WMMSIM: A Weak-Memory Model Simulator

Gowtham Kaki
Purdue University
gkaki@cs.purdue.edu

Abstract

We present WMMSIM, a solver-aided DSL to simulate weak memory models. A weak memory model can be defined in terms of acceptable violations of a) program order, and b) atomicity of writes (stores) in shared-memory concurrent program that is otherwise sequentially consistent [Lamport 1979]. WMMSIM relies on this observation to describe a memory model constructively as a non-deterministic function that takes a shared memory concurrent program, and non-deterministically re-orders instructions in the program. WMMSIM then symbolically interprets the resultant program to determine if any unexpected behaviours are possible under the memory model. If such behaviour is possible, WMMSIM outputs an execution of the shared memory program which reproduces unexpected behaviour. We believe that WMMSIM approach can be extended to a full-fledged framework that greatly aids concurrent programmers in understanding and tailoring their programs for new memory models that are constantly being developed.

1. Introduction

Recent years have witnessed the proliferation of multi-core and many-core architectures into general-purpose computing domain. Concurrent programming over these systems is complex even under programming models offering strong semantic guarantees (eg: sequential consistency [Lamport 1979]). However, sequential consistency (SC) is an expensive affair as both compilers and underlying hardware have to forego a range of optimizations to satisfy the requirements of SC [Adve and Boehm]. Instead, existing hardware and programming models provide weaker consistency guarantees by default [Adve and Gharachorloo], strengthening them only on demand. Such weaker models take an unsuspecting programmer by surprise by inducing new hard-to-understand behaviours into otherwise correct programs [Adve and Boehm]. In this paper, we describe a solver-aided DSL [Torlak and Bodik] that is intended to be used in two different ways: Firstly, it can be used by programmers to a) abstractly describe their shared-memory concurrent programs as list of lists of STOREs and LOADs, and b) define the set of outputs (A) that they intuitively expect the program to produce. Secondly, the DSL can be used by compiler writers (aware of hardware architectures) to describe the semantic guarantees offered by their language (call is L) as a non-deterministic function in the DSL. The function takes a shared-memory concurrent program in L and outputs a symbolic program that captures the entire space of behaviours possible under the reorderings of memory operations allowed by L's memory model. The DSL then checks if user's understanding of the program's behaviour (A) matches with the reality (as dictated by L's memory model). If there is a mismatch, the DSL shows the concrete set of valid reorderings that the compiler can peform to exhibit the behaviour that programmer did not expect. Furthermore, we are working on an extension to the DSL that will also suggest the placement of FENCE instructions at appropriate locations to constrain reorderings so that reality matches with user's expectations. WMMSIM is primarily intended to be a workbench to understand the implications low-level hardware optimizations (eg: non-coherent caches, store buffers etc) on high-level shared-memory programs.

The rest of the paper is organized as follows. The following section gives an overview of weak memory models, and possible anomalies resulting from programming under weak consistency guarantees. Section 3 describes how non-deterministic programming and angelic execution can be used to automatically reason about shared memory programs under weak memory models. Section 4 gives a brief introduction to Rosette [Torlak and Bodik 2014], and describes how the ideas presented in Section 3 were implemented in WMMSIM using Rosette's solver-aided features. Section 5 demonstrates the effectiveness of WMMSIM in generating witness executions exhibiting erratic behaviours for example programs. Finally, section 6 talks about future work and concludes.

2. Weak Memory Models

The memory model, or memory consistency model, of a shared memory multi-processor provides a formal specification of how the memory system will appear to the programmer. Such an artifact is essential to bridge the gap between the behaviour expected by the programmer and the actual behaviour supported by the system. Effectively, consistency model places restrictions on the values that can be returned by a read operation. In a sequential program, a read always

Initially `X = Y = 0`. Assume `r1` and `r2` are register variables.

```
Thread1    Thread2
X = 1;     Y = 1;
r1 = Y     r2 = X
```

