Memory Model Safety of Programs

Sebastian Burckhardt Madanlal Musuvathi
Microsoft Research
{sburckha,madanm}@microsoft.com

1. Introduction

Concurrency is pervasive in all systems software, including operating systems, databases, and web servers. With the future hardware performance improvements coming mainly from additional parallelism in the hardware, system designers will be forced make their programs more concurrent to exploit this trend.

A particular problem that programmers face when writing concurrent programs is to ensure correctness in the presence of memory reordering caused by the underlying hardware or the compiler. Such ordering relaxations are invisible to a single-threaded program. However, a concurrent program may exhibit more executions on a relaxed model than on a sequentially consistent (SC) machine. This additional behavior can result in subtle bugs that are very hard to find, understand, and debug.

One way to shield a programmer from these relaxations is to use appropriate concurrency abstractions, such as locks and transactional memory, provided either by a library or the compiler. Most programs will (and should) use such high-level concurrency abstractions. However, it is our position that a class of programs will still bypass these abstractions and use *ad-hoc* synchronization techniques. Such ad-hoc techniques include making direct use of hardware primitives for atomic operations (such as interlocked exchange, or compare-and-swap) and employing regular loads and stores for synchronization purposes. As such, these programs will be exposed to the effects of the relaxed memory model.

We foresee three kinds of programs that use such ad-hoc synchronization techniques. First, the libraries that implement high-level concurrency abstractions would obviously use hardware primitives to provide the abstractions. Second, the fast-paths of all systems will be heavily optimized for performance and programmers will be unwilling to pay the inherent cost of these abstractions. Last but not the least, there exists a huge body of legacy code that heavily relies on the strong memory-ordering guarantees provided by the current x86 architectures. Porting these programs to modern multi-core architectures, with possibly more relaxed memory models, remains a big challenge.

While programs with ad-hoc synchronizations are notoriously hard to get right [2, 8], they will remain crucial to the reliability of the entire system. Thus, it is necessary to develop verification methodologies to prove these programs correct. Simultaneously, it is important for the designers of future compilers and architectures to define memory models with verification of such programs in mind.

In our recent work [3], we propose a promising direction for verifying programs against relaxed memory models. Let $\mathcal{T}_\pi^Y$ denote the set of executions of program π on memory model Y. A program π is *memory model safe* for Y, or simply Y*-safe*, if $\mathcal{T}_\pi^Y = \mathcal{T}_\pi^{SC}$. In other words, a Y-safe program remains sequentially consistent when executed on Y. The verification of a Y-safe trivially reduces to the standard verification problem of checking if every execution in $\mathcal{T}_\pi^{SC}$ is correct. In addition, we show [3] that checking memory model safety for the *TSO* (total store order) [10] memory model can be efficiently combined with a model checker that systematically enumerates all executions in $\mathcal{T}_\pi^{SC}$.

```
volatile bool isIdling;
volatile bool hasWork;

//Consumer thread
void BlockOnIdle(){
   lock (condVariable){
      isIdling = true;
      if (!hasWork)
          Monitor.Wait(condVariable);
      isIdling = false;
   }
}

//Producer thread
void NotifyPotentialWork(){
   hasWork = true;
   if (isIdling)
     lock (condVariable) {
        Monitor.Pulse(condVariable);
     }
}
```

Figure 1. Violation of *TSO*-safety in a C# program.

The main motivation for memory model safety arises from our observation that programmers, even those using ad-hoc synchronizations, *expect* their programs to be sequentially consistent. They design their programs to be correct for SC executions and insert memory ordering fences to counter relaxations where necessary. In particular, any program execution that is not SC is almost always an error, resulting either from an insufficient use of fences or a misunderstanding of the underlying memory model.

As an example, Figure 1 describes a program that violates *TSO*-safety and contains a serious error. This example is a modified version of a bug that we found [3] in a production level concurrency library at Microsoft. The program uses two flags `isIdling` and `hasWork` apart from a condition variable to synchronize between consumers and producers. An idle consumer waits on the condition variable if `hasWork` is false, but only after setting `isIdling` to true. To optimize for the common case in which there are no idle consumers, the producer acquires the lock only when `isIdling` is true. Also, to account for a possible race on the `isIdling` flag, the producer sets `hasWork` to true before checking the `isIdling` flag. We can see that in all sequentially consistent executions the producer wakes up the idle consumer, if any. However, this is not guaranteed in *TSO* where a store can be reordered with a

subsequent load.[1] In particular, the consumer can read `hasWork` before its write to `isIdling` is visible to the producer. Thus, there is an execution in which the producer erroneously decides that no consumer is idling and the consumer blocks for ever.

