Multiprocessor Memory Model Verification

Paul Loewenstein, Shailender Chaudhry,
Robert Cypher, Chaiyasit Manovit

Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, CA 95054 USA

Summary. Using the system architects' specified memory ordering as a function of execution, a multiprocessing system can be verified by monitoring not only that the result of an execution conforms to the required memory model, but that the result is exactly what the system architects intended. This allows a much more thorough verification of the system implementation, because in typical simulations many implementation errors do not propagate to memory model violations. This approach amplifies the coverage of an analysis using scarce architectural/mathematical insight via "informal" verification using simulation, which does not require the same insight to execute and interpret.

1 Introduction

Today's microprocessor clock frequencies are at multiple GHz, with execution pipelines capable of executing several instructions per clock cycle. By contrast, memory access times have remained stubbornly static as the execution rate of processors has increased. A processor may be able to execute around a thousand instructions in the time taken to perform a single access to main memory.

Despite the use of caches to hide much of this latency for many memory accesses, a significant proportion of accesses miss in the caches and have to access main memory directly. If a processor stalls on such accesses, considerable performance can be lost. Another approach is to allow the processor to speculatively execute past such long latency instructions, thereby executing instructions out of order. Considerable bookkeeping is required to ensure that the results of the execution are consistent with executing instructions, including memory-accessing instructions, in program order.

The move towards chip multiprocessing (CMP) processors requires the integration of multiprocessor cache coherence on the same chip as the execution pipelines. The interaction between the execution pipelines and the memory system can be quite complex, especially when the pipelines execute instructions out of order.

For this paper, we shall use as an example the Sparc TSO memory model as specified in the Sparc Architecture Manual Version 9. We shall only concern ourselves with cpu-initiated loads and stores to memory; we shall ignore atomic load-store operations, direct memory access by input-output devices and memory barrier instructions used for imposing additional memory ordering constraints. Inclusion of these operations does not substantially change the principles behind the verification approach.

The term "hardware" refers to an executable software model used for both simulation and for ultimate generation of the silicon processor chip.

2 Relationship to Other Work

Sorin et al. [2] introduce a methodology for conceptually attaching ordering timestamps to a hardware execution so that the memory order (see Section 3) can be determined from the execution. This paper assumes that real-time suffices for these timestamps, although using real-time may not suffice for some hardware implementations.

Condon et al. [1] verify (by informal mathematical proof) the implementation of TSO and Alpha memory models by a very abstract execution model (at a similar level of abstraction to Figure K-1 in Appendix K of [3]). The paper does not address the maintenance of an in-order programming model by an implementation that executes instructions out of order.

3 The Sparc TSO Memory Model

The Sparc TSO (Total Store Order) model is a specification designed for programmers, by defining what results are legal for an arbitrary single or multiprocessor program. There is no concept of time in the specification; rather, the specification uses program order (the order that instructions are programmed to execute on any single processor) and a memory order, defined below.

The following is derived from Appendix D of The SPARC Architecture Manual, Version 9 [4], or Appendix K of The SPARC Architecture Manual, Version 8 [3][1].

In the following, $<_p$ denotes program order. Variable l ranges over loads and variable s ranges over stores.

A subscripted variable denotes a memory operation to the address donated by the subscript (e.g. a variable s_a ranges over stores to address a).

For every execution, there exists a memory order $<_m$ over all loads and stores. For the operations covered in this paper, this memory order is constrained by 4 axioms:

[1] The Sparc V8 description in [3] is easier to follow than the V9 description in [4], which is based on a more relaxed memory model.

A load program-ordered before another load is also memory-ordered before that load:

$$\forall l_a l_b . l_a <_p l_b \Rightarrow l_a <_m l_b \tag{1}$$

A store program-ordered before another store is also memory-ordered before that store:

$$\forall s_a s_b . s_a <_p s_b \Rightarrow s_a <_m s_b \tag{2}$$

A load program-ordered before any store is also memory-ordered before that store:

$$\forall l s . l <_p s \Rightarrow l <_m s \tag{3}$$

The value of a load to address a is the value of the latest store in memory order that is either program-ordered before the load or memory-ordered before the load:

$$\text{Value}(l_a) = \text{Value}(\underset{m}{\text{Max}}(\{s_a : s_a <_m l_a\} \cup \{s_a : s_a <_p l_a\}) \tag{4}$$

The result of a program execution conforms to the TSO memory model if there exists a memory order satisfying the 4 axioms above.

It is very likely, when testing faulty hardware, that a bug could change a load value, but that there still exists a memory order satisfying the axioms. A checker that merely checks the existence of a valid memory order would fail to detect the bug.

We therefore prefer stronger testing, whereby the load values are compared with the value that the architects intended to be loaded, using the existential witness of the memory order that the system architects designed the system to implement. The model that defines these loaded values in terms of the initial values in memory, and the stores to that memory, is referred to here as the "golden memory model."

4 Strand Model

In a CMP system, "strand" refers to the equivalent of a "processor" in a traditional multiprocessor system. Each strand executes an independent instruction stream, although it may share hardware resources with other strands.

In the verification environment, each hardware strand (which typically executes instructions out of order, but retires instructions in program order) is run in parallel with a model that executes Sparc instructions in program order. This model takes load values from the golden memory model and supplies stores to that golden memory model.

