Proving Sequential Consistency by Model Checking ⋆

Tim Braun[1], Anne Condon[3], Alan J. Hu[3], Kai S. Juse[1], Marius Laza[3], Michael Leslie[2], and Rita Sharma[3]

[1] Dept. of Computer Science, Technical University of Darmstadt
[2] Dept. of Electrical and Computer Engineering, Univ. of British Columbia
[3] Dept. of Computer Science, Univ. of British Columbia
(condon,ajh)@cs.ubc.ca, http://www.cs.ubc.ca/spider/ajh

Abstract. *Sequential consistency* is a multiprocessor memory model of both practical and theoretical importance. The general problem of deciding whether a finite-state protocol implements sequential consistency is undecidable. In this paper, however, we show that for the protocols that arise in practice, proving sequential consistency can be done automatically in theory and can be reduced to regular language inclusion via a small amount of manual effort. In particular, we introduce an approach to construct finite-state "observers" that guarantee that a protocol is sequentially consistent. We have developed possible observers for several cache coherence protocols and present our experimental model checking results on a substantial directory-based cache coherence protocol. From a theoretical perspective, our work characterizes a class of protocols, which we believe encompasses all real protocols, for which sequential consistency can be decided. From a practical perspective, we are presenting a methodology for designing memory protocols such that sequential consistency may be proven automatically via model checking.

1 Introduction

Model checking [7] has emerged as the dominant paradigm for formally verifying temporal properties of computer system designs. A key factor in favor of model checking is that it is fully automatic. Reducing a problem to model checking is therefore, at least in theory, a major step towards solving the problem.

One of the most successful application domains for model checking has been multiprocessor cache coherence protocols (e.g., [16, 9, 6, 8, 14, 23, 26, 4, 19, 13] are some early works). The application domain is commercially very important, since almost all high-end servers are now cache-coherent multiprocessors. Furthermore, the protocols are tricky, highly concurrent, and hence bug-prone. In addition, the protocols can be modeled in finite state, naturally supporting model checking.

Work on model checking cache coherence protocols has concentrated almost entirely on checking assorted correctness and consistency properties (e.g., shared copies of a memory block agree in value). A different verification task is to check whether the memory system implements the desired behavior with respect to the loads and stores of

⋆ This work was supported in part by research grants from the National Science and Engineering Research Council of Canada.

running programs. The specification of the desired behavior is called the *memory model* of the system. We would like to be able to verify that a protocol implements a specified memory model.

Sequential consistency is a multiprocessor memory model introduced by Lamport [15]. A memory system is sequentially consistent iff there always exists an interleaving of the program orders of all the processors such that each load returns the value of the most recent store to the same address. Sequential consistency is important both as a practical memory model that provides intuitive ease-of-programming while allowing efficient hardware optimizations (e.g. [11]) and also as an extensively studied memory model that can be used to understand other, more relaxed models (e.g. [1]).

Ideally, we would specify sequential consistency in CTL or LTL and then use model checking to determine whether or not a protocol implements sequential consistency. Unfortunately, the general problem of deciding sequential consistency of a finite-state protocol is undecidable [3]. Real protocols, however, might not be fully general, suggesting that the undecidability result may not be relevant in practice. If we can create a model-checkable specification that is sufficient to prove sequential consistency, and if such sufficient specifications exist for all real protocols that implement sequential consistency, then in practice, we can verify sequential consistency using model checking.

In this paper, we present a methodology for proving sequential consistency using model checking. The protocol being verified is augmented with additional (finite-state) bookkeeping information. We call the augmented protocol the *observer*. A finite-state checker examines runs of the observer and certifies that a run is indeed sequentially consistent. Model checking the entire finite-state system determines whether the checker will certify all possible runs, proving sequential consistency of the protocol.

In theory, sequential consistency can be checked automatically, though inefficiently, by enumerating all possible observers and testing for each if the checker is satisfied. In practice, we expect that possible observers would be manually created and then verified automatically.

For our methodology to be practical, it must satisfy three constraints: a suitable observer must exist for most or all real protocols, the process of creating the observer must not be too difficult, and the resulting observer-checker system must not be too much larger than the original protocol, in order to minimize state explosion. In the rest of this paper, we will describe a method for creating observers, along with the corresponding checker. We will argue that in principle, a finite amount of bookkeeping information should be sufficient for real protocols. Finally, we will present experimental results using our methodology to prove sequential consistency of a substantial directory-based cache coherence protocol.

1.1 Related Work

There has been considerable work over the years on verifying memory system protocols and memory models. For brevity, we mention here only closely related work, pertaining to finite-state verification of protocols with respect to sequential consistency.

Plakal et al. [18] introduce a verification approach based on logical clocks and apply it to a directory based protocol. Our approach is inspired by the logical clocks approach, but in contrast to logical clocks, which are unbounded, our approach reduces verification to a language inclusion problem between finite state automata.

Henzinger et al. [10] propose a very similar approach to ours, using a finite-state observer to reorder loads and stores to construct a witness of sequential consistency. Because of the finite-state limit on reordering, the method is too restrictive to handle most real protocols. One could view our approach as a generalization of theirs that handles many more protocols. We note that Henzinger et al. prove very strong results for protocols in their restrictive class, namely that it is sufficient to reduce verification of a protocol with arbitrarily large parameters (number of processors, number of blocks, number of values per block) to a fixed-parameter problem. In contrast, our method applies to verification of only fixed-parameter protocols.

Nalumasu et al. [17] propose the Test Model-Checking technique, in which a protocol is checked against various predefined finite-state automata that test certain memory model properties. These tests can be considered to be finite-state observers. By combining these tests, it is possible to verify memory models that are close to, but not identical to, sequential consistency. Determining exactly how these test combinations relate to sequential consistency and to the class of protocols we can handle is an open question.

At a recent, informal workshop, Qadeer proposed an approach for automatically verifying that a memory protocol implements a memory model [20]. The basic idea is to identify and formalize many assumptions that typically hold of real protocols and real memory models. In the presence of these assumptions, one can generate a finite-state witness automatically (and much more efficiently than our construction). Our method is currently more general than Qadeer's, but requires manual effort in practice. We believe that Qadeer's method can be extended to greater generality, but is likely to require human effort to match the generality of ours. Complementarily, we believe that the efficiency of our method can be improved by exploiting Qadeer's assumptions.

2 Theory

2.1 Basic Definitions

We start by formalizing our notion of cache protocols and sequential consistency. Intuitively, a protocol will be a finite-state machine parameterized by the number of processors, memory blocks, and possible values per memory block. Among the possible actions of the protocol will be load and store actions, which indicate the processor, the address (memory block number), and the value loaded or stored.

