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.

- *The authors 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.

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 MSI protocol that we will sp

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.

- *The authors 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.

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

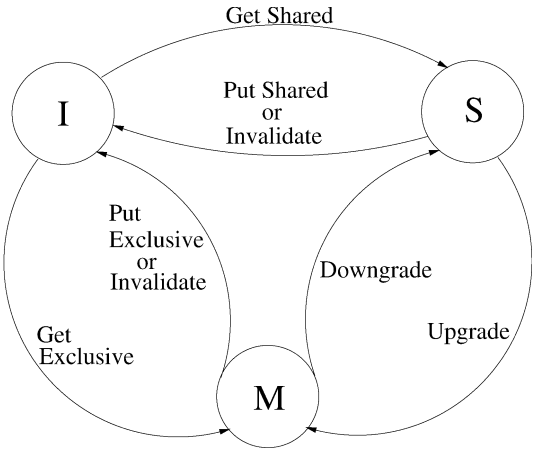

Fig. 1. High-level specification for cache controller.

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 shows a

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

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, specified using the same table-based

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

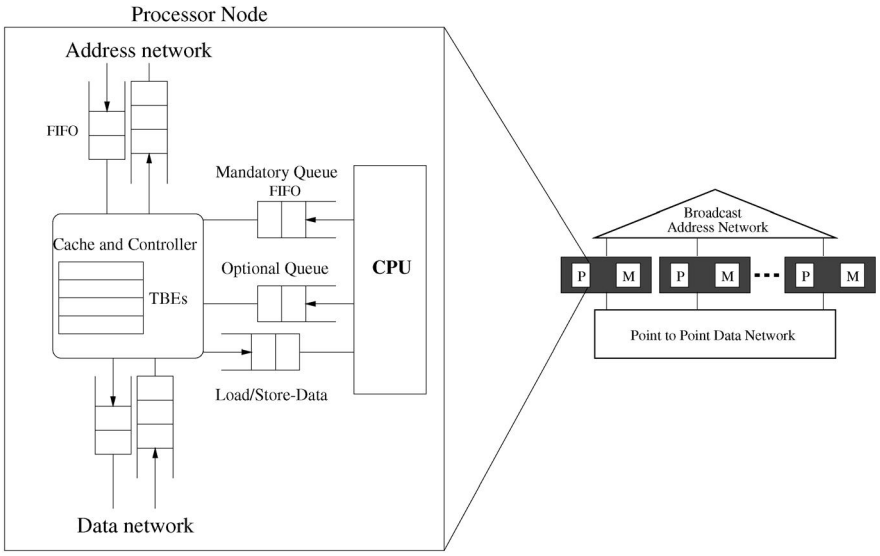

Fig. 2. Broadcast snooping system.

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

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.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.