Supervised Testing of Concurrent Software in Embedded Systems

Jasmin Jahić, Thomas Kuhn, Matthias Jung
Embedded Software Engineering
Fraunhofer ISE
Kaiserslautern, Germany
firstname.lastname@fraunhofer.iese.de

Norbert Wehn
Microelectronic Systems Design Research Group
University of Kaiserslautern, Germany
wehn@eit.uni-kl.de

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. Therefore, traditional testing approaches cannot efficiently test concurrent software. More appropriate are analysis approaches that prove the absence of software faults. Current approaches often produce false positives as they fail to consider all relevant synchronization sources. In this paper, we complement current analysis techniques by considering a scheduling scheme as a synchronization mechanism. We narrow the analysis by analyzing only relevant variants in execution timing that might produce concurrency bugs. Therefore, we eliminate a family of false positives caused by ignoring the scheduling synchronization. Engineers can optimize this scheduling scheme to satisfy different requirements. Our approach uses virtual prototyping to enable design space exploration of systems with complex scheduling schemes by investigating the influence of the scheduling scheme on the synchronization of concurrent software.

Keywords—Data Races; Testing; Concurrency; Scheduling; LLVM; FERAL; Virtual Prototyping

I. INTRODUCTION

In concurrent software, concurrent threads operate on shared memory. If operations on shared memory are not synchronized properly, shared memory may contain invalid data. In order to synchronize concurrent threads, developers use synchronization mechanisms. Therefore, testing of concurrent software includes testing of functional correctness of software, and testing of concurrent operations on shared memory. For comparison, when testing sequential algorithms, the result of a test case will only depend on the inputs and the algorithm under test. When triggered with the right input data, a faulty algorithm will always yield a failure. Concurrency bugs originate due to synchronization faults between concurrent threads, and inputs do not determine them. Only a specific interleaving of operations on shared memory yield visible failures. Testing concurrent software is a hard task, because progress of threads determines interleave-ings of memory operations. Progress of a single thread depends on numerous factors, including hard-to-predict aspects such as cache misses [1]. When several threads execute concurrently, it is even harder to predict execution progress and consequently interleave-ings of concurrent threads. By excluding specialized solutions, it is fair to say that the scheduling of thread interleave-ings on the majority of computing platforms is non-deterministic. Because there are too many interleave-ings for concurrent software, it is impossible to test them all. Therefore, traditional testing is not suitable for concurrent software.

One way to test concurrent software is to complement traditional testing with analysis. During software execution, it is necessary to collect an execution trace, and to analyze if threads synchronize their access to shared memory. An execution trace is a description of the dynamic behavior of software, including information about access to memory locations and function calls. However, developers can use different synchronization mechanisms. The most common synchronization mechanisms are based on locks (i.e., while one thread operates on a shared resource, other threads that currently require the same resource wait). Locks activate via function calls. A well known algorithm for execution trace analysis, 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). The Candidate Set, at the beginning, 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 Candidate Set with the current Lock Set of the thread. If the result of the intersection is an empty set, no common locks protect the variable and therefore it 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. It is possible to synchronize threads only with locks. However, the overuse of locking mechanisms has a devastating effect on performance, as it leads to serialization. In order to avoid serialization, developers can use other synchronization mechanisms. In systems that allow customizable scheduling schemes, developers can pin certain threads to specific cores, configure priorities and create a strict execution order of threads. In this way, it is possible to guarantee that certain threads will never execute concurrently, and avoid the need for locks. Concurrency means that two or more calculations happen within the same time frame, with a dependency between them. Parallelism means that two or more calculations happen simultaneously. In complex systems with many threads, it is beneficial to avoid parallel and concurrent execution of threads that frequently access to common shared data, by optimizing scheduling. Such optimization enables to avoid locks, and
consequently serialization, while retaining proper synchronization. However, Lockset algorithm ignores the notion of concurrency and does not consider scheduling as synchronization mechanism. If Lockset detects threads that access to shared memory without common lock, it will report concurrency bugs, even in cases when those threads can never execute concurrently. Hence, in systems that rely on scheduling as synchronization, Lockset will produce a large number of false positives.

