undertaken at a level that is independent of details that are specific to the hardware, yet models transient states, queues, and race conditions that typically introduce subtle bugs. Verifying a high-level specification without transient states and race conditions may show that invariants hold for this abstraction of the protocol, but it will not show that an implementable version of the protocol obeys these invariants.

What are the limitations of current specifications? In the industrial groups with which we are familiar, there are three classes of people—architects, implementors, and verifiers—who work together to develop systems. However, current specifications are generally not accessible to all three classes. For a specification to be accessible to all three groups, a balance must be struck between having a concise, visually informative format while still incorporating sufficient detail. Specifications that have been published in the literature are often visually accessible, but they have not been sufficiently detailed for purposes of implementation or verification. In academia, protocol specifications tend to be high-level because a complete, detailed specification may not be necessary for the goal of publishing research [5], [8], [15]. In industry, low-level, detailed specifications are necessary and exist, but, to the best of our knowledge, none have been published in the literature. Moreover, these detailed specifications often match the hardware closely, which complicates verification and limits alternative implementations but eliminates the problem of verifying that the implementation satisfies the specification. Formal specifications, which are used in both academia and in industry, are well-suited to verification with tools such as model checkers, but they are generally unusable by less mathematically-inclined implementors and architects.

A new, widely-accessible table-based specification technique. To address the need for concise, detailed specifications that are widely accessible, we have developed a table-based specification methodology for cache coherence protocols. While tables have been used widely to describe state machines [18], the concise format of our tables allows for substantial detail while retaining visual clarity. It is useful to have a complete table on one page so that, for example, a missing entry or an entry that differs slightly from all others in its column is conspicuous. Other table-based specification schemes, such as Johnson's behavior tables [19], are both formal and visually informative, but they are not tailored for coherence protocols and, as such, do not represent them concisely.

In our scheme, for each system component that participates in the coherence protocol, there is a table that specifies the component's behavior with respect to a given cache block. As an illustrative example, Table 1

TABLE 1
Simplified Atomic Cache Controller Transitions

State \ Event	Load	Store	Other GETS	Other GETX
I	a/S	c/M		
S	h	c/M		I
M	h	h	dm/S	d/I

a: perform Get-Shared d: send data to requestor h: cache hit
c: perform Get-Exclusive m: send data to memory

shows a specification for a simplified atomic cache controller. The rows of the table correspond to the states that the component can enter, the columns correspond to the events that can occur, and the entries themselves are the actions taken and resulting state that occur for that combination of state and event. The actions are coded with letters which are defined below the table. For example, the entry a/S denotes that a Load event at the cache controller for a block in state I causes the cache controller to perform a Get-Shared and enter state S.

This simple example, however, does not show the power of our specification methodology because it does not include the many transient states possessed by realistic coherence protocols. For simple atomic protocols, the traditional specification approach of drawing up state transition diagrams is tractable. However, nonatomic transactions cause an explosion in the state space, since events can occur between when a request is issued and when it completes, and numerous transient states are used to capture this behavior. Section 2 illustrates the methodology with a more realistic broadcast snooping protocol and a multicast snooping protocol [6].

In our specification methodology, we aim for a middle ground that can be used by architects, implementors, and verifiers. While the tables themselves do not enable a specific level of detail, we choose a level of detail that can be used for many purposes and in which actions that are specified as atomic could be implemented atomically. Verification of a protocol at this level must handle many of the most subtle issues, such as those that arise from considering the queues between state machines. It is also important to note that a specification at this level allows us to verify this level of implementation, but it also aids the verification of more complex implementations. To verify a system at a lower level of detail, one must now only verify that the lower level implementation is equivalent to this specification. For example, one might verify that a pipelined implementation of a given set of actions still appears to be atomic.

We have developed software that automatically maps specifications in our format to different levels of abstraction, including simulator code and documentation, and we use the specifications as input for a manual proof technique presented in this paper. Mapping specifications to input for automated verification tools is future work.

A methodology for proving that table-based specifications are correct. Using our table-based specification methodology, we present a methodology for proving that a specification is sequentially consistent, and we show how this scheme can be used to prove that our multicast protocol satisfies SC. Our method uses an extension of Lamport's logical clocks [20] to timestamp the load and store operations performed by the protocol. Timestamps determine how operations should be reordered to witness SC, as intended by the designer of the protocol. Thus, associated with any execution of the augmented protocol is a sequence of timestamped operations that witnesses sequential consistency of that execution. Logical clocks and the associated timestamping actions are, in effect, a conceptual augmentation of the protocol and are specified using the same table-based transition tables as the protocol itself. We note that the set of all possible execution operation traces of the protocol equals that of the augmented protocol, and that the logical clocks are purely conceptual devices introduced for verification purposes and are never implemented in hardware. We consider the process of specifying logical clocks and their actions to be intuitive for the designer of the protocol, and indeed the process is a valuable debugging tool in its own right.

A straightforward invariant of the augmented protocol guarantees that the protocol is sequentially consistent. Namely, for all executions of the augmented protocol, the associated timestamped sequence of loads (LDs) and stores (STs) is consistent with the program order of operations at all processors and the value of each LD equals that of the most recent ST. To prove this invariant, numerous other "support" invariants are added as needed. It can be shown that all executions of the protocol satisfy all invariants by induction on the length of the execution. This involves a tedious case-by-case analysis of each possible transition of the protocol that could possibly be automated with a model checker.

To summarize, the strengths of our methodology are that the process of augmenting the protocol with timestamping is useful in designing correct protocols, and an easily-stated invariant of the augmented protocol guarantees sequential consistency. However, our methodology also involves tedious case-by-case proofs that protocol state transitions respect certain invariants. Because the problem of automatically verifying SC is undecidable, automated approaches have been proved to work only for a limited class of protocols [16], [29]. We will discuss other verifications techniques and compare them to ours in Section 4.

What have we contributed? This paper makes four contributions. First, we develop a table-based specification methodology that allows us to concisely describe cache coherence protocols. Second, we provide a detailed specification of a modern three-state broadcast snooping protocol with an unordered data network and an address network which allows arbitrary skew. Third, we present a detailed specification of multicast snooping [6], and, in doing so, we better illustrate the utility of the table-based specification methodology. The specification of this more complicated protocol is thorough enough to warrant verification. Fourth, we demonstrate a technique for verification of the multicast snooping protocol, through the sketch of a manual proof that the specification satisfies a sequentially consistent memory model.

2 Specifying Broadcast and Multicast Snooping Protocols

In this section, we demonstrate our protocol specification methodology by developing two protocols: a broadcast snooping protocol and a multicast snooping protocol. Both protocols are MSI (Modified, Shared, Invalid) and use eight tables to document and specify:

- the states, events, actions, and transitions of the cache controller and
- the states, events, actions, and transitions of the memory controller.

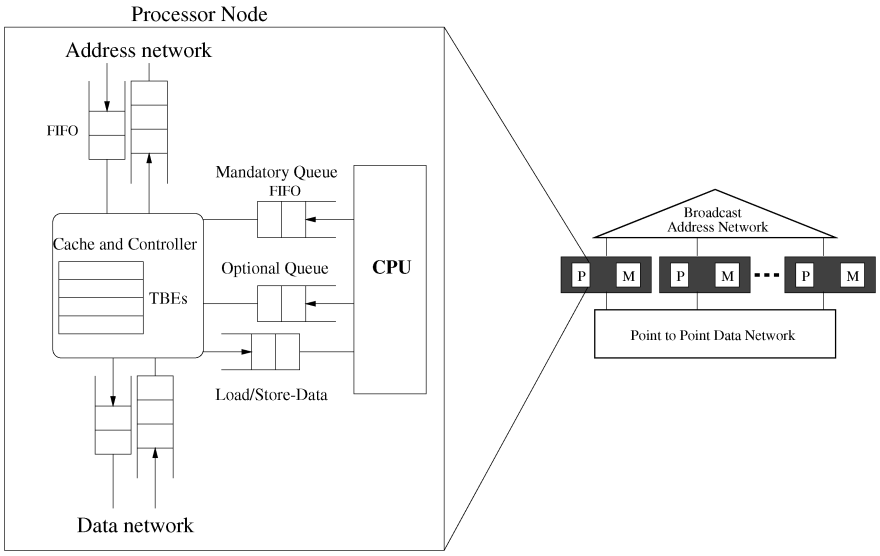

Fig. 2. Broadcast snooping system.

The controllers are state machines that communicate via queues, and events correspond to messages being processed from incoming queues. The actions taken when a controller services an incoming queue, including enqueuing messages on outgoing queues, are considered atomic.

2.1 Specifying a Broadcast Snooping Protocol

In this section, we shall specify the behavior of an MSI broadcast snooping protocol. While three state broadcast protocols are simple to describe at an abstract level, realistic protocols can have significant complexity due to transient states and nonatomic transactions.

2.1.1 System Model and Assumptions

The broadcast snooping system is a collection of processor nodes and memory nodes (possibly collocated) connected by two logical networks (possibly sharing the same physical network), as shown in Fig. 2.

A processor node contains a CPU, cache, and a cache controller which includes logic for implementing the coherence protocol. It also contains queues between the CPU and the cache controller. The Mandatory queue contains Loads (LDs) and Stores (STs) requested by the CPU, and they are ordered by program order. LD and ST entries have addresses, and STs have data. The Optional queue contains Read-Only and Read-Write Prefetches requested by the CPU, and these entries have addresses. The Load/Store Data queue contains the LD/ST from the Mandatory queue and its associated data (in the case of a LD). A diagram of a processor node is also shown in Fig. 2.

The memory space is partitioned among one or more memory nodes. It is responsible for responding to coherence requests with data if it is the current owner (i.e., no processor node has the block Modified). It also receives writebacks from processors and stores this data to memory.

The two logical networks are a totally ordered broadcast network for address messages and an unordered unicast network for data messages. The address network supports three types of coherence requests: GETS (Get-Shared), GETX (Get-Exclusive), and PUTX (Dirty-Writeback). Cache controllers issue coherence requests in response to memory accesses (LD/ST) and prefetches received from the CPUs. Protocol transactions are address messages that contain a data block address, coherence request type (GETX, GETS, PUTX), and the ID of the requesting processor. Data messages contain the data and the data block address.

All of the components in the system make transitions based on their current state and current event (e.g., an incoming request), and we will specify the states, events, and transitions for each component in the rest of this section. There are many components that make transitions on many blocks of memory, and these transitions can happen concurrently. We assume, however, that the system appears to behave as if all transitions occur atomically.