Definition 1. *A* **protocol** *is a tuple $(p, b, v, Q, q_0, \mathcal{A}, \delta)$. The constants p, b, and v specify the number of processors, memory blocks, and data values in the protocol. We assume there is a distinguished value $\perp$, which is the initial value of each block. The set of states is Q, of which q_0 is the initial state. The set $\mathcal{A}$ is the set of all actions, which includes actions of the form LD(P, B, V) and ST(P, B, V), where $1 \leq P \leq p$, $1 \leq B \leq b$, and $1 \leq V \leq v$. The transition relation is δ, with $\delta \subseteq Q \times \mathcal{A} \times Q$.*

For notational convenience, we use *'s to denote sets of LD and ST actions over all values of a parameter: e.g., ST(*,B, V) denotes the set $\{\mathrm{ST}(P, B, V) \mid 1 \leq P \leq p\}$.

Definition 2. *A* **protocol run** *is a sequence of actions $A_1, A_2, \ldots, A_k$ such that there exists a sequence of states $q_0, q_1, q_2, \ldots, q_k$ with $(q_{j-1}, A_j, q_j) \in \delta$ for all $1 \leq j \leq k$.*

Definition 3. *A* **protocol trace** *is the subsequence of a protocol run that includes exactly the ST and LD operations of the run.*

For a given protocol P, let $L(P)$ denote the set of all runs of P (the language of P). Let $T(P)$ denote the set of all traces of P.

Definition 4. *A trace $T = t_1, t_2, \ldots, t_k$ is a* **serial trace** *if for all blocks B and values V, for all $1 \leq j \leq k$:*

$$(t_j \in LD(*, B, V)) \Rightarrow \left(\begin{array}{c} (V = \perp) \wedge \forall i_{<j}[t_i \notin ST(*, B, *)] \\ \vee \\ \exists h_{<j}[t_h \in ST(*, B, V) \wedge \forall i_{h<i<j}(t_i \notin ST(*, B, *)] \end{array} \right).$$

Intuitively, a serial trace is one in which each load returns the value of the most recent (prior to the load) store to the same block. If there were no prior stores to that block, the load must return $\perp$.

Definition 5. *A* **reordering** *of a run or trace of length k is simply a permutation Π of the numbers from 1 to k.*

Definition 6. *Let $\Pi = \pi(1), \pi(2), \ldots \pi(k)$ be a reordering of a trace T. Let $T' = t_{\pi(1)}, t_{\pi(2)}, \ldots t_{\pi(k)}$. Π is called a* **serial reordering** *and T' is the corresponding serial trace if Π and T' have the following two properties. First, Π preserves the "per processor" order of T, i.e., for all processors P, if t_a and t_b are operations of processor P then $a < b$ if and only if $\pi^{-1}(a) < \pi^{-1}(b)$. Second, T' must be a serial trace.*

Definition 7. *A protocol is* **sequentially consistent** *if all of its traces have a serial reordering.*

2.2 Window Observers

Our methodology for constructing observers is based on a bookkeeping structure we call a **window**. To understand what a window is, we first note that there are two notions of time associated with a protocol: real time, and reordered (or logical) time, in which operations and actions of the protocol are serialized so that every LD gets the value of the most recent ST. Intuitively, a window summarizes the overall status of the memory system in reordered time. The window includes the active STs (i.e. those which may be read by future LDs of the protocol), their ordering in logical time, and where the most recent loads have occurred in logical time. A **window observer** annotates the original protocol run with windows. A finite-state checker (described in Section 2.3) can prove that a run is sequentially consistent by using the windows. Let us now consider these ideas in more detail.

Definition 8. *A* **window** *is a sequence of nodes. Nodes can be one of four different types: delete vectors (DV), logical pointers (LP), stores (ST), and last load indicators*

(LL). A DV is a vector of b bits, one per memory block. There are exactly p logical pointers, denoted $LP_1, \ldots, LP_p$, one for each processor. A store node contains a block number B and a value V, denoted $ST(B, V)$. There are b last load indicators, denoted $LL_1, \ldots, LL_b$.

Delete vectors summarize an unbounded sequence of no-longer-relevant stores into a bounded-size node. This capability is crucial for handling many real protocols. We will use DV_{false} to denote a delete vector with all entries set to false.

Definition 9. *A* **window observer** *for a protocol P is a protocol with actions which are either LD or ST operations, windows, or a special NULL action. If O is a window observer for protocol P, then the set of traces of O must equal the set of traces of P, i.e. $T(O) = T(P)$.*

Intuitively, for real-world protocols, a window observer may be obtained for a protocol by augmenting the protocol to output a window after each protocol action, thereby annotating the protocol runs with windows, and simplifying the run alphabet of the observer so that actions other than windows, LD and ST operations are replaced by the NULL action. The NULL action abstracts away the detailed behavior of the protocol, allowing the use of a universal checker for all observers. We present an example window observer in Section 3.

2.3 Checkers

The checker is a finite-state machine parameterized by p, b, and v, just as protocols are. The same (family of) checker is used for all protocols. We will argue in Section 2.4 why the maximum window size will be finite for real protocols, leading to a finite-state checker. In practice, the appropriate size of the checker's state space is determined by how many ST operations are active, i.e., could have their value read by a future LD.

The checker examines the run (annotated protocol run) generated by the window observer. It always saves a copy of the most recently seen window, and it checks each subsequent action/window against the most recently seen window:

Checker Rules

1. **Windows must be properly structured.** In particular, DV nodes occur only immediately preceding each LP node and each ST node. This implies that there is exactly one DV node between adjacent LP or ST nodes.
2. **Each LD must get its value from the most recent ST.** If $LD(P, B, V)$ (processor P loads value V from block B) is the protocol action, the checker looks in the most recent window for the closest ST node to block B preceding logical pointer LP_P. This ST node must have stored value V, and there must be no DV vector indicating deleted ST nodes to block B between the ST node and LP_P. If there is no ST node to block B prior to LP_P, then the LD must have returned value $\perp$.
3. **A ST cannot be retroactive.** Intuitively, we prohibit a ST operation from occuring at a point in logical time if a LD operation to the same block has already occurred later in logical time. Formally, if $ST(P, B, V)$ (processor P stores value V to block B) is the protocol action, the logical pointer LP_P must be later in the window than the last load marker LL_B.