Our work targets embedded systems that provide the possibility for defining a custom scheduling scheme. To the best of our knowledge, there does not exist an approach that relates scheduling, synchronization and concurrency bugs. In automotive systems, software decomposes to runnables that are subject to scheduling. In order to comply with embedded terminology, we will refer to threads as software runnables in the remainder of this text. Runnables are scheduled by the AUTOSAR [19] operating system with fixed priorities. Once started, runnables run to completion and can only be preempted by higher priority runnables. Runnables are additionally allowed to wait upon events and can pass the thread of control to runnables with lower priorities. We also consider Linux-based embedded systems, as Linux provides functions for pinning runnables to cores and assigning priority to runnables [20]. In this paper, we present our supervised testing approach, which complements existing dynamic analysis approaches, based on the Lockset algorithm [2] with scheduling synchronization. We execute software in a virtual environment to collect traces and analyze the scheduling scheme (Fig. 3). Our approach identifies sets of runnables that can never be concurrent because of the scheduling scheme. Instead of applying the Lockset algorithm on all execution traces, we exclude non-concurrent runnables from the analysis. Our paper presents three contributions: 1) an approach for inferring non-concurrent runnables from source code by executing runnables on virtual prototypes; 2) an algorithm for complementing Lockset with scheduling synchronization; and, as a consequence, 3) the elimination of one family of false positives. Together, our contributions provide a tool for exploring the design space in terms of runnable scheduling, which assist engineers in evaluating the influence of the scheduling scheme on synchronization. Section II discusses related work. Section III describes our overall approach and notion of mutual concurrency, while Section IV describes supervised testing, including platform scheduling in the analysis. Section V evaluates our approach and Section VI concludes this paper.

II. RELATED WORK

A. Static analysis

Static Analysis (SA) approaches build a model of the target software from the source code (e.g. by using abstract interpretation of the code [3]). If a part of the software model corresponds to the model of the concurrency bug, the SA analysis identifies the bug. SA is capable of exposing all bugs in a piece of software. Over time, many different SA approaches have emerged [5]. 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. Additionally, checking a large piece of software may lead to a state space explosion (a common challenge for model checkers [7]), which forces static analysis to terminate and to potentially produce more false positives. Some approaches tried to tune SA for a specific purpose, but even so, SA still produces a significant percentage of false positives [9]. Common tools for static code analysis are Astree [4] and Polyspace [6]. Polyspace can detect shared variables and take task interleavings into account, but reports neither data races nor lock/unlock faults. 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.

B. Testing and analysis of execution traces

A survey from 2014 in the automotive industry shows that the participants preferred dynamic testing tools to static analysis and formal methods [8]. Dynamic testing approaches for concurrent software gather and analyze execution traces. The most common algorithms for execution trace analysis are Happens-before [10], IFRit [15] algorithm monitors interference-free regions surrounding a shared variable. IFRit performs identification of interference-free regions through static analysis and does not consider properties of the target platform. ThreadSanitizer [16] uses LLVM for compile time instrumentation in order to reduce the slowdown of the target software. The authors of ThreadSanitizer increase the performance by changing the memory access sampling rate, but do not provide an analysis of sampling vs. accuracy. Some approaches focus on the anticipation of bugs and on stalling problematic threads before they make irreversible changes. The Anticipating Invariant [14] technique successfully tolerates concurrency bugs related to atomicity and order violation in some cases.

To the best of our knowledge, the previously presented analysis algorithms do not consider platform scheduling synchronization, and no other approach is offering tools to engineers for analyzing the scheduling schemes of complex systems in order to reduce the number of used locks. Neglecting this type of synchronization leads to false positives – the analysis may claim there is a bug in correct code. The Astree tool partially tackles this challenge [4], but with possibly imprecise abstractions of thread priorities and real-time scheduling.

III. CONCURRENCY BUG DETECTION

In order to detect concurrency bugs, in the ideal case, it is necessary to collect, resp. identify, the following data about software: execution trace, runnables that execute concurrently, and synchronization mechanisms between runnables. Finally, it is necessary to perform an analysis on the collected data in order to identify fail-prone behavior.
Reasoning about synchronization and concurrency between runnables is a challenging task. The scheduling of software runnables can be defined statically, completely dynamically, or as a combination in which some rules are imposed a priori (e.g., higher-priority tasks can interrupt lower-priority tasks) while others are dynamic (e.g., runnables can reallocate to different cores). Let us observe an example in Fig. 1. Software runnables (R1-R5) access shared memory locations (A-E). With a scheduling scheme, it is possible to define the relative execution order of runnables, their priorities, and the duration of the execution time slots. However, due to the non-determinism of multicores, it is hard to determine the time span during which a runnable accesses a shared variable. Under the assumption that this scheme is guaranteed by the scheduling properties (core affinity, strict scheduling), it can be used for synchronization.