2.1.2 Network Specification

The network consists of two logical networks. The address network is a totally ordered broadcast network, as in all known broadcast snooping protocols. Total ordering does not, however, imply that all messages are delivered at the same time. For example, in an asynchronous implementation, the path to one node may take longer than the path to another node. The address network carries coherence request messages. A transition of the address network is modeled as atomically transferring a coherence request from the output queue of a node to the input queues of all of the nodes, thus inserting the request into the total order of requests. Note that a total order of requests does not imply a total order of memory accesses (LD/ST), since requests are issued to gain permission to access data, but they are not the accesses themselves.

The data network is an unordered point-to-point network for delivering responses to coherence requests. A transition of the data network is modeled as atomically transferring a data message from the output queue of a node to the input queue of the destination node.

All nodes are connected to the networks via queues, and all we assume about these queues is that address queues from the network to the nodes are served in FIFO order.

TABLE 2
Broadcast Snooping Cache Controller States

	TBE State	Cache State	Description
Stable states		I	invalid
		S	shared
		M	modified
Transient states	IS^{AD}	busy	invalid, issued GETS, have not seen GETS or data yet
	IS^{A}	busy	invalid, issued GETS, have not seen GETS, have seen data
	IS^{D}	busy	invalid, issued GETS, have seen GETS, have not seen data yet
	IM^{AD}	busy	invalid, issued GETX, have not seen GETX or data yet
	IM^{A}	busy	invalid, issued GETX, have not seen GETX, have seen data
	IM^{D}	busy	invalid, issued GETX, have seen GETX, have not seen data yet
	MI^{A}	I	modified, issued PUTX, have not seen PUTX yet
	II^{A}	I	modified, issued PUTX, have not seen PUTX, then saw other GETS or GETX (reachable from MI^{A})

Data queues and address queues from the nodes to the network can be served without this restriction. For example, this allows a processor node's GETX to pass its PUTX for the victim block.

2.1.3 CPU Specification

A transition of the CPU occurs when it places a LD or ST in the Mandatory queue, places a Prefetch in the Optional queue, or removes data from the LD/ST data queue. It can perform these transitions at any time.

2.1.4 Cache Controller Specification

In each transition, a cache controller may inspect the heads of its incoming queues, inject new messages into its queues, and make appropriate state changes. All we assume about serving incoming queues is that no queue is starved and that the Address and Mandatory queues are served in strict FIFO order. The actions taken when a queue is served are considered atomic in that they are all done before another queue (including the same queue) is served. Before any of the actions are taken, however, the cache controller checks to ensure that resources, such as space in an outgoing queue or an allocated TBE, are available for all of the actions. If the sum of the resources required for all of the actions is not available, then the cache controller aborts the transition, performs none of the actions, and waits for resources to become available (where we define a cache block to be available for a LD/ST if either the referenced block already exists in the cache or there exists an empty slot which can accommodate the referenced block when it is received from external sources). The exception to this rule is having an available block in the cache, and this situation is handled by treating a LD, ST, or Prefetch for which no cache block is available as a Replacement event for the victim block.

If the request at the head of the Mandatory or Address queue cannot be serviced (because the block is not present with the correct permissions or a transaction for the block is outstanding), then no further requests from that queue can be serviced. Optional requests can be discarded without affecting correctness.

The cache controller keeps a count of all outstanding coherence transactions issued by that node and, for each such transaction, one Transaction Buffer Entry (TBE) is reserved. No transactions can be issued if there is no space in the outgoing address queue or if there is already an outstanding transaction for that block. A TBE contains the address of the block requested, the current state of the transaction, and any data received.[1]

The possible block states and descriptions of these states are listed in Table 2. Note that there are two types of "states" for a cache block: the "stable" state and the "transient" state. The *stable state* is one of M (Modified), S (Shared), or I (Invalid), it is recorded in the cache, and it indicates the state of the block before the latest outstanding transaction for that block (if any) started. The *transient state*, as shown in Table 2, is recorded in a TBE, and it indicates the current state of an outstanding transaction for that block (if any). When future tables refer to the state of a block, it is understood that this state is obtained by returning the transient state from a TBE (if there is an outstanding transaction for this block), or else (if there is no outstanding transaction) by accessing the cache to obtain the stable state. Blocks not present in the cache are assumed to have the stable state of I. Each transient state has an associated cache state, as shown in Table 2, assuming that the tag matches in the cache. A cache state of busy implies that there is a TBE entry for this block, and its state is a transient state other than MI^{A} or II^{A}.

To represent the transient states symbolically, we have developed an encoding of these transient states which consists of a sequence of two or more stable states (initial, intended, and zero or more pending states), where the second state has a superscript which denotes which part(s) of the transaction—address (A) and/or data (D)—are still outstanding. For example, a processor which has block B in state I, sends a GETS into the Address-Out queue, and sees the data response but has not yet seen the GETS, would

1. The data field in the TBE may not be required. An implementation may be able to use the cache's data array to buffer the data for the block. This modification reduces the size of a TBE and avoids specific actions for transferring data from the TBE to the cache data array.

TABLE 3
Broadcast Snooping Cache Controller Events

Event	Description	Block B
Load	LD at head of Mandatory queue	address of LD at head of Mandatory Queue
Read-Only Prefetch	Read-Only Prefetch at head of Optional queue	address of Read-Only Prefetch at head of Optional Queue
Store	ST at head of Mandatory queue	address of ST at head of Mandatory Queue
Read-Write Prefetch	Read-Write Prefetch at head of Optional queue	address of Read-Write Prefetch at head of Optional Queue
Mandatory Replacement	LD/ST at head of Mandatory queue for which no cache block is available	address of victim block for LD/ST at head of Mandatory queue
Optional Replacement	Read-Write Prefetch at head of Optional queue for which no cache block is available	address of victim block for Prefetch at head of Optional queue
Own GETS	Occurs when we observe our own GETS request in the global order	address of transaction at head of incoming address queue
Own GETX	Occurs when we observe our own GETX request in the global order	same as above
Own PUTX	Occurs when we observe our own PUTX request in the global order	same as above
Other GETS	Occurs when we observe a GETS request from another processor	same as above
Other GETX	Occurs when we observe a GETX request from another processor	same as above
Other PUTX	Occurs when we observe a PUTX request from another processor	same as above
Data	Data for this block from the data network	address of data message at head of incoming data queue

have B in state IS^A. When the GETS arrives, the state becomes S.

Events at the cache controller depend on incoming messages. The events are listed and described in Table 3. Note that, in the case of Replacements, block B refers to the address of the victim block. The allowed cache controller actions are listed in Table 4. Cache controller behavior is detailed in Table 5, where each entry contains a list of $< actions/next\ state >$ tuples. When the current state of a block corresponds to the row of the entry and the next event corresponds to the column of the entry, then the specified actions are performed in order and the state of the block is changed to the specified new state. If only a next state is listed, then no action is required. All shaded cases are impossible. The level of detail is such that actions that are specified as atomic could be implemented so as to appear atomic. An actual implementation could, for example, use pipelining with bypassing and still appear atomic.

2.1.5 Memory Node Specification

One of the advantages of broadcast snooping protocols is that the memory nodes can be quite simple. The memory nodes in this system, like those in the Synapse [11], maintain some state about each block for which this memory node is the home, in order to make decisions about when to send data to requestors. This state includes the state of the block and the current owner of the block. Memory states are listed in Table 6, events are in Table 7, actions are in Table 8, and transitions are in Table 9.

2.2 Specifying a Multicast Snooping Protocol

In this section, we will specify an MSI multicast snooping protocol with the same methodology used to describe the broadcast snooping protocol. Multicast snooping is a promising new protocol that requires less snoop bandwidth and provides higher throughput of address transactions, thus enabling larger systems than are possible with broadcast snooping. Multicast snooping also has many features, which will be discussed later, that pose new challenges for verification.

2.2.1 System Model and Assumptions

Multicast snooping, as described by Bilir et al. [6], incorporates features of both broadcast snooping and directory protocols. It differs from broadcast snooping in that coherence requests use a totally ordered multicast address network instead of a broadcast network. Multicast masks are predicted by processors, and they must always include the processor itself and the directory for this block (but not any other directories), yet they are allowed to be incorrect. A GETS mask is incorrect if it omits the current owner, and a GETX mask is incorrect if it omits the current owner or any of the current sharers. This scenario is resolved by a simple directory which can detect mask mispredictions and retry these requests (with an improved

TABLE 4
Broadcast Snooping Cache Controller Actions

Action	Description
a	Allocate TBE with Address=B
c	Set cache tag equal to tag of block B.
d	Deallocate TBE.
f	Issue GETS: insert message in outgoing Address queue with Type=GETS, Address=B, Sender=N.
g	Issue GETX:insert message in outgoing Address queue with Type=GETX, Address=B, Sender=N
h	Service LD/ST (a cache hit) from the cache and (if a LD) enqueue the data on the LD/ST data queue.
i	Pop incoming address queue.
j	Pop incoming data queue.
k	Pop mandatory queue.
l	Pop optional queue.
m	Send data from TBE to memory.
n	Send data from cache to memory.
p	Issue PUTX: insert message in outgoing Address queue with Type=PUTX, Address=B, Sender=N
q	Copy data from cache to TBE.
r	Send data from the cache to the requestor
s	Save data in data field of TBE.
u	Service LD from TBE, pop mandatory queue, and enqueue the data on the LD/ST data queue if the LD at the head of the Mandatory queue is for this block.
v	Service LD/ST from TBE, pop mandatory queue, and (if a LD) enqueue the data on the LD/ST data queue if the LD/ST at the head of the Mandatory queue is for this block.
w	Write data from data field of TBE into cache
y	Send data from the TBE to the requestor.
z	Cannot be handled right now.

mask) on behalf of the requestors. In the common case of a correct prediction, multicasting reduces network bandwidth and processor snoop bandwidth, as compared to broadcasting.

The multicast snooping protocol described here differs from that specified in Bilir et al. in a few significant ways. First, we specify an MSI protocol here instead of an MOSI protocol. Second, we specify the protocol here at a lower, more detailed level. Third, the directory in this protocol can retry requests with incorrect masks on behalf of the original requester. The directory can also nack such requests, as specified by Bilir et al., to avoid deadlock in the case where there the address network cannot accept the retry.

A multicast system is shown in Fig. 3. The processor nodes are structured like those in the broadcast snooping protocol. Instead of memory nodes, though, the multicast snooping protocol has directory nodes, which are memory nodes with extra protocol logic for handling retries, and they are also shown in Fig. 3. In the next two sections, we will specify the behaviors of processor and directory components in an MSI multicast snooping protocol.

