Hardware-Assisted Online Data Race Detection

Faustin Ahishakiye[1], José Ignacio Requeno Jarabo[1,2], Violet Ka I Pun[1], and Volker Stolz[1]

[1] Western Norway University of Applied Sciences, Norway
{fahi,jirj,vpu,vsto}@hvl.no
[2] Complutense University of Madrid, Spain
jrequeno@ucm.es

Abstract. Dynamic data race detection techniques usually involve invasive instrumentation that makes it impossible to deploy an executable with such checking in the field, hence making errors difficult to debug and reproduce. This paper shows how to detect data races using the COEMS technology through continuous online monitoring with low-impact instrumentation on a novel FPGA-based external platform for embedded multicore systems. It is used in combination with formal specifications in the high-level stream-based temporal specification language TeSSLa, in which we encode a lockset-based algorithm to indicate potential race conditions. We show how to instantiate a TeSSLa template that is based on the Eraser algorithm, and present a corresponding light-weight instrumentation mechanism that emits necessary observations to the FPGA with low overhead. We illustrate the feasibility of our approach with experimental results on detection of data races on a sample application.

Keywords: Runtime verification, data race detection, FPGA, lockset algorithm

1 Introduction

Data races occur in multi-threaded programs when two or more threads access the same memory location concurrently, with at least one write access, and the threads are not using any exclusive locks to control their accesses to that location. They are usually difficult to detect using tests as they depend on the interleaving and scheduling of tasks at runtime. Static analysis techniques frequently suffer from false positives due to over-abstraction, although precise results for source code written in a particular style is certainly feasible. We do not want to discount this field and recent advances, but focus on dynamic techniques for the present occassion.

Races and other concurrency issues have featured prominently in area of Runtime Verification (RV), where precise formal specifications are used at runtime to monitor, and possibly influence, a running system (as opposed to static verification). Our guest of honor (see [12] for an extensive account of his work) has been one of the founders of the RV workshop- and conference series, and has indeed contributed to the study of the dynamic nature of races with a contribution to the very first workshops [13,14], which has since withstood the test of time [15]. His exploits — which indicate that he is rather aiming for a marathon than a sprint in the race to formal verification — did not stop there: Together with Artho and Biere, he lifted the abstract formal concepts into a practical software engineering setting, where, although race-free in the original sense of the definition, they captured patterns that indicate flawed access to data structures [3]. Source-code instrumentation is one of the go-to solutions to inject RV mechanisms into existing software [10]. His further research with Bodden in a similar direction resulted in the suggestion of a new feature for aspect-orientated programming, a technique for manipulating programs on a higher level, that would facilitate better addressing of concurrency concerns [6].

Although runtime checking only give a limited view on the behaviour of the concretely executed code, it allows precise reporting of actual occurrences, which can be used to predict potential erroneous behaviour across different runs [16]. However, this runtime analysis is not enabled in the final product: inline dynamic data race detection techniques come with invasive instrumentation for each memory access that makes it prohibitive to deploy an executable with such checking in the field [22]. This also makes reproducing errors challenging.

In this article, we take earlier RV attempts for race analysis further and present a *non-intrusive* approach to monitoring applications on embedded system-on-chips (SoCs) for data races using the COEMS platform [8] which aims to eliminate the overhead of dynamic checking by offloading it to external hardware.[3], The platform offers control-flow reconstruction from processor-traces (here: the Arm CoreSight control-flow trace), and data-traces through explicit instrumentation [26]. Race checking is executed on an FPGA on a separate hardware-platform to minimize impact on the system under observation. Our experimental results show that the necessary instrumentation in the target application incurs the expected *fixed, predictable* overhead, and is not affected by the time required for race checking. Our use of the high-level stream-based temporal specification language, TeSSLa [21], means that the reconfiguration of the monitor is substantially faster than synthesising VHDL (few seconds vs. dozens of minutes), and allows end-users to customize the race checker specification to their needs without being FPGA-experts.

This is not possible with other specification-based approaches that directly aim to use the integrated FPGA of a SoC. These approaches do not offer the quick reconfiguration possible with the COEMS platform but require full time-consuming reconfiguration, and do not support the use of control-flow tracing due to the limited capacity of the SoC.

The approach proposed in this paper is more flexible than a dedicated race checker implemented on the FPGA: to the best of our knowledge, such a general solution does not exist, though it is of course in principle possible. It would not offer the end-user flexibility in terms of fast reconfiguration that we gain

[3] The EU Horizon 2020 project “COEMS–Continuous Observation of Embedded Multicore Systems”, `https://www.coems.eu`.

through TeSSLa, and, again due to the space restrictions on SoCs, would not be able to benefit from control-flow tracing features that are important for future optimisations and integration with our analyses.

Our *hardware*-based approach can be used in safety-critical systems such as the aerospace and railway-domains where certification is necessary. In these domains, using a software inline race-checker such as ThreadSanitizer [28] is not possible as the tooling for instrumentation and online race-checking is not certified for those systems, if it even exists. For example, ThreadSanitizer support for Arm32 SoCs is not part of the LLVM toolchain [23]. In contrast to software-based approaches, our instrumentation for the application under test has straightforward complexity and gives predictable performance overhead independant from whether or not race checking is enabled. This is especially important for software development in these safety-critical domains, as again for certification purposes, it is not permissible to, e.g., deploy a separate version for debugging or troubleshooting on demand in the field. Any debugging and trace support must be already integrated in the final product.

The COEMS FPGA requires a compiled monitor-configuration. As this configuration needs to be generated for a specific binary under test, we present in this paper our approach where we instantiate a template that monitors a fixed number of memory locations for consistent access through a fixed number of locks. Although these numbers need to be determined before starting the monitoring, the flexibility of TeSSLa allows us to also deal with an unbounded number of threads, and limited monitoring of dynamically allocated memory and locks. Additionally, our instrumentation supports recording traces in files, and offline analysis of execution traces with the TeSSLa interpreter only. This aids in quick prototyping of new specifications on vanilla developer machines without replicating a full setup of SoC and COEMS hardware.