Figure 1. The core of Dekker's Mutual Exclusion Algorithm

```
Execution1    Execution2    Execution3
X = 1;        Y = 1;        X = 1;
r1 = Y;       r2 = X;       Y = 1;
Y = 1;        X = 1;        r1 = Y;
r2 = X;       r1 = Y;       r2 = X;
//r1 == 0     //r1 == 1;    //r1 == 1
//r2 == 1     //r2 == 0;    //r2 == 1
```

Figure 2. Some Sequentially Consistent Executions of the Program in Fig. 1

returns the value of the last write to the same memory location. Consequently, memory models are simple in the absence of concurrency. On the other hand, in concurrent programs, it is not always obvious what the last write operation is. For example, consider the code snippet in Fig. 1, which shows two threads sharing variables `X` and `Y`. What values do register variables `r1` and `r2` contain? Since there are no obvious last write operations to variables `X` and `Y` before their corresponding reads, the question has no obvious answer.

To answer this question, an intuitive extension of the uniprocessor model can be applied to the multiprocessor case. This model is called *sequential consistency* (SC). SC model requires that all memory operations appear to execute one at a time, and operations of a single thread appear to execute in the program order. For example, some sequentially consistent executions of the concurrent program shown in Fig. 1 are shown in Fig. 2. As evident from the figure, `r1` and `r2` can assume different values depending on the interleaving of operations between two threads. Nonetheless, there is always one sequential execution of operations from both the threads that explains the values assumed by `r1` and `r2`.

Sequential Consistency provides simple and intuitive programming model. Unfortunately, it disallows many hardware and compiler optimizations that are possible in uniprocessors by enforcing strict order among shared memory operations. For example, consider *store buffers*, which are commonly used by the memory hardware to mask write latency. When a write is issued by the processor, memory hardware buffers the write in the local store buffer and signals success to the processor, before actually committing the write to the main memory. Since only the writes committed to the main memory are available to rest of the processors, utilizing store buffers results in writes that are considered succesful, but are not yet available at rest of the processors. This results in the violation of SC guarantee. For example, in Fig. 1, writes `X=1` and `Y=1` can be buffered at their respective processors, and therefore may not be available to the other thread. Consequently, it is possible that the reads `r1=Y` and `r2=X` may read old values of `X` and `Y` (namely `0`). Observe that this behaviour of both reads returning `0` is not possible under any sequentially consistent execution (see Fig. 2), and may confound a programmer who is unaware of workings of the underlying hardware.

Common compiler optimizations, such as prefetching loads, also violate SC guarantee. In order to support SC memory model, it is therefore necessary to forego a range of compiler and hardware optimizations that are quite fundamental for performance. For this reason, most practical multi-processor memory hardware and compilers do not support SC model. However, in the absence of a memory model, programmers have to painstakingly reason about program correctness in terms of low-level hardware optimizations. To avoid this, a number of relaxed memory consistency models have been proposed. These models describe the effect of low-level optimizations in terms of *relaxations* to the strict SC model. For example, a memory model of the hardware utlizing store buffer optimization might relax the program order the requirement of SC model that in an SC execution operations of a thread must appear in program order. The model might instead allow a write and a a subsequent read to not appear in the same order in a valid execution. Under this relaxation, the writes to `X` and `Y` can be reordered with respect to reads to `r1` and `r2`, respectively, in Fig. 1 making the following execution possible:

```
r1 = X;
r2 = Y;
X = 1;
Y = 1;
//r1 == 0
//r2 == 0
```

Similar to the store buffer optimization, more subtle hardware optimizations can also be described in terms of instruction reorderings, which can be more elaborate than the *write-read* reordering shown above.

3. WMMSim

Relaxed memory models provide an abstract description of the behaviour of a shared memory system in terms of reorderings. However, reasoning about a shared memory concurrent program becomes more complicated in presence of instruction reorderings due to large number of possible interleavings, even for small progams. It is therefore desirable if the reasoning can be performed automatically. WMMSim is a tool built for this purpose. The primary insight underlying WMMSim is that a shared memory concurrent program can be transformed to a non-deterministic sequential program capturing all possible interleavings allowed by the memory model. For example, consider the program in Fig. 1. A non-deterministic program that can generate *any* SC execution of the program can be described by the following algorithm:

```
Data: $T_1, T_2$ : Two threads as list of instructions
Result: An SC execution represented as a list of
        instructions
Exn := [];
while $T_1$ and $T_2$ not empty do
    T := oneOf($T_1$,$T_2$);
    Instr := removeFirst(T);
    Exn := addAtEnd(Exn,Instr);
end
if $T_1$ is empty list then
    Exn := append(Exn,$T_2$)
end
if $T_2$ is empty list then
    Exn := append(Exn,$T_1$)
end
return Exn
```

