Concolic Testing of Concurrent Software in the Context of Weak Memory Models

Martin Dobiasch, 0828302

March 25, 2014

1 Motivation & Problem Statement

Software is becoming more and more ubiquitous in our every day life and moreover, the complexity of software is increasing. This poses a major chal-lenge for the reliability of software. To mitigate this problem, research on software testing and verication tries to create automated solutions which exhaustively search for software bugs. Although there has been progress in developing automated testing tools for sequential testing [5], testing concur-rent software still remains as a challenge. Concurrency adds an exponen-tial amount of possibilities for tests by adding the scheduling of statements from dierent threads as an additional factor, i.e., it is also necessary to investigate in which order program statements are executed across dierent threads.

Weak Memory Models. Most of the test and verication tools for concurrent software rely on the assumption of sequential consistency. This means that the execution of a concurrent program is the same as if the statements from dierent threads had been brought to a sequential order and executed sequentially while the thread-local order remains unchanged for all threads. However, unlike sequential consistency, so-called Weak Mem-ory Models (WMMs) allow certain deviations from a thread-local program order. For example, modern processors are allowed to commit the eect of a write operation later and not immediately after the write operation was ex-ecuted. Commit in this context refers to the fact that the write to a shared variable might be stored in a local buer before it is stored in global mem-ory. Thus, the executing thread can see the new value of this variable, while other threads might still see the old value. Only when the write is com-mitted then all threads will see the new value. Another interesting aspect

1

of modern CPUs is that they have multiple pipelines for commands to be executed. This stems from the fact that they have dierent processing units (e.g. for integer operations, for oating point operations, ...). Whenever an instruction is being processed but has not been nished yet, it is referred to as being in an in-ight state. As a result of WMMs, an instruction can remain in an in-ight state while other subsequent instructions have been committed already. By doing so the processor can save time since it does not have to wait for an instruction - like a write - to nish until the next in-struction can be executed. This can result in a behaviour that for thread A it seams that thread B does not respect the thread-local order. One aspect that increases the diculty of understanding these models of computation is that they are usually not published in a comprehensible manner but instead hidden in long technical documentations of a specic processor [6]. One major problem during testing is that testing a software on a machine with a WMM can lead to unexpected program results which are dicult to explain and are hard to reproduce since WMMs allow WMM-specic eects, but do not require the computation to perform them again when running the program again. In summary, WMMs can make identifying faults via manual code inspection extremely hard if not impossible.

Figure 1 illustrates an example with two threads both executing two writes on two shared variables x and y and on two local variables r1 and r2. Moreover, it denes a state which should usually be infeasible when con-sidering sequential consistency (denoted as ’forbidden’). In the initial state both x and y are initialised with the value 0. Under sequential consistency the forbidden state cannot be reached. However, this state is feasible for ARM and POWER architectures [6]. One possibility to reach this state is as follows: First Thread 0 starts executing the write x = 1 but does not commit it. Then the second write y = 1 is executed but this time the write is committed immediately. Next, Thread 1 takes over reading the value of y (1) and writing it to r1. After this, it executes the second read, thus storing the initial value of x (0) to r2. Thereby, the forbidden state is reached.

|  |  |
| --- | --- |
| Thread 0 | Thread 1 |
| x = 1 y = 1 | r1 = y r2 = x |
| Initial state: x = 0^y = 0 | |
| Forbidden: r1 = 1^1 : r2 = 0 | |

Figure 1: Message Passing Example from [6]