4. **Consecutive windows are consistent.** The checker compares the new window against the most recent window. (If the new window is the first window the checker sees, then consistency is checked against a default initial window that consists of the LP nodes and nothing else.) First, the checker makes sure that it sees at most one memory operation (LD or ST) between the most recent window and the new window. Depending on the intervening memory operation (if any), the following are possible:
   (a) The intervening memory operation was $ST(P, B, V)$, and the only difference between the windows is that a new ST node $ST(B, V)$ and a new DV node DV_{false} are inserted immediately before logical pointer LP_P. (The DV node formerly preceding LP_P now precedes the new ST node.)
   (b) The intervening memory operation was $LD(P, B, V)$, and if LL_B (the last load to block B) was before LP_P in the old window, then LL_B is moved so that it immediately precedes the DV preceding LP_P in the new window. Otherwise, the window is unchanged.
   (c) There were no intervening memory operations, and one logical pointer has moved forward. Intuitively, a processor is updating its state to a newer one. The details of this change are tedious, but basically, the DV preceding the LP that is moving is bitwise ORed into the closest subsequent DV, the LP is free to move to any subsequent point immediately following a DV, and a new DV_{false} node is added immediately after the LP's new location.
   (d) There were no intervening memory operations, and some ST nodes have been deleted. Again, the details of this change are tedious. Basically, a sequence of ST nodes without any LP nodes separating them can be deleted. Their corresponding DV nodes are bitwise ORed, and any deleted ST nodes are also marked on the remaining DV node.

If every action and annotation the checker sees is legal, the checker accepts the run.

If any observer (whether manually or automatically generated) passes the checker, the protocol is sequentially consistent, as summarized in the following theorem.

Theorem 1. *Let P be a protocol, and let O be a window observer. Let C be a checker as described above. If $T(P) = T(O)$ and $L(O) \subseteq L(C)$, then P is sequentially consistent.*

Proof Sketch: (Proof is in Appendix A.)
Given an observer that satisfies the checker, the heart of the proof shows how to reorder each trace of the observer so as to obtain a serial trace. The construction of the reordered trace is done inductively from the observer's run (which includes both trace operations and windows). Roughly, for each trace of the observer, the windows can be pieced together in a consistent manner to provide a reordering of the ST operations in that trace. LDs can be inserted in this total order using logical pointer information to yield the reordered trace. Checks 2, 3, 4(b) and 4(d) of the checker then ensure that LDs get the value of the most recent ST in this reordered trace. Checks 4(a) and 4(c) ensure that, in the reordered trace, STs still respect program order. □

In practice, the trace equivalence $T(P) = T(O)$ is established by construction, by simply adding the observer state and actions in a way that doesn't interfere with the protocol. Thus, the problem reduces to regular language containment, which can be easily verified by model checking.

2.4 Bounding Observer Size

If we bound the maximum allowable size of the window observer, we characterize a class of protocols for which sequential consistency is decidable.

Definition 10. *A protocol P with n states belongs to the* **Window Observer Class** *with parameter $f(n)$ if there exists a window observer O with at most $f(n)$ states such that $T(P) = T(O)$ and $L(O) \subseteq L(C)$, where C is a checker.*

The parameter $f(n)$ in our definition of the Window Observer Class is needed to ensure that for a given protocol P, all possible window observers O can be enumerated to give the decidability result. We now argue that, for the Window Observer Class to encompass all real-world protocols, the bound $f(n)$ need be at most exponential in n. This exponential bound is based on a crude analysis to cover the worst case; in practice we do not expect the size of the observer to be exponential in n. The argument is based on the premise that the status of the memory system of a real-world protocol at each state can be captured by a window as in Definition 8.

First, consider how the size of such a window can be bounded by the parameters of the protocol. We assume that for a ST to be active (readable by future LDs), the protocol state must have some record of it. In other words, LD instructions must return values from ST instructions that the protocol knows has been executed. In that case, n is an obvious bound on the maximum number of active store instructions, since there must be an initial state with no active stores, and for any state, its successors can have at most one additional active store. Hence, a window contains at most n ST nodes, each of which need be only $\lg b + \lg v$ bits in size (where $\lg$ denotes $\log_2$) to hold the block number and value. There are p LP nodes, each using $\lg p$ bits, and b LL nodes, each using $\lg b$ bits. DV nodes are b bits long. The checker enforces that there is exactly one DV node for each LP or ST node, for a total of $p + n$ DV nodes. Combining all the nodes gives an impractical, but finite, size upper bound for a window of $n(\lg b + \lg v) + p \lg p + b \lg b + (p+n)b$ bits. Assuming that $b, \lg v$, and p are bounded by n, the number of bits in a window is at most polynomial in n.

Therefore, the number of different windows that could be associated with each state of P is bounded by an exponential function of n. Thus, if P were augmented so that states explicitly contain their annotation, the number of states in the annotated protocol would be at most exponential in n. Now, two adjacent windows in this annotated protocol may not be consistent as required by Rule 4 of the checker, because more than one logical pointer may move, for example. By adding further states to the augmented protocol that enable intermediate windows to be inserted between two such windows, an observer that meets the criteria of the checker can be obtained. This is because, by Rules 4c and 4d, LP nodes can only move forward in the finite window, and there are only a finite number of ST nodes that might be deleted. Thus, the number of further states added between two states of the augmented protocol is bounded by a polynomial in n, and so the total number of states of the resulting observer O is exponential in n.

In theory, therefore, the number of window observers that must be checked for any finite-state protocol is finite, so we can test membership in the Window Observer Class by generating and testing all possible window observers. In practice, a good observer candidate would be generated manually based on an understanding of the protocol. An example of such an observer follows.

3 Example

We have constructed paper-and-pencil window observers for three substantial memory protocols: a non-trivial snoopy cache coherence protocol simplified from a real protocol; Afek et al.'s Lazy Caching protocol [2], which has much more relaxed ordering requirements than most sequentially consistent protocols; and a directory-based cache coherence protocol that is sequentially consistent, but not coherent. Our previous experience with cache protocols suggests that, in the absence of automated verification, we should assume our paper-and-pencil designs to be buggy. Nevertheless, we were able to fit three very different protocols into our framework, showing the broad applicability of our approach. Here, we focus on only the directory-based protocol, for which we have created machine-readable models and completed the model checking.