2.2.2 Network Specification

The data network behaves identically to that of the broadcast snooping protocol, but the address network behaves slightly differently. As the name implies, the address network uses multicasting instead of broadcasting and, thus, a transition of the address network consists of taking a message from the outgoing address queue of a node and placing it in the incoming address queues of the nodes specified in the multicast mask, as well as the requesting node and the memory node that is the home of the block being requested (if these nodes are not already part of the mask).

Address messages contain the coherence request type (GETS, GETX, or PUTX), requesting node ID, multicast mask, block address, and a retry count. Data messages contain the block address, sending node ID, destination node ID, data message type (DATA or NACK), data block, and the retry count of the request that triggered this data message.

2.2.3 CPU Specification

The CPU behaves identically to the CPU in the broadcast snooping protocol.

TABLE 5
Broadcast Snooping Cache Controller Transitions

State	Load	Read-Only Prefetch	Store	Read-Write Prefetch	Mandatory Replacement	Optional Replacement	Own GETS	Own GETX	Own PUTX	Other GETS	Other GETX	Other PUTX	Data
I	caf/ IS^{AD}	cafl/ IS^{AD}	cag/ IM^{AD}	cagl/ IM^{AD}						i	i	i	
S	hk	l	ag/ IM^{AD}	agl /IM^{AD}	I	I				i	i/I	i	
M	hk	l	hk	l	aqp/ MI^{A}	aqp/ MI^{A}				rni/S	ri/I	i	
IS^{AD}	z	z	z	z	z	z	i/IS^{D}			i	i	i	sj/IS^{A}
IM^{AD}	z	z	z	z	z	z		i/IM^{D}		i	i	i	sj/IM^{A}
IS^{A}	z	z	z	z	z	z	uwdi/S			i	i	i	
IM^{A}	z	z	z	z	z	z		vwdi/ M		i	i	i	
MI^{A}	z	z	z	z	z	z			mdi/$I^{\ddagger}$	ymi/II^{A}	yi/II^{A}	i	
II^{A}	z	z	z	z	z	z			di/$I^{\ddagger}$	i	i	i	
IS^{D}	z	z	z	z	z	z				i	z	i	suwdj/ S
IM^{D}	z	z	z	z	z	z				z	z	i	svwdj/ M

‡*Only change the cache state to I if the tag matches.*

2.2.4 Cache Controller Specification

Cache controllers behave much like they did in the broadcast snooping protocol, except that they must deal with retried and nacked requests and they are more aggressive in processing incoming requests. This added complexity leads to additional states, TBE fields, protocol actions, and protocol transitions.

There are additional states in the multicast protocol specified here due to the more aggressive processing of incoming requests. Instead of buffering incoming requests (with the "z" action) while in transient states, a cache controller in this protocol ingests some of these requests, thereby moving into new transient states. An example is the state $IM^{D}I$, which occurs when a processor in state IM^{D} ingests an incoming GETX request from another processor instead of buffering it. The notation signifies that a processor started in I, is waiting for data to go to M, and will then go to I immediately (except for in cases in which forward progress issues require the processor to perform a LD or ST before relinquishing the data, as will be discussed below). There are also three additional states that are necessary to describe situations where a processor sees a nack to a request that it has not seen yet.

There are four additional fields in the TBE: ForwardProgress, ForwardID, RetryCount, and ForwardIDRetryCount. The ForwardProgress bit is set when a processor sees its own request that satisfies the head of the Mandatory queue. This flag is used to determine when a processor must perform a single load or store on the cache line before relinquishing the block.[2] Being able to perform at least a single access is necessary to avoid livelock when a cache line is highly contended for by multiple processors. For example, when data arrives in state $IM^{D}I$, a processor can service a LD or ST to this block before forwarding the block if and only if ForwardProgress is set. The ForwardID field records the node to which a processor must send the block in cases such as this. In this example, ForwardID equals the ID of the node whose GETX caused the processor to go from IM^{D} to $IM^{D}I$. RetryCount records the retry number of the most recent message, and ForwardIDRetryCount records

TABLE 6
Broadcast Snooping Memory Controller States

State	Description
S	Shared or Invalid
M	Modified
MS^{A}	Modified, have not seen GETS/PUTX, have seen data
MS^{D}	Modified, have seen GETS or PUTX, have not seen data

2. Another viable scheme would be to set this bit when a processor observes its own address request and this request corresponds to the address of the head of the Mandatory queue. It is also legal to set ForwardProgress when a LD/ST gets to the head of the Mandatory queue while there is an outstanding transaction for which we have not yet seen the address request. However, sequential consistency is not preserved by a scheme where ForwardProgress is set when data returns for a request and the address of the request matches the address at the head of the Mandatory queue.

TABLE 7
Broadcast Snooping Memory Controller Events

Event	Description	Block B
Other Home	A request arrives for a block whose home is not at this memory	address of transaction at head of incoming address queue
GETS	A GETS at head of incoming address queue	same as above
GETX	A GETX at head of incoming address queue	same as above
PUTX (requestor is owner)	A PUTX from owner at head of incoming address queue	same as above
PUTX (requestor is not owner)	A PUTX from non-owner at head of incoming address queue	same as above
Data	Data at head of incoming data queue	address of message at head of incoming data queue

the retry count associated with the block that will be forwarded to the node specified by ForwardID.

We use the same table-driven methodology as was used to describe the broadcast snooping protocol. Tables 10, 11, 12, and 13 specify the states, events, actions, and transitions, respectively, for processor nodes.

2.2.5 Directory Node Specification

Unlike broadcast snooping, the multicast snooping protocol requires a simplified directory to handle incorrect masks. A directory node, in addition to its incoming and outgoing queues, maintains state information for each block of memory that it controls. The state information includes the block state, the ID of the current owner (if the state is M), and a bit vector that encodes a superset of the sharers (if the state is S). The possible block states for a directory are listed in Table 14. As before, we refer to M, S, and I as stable states and others as transient states. Initially, for all blocks, the state is set to I, the owner is set to memory, and the bit-vector is set to encode an empty set of sharers. The state notation is the same as for processor nodes, although the state MX^A refers to the situation in which a directory is in M and receives data, but has not seen the corresponding coherence request yet and, therefore, does not know (or care) whether it is PUTX data or data from a processor that is downgrading from M to S in response to another processor's GETS.

A directory node inspects its incoming queues for the address and data networks and removes the message at the head of a queue (if any). Depending on the incoming message and the current block state, a directory may inject a new message into an outgoing queue and may change the state of the block. For simplicity, a directory currently delays all requests for a block for which a PUTX or downgrade is outstanding.[3]

The directory events, actions and transitions are listed in Tables 15, 16, and 17, respectively. The action "z" ("delay transactions to this block") relies on the fact that a directory can delay address messages for a given block arbitrarily while waiting for a data message. Conceptually, we have one directory per block. Since there is more than one block per directory, an implementation would have to be able to delay only those transactions which are for the specific block. Note that consecutive GETS transactions for the same block could be coalesced.

3. This restriction maintains the invariant that there is at most one data message per block that the directory can receive, thus eliminating the need for buffers and preserving the sanity of the protocol developers.

2.3 Online Interactive Protocol Tables

We have developed software that automatically maps protocol specifications to interactive web documentation. Specifications for both protocols are available at url: http://www.cs.wisc.edu/multifacet/public/ieeetpds2002. The specifications are interactive in a couple of ways. First, clicking on states, actions, and events provides descriptions of them. Second, clicking on a state shows all ways to reach that state by highlighting transition table entries for which that state is the next state. This Web documentation has proven useful to us in debugging the protocols as well as communicating the protocols with collaborators.

3 Verification of Snooping Protocols

In this section, we present a methodology for proving that a specification is sequentially consistent, and we show how this methodology can be used to prove that our multicast protocol satisfies SC. Our method uses an extension of Lamport's logical clocks [20] to timestamp the load and store operations performed by the protocol. Timestamps

TABLE 8
Broadcast Snooping Memory Controller Actions

Action	Description
c	Set owner equal to directory.
d	Send data message to requestor.
j	Pop address queue.
k	Pop data queue.
m	Set owner equal to requestor.
w	Write data to memory.
z	Delay transactions to this block.

TABLE 9
Broadcast Snooping Memory Controller Transitions

State	Other Home	GETS	GETX	PUTX (requestor is owner)	PUTX (requestor not owner)	Data
S	j	dj	dmj/M	j	j	
M	j	cj/MS^D	mj	cj/MS^D	j	wk/MS^A
MS^A	j	cj/S	mj	cj/S	j	
MS^D	j	z	z	j	j	wk/S

determine how operations should be reordered to witness SC, as intended by the designer of the protocol. Logical clocks and the associated timestamping actions are a *conceptual* augmentation of the protocol and are specified using the same table-based transition tables as the protocol itself. We note that the set of all possible execution traces of the protocol equals that of the augmented protocol.

The process of developing a timestamping scheme is a valuable debugging tool in its own right. For example, an early implementation of the multicast protocol did not include a ForwardProgress bit in the TBE, and, upon receiving the data for a GETX request when in state IM^DI, always satisfied an OP at the head of the mandatory queue before forwarding the data. Attempts to timestamp OP revealed the need for a forward progress bit, roughly to ensure that OP can indeed be timestamped so that it appears to occur just after the ("earlier") time of the GETX, and that this OP's logical timestamp also respects program order.

In brief, our methodology for proving sequential consistency consists of the following steps.

- Augment the system with logical clocks and with associated actions that assign timestamps to LD and ST operations as the protocol operates. The logical clocks are purely conceptual devices introduced for verification purposes and are never implemented in hardware.
- Associate a *global history* with any execution of the augmented protocol. Roughly, the history includes the *configuration* at each node of the system (states, TBEs, cache contents, logical clocks, and queues), the totally ordered sequence of transactions delivered by the network, and the memory operations serviced so far, in program order, along with their logical timestamps.
- Using invariants, define the notion of a *legal global history*. The invariants are quite intuitive when expressed using logical timestamps. It follows immediately from the definition of a legal global history that the corresponding execution is sequentially consistent.
- Finally, prove that the initial global history of the system is legal, that each transition of the protocol maps legal global histories to legal global histories, and that the entries labeled "impossible" in the protocol specification tables are indeed impossible. It then follows by induction that the protocol is sequentially consistent.