In the above algorithm, the function `oneOf` non-deterministically selects one of its both arguments (for eg, based on a coin toss). One way to look at the above algorithm is as a *definition* of SC memory model for a shared-memory program of two threads. Alternatively, if we redefined the algorithm for n threads, the algorithm simply serves as a constructive definition of SC. Our main observation is that such constructive definitions can be given for most[1] weak memory models.

Although the above algorithm is constructive (i.e., it can generate any SC execution of the program in Fig. 1), it is only of theoretical interest; it cannot be used to generate an SC execution that satisfies or violates a given property. To realize the practical utility of the algorithm, we replace the construct `oneOf` with `angelicOneOf`, and add an assertion at the end. `angelicOneOf` is a hypothetical construct that non-deterministically selects one of its both arguments *such that* an assertion in the future is satisfied. The assertion could be an assertion on the generated execution. For example, if we are interested in knowing if there exists an SC execution of program in Fig. 1 such that `r1==1` and `r2==0` under that execution, we modify the algorithm as following: The semantics of `angelicOneOf` lets us guarantee that any execution generated by the algorithm satisfies the stated assertion. Since the algorithm only generates SC executions, the returned execution is an SC execution satisfying the stated assertion. If there does not exist such an execution, the assertion simply fails. In the similar manner, the behaviour of a shared memory program under any relaxed memory model can be understood.

```
1 Exn := [];
2 while $T_1$ and $T_2$ not empty do
3     T := angelicOneOf($T_1$,$T_2$);
4     // ... rest remains same
5 end
6 // ...
7 assert(Exn $\vdash$ r1==1 $\wedge$ r2==0);
8 return Exn
```

The aforementioned methodology crucially relies on angelic non-determinism. In order for the method to be practical, `angelicOneOf` construct has to be realized. WMMSim realizes it with help of Rosette's[Torlak and Bodik 2014] solver-aided features. Following section elaborates.

4. Rosette Implementation

Rosette[Torlak and Bodik 2014] is a solver-aided symbolic virtual machine. In Rosette, we can express a computation by using symbols for unknown values, make assertions on the computation, and finally solve for unknown symbols such that the assertions are valid[2]. The `angelicOneOf` feature described in the previous section can be implemented with help of binary symbols as following (shown in Haskell):

```
angelicOneOf (x,y) = if (k=0) then x else y
  where k = newBinarySymbol()
```

A binary symbol takes the values of either `0` or `1`. Everytime a call to `angelicOneOf` is made, a new binary symbol is created. For a program with n threads, we create integer symbols. The values that these symbols take determines the execution generated by the algorithm.

The assertion on the execution shown on the line 7 of the 2nd algorithm is made by implementing an interpreter for the shared memory language. For the sake of simplicity, we define a very simple language with the following grammar for shared memory programs:

i	$\in$	`Integer constants`	
r	$\in$	`Register-allocated vars`	
x	$\in$	`Memory-allocated vars`	
c	$::=$	$STORE\ x\ i \mid LOAD\ x\ r$	`Commands`
t	$::=$	$c; t \mid \cdot$	`Thread`
p	$::=$	$t \parallel p \mid \cdot$	`Program`

The interpreter for the language is implemeted trivially (using a store to simulate main memory and an environment to simulate registers). The non-deterministic execution generated by the constructive definition of the memory model is

[1] Weak memory models violating write atomicity have more complex constructive definitions. Source code (wmmsim.rkt) has a demo.

[2] For the sake of demonstrating solver-aided features of Rosette that were used in implementing WMMSim, we provide a simple sudoku example (sudoku.rkt) with our source code.

given to the interpreter, which interprets the execution and generates symbolic values for register variables. The symbolic values are then asserted equal to concrete values, which generates concrete bindings for symbols (i.e., resolves symbols). The result is a concrete witness of the program trace possible under the memory model which satisfies the stated assertion.