In this paper, we explore the possibility of checking memory model safety of programs for general memory models. In particular, we identify a class of memory models for which memory model safety can be verified by only exploring executions in $\mathcal{T}_\pi^{SC}$. We believe that many practical memory models fall in this class, making this verification approach very promising.

2. Formulation

We now proceed to formulate three wishes to the designers of memory models, and explain how a model that satisfies all three wishes simplifies the task of checking the memory model safety of programs.

First, we believe that many models suffer from being vague or incomplete. Many official specifications give an informal (and occasionally mystifying) description accompanied by a number of examples that convey just enough information to implement basic synchronization primitives such as locks. However, the information is usually not sufficient to verify the memory model safety of arbitrary programs.

Wish 1. Memory model designers should give a formal description of the core memory model.

A specification for a memory model Y should provide the following components:

- It should define a set $\mathcal{T}$ of *memory traces*. A memory trace $E \in \mathcal{T}$ represents the aspects of a program execution that are relevant for the purposes of the memory model. Typically, traces contain information about the execution that is not directly observable to the program (for example, it may record information about the order of accesses performed by the program), but omits execution details that are not relevant (for example, it may abstract data values).
- For each program π, it should define a set $\mathcal{T}_\pi^Y \subset \mathcal{T}$ of traces that represents all possible partial and complete executions of π on Y. We call the problem of deciding whether a particular trace $E \in \mathcal{T}$ is in $\mathcal{T}_\pi^Y$ the *membership* problem.

Different specification styles of memory models use different methods to specify the set $\mathcal{T}_\pi^Y$. *Operational* memory models [4, 9] employ automata (usually nondeterministic and infinite-state); a trace is a member if and only if there exists a run of the automaton that accepts some linearization of the trace. Conversely, *axiomatic* memory models [11, 1, 7] directly state the logical requirements that members have to satisfy. Most memory models found in architecture manuals follow an axiomatic style.

Deciding membership is not always easy. Operational memory models often exhibit substantial nondeterminism; for axiomatic models, expensive decision procedures may be required.

Wish 2. Memory Models should allow us to decide the membership problem efficiently.

Note that the choice of what information to include in the trace influences the complexity of the membership problem. For example, Gibbons and Korach [5] show that if the trace records the relative order of stores that target the same location, the membership problem for sequential consistency can be decided in polynomial time, but without this information, it is NP-complete.

Our last wish concerns the ability to reason inductively about memory model safety. To illustrate our point, we first present a basic formalization of memory traces. Then, we define the notion of a borderline execution, and show how it can simplify the task of verifying memory model safety.

A trace is a collection of events, each representing a regular memory access (store or load), an atomic access (such as compare-and-swap), or a memory fence. Let $\mathbb{N}$ be the set of natural numbers, and let $\mathit{Proc} = \{1, \ldots, N\}$ be a finite set of processor identifiers for some fixed bound $N \in \mathbb{N}$. Then we define the set of instruction identifiers $\mathit{Id} = \mathit{Proc} \times \mathbb{N}$, where each tuple (p, n) represents the n-th instruction issued by processor p. We call n the *issue index*. We now define a *memory trace* E to be a tuple $E = (I, R, W, \mathit{src}, X)$ where

- $I \subset \mathit{Id}$ is a set of instruction identifiers such that the indexes issued by each processor p form a contiguous range $\{1, \ldots, n_p\}$ for some $n_1, \ldots, n_N \geq 0$.
- R and W are the subsets of I corresponding to instructions that read or write from memory, respectively (R and W may intersect; for instance, atomic operations both read and write the same location).
- src is a partial function $R \rightharpoonup W$. The idea is that for each instruction $r \in R$, $\mathit{src}(r)$ is the instruction that sourced the value read by r, or is undefined if r reads the initial value of the memory location.
- X captures additional properties of the events in I in a way that we leave purposefully unspecified here for the sake of brevity and generality. For example, X may track attributes of events (such as the opcode or the address of the targeted memory location), and some ordering information (such as the relative order of stores to the same location).