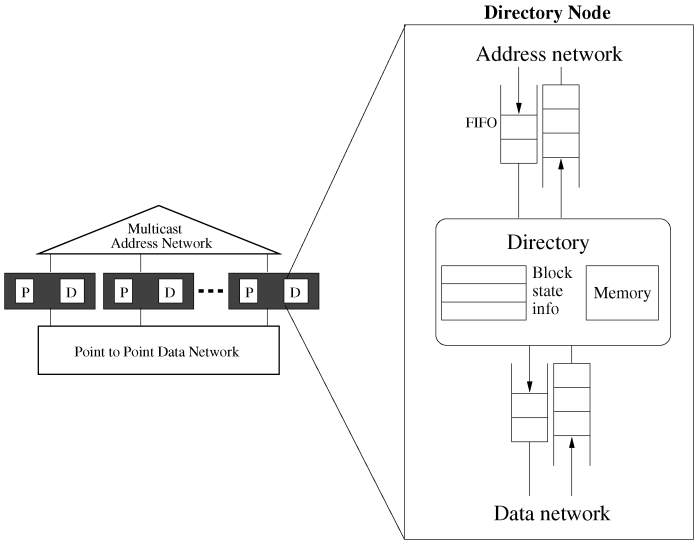

Fig. 3. Multicast snooping system.

TABLE 10
Multicast Snooping Cache Controller States

TBE State	Cache State	Description
	I	Invalid
	S	Shared
	M	Modified
IS^{AD}	busy	invalid, issued GETS, have not seen GETS or data yet
IM^{AD}	busy	invalid, issued GETX, have not seen GETX or data yet
SM^{AD}	busy	shared, issued GETX, have not seen GETX or data yet
IS^{A}	busy	invalid, issued GETS, have not seen GETS, have seen data
IM^{A}	busy	invalid, issued GETX, have not seen GETX, have seen data
SM^{A}	busy	shared, issued GETX, have not seen GETX, have seen data
IS^{A*}	busy	invalid, issued GETS, have not seen GETS, have seen nack
IM^{A*}	busy	invalid, issued GETX, have not seen GETX, have seen nack
SM^{A*}	busy	shared, issued GETX, have not seen GETX, have seen nack
MI^{A}	I	modified, issued PUTX, have not seen PUTX yet
II^{A}	I	modified, issued PUTX, have not seen PUTX, then saw other GETS or GETX
IS^{D}	busy	invalid, issued GETS, have seen GETS, have not seen data yet
$IS^{D}I$	busy	invalid, issued GETS, have seen GETS, have not seen data, then saw other GETX
IM^{D}	busy	invalid, issued GETX, have seen GETX, have not seen data yet
$IM^{D}S$	busy	invalid, issued GETX, have seen GETX, have not seen data yet, then saw other GETS
$IM^{D}I$	busy	invalid, issued GETX, have seen GETX, have not seen data yet, then saw other GETX
$IM^{D}SI$	busy	invalid, issued GETX, have seen GETX, have not seen data yet, then saw other GETS, then saw other GETX
SM^{D}	busy	shared, issued GETX, have seen GETX, have not seen data yet
$SM^{D}S$	busy	shared, issued GETX, have seen GETX, have not seen data yet, then saw other GETS

The first step above, that of augmenting the system with logical clocks, can be done hand in hand with development of the protocol. Thus, it is, on its own, a valuable debugging tool. The second step is straightforward. It is also straightforward to select a core set of invariants in the third step that are strong enough to guarantee that the execution corresponding to any legal global history is sequentially consistent. The final step of the proof methodology above requires a proof for every transition of the protocol and every invariant, and may necessitate the addition of further invariants to the definition of legal. This step of the proof, while not difficult, is certainly tedious.

In the rest of this section, we describe the first three steps of this process in more detail, namely how the multicast protocol is augmented with logical clocks, and what is a global history and a legal global history. We include examples of the cases covered in the final proof step in Appendix A.

3.1 Augmenting the System with Logical Clocks

In this section, we shall describe how we augment the system specified earlier with logical clocks and with actions that increment clocks and timestamp operations and data. These timestamps will make future definitions (of global states and legal global states) simpler and more intuitive. *These augmentations do not change the behavior of the system as originally specified.*

3.1.1 The Augmented System

The system is augmented with the following counters, all of which are initialized to zero:

- One counter (global pulse number) associated with the multicast address network.
- Two counters (global and local clocks) associated with each processor node of the system.
- One counter (pulse number) added to each data field and to each ForwardID field of each TBE.
- One counter (pulse number) field added to each data message.
- One counter (global clock) associated with each directory node of the system.

3.1.2 Behavior of the Augmented System

In the augmented system, the clocks get updated and timestamps (or pulses) are assigned to operations and data upon transitions of the protocol according to the following rules.

Network. Each new address transaction that is appended to the total order of address transactions by the network causes the global pulse number to increment by 1. The new value of the pulse number is associated with the new transaction.

Processor. Tables 18 and 19 describe how the global and local clocks are updated. The TBE counter is used to record

TABLE 11
Multicast Snooping Cache Controller Events

Event	Description	Block B
Load	LD at head of Mandatory queue	address of LD at head of Mandatory Queue
Read-Only Prefetch	Read-Only Prefetch at head of Optional queue	address of Read-Only Prefetch at head of Optional Queue
Store	ST at head of Mandatory queue	address of ST at head of Mandatory Queue
Read-Write Prefetch	Read-Write Prefetch at head of Optional queue	address of Read-Write Prefetch at head of Optional Queue
Mandatory Replacement	LD/ST at head of Mandatory queue for which no cache block is available	address of victim block for LD/ST at head of Mandatory queue
Optional Replacement	Read-Write Prefetch at head of Optional queue for which no cache block is available	address of victim block for Prefetch at head of Optional queue
Own GETS	Occurs when we observe our own GETS request in the global order	address of transaction at head of incoming address queue
Own GETX	Occurs when we observe our own GETX request in the global order	same as above
Own GETS (mismatch)	Occurs when we observe our own GETS request in the global order, but the RetryCount of the GETS does not match RetryCount of the TBE	same as above
Own GETX (mismatch)	Occurs when we observe our own GETX request in the global order, but the RetryCount of the GETS does not match RetryCount of the TBE	same as above
Own PUTX	Occurs when we observe our own PUTX request in the global order	same as above
Other GETS	Occurs when we observe a GETS request from another processor	same as above
Other GETX	Occurs when we observe a GETX request from another processor	same as above
Other PUTX	Occurs when we observe a PUTX request from another processor	same as above
Data	Data for this block arrives	address of message at head of incoming data queue
Data (mismatch)	Data for this block arrives, but the RetryCount of the data message does not match RetryCount of the TBE	address of message at head of incoming data queue

the timestamp of a request that cannot be satisfied until the data arrives. When the data arrives, the owner sends the data with the timestamp that was saved in the TBE.

Directory. Briefly, upon handling any transaction, the directory updates its clock to equal the global pulse of that transaction. The pulse attached to any data message is set to be the value of the directory's clock.

3.2 Global Histories

The global history associated with an execution of the protocol is a 3-tuple $< TransSeq, Config, Ops >$. *TransSeq* records information on the sequence of transactions requested to date: the type of transaction, requester, address, mask, retry-number, pulse (possibly undefined), and status (successful, unsuccessful, nack, or undetermined). *Config* records the configuration of the nodes: state per block, cache contents, queue contents, TBEs, and logical clock values. *Ops* records properties of all operations generated by the CPUs to date: operations along with address, timestamp (possibly undefined), value, and rank in program order.

The global history is defined inductively on the sequence of transitions in the execution. In the initial global history, *TransSeq* and *Ops* are empty. In *Config*, all processors are in state I for all blocks, have empty queues, no TBEs, and clocks initialized to zero. For all blocks, the directory is in state I, the owner is set to the directory, and the list of sharers is empty. All incoming queues are empty. Upon each transition, *TransSeq, Ops, and Config* are updated in a manner consistent with the actions of that transition.

3.3 Legal Global Configurations and Legal Global Histories

There are several requirements for a global history such as $< TransSeq, Config, Ops >$ to be legal. Briefly, these are as follows. The first requirement is sufficient to imply sequential consistency. The remaining four requirements supply additional invariants that are useful in building up to the proof that the first requirement holds.

TABLE 12
Multicast Snooping Cache Controller Actions

Action	Description
a	Allocate TBE with Address=B, ForwardID=null, RetryCount=zero, ForwardIDRetryCount=zero, ForwardProgress bit=unset.
b	Set ForwardProgress bit if request at head of address queue satisfies request at head of Mandatory queue.
c	Set cache tag equal to tag of block B.
d	Deallocate TBE.
e	Record ID of requestor in ForwardID and record retry number of transaction in ForwardIDRetryCount.
f	Issue GETS: insert message in outgoing Address queue with Type=GETS, Address=B, Sender=N, RetryCount=zero.
g	Issue GETX: insert message in outgoing Address queue with Type=GETX, Address=B, Sender=N, RetryCount=zero.
h	Service load/store (a cache hit) from the cache and (if a LD) enqueue the data on the LD/ST data queue.
i	Pop incoming address queue.
j	Pop incoming data queue.
k	Pop mandatory queue.
l	Pop optional queue.
m	Send data from TBE to memory.
n	Send data from cache to memory.
o	Send data and ForwardIDRetryCount from the TBE to the processor indicated by ForwardID.
p	Issue PUTX: insert message in outgoing Address queue with Type=PUTX, Address=B, Sender=N.
q	Copy data from cache to TBE.
r	Send data from the cache to the requestor
s	Save data in data field of TBE.
t	Copy retry field from message at head of incoming Data queue to Retry field in TBE, set ForwardID = null, and set ForwardIDRetryCount=zero.
u	Service LD from TBE, pop mandatory queue, and enqueue the data on the LD/ST data queue if the LD at the head of the mandatory queue is for this block.
v	Treat as either h or z (optional cache hit). If it is a cache hit, then pop the mandatory queue.
w	Write data from the TBE into the cache.
x	If (and only if) ForwardProgress bit is set, service LD from TBE, pop mandatory queue,and enqueue the data on the LD/ST data queue.
y	Send data from the TBE to the requestor.
z	Cannot be handled right now. Either wait or discard request (can discard if this request is in the Optional queue).
α	Copy retry field from message at head of incoming address queue to Retry field in TBE, set ForwardID = null, and set ForwardIDRetryCount=zero.
γ	Service LD/ST from TBE, pop mandatory queue, and (if a LD) enqueue the data on the LD/ST data queue if the LD/ST at the head of the mandatory queue is for this block. (If ST, store data to TBE).
λ	Optionally service LD/ST from TBE.
δ	If (and only if) ForwardProgress bit is set, service LD/ST from TBE, pop mandatory queue,and (if a LD) enqueue the data on the LD/ST data queue.

- *Ops* is legal with respect to program order. That is, the following should hold:

3.3.1. *Ops* respects program order. That is, for any two operations O_1 and O_2, if O_1 has a smaller timestamp than O_2 in *Ops*, then O_1 must also appear before O_2 in program order.