The paper is structured as follows. After this introduction, Section 2 explains the related work. Section 3 details the data race detection in the COEMS framework. Section 4 illustrates the feasibility of our approach and presents performance data on detecting data race errors in a Linux `pthreads`-based case study. Experimental results and software are published in public repositories. Finally, Section 5 gathers the conclusions and future work.

2 Our Approach and Related Work

Traditionally, data races have been approached from two sides: *static analyses* check the source code and report potential errors. To that end, over-approximations of program behaviours are used (e.g., in terms of variable accesses and lock operations), which may lead to uncertainties on whether a particular behaviour will actually occur during runtime due to general issues on decidability. This frequently generates too many warnings of potential problems for developers to be useful. These uncertainties can be minimised if decisions such as the number of threads to spawn are fixed at compile time. Static analysers may also have limited support for particular language features.

In contrast to checking the code before it runs, *dynamic analyses* look at individual executions of a program. Although this can only analyse the behaviour of the concretely executed code, it can accurately identify the actual occurrences of defects. These can then be traced back to the buggy code that resulted in the potentially erroneous behaviour. Both techniques in general rely on the availability of the source code, and, in the case of dynamic analyses, the possibility of recompilation with additional instrumentation.

As dynamic analyses for data race detection need to record historic behaviour during execution, they often interfere in terms of computation time and memory consumption. For example, the popular dynamic ThreadSanitizer integrated with the LLVM compiler toolchain slows down executions by a factor of 10 to 100, depending on the workload [28,30]. This is one reason why dynamic analysers are traditionally only employed during development and testing, but not included on the production system [29].

The COEMS project developed a hardware-based solution, in which a *field-programmable gate array* (*FPGA*) checks the execution trace in parallel to the running system with minimal interference. The hardware is adapted for analysing events described in the stream-based specification language TeSSLa [21]. Using an intermediate specification language that is executable on the FPGA can avoid the time-consuming re-synthesisation of the FPGA when changing specifications.

We have ported the gist of the Eraser algorithm [27] to the subset of the TeSSLa language that is supported on the hardware. An alternative approach already used the TeSSLa-interpreter, but was not suitable for compilation onto an FPGA due to the dynamic data structures (sets and maps) that only the interpreter offers [19].

Firstly, we adapt the software-based analysis, which relied on dynamic data structures such as sets and maps in the TeSSLa interpreter, to the hardware-specific implementation of the COEMS trace box (see Section 3 for details). As the complete specification does not fit onto the FPGA, we then split the specification into two parts: the performance-relevant portion of the TeSSLa specification is processed on the FPGA (filtering accesses), and the final tracking of which lock protects which memory is done in the interpreter which receives the intermediate output from the FPGA. Additionally, we also allow monitoring of dynamically allocated memory and locks.

The Eraser algorithm is certainly no longer the state of the art in dynamic race detection (or rather, checking locking discipline), but has the advantage that it can be captured in a state machine that is instantiated per memory location and set of locks. Even though it conceptually uses sets, assuming that the number of used memory locations and locks is statically known, we can statically derive the necessary streams.

Such a static encoding should be possible also for the more modern FastTrack-algorithm by Flanagan and Freund [11], which uses lightweight vector clocks and the happens-before relation to avoid false positives. Their article includes a detailed description of the necessary data structures, and uses thread-ids as offset into arrays. Our approach here requires focusing on a fixed number of memory locations and locks, but can deal with an arbitrary number of threads. We leave an encoding into TeSSLa of FastTrack to future work — for a statically decidable set of threads it should certainly be possible, with the necessary vector-clocks also being maintained on the FPGA-side.

An observation-based race checker that tracks memory accesses and lock operations can also be implemented through the help of virtualisation. Gem5 [5,17] is such a framework. Virtualisation means on the one hand that observation cannot be done on a deployed system in the field but only in the lab and with a limited number of supported peripherals. On the other hand, control-flow events can easily be explicitly generated, no expensive reconstruction is necessary: in full virtualisation, we can directly match on any assembly instruction, and not only branches like with the COEMS hardware. In fact, in such a scenario, it would be straightforward to use Gem5 as event source, where the virtualisation sends events on to a TeSSLa interpreter checking the trace against our specification. Gem5 does not directly offer a high-level specification language for monitoring. Given the high event rate of observations on memory accesses, we expect a similar performance impact like the one reported for ThreadSanitizer.

Another prominent example where a high-level specification is synthesised into an FPGA is RTLola [4]. This differs from our approach in the following: the specification language puts a stronger emphasis on periodic data than we do with our discrete TeSSLa events. Furthermore, RTLola is synthesized via VHDL onto the FPGA, and hence has a high turn-around time for reconfiguration. Communication between the system-under-test and the verification logic is left open to the user and requires knowledge of VHDL, though of course in principle data events can then be emitted through instrumentation. In contrast to our solution, an RTLola specification cannot benefit from control-flow tracing, since control-flow reconstruction is not available as specification and hence cannot be compiled onto the FPGA, and furthermore would exceed the capacity of current SoCs both in terms of space and execution speed [26]. We leave performance evaluation of RTLola execution for race checking purposes on the FPGA to future work, but note that providing an API for the instrumentation to the monitor requires VHLD-knowledge.

A similar direct approach via hardware-synthesis has been taken for Signal Temporal Logic (STL) [18]. It would certainly be feasible to encode a race checker in STL, but that would not be playing to STL's strength in terms of timing properties (which are not relevant for race checking) and observing signals on a wire (as opposed to a programmable interface to send values from the instrumented code to the monitor).

