Supervised Testing of Concurrent Software in Embedded Systems

Jasmin Jahić, Thomas Kuhn, Matthias Jung
Embedded Software Engineering
Fraunhofer IESE
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 interleavings 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 interleavings of concurrent threads. By excluding specialized solutions, it is fair to say that the scheduling of thread interleavings on the majority of computing platforms is non-deterministic. Because there are too many interleaving alternatives 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

978-1-5386-3437-0/17/$31.00 ©2017 IEEE

consequentially 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 *Lockset* [2] and *Happens-before* [10]. Tools usually gather execution traces by changing the source code, by using code instrumentation [11], or by using compiler support (e.g., *LLVM* [12], [21]). The *Portend* tool [13] classifies data races and focuses on the identification of harmful races, but is ignorant of features that may make code correctly synchronized on a specific platform. The *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 (R_1-R_5) 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.

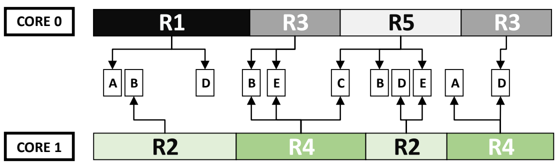

Fig. 1. Complex synchronization scheme, runnables (R1-R5) accessing shared variables (A, B, C, D, E)

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

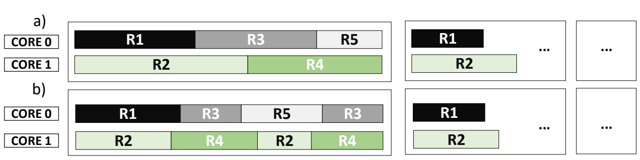

Fig. 2. a) R_2 and R_3 are concurrent. b) R_2 and R_3 do not execute at the same time, but are technically concurrent.

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 R_1 – R_5, with a fixed scheduling scheme. Every runnable executes only once within one execution cycle. In Fig. 2.a, R_2 and R_3 are concurrent. In Fig. 2.b, R_2 and R_3 do not execute at the same time due to an interrupt in the form of task R_4. The interrupt runnable R_4 and the runnable R_3 are obviously concurrent. Runnables R_3 and R_4 start while R_2 is not complete yet. R_3 and R_4 can write to memory shared with R_2. If these runnables do not properly synchronize the access to the shared memory, R_2 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, R_2, R_3, and R_4 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 are generated tuples of mutually concurrent runnables and their execution traces. Sets of runnables that are mutually concurrent are passed to Phase II. Every R_i*(ExecutionTraces)* contains a set of execution traces gathered by executing the runnable R_i, 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.

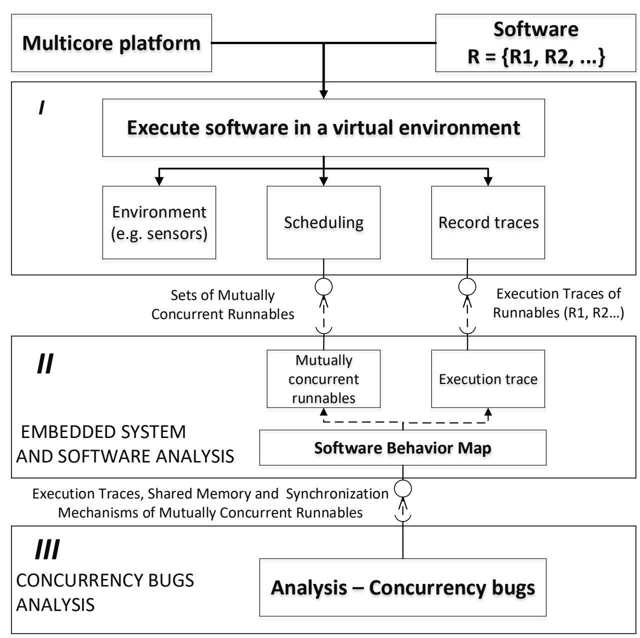

Fig. 3. Detection of concurrency faults in three phases

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).