3.3.2. Every LD returns the value of the most recent ST to the same address in timestamp order.

- *TransSeq* is legal. To describe the type of constraints that *TransSeq* must satisfy, we introduce the notion of A-state vectors. The A-state vector corresponding to *TransSeq* for a given block B records, for each processor N, whether *TransSeq* confers Shared (S), Modified (M), or no (I) access to block B to processor N. For example, in a system with three processors, if *TransSeq* consists of a successful GETS to block B by processor 1, followed by an unsuccessful GETX to block B by processor 2, followed by a successful GETS to block B by processor 3, then the corresponding A-state for

TABLE 13
Multicast Snooping Cache Controller Transitions

State	Load	read-only prefetch	Store	read-write prefetch	Mandatory Replacement	Optional Replacement	Own GETS	Own GETX	Own GETS (mismatch)	Own GETX (mismatch)	Own PUTX	Other GETS	Other GETX	Other PUTX	Data	Data (mismatch)	nack	nack (mismatch)
I	caf/ IS^{AD}	cafl/ IS^{AD}	cag/ IM^{AD}	cagl/ IM^{AD}								i	i	i				
S	hk	l	ag/ SM^{AD}	agl/ SM^{AD}	I	I						i	i/I	i				
M	hk	l	hk	l	aqp/ MI^{A}	aqp/ MI^{A}						rmi/S	ri/I	i				
IS^{AD}	z	l	z	z	z	z	bi/ IS^{D}					i	i	i	sj/ IS^{A}	stj/ IS^{A}	tj/ IS^{A*}	tj/ IS^{A*}
IM^{AD}	z	l	z	l	z	z		bi/ IM^{D}				i	i	i	sj/ IM^{A}	stj/ IM^{A}	tj/ IM^{A*}	tj/ IM^{A*}
SM^{AD}	v	l	z	l	z	z		bi/ SM^{D}				i	i/ IM^{AD}	i	sj/ SM^{A}	stj/ SM^{A}	tj/ SM^{A*}	tj/ SM^{A*}
IS^{A*}	z	l	z	z	z	z	di/I		i			i	i	i				
IM^{A*}	z	l	z	l	z	z		di/I		i		i	i	i				
SM^{A*}	v	l	z	l	z	z		di/S		i		i	i/ IM^{A*}	i				
IS^{A}	z	l	z	z	z	z	uwdi/ S		i			i	i	i				
IM^{A}	z	l	z	l	z	z		γwdi/ M		i		i	i	i				
SM^{A}	v	l	z	l	z	z		γwdi/ M		i		i	i/ IM^{A}	i				
MI^{A}	λ	z	λ	z	z	z					mdi/I‡	ymi/ II_{A}	yi/ II^{A}	i				
II^{A}	z	z	z	z	z	z					di/I‡	i	i	i				
IS^{D}	z	l	z	z	z	z			αi			i	i/ IS^{D}I	i	suwdj/ S	stj/ IS^{A}	dj/I	tj
IS^{D}I	z	z	z	z	z	z			αi/ IS^{D}			i	i	i	sxdj/I	stj/ IS^{A}	dj/I	tj
IM^{D}	z	l	z	l	z	z				αi		ei/ IM^{D}S	ei/ IM^{D}I	i	sγwdj/ M	stj/ IM^{A}	dj/I	tj
IM^{D}S	z	l	z	z	z	z				αi/ IM^{D}		i	i/ IM^{D} SI	i	sδom- wdj/S	stj/ IM^{A}	dj/I	tj
IM^{D}I	z	z	z	z	z	z				αi/ IM^{D}		i	i	i	sδodj/ I	stj/ IM^{A}	dj/I	tj
IM^{D}SI	z	z	z	z	z	z				αi/ IM^{D}		i	i	i	sδomd j/I	stj/ IM^{A}	dj/I	tj
SM^{D}	z	l	z	l	z	z				αi		ei/ SM^{D}S	ei/ IM^{D}I	i	sγwdj/ M	stj/ SM^{A}	dj/S	tj
SM^{D}S	z	l	z	z	z	z				αi/ SM^{D}		i	i/ IM^{D}SI	i	sγom- wdj/S	stj/ SM^{A}	dj/S	tj

‡*Only change the cache state to I if the tag matches.*

block B is (S,I,S). The constraints on *TransSeq* require, for example, that a GETX on block B should not be successful if its mask does not include all processors that, upon completion of the transaction just prior to the GETX, may have Shared or Modified access to B. That is, if *TransSeq* consist of *TransSeq'* followed by GETX on block B and $\underline{A}$ is the A-state for block B corresponding to *TransSeq'*, then the mask of the GETX should contain all processors whose entries in $\underline{A}$ are not equal to I. The precise definition of a legal transaction sequence is included in Appendix A.

- *Ops* is legal with respect to *TransSeq*. Intuitively, for all operations *op* in *Ops*, if *op* is performed by processor N at global timestamp t, then the A-state for processor N at logical time t should be either S or M and should be M if *op* is a ST.

TABLE 14
Multicast Snooping Memory Controller States

State	Description
I	Invalid - all processors are Invalid
S	Shared - at least one processor is Shared
M	Modified - one processor is Modified and the rest are Invalid
MX^A	Modified, have not seen GETS/PUTX, have seen data
MS^D	Modified, have seen GETS, have not seen data
MI^D	Modified, have seen PUTX, have not seen data

TABLE 15
Multicast Snooping Memory Controller Events

Event	Description	Block B
Other Home	A request arrives for a block whose home is not at this memory	address of transaction at head of incoming address queue
GETS	GETS with successful mask at head of incoming address queue	address of transaction at head of incoming address queue
GETX	GETX with successful mask at head of incoming address queue	same as above
GETS-RETRY	GETS with unsuccessful mask at head of incoming queue. Room in outgoing address queue for a retry.	same as above
GETS-NACK	GETS with unsuccessful mask at head of incoming queue. No room in outgoing address queue for a retry.	same as above
GETX-RETRY	GETX with unsuccessful mask at head of incoming queue. Room in outgoing address queue for a retry.	same as above
GETX-NACK	GETX with unsuccessful mask at head of incoming queue. No room in outgoing address queue for a retry.	same as above
PUTX (requestor is owner)	PUTX from owner at head of incoming address queue.	same as above
PUTX (requestor is not owner)	PUTX from non-owner at head of incoming address queue.	same as above
Data	Data message at head of incoming data queue	address of message at head of incoming data queue

- *Config* is legal with respect to *TransSeq*. This involves several constraints, since there are many components to *Config*. For example, if processor N is in state IS^{AD} for block B, then a GETS for block B, requested by N, with timestamp greater than that of N (or undefined) should be in *TransSeq*.
- *Config* is legal with respect to *Ops*. That is, for all blocks B and nodes N, the following should hold:

3.3.3. If N is a processor and its state for block B is one of S, M, MI^A, SM^{AD}, or SM^A, then the value of block B in N's cache equals that of the most recent ST in *Ops*, relative to N's clock. By "most recent ST relative to N's clock," we mean a ST whose timestamp is less then or equal to N's clock.

3.3.4. If N is a processor and block B is in one of N's TBEs, then its value equals that of the most recent ST in *Ops*, relative to p.0.0, where p is the pulse in the data field of the TBE.

3.3.5. If data for block B is in N's incoming data queue, its value equals the most recent ST in *Ops* (relative to the data's timestamp, not N's current time).

3.3.6. If N is the directory of block B, then for each block B for which N is the owner, its value equals that of the most recent ST in *Ops* (relative to N's clock).

3.4 Properties of Legal Global Histories

It is not hard to show that the global history of the system is initially legal. The main task of the proof is to show the following:

Theorem 1. *Each protocol transition takes the system from a legal global history to a legal global history.*

To illustrate how Theorem 1 is proven, we include in Appendix A the proof of why the transition at each entry of Table 13 (cache controller transitions) maps a legal global history, $< TransSeq, Ops, Config >$, to a new global

TABLE 16
Multicast Snooping Memory Controller Actions

Action	Description
c	Clear set of sharers.
d	Send data message to requestor with RetryCount equal to RetryCount of request.
j	Pop address queue.
k	Pop data queue.
m	Set owner equal to requestor.
n	Send nack to requestor with RetryCount equal to RetryCount of request.
q	Add owner to set of sharers.
r	Retry by re-issuing the request. Before re-issuing, the directory improves the multicast mask and increments the retry field. If the transaction has reached the maximum number of retries, the multicast mask is set to the broadcast mask.
s	Add requestor to set of sharers.
w	Write data to memory.
x	Set owner equal to directory.
z	Delay transactions to this block.

TABLE 17
Multicast Snooping Memory Controller Transitions

State	Other Home	GETS	GETX	GETS - RETRY (Unsuccessful mask)	GETS - NACK (unsuccessful mask)	GETX - RETRY (unsuccessful mask)	GETX - NACK (unsuccessful mask)	PUTX (requestor is owner)	PUTX (requestor not owner)	Data
I	j	dsj/S	dmj/M						j	
S		dsj	cdmj/M			rj	nj		j	
M		qsxj/MS^D	mj	rj	nj	rj	nj	xj/MI^D	j	wk/MX^A
MX^A		qsxj/S	mj	rj	nj	rj	nj	xj/I	j	
MS^D		z	z			rj	nj		j	wk/S
MI^D		z	z						j	wk/I

history, $< TransSeq', Ops', Config' >$ in which $TransSeq'$ is legal.

4 Related Work

We focus on papers that specify and prove a complete protocol correct, rather than on efforts that focus on describing many alternative protocols and consistency models, such as [2], [12]. There is a large body of literature on the subject of formal protocol verification,[4] which we have classified into a taxonomy along two independent axes: *automation* and *completeness* [27]. We distinguish verification methods based on the level of automation they support: manual, semiautomated or automated. Manual methods involve humans who read the specification and construct the proofs. Semiautomated methods involve some computer programs (a model checker or theorem prover) which are guided by humans who understand the specification and provide the programs with the invariants or lemmas to prove. Automated methods take the human out of the loop and involve a computer program that reads a specification and produces a correctness proof completely automatically. For our classification, we also consider a technique to be automated if it uses human-written specification invariants which are reusable across multiple protocols. We also distinguish techniques that are complete (i.e., prove that a system implements a particular consistency model) from those that are incomplete (i.e., prove coherence or selected invariants). Table 20 provides a

4. Formal methods involve construction of rigorous mathematical proofs of correctness while informal methods include such techniques as simulation and random testing which do not guarantee correctness. We only consider formal methods in this review.

