Support Development and Testing of Concurrent Software through Supervised Software Execution

Jasmin Jahić[o], Thomas Kuhn[o], Matthias Jung[o], Norbert Wehn[+]

[o]*Fraunhofer Institute for Experimental Software Engineering IESE, Fraunhofer-Platz 1, 67663 Kaiserslautern, Germany*
[+] *Microelectronic Systems Design Research Group, University of Kaiserslautern, Germany*

ABSTRACT

The migration of sequential embedded software to multicore processors is a challenging task. Parallelization of software introduces concurrency bugs (e.g., data races), which only conditionally appear during testing because they strongly depend on the timing of the execution. The migration requires design space exploration for optimal use of embedded hardware. We propose a supervised execution approach, which executes migrated software on virtual prototypes, in order to verify migrated software and correlate parallelization options with performance measurement.

KEYWORDS: Multicore; testing; design space exploration; embedded systems

1 Introduction

Over the years, manufacturers of processors have improved the performance of processors by increasing their clock frequency. Semiconductor technology has reached a point where a further increase in clock frequency is economically impractical due to high heat dissipation. The current trend is to reduce processor frequency and to prefer concurrent execution of software using multicore processors. Instead of executing software faster, multicore processors enhance performance by concurrently executing software tasks. This raises a problem for software testers and software architects.

In sequential software, it is necessary to test functional correctness. In concurrent software, it is necessary to test the functional correctness of individual software tasks as well as the interleaving of concurrent tasks. The complexity of multicore hardware makes the progress of tasks non-deterministic, therefore making their interleaving non-deterministic as well. The critical issue regarding interleaving of concurrent tasks are operations on shared memory. If operations on shared memory are not properly synchronized, the results for the shared memory may be invalid or overwritten data. Because of the non-deterministic nature of interleaving, it is hard to detect synchronization faults in software using standard testing. Failures occur only in special interleaving combinations. In sequential software, the input deterministically determines the output. In concurrent software, input data cannot deterministically trigger specific interleavings. Therefore, traditional testing is not suitable for testing concurrent software. Common approaches

used for the verification of concurrent software are static analysis and analysis of recorded execution traces.
The most common synchronization mechanisms are lock-based. However, overuse of locking mechanisms is responsible for serialization of concurrent software, which, according to Amdahl's law, has a devastating effect on performance. Non-locking synchronization uses complex data structures. In safety-critical systems, developers often use strict scheduling and synchronization points (e.g., barriers) for synchronization. As there are several synchronization approaches, architecting a concurrent system is a challenging undertaking in terms of design space options. We focus on embedded systems, where software is deployed on a unique hardware configuration, which often does not change. The challenge is what synchronization approach to use in order to optimize hardware utilization.
The goal of our work is to improve understanding of the faulty behavior of concurrent software by developing models of concurrency bugs and synchronization mechanisms. We aim to complement the existing analysis approaches with these models. We also aim to improve the precision of analyses where the main evaluation criterion is the number of false positives. Our approach obtains execution traces without changing the source code or instrumenting binaries. Such non-intrusive execution traces are necessary for precise identification of concurrency bugs. In order to give credibility to our results, we expand existing sequential coverage criteria to suit concurrent software. We provide means and tools for design space exploration in order to increase the efficiency of running concurrent code on embedded systems.

2 Related work

Static analysis (SA) approaches build a model of the target software from the source code. If a part of the software model corresponds to the model of the concurrency bug, the SA identifies the bug. The main drawback of SA is a high number of false positives, as some statements are statically undecidable (e.g., pointer arithmetic, recursive calls). It is often necessary to annotate source code in order to reduce the number of false positives to an acceptable level. A common tool for static code analysis is Astree [1]. For C programs, Astree covers all possible interleavings, uncovers all data races, and considers the software initialization and execution phases. However, Astree employs possibly imprecise abstractions of thread priorities and real-time scheduling, and assumes arbitrary preemption.
Industry prefers dynamic testing tools to static analysis and formal methods. Dynamic testing approaches for concurrent software gather and analyze execution traces. The most common algorithms for execution trace analysis are Lockset [2] and Happens-before [3]. Tools usually gather execution traces by changing the source code, by using code instrumentation, or by using compiler support.
Lockset [2], maintains for each variable the set of locks that have protected a shared variable "so far" (Candidate Set), and the set of locks at a specific access to a variable (Lock Set). At the beginning, the Candidate Set contains all locks that a thread can use. The Lockset adds or removes a lock from a Lock Set when a thread acquires or releases the lock, respectively. When the algorithm detects an access to a shared variable, it updates the Candidate Set by intersecting it with the thread's current Lock Set. If the result of the

intersection is an empty set, no common locks protect the variable and therefore this is a potential race. The idea behind Lockset is to ensure that at least one common lock protects all accesses to the same shared variable.

3 Supervised testing of concurrent software

We propose splitting concurrent software testing activities into three phases (cf. *Figure 2*). Phase I produces the execution traces by executing the runnables and analyzes the scheduling. Here, we rely on virtual platforms. The software executes on a virtual prototype of the target system, producing non-intrusive traces. Phase II receives tuples of mutually concurrent runnables and their execution traces. A set of mutually concurrent runnables is a set where every runnable is concurrent with all other runnables from that set. Phase II extracts information relevant for synchronization and identifies shared memory between concurrent runnables and the synchronization mechanisms used by the runnables. Phase III applies the algorithm for execution trace analysis (e.g., Lockset [2]) to the execution traces to identify concurrency bugs. With this division, we gradually reduce the state space in which the analysis is performed.