2

Concolic Testing. In past years various testing tools have been devel-oped featuring concolic testing [5], a testing approach that combines the concrete execution of a program with a simultaneous symbolic execution. However, only a few of these tools [9] have the ability to test concurrent software. A testing approach derived from concolic testing which is able to test concurrent software is (con)2colic testing (concrete and symbolic execution of a concurrent program) [4]. Figure 2 illustrates the high-level architecture of (con)2colic testing which can mainly be divided into two parts: an execution engine and a reasoning engine. The execution engine gathers symbolic constraints which the reasoning engine then modies to generate new execution paths. The reasoning engine explores interference scenarios by combining and modifying gathered constraint systems. The (con)2colic algorithm works in the following way: First, the program under test is executed and (symbolic) information is collected. Based on gathered information alternate program runs are then generated and tested for their realisability. However, there could be an intractable amount of such alter-nate program runs. In order to reduce the amount of tests, (con)2colic uses so-called interference scenarios: An interference scenario (IS) represents a situation where a program thread needs to read a variable which is written by another thread. For a given IS it can be determined whether a schedule for the threads in the program exists so that this scenario can be realised. Thus, the number of schedules to be tested can be reduced dramatically, while the testing routine is still complete (under some standard concolic testing assumptions). The tool CONCREST [4] implements (con)2colic test-ing as an extension of CREST1. However, CONCREST has no means to test programs with respect to the eects of weak memory models and instead relies on sequential consistency. As a consequence of this, certain aspects of concurrent behaviour remain untested. The problem to be solved in this thesis is to add support for weak memory models in (con)2colic testing.

2 Expected Results

In this thesis I will extend the execution platform of CONCREST with the capability to simulate eects of weak memory models. It will provide an interface to enable the precise control of the eects of a WMM.

Since there exist dierent WMMs, the extension has to be congurable with regard to which WMM should be used while performing program ex-ecutions. This also involves identifying important/relevant WMMs and in-

1http://code.google.com/p/crest/

3

Multi-threaded Concolic Execution

Symb.

Trace Path Exploration

Seq. Schedule & Random Inputs

Schedule & Inputs

Interference Scenarios

Interference Exploration

Yes: realizable

Realizability Check

No: not realizable

Execution Engine Reasoning Engine

Figure 2: Overview of (Con)2colic Testing [4].

vestigating their properties.

A major feature of CONCREST is that during the performed tests it is possible to use dynamic analysis tools, like Valgrind, and one goal is to preserve this feature.

3 Methodological Approach

1. Literature Review: In a rst step I will gather background information about weak memory models and (con)2colic testing by a thorough review of existing literature. During this literature review, I will identify existing WMMs and study their similarities and dierences. By doing so, I will identify central concepts of WMMs like reorderings. Later on, it has to be decided which of these models have to be supported.

2. Model: In a next step I will dene a formalism for modelling WMMs. These models should then be loadable by the testing tool. This will not only ease the development processes of the tool, but will also provide extensibility for additional WMMs in the future.

3. Implementation: I will extend CONCREST with a simulation capabil-ity which is able to inject eects of a WMM into a concolic execution. In order to enable the reasoning engine to precisely control WMM eects, I will realise a WMM-specic interface between reasoning and execution engine.

4. Evaluation: I will evaluate the implementation on a set of benchmark programs and WMMs. The benchmarks will be based on ndings during my literature review.

4

4 State of the Art

There are various other tools for software testing and verication. Like CONCREST, SAGE [5] and CUTE [9] implement the DART (Directed Auto-mated Random Testing) approach but have no support for multi-threading. While the test tool CUTE (Concolic Unit Testing and Explicit Path Model-Checking) can only test sequential C programs, jCUTE can test concurrent Java programs for data races, deadlocks, innite loops and uncaught ex-ception errors [9]. ConTest [3] takes a dierent approach: It takes a Java program and a test for the program and tries to increase the coverage of two test models. The rst model just requires a context switch for a method in the Java program, while the second model records context switches forced by a specic method in the Java program. ConTest tests a program only on a single core CPU since any concurrency issue on a single execution is also a bug for a multi-core execution. The tool instruments the program using sleep, yield and priority. During the tests the instrumentations are used to force a context switch from one thread to a dierent thread [3]. Poirot [8] takes a similar approach as ConTest and translates a concurrent program into a sequential execution. This sequential execution is forced to contain context switches which are considered to be crucial for possi-ble bugs. The sequential program is then analysed using static analysis techniques. Chess [7] uses model checking for testing concurrent programs. It tries to enumerate possible thread schedules giving priority to schedules with fewer preemptions. To incorporate Chess into an existing test the code has to be modied slightly. The framework provides wrapper functions for common concurrency API-functions such as the Win32 API such that the execution of the test can be instrumented and recorded. Fusion [11, 10] also records a concurrent trace of a program execution but then transforms this trace into a logical formula that also encodes certain properties to be checked. If the formula has a solution then a property violation has been determined or otherwise the trace does not violate the given properties. As an extension, the concept of interference abstractions, a concept similar to interference scenarios, is introduced [10].

