Proving Sequential Consistency by Model Checking *

Tim Braun,† Anne Condon,§ Alan J. Hu,§ Kai S. Juse,† Marius Laza,§ Michael Leslie,‡ and Rita Sharma§

†Dept. of Computer Science, Technical University of Darmstadt
‡Dept. of Electrical and Computer Engineering, Univ. of British Columbia
§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. Unfortunately, the general problem of verifying that a finite-state protocol implements sequential consistency is undecidable, and in practice, validating that a real-world, finite-state protocol implements sequential consistency is very time-consuming and costly. In this work, we show that for memory protocols that occur in practice, a small amount of manual effort can reduce the problem of verifying sequential consistency into a verification task that can be discharged automatically via model checking. Furthermore, we present experimental results on a substantial, directory-based cache coherence protocol, which demonstrate the practicality of our approach.

1 Introduction

When multiple processors or other devices access a shared memory, the correct behavior is often unclear. The most intuitive model is that each memory operation happens instantaneously in real-time, so that each load returns the value of the most recent store to the same memory location. Such a model, however, produces contention, arbitration, and serialization of memory accesses, which is unneeded in many applications and limits performance. At the other extreme, a system in which all processors could freely read and write inconsistent or stale values would be nearly impossible to program. A *memory model* is a specification of the desired behavior of the memory system from the programmer's point of view.

Sequential consistency is a multiprocessor memory model introduced by Lamport [16]. 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. Intuitively, this means that processors can get out-of-sync with each other, perhaps because of caching and buffers, but that the same results could have been achieved had the executions of the processors been interleaved in some other way. Sequential consistency is important both as a practical memory model that provides ease-of-programming while allowing efficient hardware optimizations (e.g. [12]) and also as an extensively studied memory model that can be used to understand other, more relaxed models (e.g. [1]).

Model checking [7] has emerged as the dominant paradigm for formally verifying temporal properties of computer system designs. One of the most successful application domains for model checking has been multiprocessor cache coherence protocols (e.g., [17, 10, 6, 8, 15, 23, 26, 3, 20, 14] are some early works). The application domain is commercially very important, since almost all high-end servers are now cache-coherent multiprocessors; the protocols are tricky, highly concurrent, and hence bug-prone; and the protocols can be modeled in finite state, naturally supporting model checking.

An important verification task is to check whether a memory system implements a specified memory model. Unfortunately, the general problem of determining whether a finite-state protocol implements sequential consistency is undecidable [2], so directly attacking this problem via model checking is impossible. More subtle approaches are needed.

In this paper, we present experimental results on a methodology for proving sequential consistency using model checking. In contrast, previous experimental results on the use of model checking for reasoning about the correctness of cache coherence protocols either prove much weaker properties, or work only on a very restricted class of protocols which don't incorporate the subtle optimizations

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

that are prone to introduce errors in real protocols. Recent theoretical results [9, 21] show that model-checking can in principle be applied to proving sequential consistency of sophisticated, real-world protocols, but provide no experimental results.

In our methodology, 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 Section 2, we describe a method for creating observers, along with the corresponding checker. We then present experimental results in using our methodology to prove sequential consistency of a substantial directory-based cache coherence protocol. The presentation here is necessarily brief; details can be found in our technical report [4].

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.

The use of an observer, or witness, to aid in reasoning about the correctness of a protocol, is an old idea. Our use of an observer was inspired by the work of Plakal et al. [19], who introduce a verification approach based on logical clocks and apply it to a directory-based protocol. In contrast to logical clocks, which are unbounded, our observer is finite state, making it possible to use it as part of a model-checking approach.

Henzinger et al. [11] 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 the types of optimizations typically found in real protocols, such as the optimization due to Scheurich that we describe in Section 3. 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. [18] 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.

Two recent works describe a model-checking approach for automatically proving sequential consistency of real-world protocols [21, 9]. However, no experimental results are presented. This paper describes experimental results for a variation of one of these approaches [9]. In the current paper, the observer is constructed manually, unlike the automatic construction of [9], but is designed to have significantly fewer states in order to mitigate the state space explosion problem of model checking. Our results show that approaches of this type are indeed feasible for realistic protocols.