The R2U2 [25] monitoring system for unmanned aerial vehicles provides a generic observation component on an SoC. Again, events must be explicitly emitted, and no control-flow reconstruction is available. Similar to our approach, and unlike in RTLola, this component is generic and is parametrised by compiled specifications. R2U2 uses Metric Temporal Logic specifications (MTL), which are very suitable to describe, e.g. timing properties. While it is certainly possible to specify our race checker in MTL, we leave it to future experimental evaluation

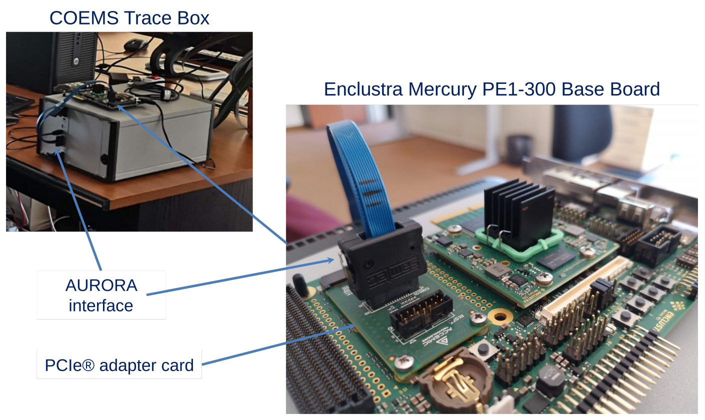

Fig. 1.1: COEMS Trace Box containing FPGA (left) and SoC (right)

how many instances of the race pattern (in terms of memory location/protecting lock) would be feasible, and how the communication bus would uphold under varying event rates.

3 Data race detection with COEMS

In the following, we first briefly introduce COEMS infrastructure. Then, we describe the workflow of data race detection with the COEMS tools. After that, we explain the idea of the lockset-based Eraser algorithm and our translation into TeSSLa.

3.1 COEMS Infrastructure

The COEMS project provides a novel observer platform for online monitoring of multicore systems. It offers a non-intrusive way to gain insights of the system behaviour, which are crucial for detecting non-deterministic failures caused by, for example, accessing inconsistent data as a result of race conditions.

To observe SoCs, the platform uses the tracing capabilities that are available on many modern multicore processors. Such capabilities provide highly compressed tracing information over a separate tracing port. This information allows the COEMS system to reconstruct the sequence of instructions executed by the processor [26]. The instruction sequence- and data trace can then be analysed online by a reconfigurable monitoring unit. Figure 1.1 shows the COEMS FPGA enclosure, the Arm-based Enclustra SoC that serves as system under test, and the AURORA interface connecting both.

As soon as the program starts running on the Enclustra board, control flow messages are generated via the Arm CoreSight module and transmitted, together with user-specified data trace messages from any instrumentation, through the AURORA interface to the COEMS trace box. Internally, the Instruction Reconstruction (IR) module reconstructs an accurate execution trace of both cores

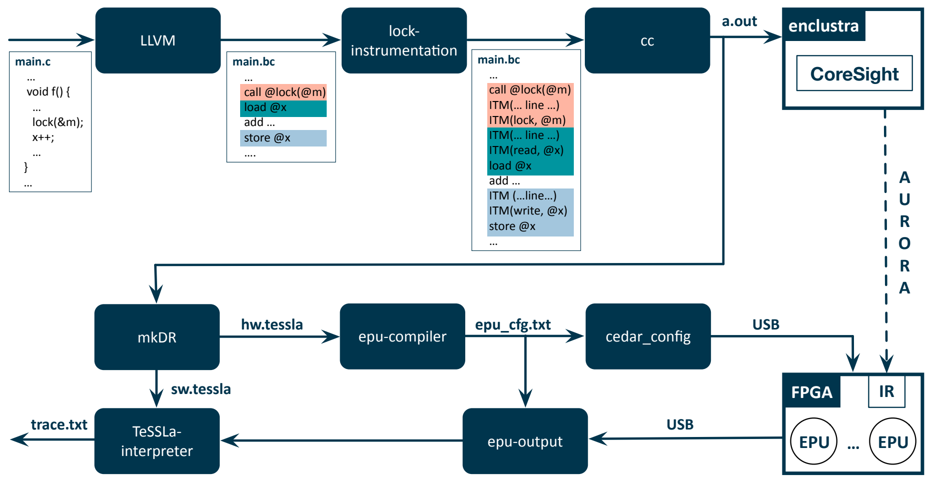

Fig. 1.2: Lock instrumentation and race monitoring using the COEMS technology

from the platform-specific compressed format into a stream-based format suitable for analysing properties defined in the TeSSLa language. The flexibility of the TeSSLa language allows expressing different kinds of analyses for functional or timing properties in terms of stream events. A TeSSLa specification is then compiled to a configuration of the Event Processing Units (EPUs) [7] of the monitoring unit, which are specialised units on the FPGA implementing low-level TeSSLa stream operations. The events of the TeSSLa streams are efficiently processed by the EPUs in the trace box. To cope with the potentially massive amount of tracing data generated by the processors, the COEMS system is implemented in hardware using an FPGA-based event processing system. The current COEMS prototype implements eight parallel EPUs.

Compared to existing monitoring approaches, COEMS provides several advantages, most notably is its non-intrusive method to observe and verify the actual behaviour of the observed system, i.e., the system behaviour will not be affected by the monitoring. As no trace data has to be stored, systems can be monitored autonomously for extended periods of time. Furthermore, the trace box reports results of a TeSSLa analysis almost immediately as the processing delay introduced by the trace box is negligible. In contrast to other hardware-based runtime verification techniques [9,29], changing the specification does not require circuit synthesis, but only a TeSSLa compilation. Hence, the focus of observation can be changed during runtime by reconfiguring the EPUs quickly.

3.2 Instrumentation and Data Race Monitoring

The current COEMS framework supports data race detection for `pthread` programs that can be recompiled using LLVM. We illustrate the workflow of instrumentation and data race monitoring in Figure 1.2.