$R = \{R_1, R_2, \ldots, R_n\}$; (1)
(where n is the number of runnables)
$MC = \{MC_1, MC_2, \ldots., MC_r\}$; and $MC_h \subseteq R$ (2)
(for each h, where h=1, …r)
$MC_1 = R$ and $MC = \{MC_1\}$ (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_g, which contains tuples of mutually concurrent runnables that are mutually concurrent according to the rules of the g^{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* $CL = \{SP_1(R), SP_2(R), \ldots SP_s(R)\}$ 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.

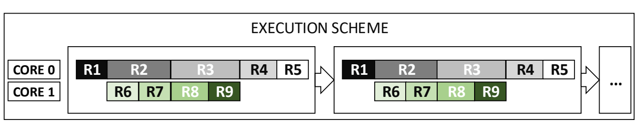

Fig. 4. Execution cycles, runnables on two cores

TABLE 1: EXAMPLE: ASSIGN RUNNABLES EXCLUSIVELY TO CORES

Core	I	II
Runnables	R_1, R_2, R_3 R_4, R_5	R_6, R_7, R_8, R_9

TABLE 2: EXAMPLE: RUNNABLES AND THEIR PRIORITIES.

Priority	I	II	III	IV	V
Runnable	R_1	R_2, R_6, R_7	R_3, R_8, R_9	R_4	R_5

We illustrate the scheduling analysis with an example. An arbitrary system with static scheduling (Fig. 4) has an initial set of runnables (4). The system assigns priorities to runnables. Preemption is disabled and every runnable has a dedicated time slot. The assumption is that runnables never exceed their time slots, and they are pinned to cores. Due to these properties, the scheduling scheme remains fixed in every execution cycle (Fig. 4). Scheduling properties – static priority policy with disabled preemption (SP_1), and core affinity with disabled preemption (SP_2) – limit concurrency (5) (6). According to the scheduling properties, software runnables are pinned to cores (Table 1) and have priorities (Table 2). The scheduling analysis takes runnable priorities as rules of concurrency of the first scheduling property SP_1 and applies them to the set of runnables R. The result are tuples of mutually concurrent runnables (7). A simple formulation summarizes the logic behind the second scheduling property (SP_2). Any element r_i of a set of runnables from R pinned to a specific core can never be concurrent with any other runnable r_k from the same set (where k is the dimension of the set R, and $i \neq k$), under the assumption that preemption is disabled. Only runnables pinned to different cores can be concurrent (under the "no preemption" assumption). Mathematically, we express this formulation as a Cartesian product between runnables pinned to core 1 and core 2. Each runnable fixed to the first core is concurrent with every runnable on the second core (8). In order to identify mutually concurrent runnables (threads) according to both scheduling properties (5), we correlate elements of Concurrency Limitations (CL). The result are tuples that contain Mutually Concurrent runnables (MC) (9).

$R = \{R_1, R_2, R_3, R_4, R_5, R_6, R_7, R_8, R_9\}$ (4)
$CL = \{SP_1, SP_2\}$ (5)
$CL = \{static\ priority, core\ affinity\}$ (6)
$SP_1(R) = \{[R_2, R_6, R_7], [R_3, R_8, R_9]\}$ (7)
$SP_2(R) = \{[R_1, R_6], [R_1, R_7], [R_1, R_8], [R_1, R_9], [R_2, R_6], [R_2, R_7], [R_2, R_8], [R_2, R_9], [R_3, R_6], [R_3, R_7], [R_3, R_8], [R_3, R_9], [R_4, R_6], [R_4, R_7], [R_4, R_8], [R_4, R_9], [R_5, R_6], [R_5, R_7], [R_5, R_8], [R_5, R_9]\}$ (8)
$MC= \{[R_2, R_6], [R_2, R_7], [R_3, R_8], [R_3, R_9]\}$ (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.