2 The Verification Method

In what follows, a protocol is a finite-state machine parameterized by the number of processors p, memory blocks b, and possible values per memory block v. Among the possible actions (transitions) of the protocol are load (LD) and store (ST) actions, which indicate the processor, the address (memory block number), and the value loaded or stored. A **protocol run** is a sequence of protocol actions that lead from state to state, starting with the initial state of the protocol. A **protocol trace** is the subsequence of a protocol run that includes exactly the ST and LD operations of the run. 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 (or some initial value, if there were no prior stores to that block). A protocol is **sequentially consistent** if every one of its traces has a reordering that respects the per-processor ordering of the trace, and is serial.

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 2.1) can prove that a run is sequentially consistent by using the windows. Let us now consider these ideas in more detail.

Definition 1 *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 delete vector node contains 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 2 *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.*

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.

2.1 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. The checker examines the 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 returned some initial value.

3. **A ST cannot be retroactive.** Intuitively, we prohibit a ST operation from occurring 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.

Combining an observer with the checker allows us to prove sequential consistency via model checking. If the run of an observer is accepted by the checker, then that run is sequentially consistent. Since both the observer and checker are finite-state, we can verify by model checking that all runs of the observer are sequentially consistent, which implies that the original protocol is sequentially consistent.

3 Example Protocol

The true test of our methodology requires experimentation. We have developed paper-and-pencil window observers for three different cache coherence protocols and selected the most challenging, a directory-based protocol, for the full model-checking experiment.

The protocol is a variant of one provided by the Multifacet group from the University of Wisconsin, to which we have added an optimization due to Scheurich [22], that allows a processor to continue to read a cache block after acknowledging an invalidation of that block. With this optimization, the protocol should be sequentially consistent, but not coherent (i.e., processors can continue to use stale data). The protocol involves several interacting entities — the processors, a directory, and a network interface — and is comparable in complexity to commercial directory-based protocols.

Roughly, 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, the processor initiates a transaction and enters a transient state. Several race conditions may arise, resulting in numerous transient protocol states to track the possibilities. Here, we present some illustrative situations. Further examples and the full protocol description can be found in our technical report [4].

1. If several processors share a block, and processor P wants Modify access, then P sends a GETX (Get Exclusive) message to the directory. 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 a block, and processor P wants Modify access, then P sends a GETX message to the directory; the directory forwards this request to Q and sets P as the new owner of the block. Processor Q (which is in state M) receives a "forwarded GETX" message from the directory, sends the data to P and goes to the I state. Processor P waits in the IM state until it gets the data from Q.

3. 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, with the main difference being that the observer updates and outputs a window, while executing the protocol. 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.

4 Experimental Results

We chose the Murphi verification system [10] 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 [14], 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. 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. 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. Eventually, we were able to debug the observer/checker as well, proving the protocol sequentially consistent.

The total effort 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. 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.

5 Conclusion and Future Work

We have presented experimental results on a methodology for proving sequential consistency of memory protocols by using 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 [13] 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.

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] 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.

[3] 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.

[4] T. Braun, A. E. Condon, A. J. Hu, K. S. Juse, M. Laza, M. Leslie, and R. Sharma. Proving sequential consistency by model checking. Technical Report TR-2001-03, Department of Computer Science, University of British Columbia, April 2001.

[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.

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	22.0s	37,317	34.1s
3	1	1	98,083	93.2s	742,984	568.0s
2	2	1	641,157	417.0s	71,242,781	47711.5s
3	1	2	754,577	636.2s	7,287,108	5741.0s
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 300Mhz Sun Ultra-60 with 2GB main memory. The state-space blow-up is moderate, showing that our method is at the edge of what is currently feasible for realistic protocols.

[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] A. E. Condon and A. J. Hu. Automatable verification of sequential consistency. In *13th Symposium on Parallel Algorithms and Architectures*, pages 113–121. ACM, 2001.

[10] 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.

[11] 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.

[12] M. D. Hill. Multiprocessors should support simple memory-consistency models. *IEEE Computer*, pages 28–34, August 1998.

[13] 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.

[14] 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.

[15] 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.

[16] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *ACM Transactions on Computer*, 28(9):690–691, September 1979.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[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.