We first instrument the application under test during compilation, so that the executable emits information to the COEMS trace box at runtime. We insert calls to instrumentation (i) after taking a lock, (ii) before releasing a lock, and (iii) on shared memory accesses with the help of LLVM, which will send the thread-id and observed action. As an optimisation, we use the LLVM analysis framework to only instrument memory accesses to potentially shared memory: through escape-analysis, this can already eliminate instrumentation e.g. on iteration variables of tight loops. Then, we compile and link the instrumented LLVM intermediate code (.bc) into a binary file (a.out). The instrumentation should hence be easy to integrate into existing build-setups.

Secondly, we copy the binary to the system under observation (enclustra) where we will later run it. The mkDR-script instantiates a TeSSLa specification template with the memory addresses and mutexes to be observed, based on the names of global variables. Expert users have the option of more fine-grained control on the instantiation, e.g., to monitor dynamically allocated memory. The specification is tailored to each program; thus, it should be regenerated from the template every time the application is recompiled. The instantiated specification is then split into two halves, as its size exceeds the currently available number of eight EPUs on the prototype hardware. The first half hw.tessla filters the high event rate stream of observations on the FPGA. It is translated by the epu-compiler into a configuration file (epu_cfg.txt), and then uploaded to the FPGA by cedar_config. The second half sw.tessla receives the output of the first stage, and does the final processing on a stream that now has a lower event rate in the TeSSLa interpreter.

Then, we run the binary file, which will automatically start sending trace data to the FPGA. The epu-output tool decodes the FPGA output into a TeSSLa event stream. Note that the behaviour of the application is independent of whether the COEMS FPGA is actually connected or not. If not, trace data is silently discarded, but does not affect the timing of the application.

Finally, we analyse this trace with the second part of the TeSSLa specification (sw.tessla) with the TeSSLa interpreter, which will emit race warnings if necessary. Our data race specification uses mostly data trace events, since we require the addresses of memory and locks, except for a control flow event when `pthread_create` is called and to signal termination of program under test.

In addition to the online (hardware-based) monitoring analysis with the COEMS trace box, the COEMS framework also supports offline (software-based) analysis of execution traces in a personal computer. In the case of software-based analysis, the user only needs the TeSSLa interpreter and the COEMS lock instrumentation tool. Most of the initial steps in Figure 1.2 such as the LLVM instrumentation or the instantiation of the TeSSLa template are similar for the software-trace analysis. Instead of compiling the TeSSLa specification for the EPUs, the software TeSSLa interpreter will now run the entire TeSSLa specification for detecting data races on a locally generated software trace-file. Writing the trace data into a file first or piping them into the TeSSLa interpreter has a higher overhead than transmitting them via the AURORA interface.

3.3 Lockset-based Algorithm in TeSSLa

Conceptually, the algorithm tracks which set of locks is held at every memory access. The current set is intersected with the previous set on a read or write (for simplicity of presentation, we do not distinguish between read- and write accesses, although only read/write and write/write-conflicts are relevant). This defines the alphabet of observations of the algorithm: pairs of reads or writes with a memory address and thread-identifier, and locking or unlocking operations with lock- and thread-identifier. The algorithm initialises the lockset for each memory address with the set containing all locks, and should the intersection ever yield the empty set, then we can conclude that an inconsistent locking discipline has been used. This means that one part of the execution uses no or disjoint locks from another part of the execution when accessing this memory, which hence gives rise to a potential data race if those executions are assumed to be possible concurrently.

As we do not have dynamic memory available to maintain potentially unbounded sets in the TeSSLa-specification on the FPGA, we need to find a static encoding. To achieve this, for each pair of memory location X and lock identifier (also an address) L, we create a boolean stream `protecting_X_with_L` that is initialised to *true*. The set of all these streams for a given X hence models the lockset as a bit-vector. Note that updates are monotone, i.e., once a stream takes the value *false*, it can no longer revert to *true*. If all these streams for a given X carry *false*, we know that no common lock is protecting the current access, and we emit a race warning on the `error_X` stream for that memory location. This encoding means that we need to know the set of all memory locations and all lock identifiers before we configure the FPGA, as we cannot declare new streams dynamically.

On every memory access to X, we check if L is being held by the current thread and update the current value if necessary. We track this through the streams `holding_L`, which carry the identity of the thread currently holding this lock, if any. Again, updating the value on these streams is trivial upon each locking or unlocking operation.

```
in threadid: Events[Int]
in readaddr: Events[Int]
in writeaddr: Events[Int]
in mutexlockaddr: Events[Int]
in mutexunlockaddr: Events[Int]
@FunctionCall("__pthread_create_2_1")
in pcreateid: Events[Unit]
in line: Events[Int]
in dyn_base: Events[Int]
in dyn_lock: Events[Int]
```

Figure 1.3: Header of the TeSSLa specification, including all the incoming events from the instrumented code.

```
 1  def lock_0 := filter(mutexlockaddr ==1, mutexlockaddr ==1)
 2  def unlock_0 := filter(mutexunlockaddr ==1, mutexunlockaddr ==1)
 3  def lock_1 := filter(mutexlockaddr ==24808, mutexlockaddr
        ==24808)
 4  def unlock_1 := filter(mutexunlockaddr ==24808, mutexunlockaddr
        ==24808)
 5  def dyn_temp := 4 * (((dyn_lock >> 1) >> 1))
 6  def slot := dyn_lock - dyn_temp
 7  def dyn_lock_0 = filter(dyn_temp, slot == 0)
 8  def lock_4 := filter(mutexlockaddr == dyn_lock_0, mutexlockaddr
        == dyn_lock_0)
 9  def unlock_4 := filter(mutexunlockaddr == dyn_lock_0,
        mutexunlockaddr == dyn_lock_0)
10  def read_0 := filter(readaddr ==24532, readaddr ==24532)
11  def write_0 := filter(writeaddr ==24532, writeaddr ==24532)
12  def access_0 := merge(read_0,write_0)
13  def access_after_pc_0 := on(last(pcreateid,access_0),line)
14  def thread_accessing_0 := last(threadid*32768 + line,
        access_after_pc_0)
15  def holding_0 := default(merge(last(threadid, lock_0), last(-1,
        unlock_0)), -1)

16  def protecting_0_with_0 := detect_change(default(
        thread_accessing_0/32768 == last(holding_0,
        thread_accessing_0), true))
17  def error_0 := on(thread_accessing_0,!(protecting_0_with_0 ||
        protecting_0_with_1 || protecting_0_with_2 ||
        protecting_0_with_3 || protecting_0_with_4))
```