The directory protocol involves several interacting entities: the processors, a directory, and a network interface. Messages, buffered in queues, are used to communicate between these entities. The system is depicted in Figure 1, with a detailed description in Appendix B. Our description is a variant of one provided by the Multifacet

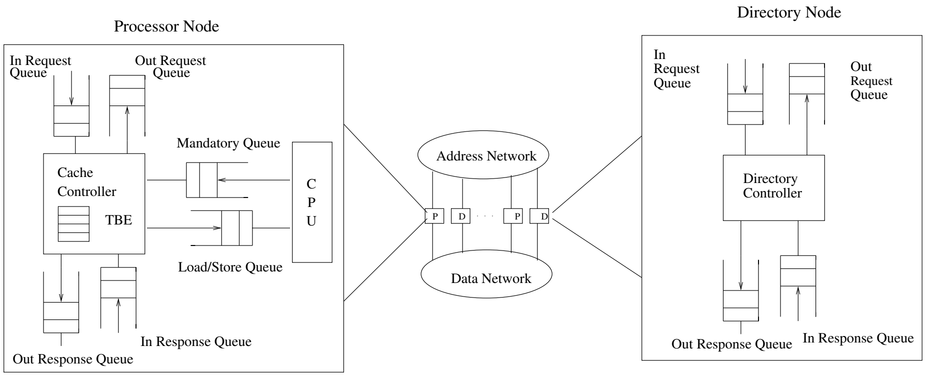

Fig. 1. Directory-Based Caching System

group from the University of Wisconsin, to which we have added an optimization due to Scheurich [21], that allows a processor to continue to read a cache block after acknowledging an invalidation of that block.

Roughly, the protocol can be understood as follows. Processors may have three types of access to a block, with three corresponding "stable" processor states per block: M(modify), S(shared), or I(invalid). A processor may do a ST only when in the M state and may do a LD only in the S or M states. For each block, at most one processor is in the M state at any given time. The directory coordinates access to blocks of memory, and is the default owner of a block when no processor has Modify access to that block.

When a processor needs to upgrade from one stable state to another in order to do a LD or ST operation, then the processor initiates a transaction and enters a transient state. How transactions are handled can be understood by the following situations. Several race conditions may also arise, which we omit here.

1. If several processors share the block, and processor P wants Modify access, then P sends a message to the directory (issue GETX); the directory returns the value of block along with the number of current sharers. The directory also sends a message to each sharer asking them to invalidate their copy of the block and to send an ACK to P once they have done so. P waits in the transient state IM (Invalid to Modify) until it gets both the data and all the ACKs before doing a LD or ST to the block.
2. If one processor Q is owner of the block, and processor P wants Modify access, then P sends a message to the directory (issue GETX); the directory forwards this request to Q and sets P as the new owner of the block. Q (which is in state M) receives a "forwarded GETX" message from the directory. Q sends the data to P and goes to I state. P waits in IM state until it gets the data from Q.
3. If several processors share the block, and processor P wants shared access, then P sends a message to the directory (issue GETS); the directory adds P to list of sharers and returns the value of block. P waits in the transient IS state until it gets the data from the directory.
4. If processor Q is owner of the block and processor P wants shared access then P sends a message to the directory (GETS); the directory adds both P and Q to the list of sharers, forwards the request to Q and clears the owner tag. Q receives the forwarded GETS message from the directory, sends the data to P and to the directory.
5. Scheurich's optimization allows a block to continue to be read after ownership has been released. We add a new cache block state I*, which indicates that the block has been invalidated, but we are in optimization mode. The I* state is entered when a processor receives an invalidate for a block that was in the shared state S, or a forwarded GETX for a block that was in the exclusive state M. While in optimization mode, the processor can continue to read the block, even though the invalidation has been acknowledged. As soon as the cache receives a request from any other entity, however, optimization mode ends, and the cache block state changes from I* to I.

The **window observer** for the directory protocol behaves just like the protocol itself, except for two differences. First, when the protocol takes actions other than LD and ST, the observer takes (i.e. outputs) an action called NULL, but moves to the same state as would the protocol. Second, the observer updates and outputs a window, while executing the protocol.

A detailed description of how windows are updated is given in Appendix B. Briefly, a window can be changed in three ways: addition of a ST node, moving a logical pointer node, or deletion of a ST node:

- Initially, the observer outputs a window containing just the p LP nodes, in any order.
- Each time a processor or the directory sends data to another processor, if the sender's LP is later than the recipient's LP, then the recipient's LP gets moved immediately after the sender's LP. Intuitively, when the recipient receives the data, it must have moved forward in time at least to pass the sender. We found it convenient to introduce a LP node for the directory. This is purely an implementation detail that makes it easy to determine where to advance the processors' LP nodes in certain cases.
- Upon a ST operation, a new ST node is created in the window and is placed just before LP_P. Upon a LD operation, the observer makes no changes to the window.
- To keep the window size finite, the observer deletes those ST nodes which will never be read in the future: for each pair of successive LP nodes, for each block B, the observer deletes all but the latest ST B node between the two LP nodes. Also, for each block B, all but the last ST B node to the earliest LP node is deleted.

As we can see, the construction of an observer is reasonably intuitive.

4 Experimental Results

The true test of our methodology requires experimentation. The directory-based protocol was the most challenging of the three on which we had worked, so we chose that for the full model-checking experiment.

We chose the Murphi verification system [9] for our experiments, mainly for ease-of-use and out of familiarity, and also because Murphi has proven successful for many cache protocol verification efforts. Modeling cache protocols in Murphi is routine [13], and many examples are available as part of the standard Murphi distribution. The main downside is that Murphi does not use symbolic model checking [5], precluding one of the most powerful techniques for combatting state explosion.

We started with verifying basic correctness properties of the protocol itself. Proving sequential consistency should wait until after the protocol is debugged. In order to reduce state explosion, we made several modeling decision and simplifications:

1. Trade speed for size. We chose not to keep track of information if we could compute it, even if the cost of the computation was high. The goal was to reduce the state space, although the use of hash compaction [25, 24] reduces the importance of this optimization.
2. Remove the network. We completely removed the communication network between processor nodes and the directory node. Instead, messages are automatically inserted into the appropriate recipient's incoming message queue. This reduces the state space, but does not change the protocol behavior because the incoming message queue can still model arbitrary delay.
3. Remove the load/store queue. The load/store queue returns load results and store completion signals to the CPU. Since we are verifying the caching protocol, we can ignore this part of processor/cache communication.
4. Reduce the mandatory queue. The mandatory queue carries load/store requests from the CPU to the cache. We reduced this queue to a single-entry buffer. This

simplification does not change the behavior of the protocol because the CPU can generate requests non-deterministically, simulating the rest of the queue.

5. Throttle the CPUs. To avoid overflowing the message queues, we allowed the cache to process CPU requests only when the number of incoming response messages was below a set threshold — 2 messages in our experiments. This was the only change that reduced the set of possible behaviors of the protocol. Such changes are often needed in practice to use model checking on real designs, although they create the possibility of missed bugs.

