

Specifying Relaxed Memory Models for State Exploration Tools

Sela Mador-Haim

Rajeev Alur

Milo M.K. Martin

University of Pennsylvania

April 6, 2009

Introduction

In recent years there is a renewed interest in memory consistency models [5, 6, 2, 4, 7]. Although most of the research on verification and concurrent program implicitly assumes that memory is sequentially consistent, many common microprocessor and compiler optimizations lead to relaxed memory models. To ensure that software behaves correctly under various hardware and software optimizations such as store buffers and statement reordering, both hardware and software designers develop various memory models that aim to capture the trade-offs between optimizations and programmability (i.e., the ease of writing correct code).

there is a large number of different specifications for both hardware and software memory models, such as Sun SPARC's RMO [12], Intel's x86 [3] and PowerPC [2] for hardware, and the Java and C++ software memory models. Memory models have been studied extensively during the last 30 years, but most of the earlier work [11, 9, 8, 10, 1] focused on the design of memory models and the way it affects microprocessor design. Recent development in multi-core processors and model checking techniques stirred new interest in memory models, focusing on new perspective such as verification of software in the context of weak memory models.

One issue we have to tackle while dealing with relaxed memory models is the handling of memory models within state exploration tools. We would like to be able to explore the state space of programs running on a relaxed memory model for two reasons:

1. We would like to be able to verify the correctness of a program running on a relaxed memory model. Even though most programmers assume a sequentially consistent execution, some sophisticated concurrent algorithms are highly sensitive to the memory model in use. In order to test and verify those algorithms, we need tools that support relaxed memory models.
2. There is a large number of different specifications for both hardware and memory models. Some of them are quite complex and could be difficult to understand. Space exploration tools allow us to explore the behavior of memory models and understand them better, as well as to compare between different memory models.

Operational specification

In order to explore programs on relaxed memory model, we need to specify the memory models in a way which is compatible with state exploration tools. Memory model specifications typically fall into two categories: *axiomatic* and *operational*. Memory models are often specified axiomatically, as a set of constraints on the order in which memory operations are observed by the various concurrent processes in the system. Axiomatic specifications are usually easier to write, and naturally decompose into a collection of simple axioms. However, these axioms describe global constraints on the complete execution trace, and therefore they are not well-suited for step-by-step execution where the complete execution trace is not known in advance. Operational specifications, on the other hand, are designed to work in a step by step manner and are therefore much better suited for the exploration of memory models. The hardware designs of concrete microprocessors are such operational models, but those models are too low level to be useful for software exploration. The translation of the axiomatic specification of memory models into simple, easy to understand and to execute operational specification is an open challenge.

In some cases, the specification of memory model in an operational style is quite intuitive. For instance, SPARC's TSO [12] is a memory model that relaxes Sequential Consistency (SC) by allowing to reorder writes after later reads, which models the effect of a store buffer. In order to specify this memory model operationally, all we need is to add a FIFO buffer between the processor and the main memory. This FIFO buffer stores write operation in order and perform them at a later time (while the processor keeps on running and perform later reads), and then at some later point in time perform the right. This is, however, one of the simplest models, and other memory models become extremely complex.

