WMMSIM: A Weak-Memory Model Simulator

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 [**?**]. 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 [**?**]). 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 [**?**]. Instead, existing hardware and programming models provide weaker consistency guarantees by default [**?**], 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 [**?**]. In this paper, we describe a solver-aided DSL [**?**] 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 programming implications of weak consistency guarantees. Section 3 gives a brief introduction to Rosette [**?**] and the paradigm of solver-aided programming pioneered by Rosette. Section 4 describes the design of WMMSIM and demonstrates its utility with help of examples. Section 5 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 returns the value of the last write to the same memory location. Consequently, memory models are simple in 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`.

Initially `x = Y = 0`.

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

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