Not surprisingly, we discovered several minor bugs and one subtle bug (with an error trace requiring 10 network messages) in the initial protocol. This first phase of the project corresponds to a typical cache protocol formal verification effort.

After fixing these bugs, we proceeded to add the observer and checker to the model. Adding the observer/checker consisted of adding a variable to store the most recently seen window, and then weaving additional actions to manipulate this variable into the rules that implement the protocol. Whenever the window is updated or a load or store is performed, the checker is invoked to make sure the action was legal. The window data structure needed to hold at most $(p+1)b + b + (p+1)$ nodes, so we implemented it as an array. (The $(p+1)$ results from the extra LP node for the directory.) No DV nodes were needed for this protocol, so we omitted them. (DV nodes track deleted ST nodes to prevent an LP node from jumping between a ST and the subsequent LP, where a LD may have executed. In this protocol, LP nodes always jump to a position immediately following another LP node, so the problem does not arise.) Model checking uncovered several bugs in the combined protocol/observer/checker, including one serious protocol bug, involving staying in optimization mode in a situation when it should have been canceled. This bug had eluded our earlier model-checking without the observer/checker. The most difficult problems to debug involved counterexample traces which failed the checker but were nevertheless serial traces, implying that the observer was not providing adequate information to the checker. Eventually, we were able to debug the observer/checker as well, proving the protocol sequentially consistent. The final protocol and observer are shown in Appendix B. The Murphi model is available for download from `http://www.cs.ubc.ca/spider/ajh/cav-review-model`.

The total effort for the model checking, including the protocol itself as well as the observer/checker, was three students, as a class project, working part-time, for approximately two months. In other words, the total effort was comparable to that required to model check only simple correctness properties, but the result is much stronger. Adding the observer and checker was neither easy, nor extremely difficult. The complexity was much like handling a somewhat more sophisticated protocol.

The other practical concern is state explosion. Table 1 shows run times and reachable state counts for the protocol with and without the observer/checker. As can be seen, the observer/checker adds a substantial amount of state, but the blow-up isn't outrageous. Again, the results with observer/checker are comparable to what one would expect if verifying a somewhat more complex protocol without observer/checker.

Obviously, additional work is needed on reducing state explosion, but the results show that our method is clearly on the edge of feasibility for realistic protocols.

Model Size			w/o Observer/Checker		w/ Observer/Checker	
p	b	v	Reached States	Run Time	Reached States	Run Time
1	1	1	41	19.5s	82	19.5s
2	1	1	2,272	20.5s	8,738	22.8s
2	1	2	7,628	22s	37,317	34.1s
3	1	1	98,083	93.2s	742,984	568s
2	2	1	641,157	417s	71,242,781	47711.5s
3	1	2	754,577	636.2s	7,287,108	5741s
2	2	2	9,413,564	7091.6s	space out	
2	3	2	space out			

Table 1. Summary of protocol runs with and without the window/checker. p is the number of processors, b is the number of memory blocks (addresses), and v is the number of values. Experiments were run on a variety of machines (Sun Ultra-60, up to 2GB main memory, 300Mhz or 360Mhz processors), so run times are only to give a rough picture.

5 Conclusion and Future Work

We have presented a methodology for proving sequential consistency of memory protocols by using model checking. From a theoretical perspective, our work characterizes a class of protocols, which we argue includes all real protocols, for which sequential consistency is decidable. From a practical perspective, we provide a concrete way to use a bit of human insight to reduce the problem of proving sequential consistency of a memory system protocol to automatic model checking. Our experiments indicate that the method is indeed feasible in practice, although additional research to reduce state explosion is needed.

The main directions to reduce state explosion are to try symbolic model checking and related techniques, and to search for domain-specific reductions. For example, the state of the window is likely to be highly determined by the state of the protocol, suggesting that techniques like functionally dependent variables [12] may be very helpful. Another possibility is to partition the checker into several smaller sub-checkers, each of which using only part of the window, that can be model-checked separately, thereby substantially reducing the state space.

On the theoretical side, we have not computed tight size bounds for the observer and checker, nor have we analyzed the complexity of the decision procedure. A hardness proof for deciding membership in the Window Observer Class, coupled with tighter upper bounds on observer and checker size would clarify the worst-case behavior of our approach. We also need to clarify the relationship between our Window Observer Class and other classes, such as can be verified using the Test Model Checking approach [17].

Sequential consistency is only one of many important memory models. We believe our approach can be generalized to handle other, more relaxed memory models.

Finally, we are currently developing a cleaner theoretical framework based on partial orders instead of our ad hoc window structure. We believe the new framework will be easier to adapt to different protocols as well as to different memory models.

Acknowledgment

We would like to thank the Multifacet Group at the University of Wisconsin for providing us with preliminary versions of several memory system protocols as well as answering our questions about them.

References

1. S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, pages 66–76, December 1996.
2. Y. Afek, G. Brown, and M. Merritt. Lazy caching. *ACM Transactions on Programming Languages and Systems*, 15(1), January 1993.
3. R. Alur, K. McMillan, and D. Peled. Model-checking of correctness conditions for concurrent objects. In *Eleventh Symposium on Logic in Computer Science*, pages 219–228. IEEE, 1996.
4. Ásgeir Th. Eiríksson and K. L. McMillan. Using formal verification/analysis methods on the critical path in system design: A case study. In P. Wolper, editor, *Computer-Aided Verification: Seventh International Conference*, pages 367–380. Springer-Verlag, July 1995. Lecture Notes in Computer Science Number 939.
5. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Conference on Logic in Computer Science*, pages 428–439, 1990. An extended version of this paper appeared in *Information and Computation*, Vol. 98, No. 2, June 1992.
6. E. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. Long, K. McMillan, and L. Ness. Verification of the Futurebus+ cache coherence protocol. Technical Report CMU-CS-92-206, Carnegie Mellon University, October 1992.
7. E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In D. Kozen, editor, *Workshop on Logics of Programs*, pages 52–71, May 1981. Published 1982 as Lecture Notes in Computer Science Number 131.
8. E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the Futurebus+ cache coherence protocol. In L. Claesen, editor, *11th International Symposium on Computer Hardware Description Languages and their Applications*. North-Holland, 1993.
9. D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *International Conference on Computer Design*. IEEE, October 1992.
10. T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Verifying sequential consistency on shared-memory multiprocessor systems. In *Computer-Aided Verification: 11th International Conference*, pages 301–315. Springer, 1999. Lecture Notes in Computer Science Vol. 1633.
11. M. D. Hill. Multiprocessors should support simple memory-consistency models. *IEEE Computer*, pages 28–34, August 1998.
12. A. J. Hu and D. L. Dill. Reducing BDD size by exploiting functional dependencies. In *30th Design Automation Conference*, pages 266–271. ACM/IEEE, 1993.
13. A. J. Hu, M. Fujita, and C. Wilson. Formal verification of the HAL S1 system cache coherence protocol. In *International Conference on Computer Design*, pages 438–444. IEEE, 1997.
14. C. N. Ip and D. L. Dill. Efficient verification of symmetric concurrent systems. In *International Conference on Computer Design*, pages 230–234. IEEE, October 1993.
15. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *ACM Transactions on Computer*, 28(9):690–691, September 1979.