Embedded systems may use numerous strategies for runnable scheduling. One of the most common scheduling strategies is OSEK conforming scheduling [19] (AUTOSAR is OSEK based OS). It is possible to implement OSEK scheduling as a preemptive or nonpreemptive strategy. OSEK scheduling supports time- and event-triggered runnables with defined priorities. Time-triggered runnables activate at specific times. Preemptive schedulers preempt running runnables if the newly ready runnable has a higher priority than the currently executed runnable. Nonpreemptive schedulers wait until the running runnable releases the CPU. In addition to time-triggered runnables, it is possible to use event-triggered runnables as well. The scheduler activates them when a specific event happens. This may be an interrupt or a signal from another runnable.

A. Mutually concurrent runnables

Fig. 2. a) R2 and R3 are concurrent. b) R2 and R1 do not execute at the same time, but are technically concurrent.

In this paper, we are limiting our study to data races. For a data race to occur, there must exist at least two concurrent runnables accessing the same memory location, and at least one of the accesses must be for writing. Each runnable requires a time slot for its execution. Runnables are concurrent if the order of their execution time slots is not sequential. It is not necessary for two runnables to execute in parallel in order to create a data race. It is enough that due to interrupts and other scheduling effects, a second runnable starts execution while the first has not yet completed its execution. In complex systems, it is challenging to determine concurrent runnables manually. Fig. 2 illustrates this explanation and shows an example set of runnables R1 – R5, with a fixed scheduling scheme. Every runnable executes only once within one execution cycle. In Fig. 2.a, R2 and R3 are concurrent. In Fig. 2.b, R2 and R3 do not execute at the same time due to an interrupt in the form of task R4. The interrupt runnable R4 and the runnable R3 are obviously concurrent. Runnables R3 and R4 start while R2 is not complete yet. R2 and R4 can write to memory shared with R2. If these runnables do not properly synchronize the access to the shared memory, R2 can theoretically operate on outdated data. This clearly demonstrates the need to analyze memory operations of mutually concurrent runnables during their entire execution span, and not only at the times when these runnables overlap during one specific test case. A set of mutually concurrent runnables is a set where every runnable is concurrent with all other runnables from that set. Hence, R2, R3, and R4 are mutually concurrent. Lockset only needs to analyze their execution traces against each other.

B. Testing concurrent software in three phases

We propose splitting concurrent software testing activities into three phases (cf. Fig. 3). Phase I produces the execution traces by executing the runnables and analyzes the scheduling. The input for Phase II is generated tuples of mutually concurrent runnables and their execution traces. Sets of runnables that are mutually concurrent are passed to Phase II. Every Ri(ExecutionTraces) contains a set of execution traces gathered by executing the runnable Ri, and i = 0, ..., n, where n is the number of runnables. Phase II extracts information relevant for synchronization and identifies shared memory between concurrent runnables and the synchronization mechanisms used by runnables. Phase III applies the Lockset [2] algorithm to the execution traces to identify concurrency bugs. With this division, we gradually reduce the state space on which Lockset works.

Fig. 3. Detection of concurrency faults in three phases
C. Scheduling: a synchronization mechanism

We define a finite set of runnables \( R \) (1) and a finite set of Mutually Concurrent runnables \( MC \) (2) containing tuples; tuple elements represent mutually concurrent runnables according to the scheduling scheme. Runnables are mutually concurrent and belong to the same tuple, if and only if each runnable from the tuple is concurrent with all other runnables from the same tuple. Two runnables are concurrent if their execution time span overlaps for at least one instruction (Section III, Fig. 2). If we do not consider scheduling or core affinity as synchronization mechanisms, there are no concurrency limitations and all runnables from \( R \) are mutually concurrent (3).

\[
R = \{R_1, R_2, ..., R_n\}; \quad \text{(where } n \text{ is the number of runnables)}
\]

\[
MC = \{MC_1, MC_2, ..., MC_h\}; \text{ and } MC_h \subseteq R \quad \text{(2)}
\]

\[
MC_1 = R \text{ and } MC = \{MC_1\} \quad \text{(3)}
\]

In order to evaluate the influence of a specific scheduling on concurrency, it is necessary to analyze each scheduling property. It is possible to represent each Scheduling Property (SP) with a set of rules with which the property influences the execution of runnables. The analysis component of each SP applies its set of rules to the set of all runnables \( R \) and produces the set \( SP_m \), which contains tuples of mutually concurrent runnables that are mutually concurrent according to the rules of the \( m \)th scheduling property. It is necessary to apply the rules of every scheduling property to the set of runnables \( R \). The result of this analysis is the set of Concurrency Limitations \( CL = \{SP_1, SP_2, ..., SP_i\} \) where elements of \( SP_i(R) \) are tuples of runnables that are mutually concurrent considering the \( i \)th scheduling property. Runnables that are not part of any tuple in \( SP_i(R) \) cannot be concurrent according to the \( i \)th scheduling property. To calculate the final sets of mutually concurrent runnables in the system, it is necessary to intersect all \( SP_i \). By intersecting all \( SP_i \), we produce sets of runnables that are mutually concurrent according to all considered scheduling properties.