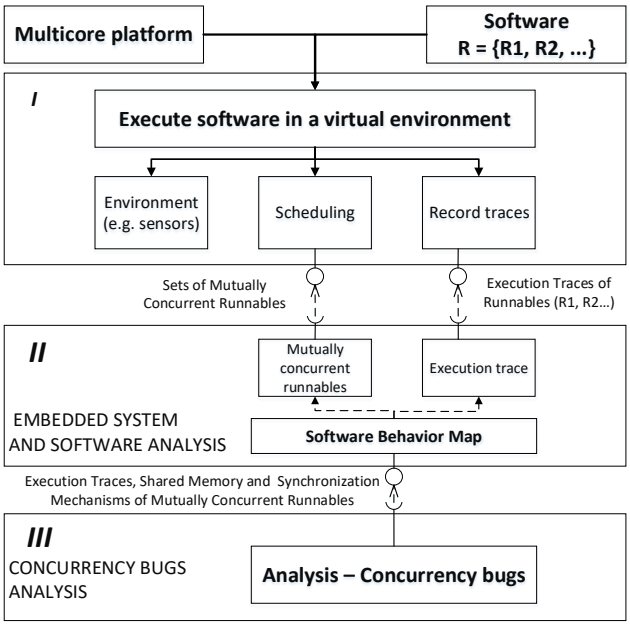

Figure 2: Detection of concurrency faults in three phases

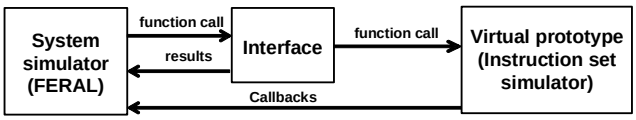

Figure 1: Supervised testing concept

We execute the software on a virtual prototype under the control of the FERAL framework (Fast Evaluation on Requirements and Architecture Level) [4] for two reasons: FERAL simulates the necessary runtime and platform components, such as CAN communication, and simulates the task scheduler that controls the execution of tasks. FERAL supports several task schedulers appropriate for creating realistic platform simulation models on the scheduling level. *Figure 1* illustrates the coupling between FERAL and the virtual prototype. The role of the virtual prototype is that of an instruction simulator. The FERAL platform simulation can trigger the execution of runnables multiple times and with an arbitrary schedule. FERAL can also mock up unimplemented functions and sensor values.
Our analysis in FERAL provides a report about the variables and memory locations accessed by the software tasks and about the access frequency. We are able to correlate scheduling with synchronization between tasks and concurrency bugs. We use virtual prototyping platforms that closely resemble real hardware. This enables us to gather low-level execution details and to provide a variety of parametrization options. Our approach enables rapid prototyping of concurrent systems with diverse optimization parameters (e.g., scheduling schemes [5]) and evaluation of their influence on software concurrency aspects.

4 Code coverage in concurrent software

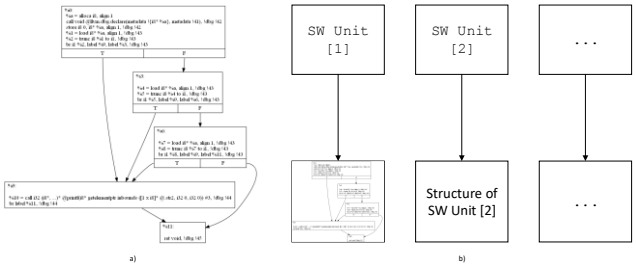

Figure 3: (a) Function structure, (b) SW composed from distinct functions

Concurrent software consists of tasks, composed of software functions. Software conditions (e.g., if-statements) increase the number of potential execution paths through functions. An access to a shared variable is part of an execution path. In the case of locks, synchronization mechanisms invoked on the execution path define the context of the access to the shared variable. The complete execution trace of a software function contains access to every variable, in every possible context [6]. In the case of non-locking synchronization, the access pattern to the shared structure is a synchronization mechanism. In order to properly reason about coverage, it is necessary to observe all possible memory patterns. Finally, it is necessary to understand the platform-executing capabilities in order to reason about synchronization that relies on static scheduling. Common cases of concurrency bugs include situations where developers forget to synchronize an access to a shared variable for some execution paths, or where developers, by mistake, use different locks to synchronize access to the same shared variable. Therefore, the context of access to shared resources is the key in exposing concurrency bugs. Our reasoning is that an analysis is capable of and responsible for finding all concurrency bugs if the execution traces contain all contexts under which concurrent tasks access shared memory. This reasoning reduces the problem of concurrent software coverage to the sequential coverage of software functions.

References

[1] Antoine Miné et al., “Taking Static Analysis to the Next Level: Proving the Absence of Run-Time Errors and Data Races with Astrée”, In Embedded Real Time Software and Systems - ERTSS 2016

[2] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, Thomas Anderson, “Eraser: a dynamic data race detector for multithreaded programs”, ACM Transactions on Computer Systems (TOCS), v. 15 no. 4, pp. 391-411, Nov. 1997

[3] R.H.B. Netzer and B.P. Miller, “Improving the Accuracy of Data Race Detection,” Proc. 3rd ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPOPP 91), ACM Press, pp. 133-144, 1991

[4] T. Kuhn et al., “FERAL - Framework for Simulator Coupling on Requirements and Architecture Level”, In Eleventh ACM-IEEE International Conference on Formal Methods and Models for Codesign, October 2013.

[5] J. Jahić et al., “Supervised Testing of Concurrent Software in Embedded Systems”, International Conference on Embedded Computer Systems Architectures Modeling and Simulation (SAMOS), July 2017

[6] J. Jahić et al., "Automatic Test Coverage Measurements to support Design Space Exploration", IFIP First International Workshop on Design Space Exploration of Cyber-Physical Systems (IDEAL), February 2014