16. K. L. McMillan and J. Schwalbe. Formal verification of the Gigamax cache-consistency protocol. In *International Symposium on Shared Memory Multiprocessing*, pages 242–251. Information Processing Society of Japan, 1991.
17. R. Nalumasu, R. Ghughal, A. Mokkedem, and G. Gopalakrishnan. The 'test model-checking' approach to the verification of formal memory models of multiprocessors. In *Computer-Aided Verification: 10th International Conference*, pages 464–476. Springer, 1998. Lecture Notes in Computer Science Vol. 1427.
18. M. Plakal, D. Sorin, A. Condon, and M. Hill. Lamport Clocks: Verifying a directory cache coherence protocol. In *Symposium on Parallel Algorithms and Architectures*, pages 67–76, 1998.
19. F. Pong, A. Nowatzyk, G. Aybay, and M. Dubois. Verifying distributed directory-based cache coherence protocols: S3.mp, a case study. In *International Conference on Parallel Processing, EuroPar '95*, August 1995.
20. S. Qadeer. On the verification of memory models of shared-memory multiprocessors. In *Workshop on Formal Specification and Verification Methods for Shared Memory Systems*. Unpublished Proceedings, October 31, 2000. Workshop affiliated with FMCAD 2000, Austin, TX.
21. C. Scheurich. *Access Ordering and Coherence in Shared Memory Multiprocessors*. PhD thesis, University of Southern California, May 1989. Published as USC Tech Report CENG 89-19.
22. D. Sorin, M. Plakal, A. E. Condon, M. D. Hill, M. M. Martin, and D. A. Wood. Specifying and verifying a broadcast and a multicast snooping cache coherence protocol. Technical Report CS-TR-2000-1412, University of Wisconsin Department of Computer Science, March 2000.
23. U. Stern and D. L. Dill. Automatic verification of the SCI cache coherence protocol. In *Correct Hardware Design and Verification Methods, CHARME '95*, pages 21–34. IFIP WG 10.5 Advanced Research Working Conference, 1995.
24. U. Stern and D. L. Dill. Improved probabilistic verification by hash compaction. In P. Camurati and H. Eveking, editors, *Correct Hardware Design and Verification Methods, IFIP WG 10.5 Advanced Research Working Conference, CHARME '95*, pages 206–224, 1995. Frankfurt/Main, Germany, October 2–4.
25. P. Wolper and D. Leroy. Reliable hashing without collision detection. In *Computer Aided Verification, Proc. 5th Int'l Workshop*, volume 697 of *Lecture Notes in Computer Science*, pages 59–70. Springer-Verlag, June 1993.
26. L. Yang, D. Gao, J. Mostoufi, R. Joshi, and P. Loewenstein. System design methodology of UltraSPARC-I. In *32nd Design Automation Conference*, pages 7–12. ACM/IEEE, 1995.

A Proof of Theorem 1

Let P be a protocol and let O be a window observer for which $T(P) = T(O)$ and $L(O) \subseteq L(C)$. Since the set of traces of P and O are identical, P must be sequentially consistent if O is. We now show that the condition $L(O) \subseteq L(C)$ implies that every trace of O has a serial reordering.

Let T be a trace of O. Note that T is a subsequence of some run R of O, where the run R includes windows (and NULL symbols) in addition to the operations of T.

From R we construct a trace T' that corresponds to a serial reordering of T. To do this, we first inductively construct *augmented traces* $\Gamma_j, j = 0, \ldots, n$, where n is the length of R. An augmented trace contains not only LD and ST operations, but also *flagged* ST operations and logical pointers. Then, T' is obtained from Γ_n by changing each flagged ST operation to a (standard) ST operation and removing the logical pointers.

We define $R[0]$ to be the window consisting of just one logical pointer per processor, namely $R[0] = LP_1, LP_2, \ldots, LP_p$ (where LP_i denotes the logical pointer for processor i). We set Γ_0 equal to $R[0]$.

For $j \in \{1, 2, \ldots, n\}$, Γ_j is obtained from Γ_{j-1} and $R[j]$, the j^{th} element of R, as follows.

1. If $R[j] = \text{LD}(P_i, B, V)$, then obtain Γ_j by inserting $\text{LD}(P_i, B, V)$ just before the logical pointer of processor i in Γ_{j-1}. That is, if $\Gamma_{j-1} = \Gamma, LP_1, \Gamma'$ then $\Gamma_j = \Gamma, \text{LD}(P_i, B, V), LP_1, \Gamma'$.
2. If $R[j] = \text{ST}(P_i, B, V)$, then obtain Γ_j by inserting $\text{ST}(P_i, B, V)$ just before the logical pointer of processor i in Γ_{j-1}.
3. If $R[j] = \text{NULL}$, then $\Gamma_j = \Gamma_{j-1}$.
4. Suppose that $R[j]$ is a window. Let j' be the last position in R with $0 \leq j' < j$ for which $R[j']$ is a window.
   (a) If $R[j]$ is obtained from $R[j']$ by the addition of node $\text{ST}(P_i, B, V)$, then $\Gamma_j = \Gamma_{j-1}$.
   (b) If $R[j]$ is obtained from $R[j']$ by deleting ST nodes, then flag as deleted the corresponding ST operations in Γ_{j-1}, to obtain Γ_j.
   (c) If $R[j]$ is obtained from $R[j']$ by moving logical pointers, then move the logical pointers in Γ_{j-1} accordingly to obtain Γ_j. (We note that construction of Γ_j maintains the invariant that the sequence of unflagged ST nodes and LP's of $\Gamma_{j'}$ equals the sequence of ST nodes and LP's of $R[j']$. Thus, the new positions of the logical pointers in Γ_j is specified unambiguously as the rightmost positions that maintain the invariant for Γ_j and $R[j]$.

We now summarize why T' corresponds to a serial reordering of T.