None of the previously mentioned tools is able to test using WMMs. CBMC has an extension which makes bounded model checking with respect to a given WMM possible. One problem with this approach is that bounded model checking does not scale well for larger concurrent software [1]. An-other tool which is capable of testing software with respect to WMM prop-erties is RELAXER [2], a tool facilitating an active random testing technique and combining it with the search for data races using dierent WMMs. This

5

Tool

CBMC [1] CREST 2 CONCREST [4] SAGE [5]

Cute [9] jCute [9] CHESS [7] Poirot [8]

Fusion [11, 10] ConTest [3]

RELAXER [2]

Basic principle

bounded model checking concolic testing (con)2colic testing concolic testing

concolic testing concolic testing model checking static analysis model checking dynamic analysis

dynamic analysis

WMM support

yes no no no no no no no no no

yes

Source code available

yes no yes no yes yes yes no no no

no

Table 1: Overview of concurrency testing tools

means that the program uses an analysis to predict potential bugs. After this initial analysis the program tries to trigger these bugs by actively con-trolling the scheduler. Table 1 summarizes the above mentioned tools.

5 Relevance to the Curriculum of Computational Intelligence

This thesis ts into the Computational Intelligence module Programming Languages and Verication as it requires deeper knowledge of the semantics of programming languages (such as C and C++), software testing and software verication.

References

[1] Jade Alglave, Daniel Kroening, and Michael Tautschnig, Partial or-ders for ecient bounded model checking of concurrent software, CAV (Natasha Sharygina and Helmut Veith, eds.), Lecture Notes in Com-puter Science, vol. 8044, Springer, 2013, pp. 141{157.

[2] Jacob Burnim, Koushik Sen, and Christos Stergiou, Testing concurrent programs on relaxed memory models, ISSTA (Matthew B. Dwyer and Frank Tip, eds.), ACM, 2011, pp. 122{132.

2http://code.google.com/p/crest/

6

[3] Orit Edelstein, Eitan Farchi, Yarden Nir, Gil Ratsaby, and Shmuel Ur, Multithreaded java program test generation, IBM Systems Journal 41 (2002), no. 1, 111{125.

[4] Azadeh Farzan, Andreas Holzer, Niloofar Razavi, and Helmut Veith, Con2colic testing, ESEC/SIGSOFT FSE (Bertrand Meyer, Luciano Baresi, and Mira Mezini, eds.), ACM, 2013, pp. 37{47.

[5] Patrice Godefroid, Nils Klarlund, and Koushik Sen, Dart: directed au-tomated random testing, PLDI (Vivek Sarkar and Mary W. Hall, eds.), ACM, 2005, pp. 213{223.

[6] LucMaranget, SusmitSarkar, andPeterSewell, A Tutorial Introduction to the ARM and POWER Relaxed Memory Models.

[7] Madanlal Musuvathi, Shaz Qadeer, and Thomas Ball, CHESS: A Sys-tematic Testing Tool for Concurrent Software, 2007.

[8] Shaz Qadeer, Poirot - a concurrency sleuth, Formal Methods and Soft-ware Engineering - 13th International Conference on Formal Engineer-ing Methods, ICFEM 2011, Durham, UK, October 26-28, 2011. Pro-ceedings (Shengchao Qin and Zongyan Qiu, eds.), Lecture Notes in Computer Science, vol. 6991, Springer, 2011, p. 15.

[9] Koushik Sen, Darko Marinov, and Gul Agha, Cute: a concolic unit testing engine for c, ESEC/SIGSOFT FSE (Michel Wermelinger and Harald Gall, eds.), ACM, 2005, pp. 263{272.

[10] Nishant Sinha and Chao Wang, On interference abstractions, Proceed-ings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Prin-ciples of Programming Languages (New York, NY, USA), POPL ’11, ACM, 2011, pp. 423{434.

[11] Chao Wang, Sudipta Kundu, Rhishikesh Limaye, Malay Ganai, and Aarti Gupta, Symbolic predictive analysis for concurrent programs, For-mal Aspects of Computing 23 (2011), no. 6, 781{805 (English).

(Unterschrift Betreuer)

7