TABLE 18
Processor Clock Actions

Action	Description
g	Set global clock equal to pulse of transaction being handled, and set local clock to zero
h	Increment local clock. The timestamp of the LD/ST is set equal to the associated global and local clock values.
i	Set TBE ForwardID pulse equal to transaction pulse.
k	Optionally treat as h.
o	Set data message pulse equal to TBE ForwardID pulse.
t	Set TBE data pulse equal to pulse of incoming data message.
u	If first Op in Mandatory queue is a LD for this block, then increment local clock. The timestamp of the LD/ST is set equal to the associated global and local clock values.
v	If first Op in Mandatory queue is a LD/ST for this block, then increment local clock. The timestamp of the LD/ST is set equal to the associated global and local clock values.
x	If ForwardProgress bit is set (i.e., head of Mandatory Queue is a LD or this block), then no clock update, set global timestamp of LD equal to pulse of incoming data message, and set local clock value equal to 1.
y	Set data message pulse equal to transaction pulse.
z	Same as x, but allow a LD or ST for this block.

summary of our taxonomy. We discuss each column of the table separately below.

Manual Techniques. Lazy caching [3] is one of the earliest examples of a formal specification and verification of a protocol (lazy caching) that implements sequential consistency. The authors use I/O automata as their formal system models and provide a manual proof that a lazy caching system implements SC. Their use of history variables in the proof is similar to the manner in which we use Lamport Clock timestamps in our proofs. Verification of the Alpha 21264 [4], [34] is a manual proof that the implementation conforms to the Alpha memory model. Both the implementation and memory model are specified in TLA+ (a form of temporal logic). Although they did find a bug that would not have been caught by simulation, their manual proofs are quite large and only a small portion could be finished even with 4 people and 7 person-months of effort. Gibbons et al. [14] provide a framework for verifying that shared memory systems implement relaxed memory models. The method involves specifying both the system to be verified as well as an operational definition of a memory model as I/O automata and then proving that the system automaton implements the model automaton. As an example, they provide a specification of the Stanford DASH memory system and manually prove that it implements the Release Consistency memory model. Our table-based specification methodology is complementary in that it could also be used to describe I/O automata.

Our previous papers [31], [25], [7], [17] specified various shared memory systems (directory and bus protocols) at a high level, and employed manual proofs using our Lamport Clocks technique to show that these systems implemented various memory models (SC, TSO, Alpha). This paper is our latest effort which demonstrates our technique applied to more detailed table-based specifications of snooping protocols. Shen and Arvind [30] propose using term rewriting systems (TRSs) to both specify and verify memory system protocols. Their verification technique involves showing that the system under consideration and the operational definition of a memory model, when expressed as TRSs, can simulate each other. This proof technique is similar to the I/O automata approach used by Gibbons et al. [14]. Both TRSs and our table-based specification method can be used in a modular and flexible fashion. A drawback of TRSs is that they lack the visual clarity of our table-based specification. Although their current proofs are manual, they mention the possibility of using a model checker to automate tedious parts of the proof.

Semiautomated Techniques. Henzinger et al. [16] provide semiautomated proofs of lazy caching and a certain snoopy cache coherence protocol using the MOCHA model checker. Their protocol specifications—with the system being expressed in a language similar to a typical imperative programming language and the proof requirements expressed in temporal logic—are augmented with a (manually constructed) specification of a "finite observer" which can reorder protocol transactions in order to produce a witness ordering which satisfies the definition of a memory model. They provide such observers for the two protocols they specify in the paper. However, the general problem of automatically verifying sequential consistency is undecidable and such finite observers do not exist for the protocols we specify in this paper or in the protocols used in modern high-performance shared-memory multiprocessors. Park and Dill [24] express both the definition of the memory model and the system being verified in the same specification language and then use manually constructed aggregation functions to map the system specification to the model specification (similar to the use of TRSs by Shen and Arvind [30] and I/O automata by Gibbons et al. [14]). As an example, they specify the Stanford FLASH protocol in the language of the PVS theorem prover (the language is a typed high-order logic) and use this aggregation technique to prove that the "Delayed" mode of the FLASH memory

TABLE 19
Processor Clock Updates

Current State	Processor/Cache Request						See Own					See Other		See Own			
							Retry Match		Retry Mismatch					DATA		NACK	
	LD	read-only prefetch	ST	read-write prefetch	Mandatory Replacement	Optional Replacement	GETS	GETX	GETS	GETX	PUTX	GETS	GETX	Retry Match	Retry Mismatch	Retry Match	Retry Mismatch
I												g	g				
S	h											g	g				
M	h		h									gy	gy				
IS^{AD}							g					g	g	t	t		
IM^{AD}								g				g	g	t	t		
SM^{AD}	k							g				g	g	t	t		
IS^{A*}							g		g			g	g				
IM^{A*}								g		g		g	g				
SM^{A*}	k							g		g		g	g				
IS^{A}							gu		g			g	g				
IM^{A}								gv		g		g	g				
SM^{A}	k							gv		g		g	g				
MI^{A}	k		k								gy	gy	gy				
II^{A}											gy	g	g				
IS^{D}									g			g	g	ht	t		
$IS^{D}I$									g			g	g	xt	t		
IM^{D}										g		gi	gi	ht	t		
$IM^{D}S$										g		g	g	zot	t		
$IM^{D}I$										g		g	g	zot	t		
$IM^{D}SI$										g		g	g	zot	t		
SM^{D}	k									g		gi	gi	ht	t		
$SM^{D}S$	k									g		g	g	zot	t		

system is sequentially consistent. Akhiani et al. [4] and Yu and Tuttle [34] summarize their experience with using TLA+ and a TLA+ model checker (TLC) to specify and verify the Compaq Alpha 21364 memory system protocol. The TLA+ specification is complete and formal, but it requires nearly two thousand lines to specify this complicated protocol. The invariants are hardware-dependent and, thus, need to be constructed manually in a protocol-specific fashion.

Automated Techniques. Qadeer [29] develops a model checking algorithm that proves that a memory system conforms to a memory model, given assumptions about fixed system size and certain properties of the system and the memory model. The state space explosion is still a problem for the model checker, but this is a promising technique. Eiriksson et al. [10], [9] describe a methodology which integrates design and verification where common state machine tables drive a model checker and generators of simulators and documentation. The protocol specification tables they describe were designed to be consumed by automated generators rather than by humans, and they do not describe the format of the text specifications generated from these tables. They use the SMV model checker (which accepts specifications in temporal logic) to prove the coherence of the protocol used in the SGI Origin 2000. However, the system verified had only one cache block (which is sufficient to prove coherence, but not consistency). Nalumasu et al. [13], [22] propose an extension of Collier's ArchTest suite which provides a collection of programs that test certain properties of a memory model. Their extension

TABLE 20
Classification of Related Work

	Manual	Semi-automated	Automated
Complete method	lazy caching [3] Alpha 21264 [4,34], DASH memory model [14] Lamport Clocks [31,25,7,17], Lamport Clocks (this paper), term rewriting [30]	lazy & snoopy caching [16]	Compaq model checking [29]
Incomplete method		FLASH coherence [24], Alpha 21364 [4,34]	Origin2000 coherence [10,9], HP Runway testing [13,22], RMO testing [23], S3.mp coherence [26], delayed consistency [28] Cray SV2 testing [1]

creates the effect of having infinitely long test programs (and, thus, checking all possible interleavings of test programs) by abstracting the test programs into nondeterministic finite automata which drive formal specifications of the system being verified. Both the automata and the implementations are specified in Verilog and the VIS symbolic model checker is used to verify that various invariants are satisfied by the system when driven by these automata. The technique is useful in practice and has been applied to commercial systems such as the HP PA-8000 Runway bus protocol. However, it is incomplete in that the invariants being tested do not imply SC (they are necessary, but not sufficient). Park and Dill [23] provide an executable specification of the Sun RMO memory model written in the language of the Murφ model checker. They use this specification to verify properties of small synchronization routines. Pong et al. [26] verify the memory system of the Sun S3.mp multiprocessor using the Murφ and SSM (Symbolic State Model) model checkers, but again the verified system had only one cache block and, thus, their method cannot verify whether the system satisfies a memory model. Pong and Dubois [28] use SSM to verify the coherence, but not the consistency, of a system that implements delayed consistency. Abts et al. [1] specify the Cray SV2 protocol in the Murφ input language, and they then verify that the protocol and its implementation (specified in RTL) meet certain requirements, but they do not verify that either satisfy a consistency model. The protocol verification uses Murφ to ensure that several coherence invariants are satisfied, and the implementation verification uses protocol execution traces as input to an RTL simulator.

5 Conclusions

In this paper, we have developed a specification methodology that documents and specifies a cache coherence protocol in eight tables: the states, events, actions, and transitions of the cache and memory controllers. We have used this methodology to specify in detail a modern three-state broadcast snooping protocol with an unordered data network and an ordered address network which allows arbitrary skew. We have also presented a detailed specification of the Multicast Snooping protocol [6], and, in doing so, we have shown the utility of the table-based specification methodology. Lastly, we have demonstrated a technique for verification of the Multicast Snooping protocol, through the sketch of a manual proof that the specification satisfies a sequentially consistent memory model.

Appendix A

Excerpt from Proof of Sequential Consistency

In this appendix, we include a precise definition of a legal transaction sequence, and we prove that processor transitions map legal histories to histories in which the transaction sequence is legal. Other parts of the proof can be done in a similar manner.

A.1 Legal Transaction Sequences

Intuitively, the definition of a legal transaction sequence rules out sequences that do not make sense. For example, a GETX on block B in which the mask does not include all processors that "currently" may have Shared or Modified access to B should not be successful. By "currently," we are referring to a moment in which all transactions occurring before the GETX to block B in the transaction sequence are completed, and no further transactions are yet handled. We use an *A-state vector* to record the type of access each processor may have to a given block upon completion of a sequence of transactions. The A-state vector for block B has P elements, each of which is either Invalid, Shared, or Modified. Also, throughout the appendix, we denote an entry of the transaction sequence as a tuple

$$< Trans, Mask, RetryNum, Pulse, Status >,$$

where *Trans* is a triple denoting the requester, address, and transaction type (GETX, GETS, or PUTX) and the meaning of the remaining entries of the tuple should be clear from the description in Section 3.2.

We first define the notion of a *determined legal transaction sequence* and its associated A-state. Here, *determined* simply refers to the fact that the outcomes of

TABLE 21
Successful Transactions

GETS	$\underline{A}_r$' = S and for any i with $\underline{A}_i$ = M, $\underline{A}_i$' = S.
GETX	$\underline{A}_r$' = M and for i not equal to r, $\underline{A}_i$' = I.
PUTX	$\underline{A}_r$' = I.

all transactions in the sequence have been determined to be success, failure, or nack.