– T' **is a permutation of** T**.** This follows directly from Invariant 1 below, and the fact that the LD and ST operations of T' are exactly those in Γ_n, and thus those in R. The invariant is a straightforward consequence of the construction of Γ_j.

  Invariant 1: For all j, the sequence of LDs and STs (both flagged and unflagged) in Γ_j is a permutation of those in $R[1, \ldots, j]$.

– T' **respects program orders of** T**.** To show this, we use the following invariants:

  Invariant 2: For all j, the logical pointer of processor i is after all operations of processor i in Γ_j.
  Invariant 3: Furthermore, the relative order of operations in Γ_j is preserved in Γ_{j+1}, for all $j, 1 \leq j \leq n - 1$.

Invariant 2 follows from the facts that (i) if an operation of processor i is added to Γ_j, then it is placed directly before processor i's logical pointer, and (ii) logical pointers always move forward in windows (since the windows satisfy the checker), and thus always move forward when rearranged to obtain Γ_j from Γ_{j-1} using part 4 (c) of the inductive definition of Γ_j.
Invariant 3 is straightforward.
Fix a processor i. Suppose that the lth trace operation of processor i is inserted to Γ_j by steps 1 or 2 of the definition of Γ_j. Then this trace operation is inserted directly before the logical pointer of processor i in Γ_j, and thus, by Invariant 2, after the first $l-1$ trace operations of processor i. Furthermore, by Invariant 3, the relative order of operations in Γ_j are preserved in Γ_n and thus in T'. Hence, T' respects the program order of processor i.

– **Every LD gets the value of the most recent ST to the same block.** The proof of this property uses a final invariant:

  Invariant 4: Suppose contraction(Γ_j) is obtained from Γ_j as follows:
  1. For each subsequence Δ, N of maximum length in Γ_i, where N is a logical pointer or unflagged ST operation and no symbol of Δ is a logical pointer or unflagged ST operation, do the following. First, remove from Γ_i all flagged ST operations of Δ. Then, just before N, add a delete vector in which the Bth entry is true if and only if there is at least one flagged ST operation to B in Δ.
  2. For each block B, replace the last LD in Γ_i which is from the set LD($*, B, *$) by LL_B and remove all other LD operations.

  Then for all j for which $R[j]$ is a window, contraction(Γ_j) $= R[j]$.

Now, consider a LD(P_i, B, V) operation $R[j]$. Let $R[j']$ be the last window prior to $R[j]$ in R. Since O satisfies the checker, it must be that in $R[j']$, the last ST operation to B before LP_i has value V (by test 4 (b) of the checker) and that the Bth entry of all delete vectors separating this ST from LP_i is false. (We omit the case where V is undefined here.) By Invariant 4, it must be that in $\Gamma_{j'}$, there are no flagged ST nodes separating LP_i from the last (unflagged) ST node to block B with value V, say $R[m]$, prior to LP_i in $\Gamma_{j'}$. Hence, the same is true of Γ_j, and so in Γ_j, no ST to B separates $R[m]$ and $R[j]$.
A further application of Invariant 4, together with Invariant 3, shows that no further ST node to block B, say $R[l]$ ($l > j$), which is added to Γ_l, can separate $R[m]$ from $R[j]$ in Γ_l. Thus, in Γ_n and T', it is still the case that $R[m]$ is the last ST node to block B prior to $R[j]$, as required.

B Directory Protocol and Observer Specification

Our directory-based protocol involves processor nodes with caches, a (possibly distributed) directory, and an interconnect network, as depicted in Figure 1.

Processor nodes contain a CPU, a cache, and a cache controller with the logic for the coherence protocol. The Mandatory queue contains LD and ST operations that are generated by the CPU, in program order.

The processors and directory are connected by a point to point address network and data network, both of which preserve order of messages between any two points. The Mandatory, and In Request, and Out Request queues are always handled in FIFO order, but this constraint can be relaxed for In and Out Response queues.

To describe the possible protocol actions, we use the table-based method of Sorin et al. [22]. The protocol is specified using tables for the cache controller, directory, and network interface. Each table entry contains a sequence of actions that are executed in sequence. The protocol proceeds one entry at a time, where any entry that is a valid transition may be chosen as the next "step" of the protocol.

We describe how the tables should be interpreted for the cache controller; the directory and network interface tables can similarly be interpreted. In the cache controller table, Table 2, the rows correspond to the internal stable and transient states of the processor. The columns correspond to the various types of events that can trigger a transition of the cache controller. A description of each type of event is given in Table 3. Each table entry lists zero or more actions that the controller can take as part of a single transition. The actions are represented as letters, and their meaning is explained in Table 4. Actions are done in the order that they are listed. An action is only done if the resources needed (such as space in an outgoing queue) is available. Table entries that are empty are impossible.

For each outstanding GETX transaction, a Transaction Buffer Entry (TBE) is allocated. The TBE contains two entries: a forward ID which is used to record the ID of a processor which may need to receive data in the future, and a count of the number of pending acks expected from other sharers (in the case that a processor is waiting to have Modify access).

The Window Observer for the directory protocol is also specified using the table method in Tables 8 and 9. The language of the observer is the set of all possible output sequences that it can produce. In this case, the only non-NULL outputs of the observer arise for table entries of the cache controller. Empty entries in this table represent transitions in which the observer outputs NULL. Also, the observer outputs NULL for all transitions corresponding to directory or network interface transitions. For easier reading, we have split the observer specification into two tables: one for when a message is received by a cache controller, the other for when a message is received at the directory.

	Load	Store	Evic-tion	For-warded GETS	For-warded GETX	INV	Proc ack	Proc last ack	Proc data	Data ack 0	Data ack not 0	Data ack not 0 last	Ack 0	Nack	Busy ack	
I	g a c /IS	g i b c /IM				t 1										I
I^*	vk	i b c /IM	r /I													I^*
S	v k	i b c /IM	r /I			t 1 /I^*										S
M	v k	y k	d /MI	e w 1 /S	e f 1 /I^*											M
IS	z	z	z			t 1 /ISI			u v k o c/S	u v k o c/S				o r /I		IS
ISI	z	z	z			t 1 /ISI			u v k o c r /I	u v k o c r /I				o r /I		ISI
IM	z	z	z	δ 1 /IMS	δ 1 /IMI	t 1 /IM	q o	y k s o c /M	u y k s o c /M	u y k s o c/M	u p o	u y k s o c/M		s o r /I		IM
IMI	z	z	z				q o	y ε k f s o c r/I	u y ε k f s o c r/I	u y ε k f s o c r/I	u p o	u y ε k f s o c r/I				IMI
IMS	z	z	z				q o	y ε k w s o c/S	u y ε k w s o c/S	u y ε k w s o c/S	u p o	u y ε k w s o c/S				IMS
MI	z	z	z	1 /MI^{B}	1 /MI^{B}								o r /I		o c/MI^{F}	MI
MI^{B}	z	z	z												o /I	MI^{B}
MI^{F}	z	z	z	1 r /I	1 r /I											MI^{F}