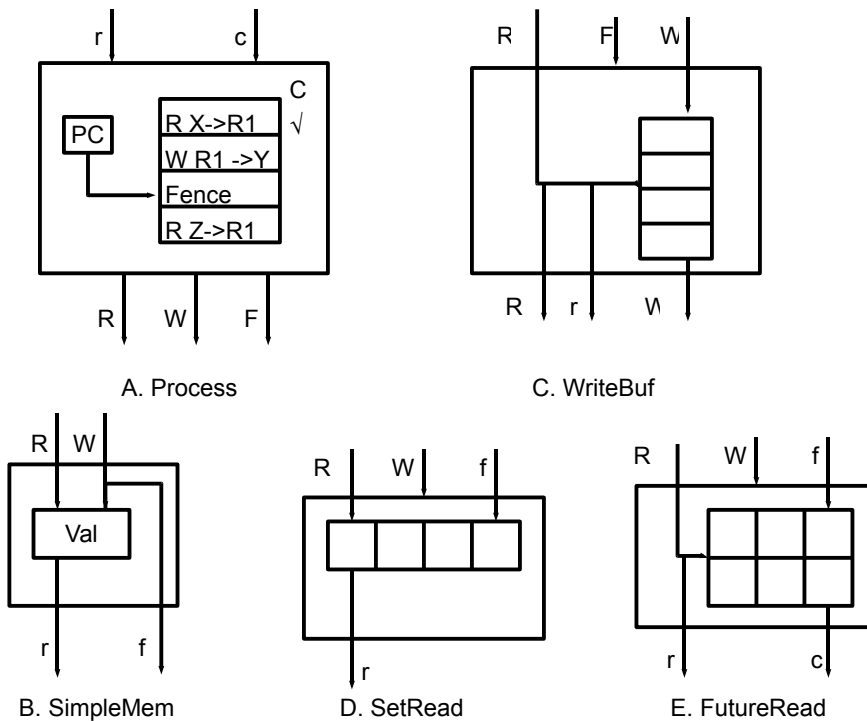


Figure 1: Plug and play components

To meet this challenge, we are exploring a modular specification of memory consistency. We can break the model into several smaller submodels, where each represents a different aspect of the memory model. We present a framework for specifying memory models in a way which is both operational and modular. We present a framework that simplifies the development of operational memory models by decomposing the model into four distinct independent parts that model different aspects of the specification, and then design separate components for each of those aspects. These four aspects are: reordering of writes, reordering of reads, the main memory and program dependencies. This approach has two advantages: first, as a result of decomposing the specification into smaller sub-properties, we get simpler properties that are easier to convert into an operational specification. Furthermore, once we designed a component that implements one aspect of this behavior, we can reuse it in other memory models that share the same behavior in one or more of those four aspects, and we could “plug-and-play” existing components and combine them to generate new memory models. We demonstrate this approach on five different memory models: Sequential Consistency, the three SPARC memory models (TSO, PSO and RMO) as well as a fourth memory model we call PSLO. We show that using only five simple components we can specify the behaviors of those four models, and well as some additional models that can be created by mixing these components in different ways.

The Component Model

Our specification of the operational semantics of weak memory models is based on a set of components that run as concurrent processes. The components can communicate with each other by sending and receiving messages. All communication between components are done via rendezvous message passing. In order to find which components we need for each model and how to specify them, we decompose the description of the model into four different aspects, and provide a component that handles each of those aspects separately. The decomposition we use is into components that handle the reorder of writes, the reorder of reads, the behavior of the main memory and program dependencies.

We demonstrate our approach by showing the components that are necessary in order to specify SC, Sun Sparc’s three memory models: TSO, PSO and RMO as well as one additional memory model we call PSLO (Partial Load Store Order). The components we need in order to specify these models are:

Processor: The job of the processor is to produce a sequence of memory operations and receive information back from the memory system. In this paper we describe a simple processor component that executes a pre-defined