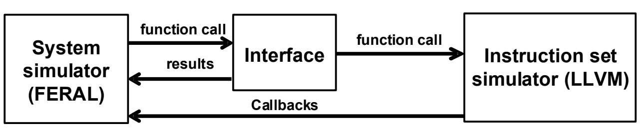

Fig. 5: Supervised testing concept

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

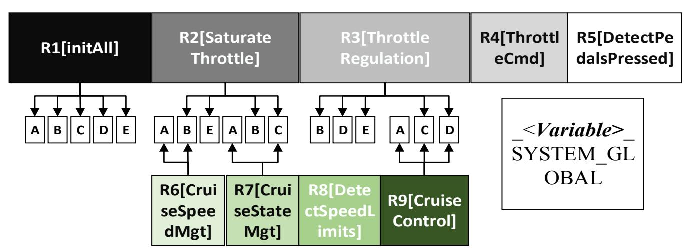

Fig. 6: Access to shared *_SYSTEM_GLOBAL(_A_, _B_, _C_, _D_, _E_)*

In order to evaluate our approach, we used an industry-like example 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 (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 R6, with some partially synchronized accesses to *_B_SYSTEM_GLOBAL*, and to R2, 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.

Analysis	Accesses to shared variables	Data races
Without scheduling	168	136
With scheduling	8	6

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 Namingha from Fraunhofer IESE for reviewing a first version of this article.

REFERENCES

[1] Bruce Jacob et ai., “Memory Systems Cache, DRAM, Disk”, ISBN: 978-0-12-379751-3

[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 n.4, p.391-411, Nov. 1997

[3] Patrick Cousot, Radhia Cousot, “Abstract interpretation: past, present and future”, Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), July 14-18, 2014

[4] 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

[5] M. C. Rinard, “Analysis of multithreaded programs”, In Proc. of the 8th Int. Symp. on Static Analysis (SAS’01), volume 2126 of LNCS, pages 1–19. Springer, Jul 2001

[6] A. Deutsch, “Static Verification of Dynamic Properties”, ACM SIGAda 2003 Conference, 2003

[7] G. J. Holzmann, "The model checker spin", IEEE Trans. Softw. Eng., vol. 23, no. 5, pp. 279-295, 1997

[8] Harald Altinger, Franz Wotawa, Markus Schurius, “Testing methods used in the automotive industry: results from a survey”, Proceedings of the 2014 Workshop on Joining AcadeMiA and Industry Contributions to Test Automation and Model-Based Testing, July 21-21, 2014

[9] S. Keul, “Tuning Static Data Race Analysis for Automotive Control Software”, Proceedings of the 11th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM) 25-26 September, pp. 45-54, September 2011

[10] 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

[11] S. Lu, J. Tucek, F. Qin, and Y. Zhou. “AVIO: Detecting atomicity violations via access interleaving invariants”, In International Conference on Architectural Support for Programming Languages and Operating Systems, 2006.

[12] C. Lattner and V. Adve, “LLVM: A compilation framework for lifelong program analysis & transformation”, In Code Generation and Optimization, 2004. CGO 2004. International Symposium on, pages 75{86. IEEE, 2004

[13] Baris Kasikci, Cristian Zamfir, George Candea, “Automated, Classification of Data Races Under Both Strong and Weak Memory Models”, ACM Transactions on Programming Languages and Systems (TOPLAS): Volume 37 Issue 3, June 2015

[14] Mingxing Zhang, Yongwei Wu, Shan Lu, “AI - A Lightweight System for Tolerating Concurrency Bugs”, FSE 2014 Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2014

[15] Effinger-Dean, L. et al., “IFRit: interference-free regions for dynamic data-race detection”, In: OOPSLA, pp. 467–484, 2012

[16] Konstantin Serebryany et al., “Dynamic race detection with LLVM compiler”, Proceedings of the Second international conference on Runtime verification, September 27-30, 2011

[17] 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.

[18] http://frama-c.com/download_neon.html, version 01.03.2014

[19] OSEK/VDX Communication Specification 2.1 revision 1, 17 June 1998, https://www.autosar.org/fileadmin/files/releases/2-0/software-architecture/communication-stack/standard/AUTOSAR_SWS_COM.pdf

[20] http://man7.org/linux/man-pages/man2/sched_getaffinity.2.html

[21] Jasmin Jahic et al., “Test Coverage Measurements to support Design Space Exploration”, IDEAL 2014, IFIP First International Workshop on Design Space Exploration of Cyber-Physical Systems