Table 2. Cache controller specification

Load	Load request in Mandatory queue for which block is already in cache or GETS resources (such as outgoing queue space or victim cache block) to request shared access are all available
Store	Store request in Mandatory queue for which block is already in cache or GETX resources to request shared access are all available
Replacement	Victim block for a Load or Store request for which no cache block is available
Forwarded GETS	A forwarded GETS from the directory is at the head of the In Request queue
Forwarded GETX	A forwarded GETX from the directory is at the head of the In Request queue
INV	Invalidation request (including ID of requester) is at head of In Request queue
Proc ack	Ack from processor is in In Response queue and number of pending acks (in TBE) is greater than 1
Proc last ack	Ack from processor is in In Response queue and number of pending acks is 1
Proc data	Data from processor is in In Response queue
Data ack 0	Data from directory, along with an ack count of 0, is in In Response queue and number of pending acks = 0
Data ack not 0	Data from directory, along with an ack count, is in In Response queue, and this ack count is not equal to the negation of the number of pending acks
Data ack not 0 last	Data from directory, , along with an ack count, is in In Response queue, and this ack count is equal to the negation of the number of pending acks
Ack 0	Ack from directory, along with ack count = 0, is in In Response queue
Ack not 0	Ack from directory, along with with ack count not equal to 0, is in In Response queue, and this ack count is not equal to the negation of the number of pending acks
Ack not 0 last	Ack from directory, along with ack count not equal to 0, is in In Response queue, and this ack count is equal to the negation of the number of pending acks
Nack	Nack from directory is in In Response queue
Busy ack	Busy_ack from directory is in In Response queue

Table 3. Description of events (column labels) of cache controller specification

a	Issue GETS
b	Issue GETX
c	cancel optimization mode i.e. change the state of any block that is in I^* to I
d	Issue PUTX [and send data]
δ	Record requestor for future forwarding (forward ID field of TBE)
e	Send data from cache to requestor
ε	Send data from cache to forward ID (of TBE)
f	Send ack to memory
g	Allocate cache block
h	Cache hit (do LD/ST from cache)
i	Allocate TBE (number of pending acks=0, forward ID = null)
k	Pop Mandatory queue
l	Pop In Request queue
o	Remove event from In Response queue
p	Add number of pending acks to TBE
q	Decrement number of pending acks by one
r	Deallocate cache block
s	Deallocate TBE
t	Send [proc] ack to invalidator
u	Write data to cache
v	Load from cache
w	Send data from cache to memory
y	Store from cache
z	Stall

Table 4. Description of actions in cache controller specification

	GETS	GETX	PUTX (requestor is owner)	PUTX (requestor not owner)	Ack	Data
I	a b j /S	f b j /M				
S	a b j	q f b h g j /M				
M	a r d p j /MS	d f j /MM	l n j /I			
MI	e j	e j			k /I	
MS	e j	e j		u l s o j /S		m k /S
MM	e j	e j	l p n j /MI	l t o j /M	k /M	

Table 5. Directory specification

a	Add requestor to list of sharers
b	Send data to requestor
d	Forward request to owner
e	Send nack to requestor
f	Set owner equal to requestor
g	Clear list of sharers
h	Send INVs to all sharers
j	Pop incoming request queue
k	Pop processor response queue
l	Write PUTX data to memory
m	Write data to memory
n	Send ack to requestor
o	Send busy-ack to requestor
p	Clear owner
q	Remove requestor from list of sharers if in list
r	Add owner to list of sharers
s	Send PUTX data to sharers
t	Send PUTX data to owner
u	Remove requestor from list of sharers

Table 6. Description of actions in directory specification

Outgoing Request From Cache	b	Send request message from cache to network
	l	Pop request queue from cache
Outgoing Response From Cache	a	Send response message from cache to network
	k	Pop response queue from cache
Outgoing Forwarded Request From Dir	d	Send response message from dir to network
	n	Pop forwarded request queue from dir
Outgoing Response From Dir	c	Send response message from dir to network
	m	Pop response queue from dir
Incoming Request	f	Send request message from network to dir
	i	Pop Incoming Request Network
Incoming Forwarded Request	g	Send forwarded request message from network to cache and dir
	j	Pop Incoming Forwarded Request Network
IncomingResponse	e	Send response message from network to cache or dir
	h	Pop Incoming Response Network

Table 7. Network Interface specification. Events are listed in the first column, actions taken for each event in the second column, and description of actions in the third column.

	Load	Store	Eviction	Forwarded GETS	Forwarded GETX	INV	Proc ack	Proc last ack	Proc data	Data ack 0	Data ack not 0	Data ack not 0 last	Ack 0	Nack	Busy ack	
I						L										I
I^*	C															I^*
S	C					L										S
M	C	W	L	L	L											M
IS						L			C	C						IS
ISI						L			C	C						ISI
IM						L		W	W	W		W				IM
IMI								L'W	L'W	L'W		L'W				IMI
IMS								L'W	L'W	L'W		L'W				IMS
MI																MI
MI^{B}																MI^{B}
MI^{F}																MI^{F}

Table 8. Observer Specification for Cache Controller

Actions:

L Let x be the receiver of the message. The message requires x to send a message to a processor (or directory) y specified in the message. If LP_y is before LP_x, move LP_y forward to just after LP_x. Remove duplicate ST-nodes.

L′ Similar to L, but y is determined by the forwardID stored in x's TBE.

W Add a new ST-node to the window. Remove duplicate ST-nodes.

C If the LL-node for the address given in the message is before the LP-node of the receiver of the message, move the LL-node forward to immediately precede the LP-node. If there is no such LL-node, create one and place it immediatly preceeding the LP-node.

	GETS	GETX	PUTX (requestor is owner)	PUTX (requestor not owner)	Ack	Data
I	L	L				
S	L	L				
M			L			
MI						
MS				L''		
MM			L	L'''		

Table 9. Observer Specification for Directory

Actions:

L Identical to action L in Table 8.
L$''$ Similar to L, but the action is performed for all y in the directory's list waiting for shared access.
L$'''$ Similar to L, but y is the new owner.