5. Evaluation

To evaluate our approach, we implemented a constructive definition of a memory model that violates atomicity of writes. Such a weak memory model is supported by multiprocessor systems that make use of non-coherent caches, where a processor's write is propagated eventually to all other processors. The result of violating atomicity of writes is that the order of two writes that are not related by the program order can be different in different threads. For example, consider the program shown in Fig. 1. Under the memory model that violates atomicity of writes, the writes to X and Y, by the virtue of not being related by the program order, can appear in different order in both the threads, making the following execution possible:

```
Thread1 Exec    Thread2 Exec
X = 1;          Y = 1;
r1 = Y          r2 = X
Y = 1;          X = 1;
//r1 == 0       //r2 == 0
```

Notice that the execution above, unlike the SC execution or an execution on hardware with store buffers (Sec. 2) cannot be represented as a sequence of instructions. Instead, the execution is a collection of instruction sequences, where each sequence corresponds to the execution of a single thread in the program. Notice that the confounding result of r1==0 and r2==0 is also possible with non-coherent caches, but the witness execution (shown above) is remarkably different for the execution producing similar result under store buffer optimization (Sec. 2). When given the constructive definition of the memory model violating atomicity of writes, along with the assertion of r1 == 0 ∧ r2 == 0, WMMSIM produces the above execution as witness instead of the execution in Sec. 2. This ability to produce appropriate witness executions helps programmer to device appropriate synchronization strategies tailor made for each memory model.

As another example, consider the program shown in Fig. 3. Under the weak memory model violating atomicity of writes, is it possible for the program to produce an execution where r1−3 all hold the value 1, but either one of r4 or r5 is 0? Such an execution is not possible under SC or a weaker model abstracting store buffers, as there does not exist any sequence of instructions that produce the stated result. However, such an execution is possible under the memory model violating atomicity of writes, as the model does not require the execution to be a sequence of instructions. WMMSIM produces following witness execution for the program in Fig. 3 (execution of only threads 3 and 4 shown):

Initially x = Y = 0. Assume r1−5 are register variables.

```
Thread1   Thread2   Thread3   Thread4
X = 1;    Y = 1;    r1 = X;   r3 = Z;
                    r2 = Y;   r4 = X;
                    Z = 1;    r5 = Y;
```

Figure 3. How does the violation of write atomicity effect this program?

```
Thread3 Exec    Thread4 Exec
X = 1;          Z = 1;
Y = 1;          r3 = Z;
r1 = X;         r4 = X;
r2 = Y;         r5 = Y;
Z = 1;          X = 1;
                Y = 1;
```

6. Conclusion and Future Work

The implications of weakening consistency guarantees offered by memory systems over shared-memory concurrent programming are hard to understand. In this paper, we showed that non-deterministic programming and angelic execution can be used to automate complex reasoning about correctness of shared-memory programs under weak memory models. Making use of the state-of-the art in solver-aided programming we built a tool called WMMSIM that materializes this approach. Currently, WMMSIM is only a protototype implementation and we haven't tested it yet on real-world benchmarks (such as Treiber stack and Chase-lev queue). As a future work, we intend to develop WMMSIM into an end-to-end reasoning tool for real-world shared-memory concurrent programs.

References

S. V. Adve and H.-J. Boehm. Memory models: A case for rethinking parallel languages and hardware. *Commun. ACM*. URL http://doi.acm.org/10.1145/1787234.1787255.

S. V. Adve and K. Gharachorloo. Shared memory consistency models : A tutorial. *H.P Tech Report*.

L. Lamport. A new approach to proving the correctness of multiprocess programs. *ACM Trans. Program. Lang. Syst.*, 1979. URL http://doi.acm.org/10.1145/357062.357068.

E. Torlak and R. Bodik. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013. URL http://doi.acm.org/10.1145/2509578.2509586.

E. Torlak and R. Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, 2014. URL http://doi.acm.org/10.1145/2594291.2594340.