Now, suppose we are given two traces $E = (I, R, W, \mathit{src}, X)$ and $E' = (I', R', W', \mathit{src}', X')$. We call E a *prefix* of E' (and write $E \sqsubseteq E'$) if $I \subset I'$, $R \subset R'$, $W \subset W'$, $\mathit{src} = \mathit{src}'|_I$, and (so to speak) $X = X'|_I$. Then $\sqsubseteq$ defines a partial order on traces. We say E' is a *successor* of E if $E \sqsubseteq E'$ and $|I'| = |I| + 1$. If E' is a successor of E, we call E a *predecessor* of E'. We say E is *nontrivial* if $|I| > 0$.

We call $E \in \mathcal{T}_\pi^{SC}$ a *borderline trace* for π and Y if there exists a successor E' of E such that $E' \in (\mathcal{T}_\pi^Y \setminus \mathcal{T}_\pi^{SC})$. We say a memory model Y *guarantees borderline executions* if has the following property for all programs π: if there exists a trace $E \in \mathcal{T}_\pi^Y \setminus \mathcal{T}_\pi^{SC}$, then there exists a borderline trace E for π and Y.

Wish 3. Memory Models should guarantee borderline executions.

A memory model that guarantees borderline executions provides us a way to check the memory model safety of a program by only reasoning about successors of executions in $\mathcal{T}_\pi^{SC}$, instead of reasoning about all executions in $\mathcal{T}_\pi^Y$.

THEOREM 1. *If the memory model Y guarantees borderline executions, then a program π is Y-safe if and only if for all sequentially consistent executions $E \in \mathcal{T}_\pi^{SC}$, there exist no successors E' of E such that $E' \in \mathcal{T}_\pi^Y - \mathcal{T}_\pi^{SC}$.*

Because the above definition gives a precise characterization of memory model safety, we can use it in combination with a variety of (under- or over-approximating) techniques, such as runtime verification, model checking, or static program analysis. As a particular

[1] Unlike Java, the C# does not guarantee strong sequential ordering of volatile accesses.

case, see our work on checking TSO-safety using stateless model checking [3].

For some memory models, it is particularly easy to prove that they guarantee borderline executions. Specifically, call a memory model Y *inductive* if for all programs π and nontrivial traces $E \in \mathcal{T}_\pi^Y$, there exists a predecessor E' of E such that $E' \in \mathcal{T}_\pi^Y$. It is easy to see that all inductive memory models guarantee borderline executions: Given a trace $E \in (\mathcal{T}_\pi^Y - \mathcal{T}_\pi^{SC})$, we can prove that there exists a borderline trace by induction over the number of instructions in E.

To show that a particular memory model is inductive, we need to show that for any program π and any trace E in $\mathcal{T}_\pi^Y$, there exists a processor p such that we can remove the last instruction issued by p from E and obtain a predecessor in $\mathcal{T}_\pi^Y$. Note that we can never remove stores that are loaded by some load, because the resulting execution would not be a prefix.

3. Discussion of Memory Models

In this section, we examine some commonly used memory models in light of the wishes we formulated in the previous section.

3.1 Intel 64

The recent whitepaper by Intel [6] contains a large number of example traces (also known as "litmus tests") to describe the memory model. It is not formal, and does thus not satisfy wish 1. As a consequence, we also do not know how to decide the membership problem for general programs. However, we believe that with some more work, it is possible to develop a formal model that is (1) consistent with the examples in the paper and (2) for which membership can be decided with reasonable efficiency. Both wish 1 and wish 2 thus seem "within reach". As for wish 3, it appears that it is indeed satisfied: the model explicitly disallows stores to be ordered before older loads. This implies that the model is inductive, by the following argument. For a trace $E = (I, R, W, src, X)$, define a binary relation $\rightarrow_{cs}$ on I by specifying that $i \rightarrow_{cs} j$ if and only if either (1) $i <_p j$ and $i \in R$ and $j \in W$, or (2) $src(j) = i$. The requirement that stores are not ordered before older loads then implies that $\rightarrow_{cs}$ is acyclic, which in turn implies that the memory model is inductive, as follows: for any trace, look at all the last instructions issued by each processor. If any of them is a load, we can safely remove it, obtaining a predecessor execution. If any of them is a store that is not the source of any load, it can also be safely removed. However, if all last instructions are stores that are loaded by some load, a $\rightarrow_{cs}$-cycle results.