Figure 1.4: TeSSLa fragment, where lines 1–15 are from the hardware stage, while lines 16–17 are from the software stage.

Figures 1.3–1.4 show an excerpt of the resulting TeSSLa specification. It has been created for one dynamic lock and three static locks, and tracks accesses to memory address 24532. Static locks are stored at memory addresses 24808, 24528 and 24560 (only lock 24808 is shown in Figure 1.4), while the memory address of the dynamic lock is emitted at runtime. The static lock at address 1 is artificial and is used for the main-thread only. TeSSLa built-in stream operations are emphasised. As input streams, we receive instrumented events on `mutexlockaddr`, `mutexunlockaddr`, `readaddr`, `writeaddr`, `threadid`, `dyn_base` and `dyn_lock`. The `pcreateid` event reduces false-positives by signalling to only start observing memory accesses after additional threads have actually been created. The annotation `@FunctionCall` indicates that this is a control flow-event which is triggered by a function call, and not through instrumentation. The symbol name corresponds to the function `pthread_create` from Linux' system library. To aid in debugging, the instrumentation also emits the current source code `line` number with each event.

Currently, due to the limited availability of EPUs on the prototype FPGA, the specification is actually split into two halves, with the fast address-filtering done on hardware, and only processing the derived information in the coloured lines 16–17 as a post-processing stage in the interpreter. The multiplication and division are binary left- and right-shifts that reduce the amount of events emitted from the hardware stage to the software stage by encoding the line number of an instruction with its event in the unused lower 16 bits, and encoding dynamically allocated locks (see below).

4 Case study

In this section, we illustrate our approach with a case study, where we simulate a set of bankers sending random amounts of money from one bank account to another [20]. The bankers lock the source and target bank accounts before committing a transaction, so that transfers are protected against data races and deadlocks. We introduce a special case where one banker (id 0) forgets one lock operation and hence, data may get corrupted. Figure 1.5 shows the core of the example with locks and memory accesses. The complete example, including source code, execution traces and TeSSLa reports, is available at [1].

```
/* get an exclusive lock on both balances before
  updating (there's a problem with this, see below) */
pthread_mutex_lock(transaction_mtx);
if( !DATA_RACE || (DATA_RACE & (id != 0)) ){
    /* In case of DATA_RACE flag is 'on', the thread_id 0
    forgets to lock the accts[from].mtx mutex */
    pthread_mutex_lock(&accts[from].mtx);
                }
pthread_mutex_lock(&accts[to].mtx);
pthread_mutex_unlock(transaction_mtx);
/* Do the actual transfer. */
if (accts[from].balance > 0) {
        payment = 1 + rand_range(accts[from].balance);
        accts[from].balance -= payment;

pthread_mutex_unlock(&accts[to].mtx);
if( !DATA_RACE || (DATA_RACE & (id != 0)) ){
    /* For symmetry -- don't unlock if racy: */
    pthread_mutex_unlock(&accts[from].mtx);
}
```

Figure 1.5: Example of incorrect locking

TeSSLa specification. We instantiate the corresponding COEMS data race template (see [2] for all files used here) using the mkDR-script and the instrumented binary, and the names of all mutexes (`accts[0].mtx`, ...) and shared variables (`accts[0].balance`, ...) as parameters.

The size of the TeSSLa specification for the Eraser algorithm is proportional to the number of locks and shared memory addresses to monitor, and independant from the number of threads. More precisely, the first half hw.tessla grows linearly with respect to both variables.

For each lock L, the TeSSLa specification includes the lock/unlock pairs and holding_L (plus an additional stream in the case of dynamic locks). For each memory address X, the TeSSLa specification includes a block of five streams (i.e., read_X … thread_accessing_X). Hence, 3L + 5X streams are generated for the first half, where L is the number of locks and X the number of memory addresses. Regarding the second half, the sw.tessla file, the number of streams grows proportionally to (X+1)*L (i.e., protecting_X_with_L plus error_X).

As the TeSSLa specification is a text file and the size is constrained by the previous parameters, the instantiation of the TeSSLa template for the data race checking is done almost instantaneously, and compilation of the largest specification for the hardware stage into the FPGA completes in around 5 minutes (see below for detailed measurements) including uploading the configuration to the FPGA, hence a big advantage over approaches that need to translate via VHDL.

Working with dynamic allocations. As we have noted above, the addresses of memory locations and locks need to be available at compile-time of the specification. This limits our approach to statically declared resources in a program. However, it fits our application domain in embedded systems for these SoCs, where development may follow the MISRA guidelines [24], which strongly recommends against using e.g., malloc. For more flexibility, we have developed an extension where developers can at runtime send the location of dynamically allocated memory or locks through additional instrumentation.