There are two temporal events associated with each load or store:

1. Retiring, when the architectural state of the strand is irrevocably updated.

2. Committing[2], when a store can affect other strand's loads, or when a load ceases to see other strand's stores.

For stores, committing cannot occur before retiring; the committing of the store irrevocably changes the state of memory, which should not be performed until the strand's architectural state reflects the execution of the store. For loads, retiring cannot occur before committing, because the new architectural state the strand is dependent on the load's value. For any given hardware strand, committing occurs in memory order. In this paper, because the real-time committing events are used for creating the memory order witness, committing occurs in memory order even when viewed across multiple strands. This would not be the case if the caches were not maintained coherent in real-time.

5 Golden Memory Model

The Golden Memory Model allows the hardware to select a memory order that conforms to Axioms (1) through (3), and supplies load values conforming to Axiom 4.

Central to the golden memory model is a sparse address-indexed array of data values. We chose to implement this on a 64-bit granularity, because this is the coherence unit specified for Sparc. It could have been implemented on a cacheline granularity.

Also, within the golden memory model, is an abstract store queue per strand. Each store queue contains, in program order, retired stores that are yet to be committed. The contents of the queue is used for determining the value of loads by that strand, supplying the store values for stores program-ordered before the load but memory-ordered after the load (the RHS of the union in Axiom (4)).

Although this golden memory model has not been formally verified to implement Sparc TSO, it has been constructed such that an informal proof is quite straightforward. If the caches were not maintained coherent in real-time, (for example, if coherence operations were queued from a central bus to the individual caches).then further work would be required (Section 7).

5.1 Store Handling

For each hardware strand the golden memory model also contains an abstract store queue, containing retired stores that have yet to be committed. The retiring of a store is signaled by the strand model (executing in program order), and results in the enqueuing of the store into the abstract store queue. The committing of a store is signaled by the hardware and results in the

[2] This use of the term "committing" is not universal; in some literature, this term is used to indicate retiring of instructions.

popping of the oldest store from the abstract store queue and performing the store into the address-indexed memory array.

No matter how the hardware signals the committing of stores, the conformance to Axiom (2) is guaranteed by the FIFO properties of the abstract store queue in the verification environment. Hardware errors therefore cannot cause a store-ordering violation, but can affect what legal memory order results from the execution.

This simple approach to store handling works because in the hardware being verified, stores are committed to caches that are maintained real-time coherent.

5.2 Load Handling

The signal indicating the committing of a load is indicated by hardware. The load's value (referred to as a "snapshot") is read from the sparse reference memory at this time. This simple approach depends on the coherent caches that source loads being maintained coherent in real-time.

Care must be taken to ensure that the eventual value calculated for the load takes into account all older (in program order) stores to the same address and that commit after the committing of the load (and were therefore not reflected in the value of the snapshot when taken) (the stores on the RHS of the union in Axiom (4). The stores that commit after the retirement of the load are in the abstract store queue when the load retires, so that they can be taken into account at that time. However, stores that commit before the retirement of the load are popped from the abstract store queue before the load retires. To take these stores into account, store commits update the load snapshots for the same strand and address, as well as updating memory.

The snapshots are queued in a FIFO; retiring loads must obtain their values from the head of that FIFO, subject to modification by has yet uncommitted stores to the same address in the strand's abstract store queue. Not all the entries in the load snapshot FIFO need be consumed by loads; it is permissible to discard entries, without violating Axiom (1). What entries to consume and what entries to discard is a property of the hardware design, and is determined by signals supplied by the hardware.

Axiom (3) is satisfied because the committing of loads is before their retirement, retirement is in program order and stores commit after their retirement.

Determining from the hardware when to take snapshots and what snapshots to discard, is very hardware-specific and can be quite difficult; complicating issues include:

- Older instructions could trap before retiring.
- Loads could read memory speculatively out of order, before they are committed.
- A snapshot could be taken for a load in the "wrong" path following a mispredicted branch.

6 Conclusion

The architects' reasoning as to how a multiprocessor design implements the required memory model is used to increase the effectiveness of conventional "informal" simulation-based verification.

It is up to the system architects to determine precisely how the system implements the required memory model and to provide the hardware-specific existential witness of the memory order as a function of the execution.

It is only the system architects that are required to have the skills to bridge the gap between the state-transition system that represents the hardware and the partial order specification of the memory model. The bulk of the verification team just use simulation-based verification techniques.

7 Further Work

Should it be necessary to perform this type of validation on a design where the caches are not maintained coherent in real-time, we would need to adopt a logical timestamping approach along the lines described in [2]. The "golden model," against which the simulation is being verified, would become considerably more complex, because a simulation environment develops a system execution in real time, rather than in memory order.

For example, in a system where loads and stores are committed from caches that are time-skewed between strands, it is possible that a load could be committed after (in real-time) the committing of a store, memory-ordered after the load and to the same address as the load.

To ensure the correctness of a suitable but more complex model, it would be wise to formally prove its correctness with respect to the memory model specification.

References

1. A. E. Condon, M. D. Hill, M. Plakal, and D. J. Sorin. Using Lamport clocks to reason about relaxed memory models. In *Proceedings of Fifth International Symposium on High-Performance Computer Architecture*, Orlando, Florida, Jan. 1999.
2. D. J. Sorin, M. Plakal, M. D. Hill, and A. E. Condon. Lamport clocks: Reasoning about shared memory correctness. Technical Report CS-TR-1367, University of Wisconsin-Madison, Mar. 1998.
3. SPARC International. *The SPARC Architecture Manual: Version 8*. Prentice-Hall, 1992.
4. SPARC International: David L. Weaver & Tom Germond, Editors. *The SPARC Architecture Manual: Version 9*. Prentice-Hall, 1994.