\[
R = \{R_1, R_2, R_3, R_4, R_5, R_6, R_7, R_8, R_9\}
\]

\[
CL = \{SP_1, SP_2\}
\]

\[
SP_1(R) = \{[R_2, R_6, R_7], [R_3, R_8, R_9]\}
\]

\[
MC= \{[R_3, R_d], [R_3, R_7], [R_3, R_9]\}
\]

IV. SUPERVISED TESTING

In our approach, we collect execution traces by sequentially executing unmodified runnables in a common memory space. We achieve this by using the LLVM compiler infrastructure [12]. The LLVM front end translates source code into a byte code called Intermediate Representation (IR). We use the LLVM interpreter to execute the IR. We have modified the LLVM interpreter to observe and record the internal state of software runnables – and relate executed instruction to the source code. For the concurrency analysis, what is important are memory manipulation instructions and function calls. The memory manipulation, store, and load instructions mark accesses to potentially shared memory. Locking mechanisms operate using function calls. Common functions for scheduling, affinity, and priority manipulation use built-in (Linux) system functions (e.g. sched_setscheduler()).

A. Building an execution trace

We execute software runnables under the control of the FERAL framework (Fast Evaluation on Requirements and Architecture Level) [17] for two reasons: FERAL simulates necessary runtime and platform components like CAN communication, and simulates the task scheduler that controls the execution of runnables. FERAL supports several task schedulers appropriate for creating realistic platform simulation models on the scheduling level. Fig. 5 illustrates the coupling between FERAL and LLVM. FERAL loads the LLVM
Intermediate representation, creates communication ports, and
controls the progress of the execution. Communication ports is a
term used to describe a mechanism we created to execute
software on a virtual prototype. The idea behind these ports is to
intercept unimplemented or system functions and provide
arbitrary results. We are also able to intercept access to global
variables holding values of sensor components, and change their
value according to specific needs. Before the execution, it is
enough to specify the names of the global variables or functions
we want to intercept. These functions and variables become
ports. When the LLVM interpreter accesses a port, the port
callback function activates and contacts FERAL. In FERAL, we
start the desired procedure to return an arbitrary value to LLVM,
or to perform any other operation. This process enables us to
execute software, including operating system functions, on a
virtual prototype without a full system stack. The LLVM
backend communicates with FERAL via the callback functions
in order to report to FERAL all functions and their blocks from
IR to FERAL as well as details of the executed instructions
(instruction type, memory address accessed, additional
parameters such as variable name, and the line number of an
instruction in the source code). FERAL executes the runnables
to observe the software’s dynamic behavior. All runnables
deployed to one memory domain (i.e., one LLVM instance) share
a common memory space. We identify access to shared variables
by analyzing the access to the memory addresses. 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 values of shared
variables through the previously mentioned port mechanism.
The results of executions are execution traces, which contain
consecutive sets of instruction details, organized into execution
cycles.

![Fig. 5: Supervised testing concept](image)

**B. Inferring mutually concurrent runnables**

It is possible to implement the scheduling scheme of
embedded software in various ways. We focus on two
implementation types. The first case is that of POSIX-based
systems. Linux provides functions for assigning priorities,
scheduling policies, and pinning runnables to cores. Typically,
one runnable is responsible for creating other software
runnables, assigning affinities and priorities, and defining an
overall scheduling scheme. In other implementations, each
runnable contains, at the beginning of its execution, a part of the
code for self-assigning a priority in the scheduling scheme. The
scheduling scheme remains static after the initiation of all
software runnables. Linux provides a wide range of system
functions for controlling scheduling (e.g., sched()), affinity (e.g.,
sched_setaffinity()), and various real-time scheduling policies,
for special time-critical applications that need precise control
over the runnables (e.g., FIFO, Round-Robin) [20]. With these
functions, it is possible to define a precise scheduling scheme
and even to redirect IRQ to specific cores. As we will explain in
section IV.A, with our approach, we are able to intercept any
function call and handle it arbitrarily – to decide to execute it or
to simply skip its execution, providing the desired return result.
The execution traces that our approach collects contain function
calls. For desired functions, we are also able to extract function
parameters. Hence, with such rich execution traces, we are able
to relate scheduling system functions and the respective
runnables in order to extract the scheduling scheme that the
developers implemented. It is only necessary to execute all
software runnables once. The assumption that the scheduling
scheme is static and determined at the beginning of the execution
guarantees that, once identified, the scheduling scheme will not
change. In the second case, we consider AUTOSAR
(AUTomotive Open Systems ARchitecture). The entire
AUTOSAR configuration is static and contains runnable
attributes (priorities, triggers for event-triggered runnables,
periods for time-triggered tasks, etc.). For each runnable,
AUTOSAR generates a deployment configuration OIL (OSEK
Implementation Language) file. We implement a parser for OIL
files and reconstruct the scheduling scheme of runnables. For
design space exploration, we leave an option in our approach to
specify the scheduling scheme manually.