Conceptually, we parametrise the specification with a placeholder for the argument of the comparison operations above. A write to a particular stream will make the specification use that value in addition to any hard-coded values. A programmer can use the instrumentation to send the address of a (potentially dynamically allocated) lock to the monitor using the function

```
emit_dynamic_lock_event(const short slot,
                                    const pthread_mutex_t* addr).
```

We can encode potentially multiple "slots" into the lower four bits, since these pointers are suitably aligned (see lines 5–8 in Figure 1.4). Similarly, the developer can register the base address of dynamically allocated storage for monitoring through emit_dynamic_addr_event(const uintptr_t base). The range of bytes to monitor is given when instantiating the specification.

Due to the size limitation on the FPGA, we currently can only provide this filter for a limited number additional memory addresses or locks. As the number of possible EPUs on the external FPGA increases, these numbers for dynamic allocations should also go up. Note that the number of statically encoded resources underlies different resource constraints, and we report the general numbers below in the performance characteristics.

Running the experiment. The epu-compiler translates the first part (hw.tessla) into a configuration file (epu_cfg.txt), and then uploads it via USB to the FPGA through cedar_config. The compilation time depends on the number of streams

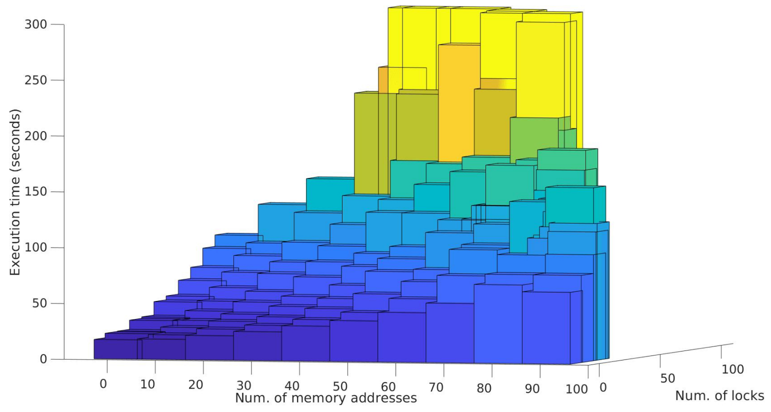

Fig. 1.6: EPU compilation time for hw.tessla

in the TeSSLa specification and, hence, the number of locks and shared memory addresses in the C code. Figure 1.6 shows the time required for compiling the TeSSLa specification into the binary configuration format for different scenarios of the bankers example in terms of the number of locks and memory locations. As this involves allocating streams and their operations to the available on-board resources of the FPGA, the following outcomes are possible: (i) successful configuration, (ii) aborted because FPGA-resources have been exceeded, (iii) timeout. Due to the combinatorial growth of the specification, we see that compilation times go up towards the timeout that we have chosen (300 seconds) for growing numbers of locks/memory locations. After that we reach ranges where it is quickly obvious for the compiler that a configuration cannot be fit onto the FPGA. As compilation is a resource-allocation problem that involves constraint-solving, this slope for compilation time is to be expected: the closer a specification gets to the available resource limits, the harder the constraint-solver has to try searching for a suitable allocation.

After compilation, uploading a new binary configuration into the FPGA after compilation is done in between 3 to 7 seconds.

The second half of the specification, sw.tessla, receives the output of the first stage, and does the final processing on a stream that now has a lower event rate in the TeSSLa interpreter. Naturally, the interpreter has some startup-cost that also scales with the size of the specification due to parsing and type-checking. Figure 1.7 shows a similar slope as for the EPU compilation, where startup-time goes up towards the upper right corner, where we also reach up to 300 seconds.

Since compilation and start-up of the interpreter can be done in parallel and hence lead to the envisioned advantage of quick reconfiguration over approaches going via VHDL.

The COEMS trace box currently supports TeSSLa specifications in the range of hundreds to a few thousands of streams depending on the complexity in the logic of the TeSSLa streams. For our race checker this translates into checking

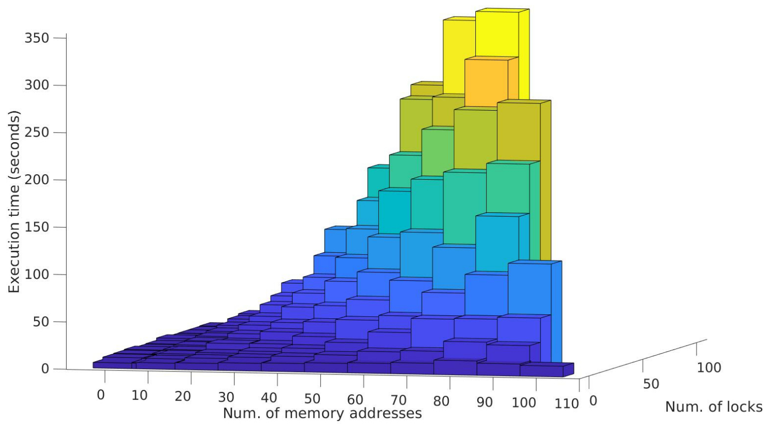

Fig. 1.7: TeSSLa interpreter startup time for sw.tessla

between around 40 memory locations with 100 locks and 90 memory locations with 20 locks.

```
105923234788: holding_1 = 21653
105923846011: holding_3 = 21653
105923949854: holding_1 = -1
105924265689: holding_1 = 21525
105924621853: thread_accessing_1 = 709525559
105924716727: thread_accessing_1 = 709525559
105924764106: holding_2 = 21525
105924822656: holding_3 = -1
105924872330: holding_1 = -1
105925531285: thread_accessing_0 = 705331255
105925621081: thread_accessing_0 = 705331255
105925674037: holding_1 = 21653
105925731366: holding_2 = -1
105926266497: holding_2 = 21653
```

Figure 1.8: Events emitted by the COEMS trace box after processing the hardware half of the TeSSLa specification

```
105847960743: error_0_in_line = 52
105847960743: error_0 = true
105848052196: error_0_in_line = 53
105848052196: error_0 = true
105848316093: error_0_in_line = 54
105848316093: error_0 = true
```