3.2 Sparc TSO, PSO, and RMO

All three of these models are formally defined in the SPARC manual [10]. For a detailed description on why TSO ("total store order") satisfies all our wishes, see our previous work [3]. We believe that the same results can be obtained for PSO ("partial store order"). Also, we believe that RMO ("relaxed memory order") satisfies wish 2. However, RMO does not satisfy wish 3, as exemplified by the trace in Fig. 2. RMO allows the reordering of the store in the third instruction of processor 1 with the load in the first instruction as there are no dependencies between them. This speculative store allows process 2 to read the stored value and subsequently set y to 1. Processor 1 then reads this value to satisfy the final conditions of the trace. However, there exists no SC execution which is a predecessor of this execution. We believe that this failure to guarantee borderline executions is not intentional, and may in fact be an artifact of the formalization (and could be fixed by a slight modification of the definition of the dependency relation): our intuitive understanding suggests that actual hardware implementations guarantee borderline executions because they refrain from performing speculative stores.

Initially: $x[0] == x[1] == y == 0$

processor 1		processor 2	
(1,1)	r1 = y	(2,1)	r2 = x[0]
(1,2)	x[r1] = 1	(2,2)	y = r2
(1,3)	x[0] = 1		

Eventually: $r1 == r2 == 1$

Figure 2. A program trace that it allowed by RMO, but for which there exists no borderline execution for RMO.

3.3 PowerPC

The PowerPC manual is not formal and gives only very few examples, leaving some room for guesses. It seems to aim for a similar position as the RMO model as far as local instruction reorderings of stores and loads are concerned (independent accesses can be reordered, but not dependent accesses), but takes a much more liberal approach to store atomicity. Our preliminary experiences suggest that PowerPC traces can be formalized in a such a way that wishes 1 and 2 are satisfied. As for wish 3, it seems that just like with RMO, the definition of dependency is not precise enough to guarantee borderline executions.

3.4 Java

As typical for a language-level model, the JMM (Java Memory Model) (formally specified in [7]) distinguishes between synchronization accesses (including locks, unlocks, and volatile loads and stores) and regular data accesses (all others). It provides very strong guarantees for synchronization accesses (any projection of a JMM trace onto the subset of synchronization accesses is sequentially consistent), and implies full sequential consistency for race-free programs. However, it provides very weak guarantees for programs with races. Wish 2 is thus at least partially satisfied: the membership of a trace can be decided efficiently if it is race free; otherwise, however, the task is daunting, as the justification of a particular trace requires the construction of a sequence of program traces that may involve different paths through the program. As for wish 3, we have not yet determined if it is satisfied (we believe it not to be satisfied, but have not found a witness yet).

4. Conclusions and Future Work

In this paper, we show that for a general class of memory models, checking the memory model safety can be efficiently performed while reasoning about the immediate successors of sequentially consistent executions only. We believe that this class includes many useful memory models and presented various challenges that stymied our effort. In future, we hope to appropriately characterize the RMO and the PowerPC memory models so that they admit borderline executions. Also, we would like to explore the possibility of combining memory model safety verification with a variety of static and dynamic analysis techniques.

References

[1] H.-J. Boehm and S. Adve. Foundations of the C++ concurrency memory model. In *Programming Language Design and Implementation (PLDI)*, 2008.

[2] S. Burckhardt, R. Alur, and M. Martin. CheckFence: Checking consistency of concurrent data types on relaxed memory models. In *Programming Language Design and Implementation (PLDI)*, pages 12–21, 2007.

[3] S. Burckhardt and M. Musuvathi. Effective program verification for relaxed memory models. Technical Report MSR-TR-2008-12, Microsoft Research, 2008.

[4] D. Dill, S. Park, and A. Nowatzyk. Formal specification of abstract memory models. In *Symposium on Research on Integrated Systems*, pages 38–52. MIT Press, 1993.

[5] P. B. Gibbons and E. Korach. The complexity of sequential consistency. In *Parallel and Distributed Processing*, pages 317–325. IEEE, 1992.

[6] Intel Corporation. *Intel 64 Architecture Memory Ordering White Paper*, August 2007.

[7] J. Manson, W. Pugh, and S. Adve. The Java memory model. In *Principles of Programming Languages (POPL)*, pages 378–391, 2005.

[8] V. Morrison. Understand the impact of low-lock techniques in multithreaded apps. *MSDN Magazine*, 20(10), October 2005.

[9] S. Park and D. L. Dill. An executable specification, analyzer and verifier for RMO (relaxed memory order). In *Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 34–41, 1995.

[10] D. Weaver and T. Germond, editors. *The SPARC Architecture Manual Version 9*. PTR Prentice Hall, 1994.

[11] Jason Yue Yang. *Formalizing Shared Memory Consistency Models for Program Analysis*. PhD thesis, University of Utah, 2005.