- The empty sequence () is a determined legal transaction sequence with associated A-state vectors $\underline{A} =< \mathrm{I}, \mathrm{I}, \ldots, \mathrm{I} >$ for each block.
- If *TransSeq* denotes a determined legal transaction sequence, then

$$TransSeq' = (TransSeq, < Trans, Mask, RetryNum, Pulse, Status >)$$

is also a determined legal transaction sequence if the following conditions are true. In what follows, let *Trans* be on block *B*, and let the requester of *Trans* be *r*.

A.1.1. Status cannot be UNDETERMINED.

A.1.2. If *Status* = SUCCESS, then *Mask* is *sufficient* with respect to *TransSeq*. A mask $\underline{M}$ is sufficient with respect to *TransSeq* if, when $\underline{A}$ is the A-state vector for block B associated with *TransSeq*, we have $\underline{M}_i = 1$ for all nodes i with $\underline{A}_i = M$ or $\underline{A}_i = S$.

A.1.3. If *RetryNum* is 0, then the most recent tuple in *TransSeq* with requester = N on block B (if any) has status that is either SUCCESS or NACK. If *RetryNum* is greater than 0 then the most recent tuple in *TransSeq* with requester = N on block B must have the same transaction type as *Trans*, must have a retry number that is less than *RetryNum*, and must have *Status* = FAILURE.

The A-state associated with *TransSeq'* for all blocks other than block B is the same as that associated with *TransSeq*. For block B, the A-state $\underline{A}'$ associated with *TransSeq'* is the same as $\underline{A}$ except for the changes in Table 21).

Finally, a transaction sequence *TransSeq* is a *legal transaction sequence* if the following conditions hold:

A.1.4. *TransSeq* is a concatenation of a determined legal transaction sequence, $TransSeq_D$, with a sequence of tuples whose *Status* is UNDETERMINED.

A.1.5. Tuples in *TransSeq* are ordered by *Pulse*, with UNDEFINED pulses occurring in arbitrary order at the end of the sequence.

A.1.6. Tuples in *TransSeq* with determined *Status* must also have a defined *Pulse*.

A.1.7. For all N and B there is at most one tuple in *TransSeq* with requester=N, address=B, and *Status*=UNDETERMINED.

A.1.8. For each tuple T in *TransSeq* with *Status*=UNDETERMINED, if the *Status* of T is replaced by FAILURE or NACK and *Pulse* is set to a defined value, then $TransSeq_D$, T is a determined legal transaction sequence.

A.1.9. For each tuple T in TransSeq with *Status*=UNDETERMINED, if the mask of T is sufficient with respect to $TransSeq_D$ (as defined in Condition A.1.2 above), the status of T is replaced by SUCCESS, and *Pulse* is set to a defined value, then $TransSeq_D$, T is a determined legal transaction sequence.

In what follows, suppose that a determined legal transaction sequence of length at least t is fixed and a block B is fixed. Let $\underline{A}$ be the A-state for block B associated with the prefix of this transaction sequence of length t. Then, we say that the A-state of processor i at time t is $\underline{A}_i$ and we denote it by $\underline{A}_i(t)$.

A.2 Cache Controller Transitions Map Legal Histories to Histories with Legal Transaction Sequences

Each entry of Table 22 points to the proof of why the transition at the corresponding entry of 13 (cache controller transition specification), maps a legal global history, <*TransSeq*,*Ops*,*Config*>, to a new global history, <*TransSeq'*,*Ops'*,*Config'*> in which *TransSeq'* is legal. As usual, the transition is done by node N on block B, and we assume that the logical time of N (in *Config*) is t.

In the case of (a) in Table 22, processor N's state is I, S, or M. By construction of the protocol, Tables 12 and 13, actions f, g, or p, a transaction T is issued with type GETS, GETX, or PUTX. By action a, the retry number must be 0. Therefore, *TransSeq'* = *TransSeq*, T, where

$$T = \begin{array}{l} << B, TYPE, N >, \underline{M}, 0, UNDEFINED, \\ \quad UNDETERMINED > . \end{array}$$

For each condition of the definition of a legal transaction sequence, we list the reasons why *TransSeq'* satisfies that condition. Throughout, we denote the determined legal prefix of *TransSeq* by $TransSeq_D$; note that this is also the determined prefix of *TransSeq'*.

A.1.4. *TransSeq'* is a concatenation of a determined legal transaction sequence with a sequence of tuples whose status is UNDETERMINED, since *TransSeq* is such a sequence and since T has UNDETERMINED status.

A.1.5. Tuples in *TransSeq'* are ordered by pulse, with UNDEFINED pulses occurring in arbitrary order at the end of the sequence, since *TransSeq* satisfies this property and T has UNDEFINED pulse.

A.1.6. Tuples in *TransSeq'* with determined status must also have a defined pulse, since all tuples of *TransSeq'* with determined status are in *TransSeq* and *TransSeq* satisfies A.1.6.

TABLE 22
Legality of Transition from $< Trans, Ops, Config >$ to $< Trans', Ops', Config' >$, where the Transition Is Done by Processor N at Logical Time t, with Respect to Block B

State	Load	read-only prefetch	Store	read-write prefetch	Mandatory	Optional Replacement	Own GETS	Own GETX	Own GETS (mismatch)	Own GETX (mismatch)	Own PUTX	Other GETS	Other GETX	Other PUTX	Data	Data (mismatch)	nack	nack (mismatch)
I	a	a	a	a								z	z	z				
S	a	a	a	a	z	z						z	z	z				
M	z	z	z	z	a	a						z	z	z				
IS^{AD}	z	z	z	z	z	z	z					z	z	z	z	z	z	z
IM^{AD}	z	z	z	z	z	z		z				z	z	z	z	z	z	z
SM^{AD}	z	z	z	z	z	z		z				z	z	z	z	z	z	z
IS^{A*}	z	z	z	z	z	z	z		z			z	z	z				
IM^{A*}	z	z	z	z	z	z		z		z		z	z	z				
SM^{A*}	z	z	z	z	z	z		z		z		z	z	z				
IS^{A}	z	z	z	z	z	z	z		z			z	z	z				
IM^{A}	z	z	z	z	z	z		z		z		z	z	z				
SM^{A}	z	z	z	z	z	z		z		z		z	z	z				
MI^{A}	z	z	z	z	z	z					z	z	z	z				
II^{A}	z	z	z	z	z	z					z	z	z	z				
IS^{D}	z	z	z	z	z	z			z			z	z	z	z	z	z	z
$IS^{D}I$	z	z	z	z	z	z			z			z	z	z	z	z	z	z
IM^{D}	z	z	z	z	z	z				z		z	z	z	z	z	z	z
$IM^{D}S$	z	z	z	z	z	z				z		z	z	z	z	z	z	z
$IM^{D}I$	z	z	z	z	z	z				z		z	z	z	z	z	z	z
$IM^{D}SI$	z	z	z	z	z	z				z		z	z	z	z	z	z	z
SM^{D}	z	z	z	z	z	z				z		z	z	z	z	z	z	z
$SM^{D}S$	z	z	z	z	z	z				z		z	z	z	z	z	z	z

A.1.7. For (node, block) pairs other than (N,B), *TransSeq'* has at most one tuple with requester=node, address=block, and UNDETERMINED status since *TransSeq* satisfies this condition and since T has requester = N and address = B. It remains to show that among the tuples in *TransSeq* with status = UNDETERMINED, there are none with both requester = N and address = B. This follows because the definition of legal configuration (not included in this document) states that, if a processor N at logical time t is in one of states I, S, or M, then there is no transaction in *TransSeq* with requester = N, address = B, and pulse either $> t$ or undefined.

A.1.8. *TransSeq* satisfies A.1.8; thus it remains to show that $TransSeq_D$, $<< B, TYPE, N >, \underline{M}, 0, P, FAILURE >$ is a determined legal transaction sequence. This is true for the following reasons. First, $TransSeq_D$ is a determined legal transaction sequence, and so we need to show that Conditions A.1.1-A.1.3 are satisfied.

A.1.1. The status of T″ is not UNDETERMINED since it is FAILURE.

A.1.2. This does not apply, since the status is FAILURE.

A.1.3. Since *RetryNum* equals 0 it is sufficient to show that the most recent transaction in $TransSeq_D$ with requester = N and address = B has status equal to either SUCCESS or NACK. As in A.1.7 above, this follows because the definition of legal configuration states that, if a processor N at logical time t is in one of states I, S, or M, then there is no transaction in *TransSeq* with requester = N, address = B, and pulse either > t or undefined; moreover, the most recent transaction with requester = N and address = B is either SUCCESS or NACK. Since $TransSeq_D$ is a subsequence of *TransSeq* of length at least t, the same two properties must hold for $TransSeq_D$.

A.1.9. *TransSeq* satisfies A.1.9; thus it remains to show that $TransSeq_D$, $<< \mathrm{B}, \mathrm{TYPE}, \mathrm{N} >, \underline{M}, 0, \mathrm{P}, \mathrm{SUCCESS} >$ is a determined legal transaction sequence, assuming that the

Mask of T is sufficient. First, $TransSeq_D$ is a determined legal transaction sequence, and so we need to show that conditions A.1.1-A.1.3 are satisfied.

A.1.1. The status of T'' is not UNDETERMINED since it is SUCCESS.

A.1.2. The mask $\underline{M}$ is sufficient by assumption.

A.1.3. Identical argument as for A.1.8 above.

In the case of (z) in Table 22, $TransSeq' = TransSeq$ and *TransSeq* is legal. Therefore, $TransSeq'$ is legal.

ACKNOWLEDGMENTS

This work is supported in part by the US National Science Foundation with grants EIA-9971256, MIPS-9625558, MIP-9225097, CCR 9257241, and CDA-9623632, a Wisconsin Romnes Fellowship, and donations from Sun Microsystems and Intel Corporation. D. Sorin is supported by an Intel graduate fellowship. M. Martin is supported by an IBM graduate fellowship. Members of the Wisconsin Multifacet Project contributed significantly to improving the protocols and protocol specification model presented in this paper, especially Anastassia Ailamaki, Ross Dickson, Charles Fischer, and Carl Mauer.

REFERENCES

[1] D. Abts, D.J. Lilja, and S. Scott, "Toward Complexity-Effective Verification: A Case Study of the Cray SV2 Cache Coherence Protocol," *Proc. 27th Int'l Symp. Computer Architecture Workshop Complexity-Effective Design,* June 2000.

[2] S.V. Adve, "Designing Memory Consistency Models for Shared-Memory Multiprocessors," PhD thesis, Computer Sciences Dept., Univ. of Wisconsin-Madison, Nov. 1993.