Figure 1.9: Race report obtained by processing Figure 1.8 including debug information

When we run our C code with the DATA_RACE flag on, the first stage produces the output stream shown in Figure 1.8. This stream contains summaries on which thread is holding which lock and accessing any of the selected memory

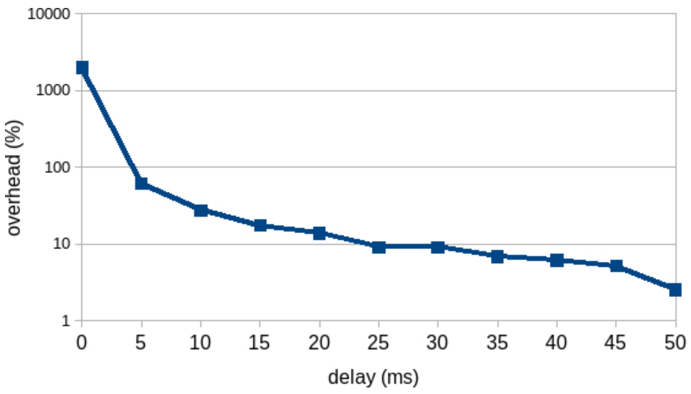

Fig. 1.11: Overhead introduced by the instrumentation.

addresses. We then pipe those events into the TeSSLa interpreter with the other half of the specification.

```
1 for (i = 0; i < N_THREADS; i++)
2       pthread_join(ts[i], NULL);
3 for (total = 0, i = 0; i < N_ACCOUNTS; i++)
4       total += accts[i].balance;
5 printf("Total money in system: %ld\n", total);
```

Figure 1.10: Example of false positive after threads have terminated

The TeSSLa interpreter correctly reports (Figure 1.9) the data race errors on the `error_X_in_line` streams, triggered by the accesses in lines 12–15 in Figure 1.5 (corresponding to lines 52, 53 and 54 in the source code). These data race errors are caused by the missing lock of mutex `accts[from].mtx` by thread `id_0`. Our tool also detects a false positive in line 4 in Figure 1.10. that happens when the `main()` thread accesses the balance in each account once all banker-threads have terminated. The barrier introduced by `pthread_join()` cannot be detected by our lockset-based approach, as it does not actually keep track of the threads in use by the program.

Performance. We have so far only obtained partial performance measurements, as currently our instrumentation needs a global lock to serialize transmission of three events per observation to the FPGA. We transmit the thread-id, the address of the relevant datum, and the line number in the source code to aid in debugging. The lock guards against interleaving between the two cores and context-switches on the same core. This leads to a penalty factor of about 20 in execution time over the original uninstrumented code. At least 50% of that overhead can be attributed to the lock, effectively linearising the above example.

Figure 1.11 shows the overhead in percent of the instrumented version against the original code. To simulate other workloads with less contention (i.e., larger regions where no race checking is required), we have introduced a configurable `usleep()` instruction between both accesses to the bank accounts. The graph shows that the high overhead is due to the tight loop accessing the accounts, and naturally becomes less prominent as contention goes down.

Another factor affecting overhead is that our instrumentation has not yet been optimised and shares code with the software-tracing for prototyping, which among other things means that even when doing hardware-tracing, additional arguments are passed on the stack that the hardware-tracing does not actually consume. Future improvements in the low level runtime support for data trace events may also bring better performance by allowing larger payloads in a single message, which would allow our instrumentation to avoid the explicit global lock.

We have not yet devised a setup that can measure the performance of evaluating the specification, apart from considerations based purely on the use of EPUs and clock rate of the FPGA: our measurements are currently completely dominated by the USB-interface overhead of polling the output events from the FPGA, and do not allow to precisely factor out the processing time. Additionally, intermediate output from the FPGA to the interpreter is transferred in a verbose ASCII-format.

As for memory consumption, our instrumentation does not need to maintain any data structures, and only passes primitive values such as pointer addresses that are already computed and presumably available in registers anyway.

5 Conclusion

In this paper, we followed the path initiated by our guest of honor in the direction of practical approaches for runtime verification. More in detail, we have shown how to use the COEMS technology, a novel platform for online monitoring of multicore systems, and contextualized it to check for potential data races in applications that use locks for synchronisation, one of our guest's research areas.

Through the COEMS platform, developers can observe the control-flow in a digital twin of their application under test on an embedded systems without affecting the behaviour. Additional instrumentation of the application can send more detailed data at negligible cost.

We presented an outline on how the lockset-based Eraser algorithm can be encoded in the TeSSLa-specification language for a given application. This specification is then compiled onto the external COEMS FPGA and uses the data- and control flow trace emitted from the system under test to observe a specified set of locks and memory locations. As the full specification exceeds the capabilities (in terms of size) of the available prototype, we combine a hardware- and a software stage to report on potential races.

Races may still hide in parts of the code that have not been executed, and our checker may report false positives, which is also a general limitation of tools based on the Eraser algorithm. On the positive side, the COEMS hardware race checker does not negatively affect the performance of the application, so potential users need to carefully assess this tradeoff and structure their code to minimize warnings.

The data race analysis uses the LLVM compiler framework, and currently works with threads using `pthread_mutex_lock/unlock` operations for protecting the shared variables. For other ways of synchronization, e.g., through compare-and-swap instructions, or baremetal execution, we do not provide instrumentation and a template yet, but they can easily be adapted from our code.

A practical limitation of the data trace is the currently restricted value-range of the trace messages to 16 bits, which complicates e.g. the use of pointers in the trace. As currently we need multiple messages per event to transmit additional data such as debugging information (the current line number) and the thread identifier, we need to serialize use of the trace bus. This additional locking that is introduced through the instrumentation affects the performance of the application under test, whereas transmitting a single datum in principle has negligible execution overhead.