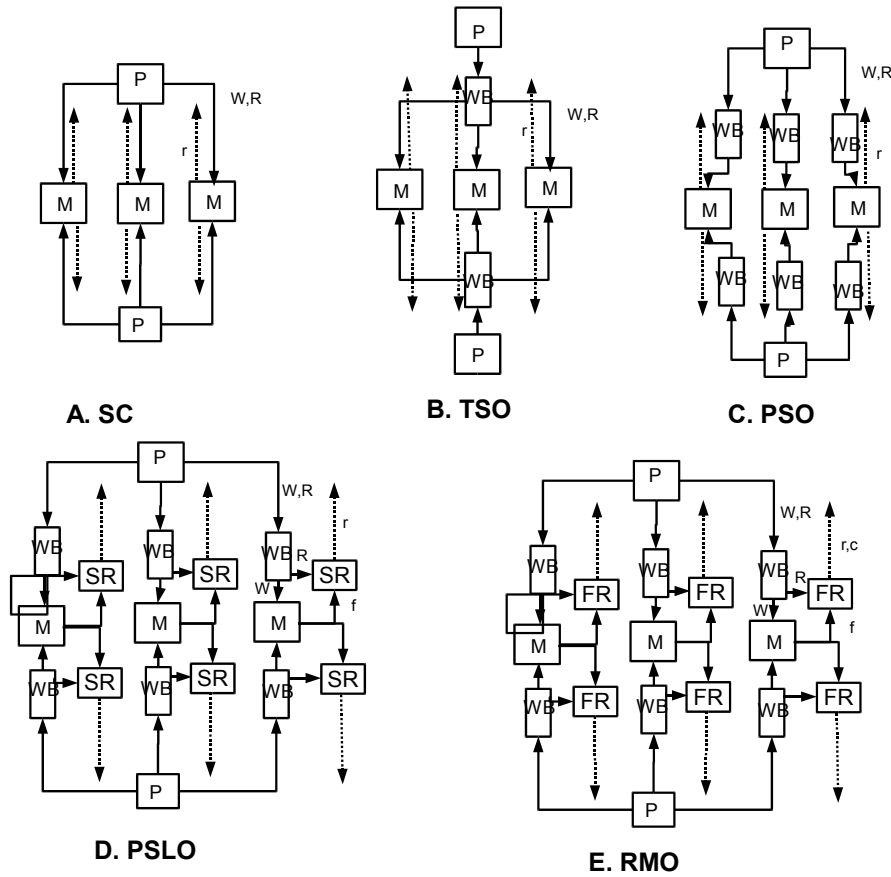


Figure 2: Component diagram of memory models

sequence of memory operations. However, it could be replaced by a component that accepts actual programs in any language and implement the same interface.

SimpleMem: This component represents the main memory. It contains one variable, *value*, which represents the content of this memory location. The value is updated when it receives write requests and it sends this value in response to read requests.

WriteBuf: Implements a FIFO buffer of write requests, as well as load forwarding feature, which returns the value in the buffer in case the corresponding read address is in the buffer. This component can be used to reorder writes with later reads, as well as to reorder writes with other writes (if we use a buffer for each address).

SetRead: Reorder reads by keeping track of past values. A value that was written in the past to a certain location is feasible unless there was a fence operation after this write or a local write to the same location. In both of those cases, SetRead empties everything from the set of values except for the latest written value.

FutureRead: Reordering writes before reads means we need to either read from the future or effectively write to the past. Both are quite tricky, but the former option is easier. We can read a value from the future by responding to a read request speculatively with a possible value we may see in the future. Each speculative guess becomes an obligation for the future: we keep track of all the guesses we made, and in case some processor writes a matching value, we remove this obligation. In case we get to the end of the program, to a fence, or to a local write to this address before all obligations are satisfied it means we made a wrong guess and we should not be able to proceed.

Using those five components, we can specify SC, TSO, PSO, PSLO and RMO, as shown in Figure 2.

Work in progress

We aim to demonstrate this approach by implementing a state exploration tool that uses the above component framework in order to specify memory models. The state exploration tool can be used both for program verification and model exploration. We need to address several different issues in order to use the models we designed for these two purposes:

1. One approach we take in order to enable designers to explore memory models and gain a better understanding of their behavior is by comparing the behaviors of two different models. A model comparison tool can compare the behavior of two (or more) memory models running all possible programs up to a certain bounded length, and it can produce concurrent programs that uncover differences in the behavior of those memory models. The main challenge here is to reduce the program space we explore by exploiting symmetry and other techniques.
2. We can use a state exploration tool to verify the behavior of a specific program by exhaustively exploring all its executions. The main challenge here is developing a tool with sufficient capacity in order to be useful to run actual program. We may need to develop optimization techniques that are specifically geared towards relaxed memory models to tackle the capacity problem.

References

- [1] Sarita V. Adve and Mark D. Hill. Weak ordering - a new definition. In *In Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, 1990.
- [2] Jade Alglave, Anthony Fox, Samin Ishtiaq, Magnus O. Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. The semantics of power and arm multiprocessor machine code. In *DAMP 2009 (to appear)*, 2009.
- [3] Jade Alglave, Anthony Fox, Samin Ishtiaq, Magnus O. Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. The semantics of x86-cc multiprocessor machine code. In *POPL 2009 (to appear)*, 2009.
- [4] Arvind Arvind and Jan-Willem Maessen. Memory model = instruction reordering + store atomicity. *SIGARCH Comput. Archit. News*, 34(2):29–40, 2006.
- [5] Gerard Boudol and Gustavo Petri. Relaxed memory models: an operational approach. In *POPL 2009 (to appear)*, 2009.
- [6] Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. Checkfence: checking consistency of concurrent data types on relaxed memory models. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 12–21, New York, NY, USA, 2007. ACM.
- [7] Jason F. Cantin, Mikko H. Lipasti, and James E. Smith. The complexity of verifying memory coherence and consistency. *IEEE Trans. Parallel Distrib. Syst.*, 16(7):663–671, 2005.
- [8] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Performance evaluation of memory consistency models for shared-memory multiprocessors. *SIGPLAN Not.*, 26(4):245–257, 1991.
- [9] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. *SIGARCH Comput. Archit. News*, 18(3a):15–26, 1990.
- [10] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, 1989.
- [11] David Mosberger. Memory consistency models. *SIGOPS Oper. Syst. Rev.*, 27(1):18–26, 1993.
- [12] Sparc. *The SPARC Architecture Manual Version 9*. Prentice Hall PTR, November 1993.