**C. Identification of unnecessary locks**

Locking mechanisms are computationally expensive
operations and have a negative effect on parallelism. Besides
standard locking errors, we are able to detect unnecessary use of
locks. This is an important hint to developers in terms of
software maintainability. It is a common case that due to some
changes, a previously shared variable becomes accessible by
only a single runnable, or a group of non-concurrent runnables.
Developers might forget to remove synchronization at some
point. Our approach can detect such cases.

**V. EVALUATION OF THE APPROACH**

In order to evaluate our approach, we used an industry-like
e xample of a Cruise Control software [18]. The Cruise Control
software consists of functions that communicate over shared
data structures. We parallelized functions of this software into
individual runnables and introduced additional shared variables
(see Fig. 6). In R_2 and R_7, we synchronize, with locks, the accesses
to _C_SYSTEM_GLOBAL. In R_3 and R_9, we synchronize, with
locks, the accesses to _D_SYSTEM_GLOBAL. We introduce
data races in R_6, with some partially synchronized accesses to
_B_SYSTEM_GLOBAL, and to R_2, with some partially
synchronized accesses to _D_SYSTEM_GLOBAL and
_E_SYSTEM_GLOBAL. Other accesses to introduced variables
are unsynchronized. We executed parallelized software on our
LLVM and FERAL infrastructure, and performed an analysis on
the execution trace. Our analysis provided a report on which variables and memory locations our runnables are accessing and with which frequency. Based on the analysis report, we designed a static scheduling scheme to separate the runnables with the highest frequency accesses to the shared variables (Fig. 6). We repeated the analysis of the execution, considering the scheduling scheme as a synchronization mechanism. When considering scheduling, our analysis identified a lower number of accesses to shared variables and data races (Table 3). This is due to the fact that the second analysis considered only the execution traces of mutually concurrent runnables. If the analysis does not consider synchronization in software that relies on scheduling as a synchronization mechanism, the outcome of the analysis will contain a large number of false positives.

| Table 3: The number of accesses to shared variables and data races. Inclusion of scheduling in concurrency bugs analysis eliminates one source of false positives. |

<table>
<thead>
<tr>
<th>Analysis</th>
<th>Accesses to shared variables</th>
<th>Data races</th>
</tr>
</thead>
<tbody>
<tr>
<td>Without scheduling</td>
<td>168</td>
<td>136</td>
</tr>
<tr>
<td>With scheduling</td>
<td>8</td>
<td>6</td>
</tr>
</tbody>
</table>

VI. CONCLUSION, DISCUSSION AND FUTURE WORK

Our approach detects shared variables between runnables, data races, and frequency with which runnables are accessing to shared memory. These data is useful for design space exploration in terms of organizing a scheduling scheme to improve the efficiency of concurrent software. With the experiment setup and the results, we demonstrate how to relate scheduling with synchronization between runnables and concurrency bugs. This enables rapid prototyping of scheduling schemes and evaluation of their influence on software concurrency aspects. We are also able to detect cases of unnecessary use of locks (e.g., locking runnables do not execute concurrently). We have implemented code coverage analysis alongside our testing approach to quantify the percentage of tested code and to generate test cases. These results will be the subject of future publications.

Our approach is applicable only in systems that rely on a customized scheduling scheme. The assumption is that system engineers already guarantee the timing properties. In the future, we plan to expand our approach and include other types of concurrency bugs and synchronization mechanisms. We will evaluate our approach on real-world software and compare it with existing tools in order to reason about its efficiency and precision.

ACKNOWLEDGMENTS

This work was supported by the Fraunhofer High-Performance Center Simulation- and Software-based Innovation in Kaiserslautern, Germany. We thank Sonnhild Namlingha from Fraunhofer IESE for reviewing a first version of this article.

REFERENCES