[3] Y. Afek, G. Brown, and M. Merritt, "Lazy Caching," *ACM Trans. Programming Languages and Systems,* vol. 15, no. 1, pp. 182-205, Jan. 1993.

[4] H. Akhiani, D. Doligez, P. Harter, L. Lamport, J. Scheid, M. Tuttle, and Y. Yu, "Cache Coherence Verification with TLA+," *FM '99-Formal Methods, Volume II,* 1999.

[5] J. Archibald and J.-L. Baer, "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model," *ACM Trans. Computer Systems,* vol. 4, no. 4, pp. 273-298, Nov. 1986.

[6] E.E. Bilir, R.M. Dickson, Y. Hu, M. Plakal, D.J. Sorin, M.D. Hilland, D.A. Wood, "Multicast Snooping: A New Coherence Method Using a Multicast Address Network," *Proc. 26th Ann. Int'l Symp. Computer Architecture,* pp. 294-304, May 1999.

[7] A.E. Condon, M.D. Hill, M. Plakal, and D.J. Sorin, "Using Lamport Clocks to Reason About Relaxed Memory Models" *Proc. Fifth IEEE Symp. High-Performance Computer Architecture,* pp. 270-278, Jan. 1999.

[8] D.E. Culler and J.P. Singh, *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann, 1999.

[9] A. Eiriksson, A. Silbey, S. Venkataraman, and M. Woodacre, "Origin System Design Methodology and Experience: 1M-Gate ASICs and Beyond," *Proc. COMPCON '97,* 1997.

[10] A.T. Eiriksson and K.L. McMillan, "Using Formal Verification/Analysis Methods on the Critical Path in Systems Design: A Case Study," *Proc. Computer Aided Verification Conf.,* pp. 367-380, 1995.

[11] S.J. Frank, "Tightly Coupled Multiprocessor System Speeds Memory-access Times," *Electronics,* vol. 57, no. 1, pp. 164-169, Jan. 1984.

[12] K. Gharachorloo, "Memory Consistency Models for Shared-Memory Multiprocessors," PhD thesis, Computer System Laboratory, Stanford Univ., Dec. 1995.

[13] R. Ghughal, A. Mokkedem, R. Nalumasu, and G. Gopalakrishnan, "Using 'Test Model-Checking' to Verify the Runway-PA800 Memory Model," *Proc. 10th ACM Symp. Parallel Algorithms and Architectures,* pp. 231-239, June 1998.

[14] P.B. Gibbons, M. Merritt, and K. Gharachorloo, "Proving Sequential Consistency of High-Performance Shared Memories," *Proc. Third ACM Symp. Parallel Algorithms and Architectures,* pp. 292-303, July 1991.

[15] J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach,* second ed. Morgan Kaufmann, 1996.

[16] T.A. Henzinger, S. Qadeer, and S.K. Rajamani, "Verifying Sequential Consistency on Shared-Memory Multiprocessor Systems," Lecture Notes in Computer Science, pp. 301-315, 1999.

[17] M.D. Hill, A.E. Condon, M. Plakal, and D.J. Sorin, "A System-Level Specification Framework for I/O Architectures," *Proc. 11th ACM Symp. Parallel Algorithms and Architectures,* pp. 138-147, June 1999.

[18] J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, Apr. 1979.

[19] S.D. Johnson, "A Tabular Language for System Design," *Proc. Fourth NASA Langley Formal Methods Workshop,* Sept. 1997.

[20] L. Lamport, "Time, Clocks and the Ordering of Events in a Distributed System," *Comm. ACM,* vol. 21, no. 7, pp. 558-565, July 1978.

[21] L. Lamport, "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs," *IEEE Trans. Computers,* vol. 28, no. 9, pp. 690-691, Sept. 1979.

[22] R. Nalumasu, R. Ghughal, A. Mokkedem, and G. Gopalakrishnan, "The 'Test Model-checking' Approach to the Verification of Formal Memory Models of Multiprocessors," *Proc. Computer Aided Verification, 10th Int'l Conf.,* A.J. Hu and M.Y. Vardi, eds., pp. 464-476, June 1998.

[23] S. Park and D.L. Dill, "An Executable Specification, Analyzer and Verifier for RMO (Relaxed Memory Order)," *Proc. Seventh ACM Symp. Parallel Algorithms and Architectures,* pp. 34-41, July 1995.

[24] S. Park and D.L. Dill, "Verification of FLASH Cache Coherence Protocol by Aggregation of Distributed Transactions," *Proc. Eighth ACM Symp. Parallel Algorithms and Architectures,* pp. 288-296, June 1996.

[25] M. Plakal, D.J. Sorin, A.E. Condon, and M.D. Hill, "Lamport Clocks: Verifying a Directory Cache-Coherence Protocol," *Proc. 10th ACM Symp. Parallel Algorithms and Architectures,* pp. 67-76, June 1998.

[26] F. Pong, M. Browne, A. Nowatzyk, and M. Dubois, "Design Verification of the S3.mp Cache-Coherent Shared-Memory System," *IEEE Trans. Computers,* vol. 47, no. 1, pp. 135-140, Jan. 1998.

[27] F. Pong and M. Dubois, "Verification Techniques for Cache Coherence Protocols," *ACM Computing Surveys,* vol. 29, no. 1, pp. 82-126, Mar. 1997.

[28] F. Pong and M. Dubois, "Formal Automatic Verification of Cache Coherence in Multiprocessors with Relaxed Memory Models," *IEEE Trans. Parallel and Distributed Systems,* vol. 11, no. 9, pp. 989-1006, Sept. 2000.

[29] S. Qadeer, "On the Verification of Memory Models of Shared-Memory Multiprocessors," *Proc. Tutorial and Workshop on Formal Specification and Verification Methods for Shared Memory Systems,* Oct. 2000.

[30] X. Shen and Arvind, "Specification of Memory Models and Design of Provably Correct Cache Coherence Protocols," Group Memo 398, Mass. Inst. of Technology, June 1997.

[31] D.J. Sorin, M. Plakal, M.D. Hill, and A.E. Condon, "Lamport Clocks: Reasoning About Shared-Memory Correctness," Technical Report CS-TR-1367, Univ. of Wisconsin-Madison, Mar. 1998.

[32] P. Sweazey and A.J. Smith, "A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus," *Proc. 13th Ann. Int'l Symp. Computer Architecture,* pp. 414-423, June 1986.

[33] D.A. Wood, G.A. Gibson, and R.H. Katz, "Verifying a Multiprocessor Cache Controller Using Random Test Generation," *IEEE Design and Test of Computers,* Aug. 1990.

[34] Y. Yu and M. Tuttle, "Analyzing Cache Coherence with TLA+," *Proc. Tutorial and Workshop on Formal Specification and Verification Methods for Shared Memory Systems,* Oct. 2000.

Daniel J. Sorin received the MS degree in electrical and computer engineering from the University of Wisconsin-Madison in 1998 and the BSE degree in electrical and computer engineering from Duke University in 1996. He is currently a graduate student and research assistant in the Electrical Engineering Department at the University of Wisconsin-Madison. His research interests are in multiprocessor memory systems, with an emphasis on availability, verification, and analytical performance evaluation. He is a student member of the ACM, IEEE, and the IEEE Computer Society.

Manoj Plakal received the BTech degree in computer science and engineering from the Indian Institute of Technology, Kanpur, in 1996, and the MS degree in computer sciences from the University of Wisconsin-Madison in 1998. He is currently a PhD candidate in the Computer Sciences Department at the University of Wisconsin-Madison. His primary research interests lie in the interactions between compilers and modern computer architectures. He is currently working with Professor Charles Fischer on innovative uses of software assist threads for multithreaded processors. Through his work with Professors Condon, Hill, and Wood, he also retains an interest in the design and verification of cache coherence protocols and shared-memory multiprocessors. He is a student member of the IEEE and the IEEE Computer Society.

Anne E. Condon received the BSc degree from University College, Cork, Ireland, in 1982, and the PhD from the University of Washington in 1987. Her thesis, a study of game-theoretic complexity classes, won an ACM Distinguished Dissertation Award in 1988. Condon received a National Young Investigator Award in 1992. She was named Distinguished Alumna of University College Cork in 2001, for her contributions in the area of DNA computing. Her research contributions are in the areas of probabilistic and interactive complexity classes, design and analysis of algorithms for computationally intractable problems, and DNA computing. She is currently a professor in the Department of Computer Science at the University of British Columbia. She was a faculty member of the Computer Sciences Department at the University of Wisconsin from 1987 to 1999.

Mark D. Hill received the BSE in computer engineering from the University of Michigan in 1981, the MS degree in computer science from the University of California–Berkeley in 1983, and the PhD degree in computer science from the University of California–Berkeley in 1987. He is professor and Romnes Fellow in both the Computer Sciences Department and the Electrical and Computer Engineering Department at the University of Wisconsin-Madison. He currently codirects the Wisconsin Multifacet project with Prof. David Wood. He has made contributions to cache design, cache simulation, translation buffers, memory consistency models, parallel simulation, and parallel computer design. He won a US National Science Foundation Presidential Young Investigator award in 1989, was named an IEEE Fellow in 2000 for "contributions to cache memory design and analysis," and co-won the best paper award in VLDB 2001.

Milo M.K. Martin received the BA degree in computer science from Gustavus Adolphus College in 1996, the MS in computer science from the University of Wisconsin-Madison in 1998, and is currently a PhD candidate and member of the Wisconsin Multifacet Project in the Department of Computer Sciences at the University of Wisconsin-Madison. His research interests include memory system performance of commercial workloads, techniques to improve multiprocessor availability, and the use of dynamic feedback to build adaptive and robust systems. He is supported by an IBM Graduate Fellowship and is a student member of the ACM, the IEEE, and the IEEE Computer Society.

David A. Wood received the BS in 1981 and his PhD in 1990, both at the University of California, Berkeley. He is a professor and Romnes Fellow in both the Computer Sciences and Electrical and Computer Engineering Departments at the University of Wisconsin, Madison. He joined the faculty at the University of Wisconsin in 1990. He is a 1991 recipient of the US National Science Foundation's Presidential Young Investigator award, area editor (computer systems) of *ACM Transactions on Modeling and Computer Simulation*, and a member of the ACM, IEEE, and the IEEE Computer Society. He has published more than 50 papers, is an inventor on six US and international patents, and was awarded an H.I. Romnes Faculty Fellowship by the University of Wisconsin in 1999. He coleads the Wisconsin Multifacet project with Professor Mark Hill (http://www.cs.wisc.edu/multifacet) which is exploring techniques for exploiting prediction and speculation in multiprocessor memory systems.