Our unoptimised performance measurements already puts us in a competitive range with other approaches such as ThreadSanitizer, and we have the advantage that COEMS-based tracing can remain enabled in production. Future developments of the COEMS platform beyond its current prototype will make splitting the specification and post-processing in the interpreter superfluous: 18 (instead of the currently 8) available EPUs will already allow for setups without dynamic values to be handled completely in hardware. In the meantime, we are improving the instrumentation to produce effect summaries for basic blocks of code instead of instrumenting single instructions, which should decrease the overhead especially for tight loops.

We are also preparing additional concurrency patterns that monitor actual deadlocks and so-called lock-order-reversal.

References

1. Ahishakiye, F., Jarabo, J.I.R., Pun, K.I., Stolz, V.: Open data for banker example. `https://doi.org/10.5281/zenodo.4381982` (December 2020)
2. Ahishakiye, F., Jarabo, J.I.R., Stolz, V.: Lock instrumentation tool. `https://github.com/selabhvl/coems-racechecker` (2020)
3. Artho, C., Havelund, K., Biere, A.: High-level data races. Softw. Test., Verif. Reliab. **13**(4), 207–227 (2003), `https://doi.org/10.1002/stvr.281`
4. Baumeister, J., Finkbeiner, B., Schwenger, M., Torfah, H.: FPGA stream-monitoring of real-time properties. ACM Trans. Embed. Comput. Syst. **18**(5s) (Oct 2019), `https://doi.org/10.1145/3358220`
5. Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., Hestness, J., Hower, D.R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M.D., Wood, D.A.: The Gem5 simulator. SIGARCH Comput. Archit. News **39**(2), 1–7 (Aug 2011), `https://doi.org/10.1145/2024716.2024718`
6. Bodden, E., Havelund, K.: Aspect-oriented race detection in Java. IEEE Trans. Software Eng. **36**(4), 509–527 (2010), `https://doi.org/10.1109/TSE.2010.25`

7. Convent, L., Hungerecker, S., Scheffel, T., Schmitz, M., Thoma, D., Weiss, A.: Hardware-based runtime verification with embedded tracing units and stream processing. In: Colombo, C., Leucker, M. (eds.) 18th Intl. Conf. on Runtime Verification, RV 2018. Lecture Notes in Computer Science, vol. 11237, pp. 43–63. Springer (2018)
8. Decker, N., Dreyer, B., Gottschling, P., Hochberger, C., Lange, A., Leucker, M., Scheffel, T., Wegener, S., Weiss, A.: Online analysis of debug trace data for embedded systems. In: Madsen, J., Coskun, A.K. (eds.) Design, Automation & Test in Europe Conference & Exhibition, DATE 2018. pp. 851–856. IEEE (2018)
9. Drzevitzky, S., Kastens, U., Platzner, M.: Proof-carrying hardware: Towards runtime verification of reconfigurable modules. In: 2009 International Conference on Reconfigurable Computing and FPGAs. pp. 189–194. IEEE (2009)
10. Filman, R., Havelund, K.: Source-code instrumentation and quantification of events. In: Foundations of Aspect-Oriented Languages (FOAL 2002). No. TR 02-06 (April 2002), http://www.cs.ucf.edu/~leavens/FOAL/papers-2002/TR.pdf
11. Flanagan, C., Freund, S.N.: Fasttrack: efficient and precise dynamic race detection. In: Hind, M., Diwan, A. (eds.) Proc. 2009 ACM SIGPLAN Conf. on Programming Language Design and Implementation, PLDI 2009. pp. 121–133. ACM (2009)
12. Havelund, K., Reger, G., Rosu, G.: Runtime verification past experiences and future projections. In: Steffen, B., Woeginger, G.J. (eds.) Computing and Software Science - State of the Art and Perspectives, Lecture Notes in Computer Science, vol. 10000, pp. 532–562. Springer (2019), https://doi.org/10.1007/978-3-319-91908-9_25
13. Havelund, K., Rosu, G.: Monitoring Java programs with Java PathExplorer. Electron. Notes Theor. Comput. Sci. **55**(2), 200–217 (2001), https://doi.org/10.1016/S1571-0661(04)00253-1
14. Havelund, K., Rosu, G.: An overview of the runtime verification tool Java PathExplorer. Formal Methods Syst. Des. **24**(2), 189–215 (2004), https://doi.org/10.1023/B:FORM.0000017721.39909.4b
15. Havelund, K., Rosu, G.: Runtime verification - 17 years later. In: Colombo, C., Leucker, M. (eds.) 18th Intl. Conf. on Runtime Verification, RV 2018. Lecture Notes in Computer Science, vol. 11237, pp. 3–17. Springer (2018), https://doi.org/10.1007/978-3-030-03769-7_1
16. Hong, S., Kim, M.: A survey of race bug detection techniques for multithreaded programmes. Software Testing, Verification and Reliability **25**(3), 191–217 (2015)
17. Jahic, J., Jung, M., Kuhn, T., Kestel, C., Wehn, N.: A framework for non-intrusive trace-driven simulation of manycore architectures with dynamic tracing configuration. In: Colombo, C., Leucker, M. (eds.) Runtime Verification. pp. 458–468. Springer (2018)
18. Jaksic, S., Bartocci, E., Grosu, R., Kloibhofer, R., Nguyen, T., Nickovic, D.: From signal temporal logic to FPGA monitors. In: 13. ACM/IEEE Intl. Conf. on Formal Methods and Models for Codesign, MEMOCODE 2015. pp. 218–227. IEEE (2015)
19. Jakšic, S., Li, D., Ka I Pun, Stolz, V.: Stream-based dynamic data race detection. In: 31st Norsk Informatikkonferanse, NIK 2018. Bibsys Open Journal Systems, Norway (2018), https://ojs.bibsys.no/index.php/NIK/article/view/511
20. Joe, N.: Concurrent programming, with examples. https://begriffs.com/posts/2020-03-23-concurrent-programming.html (March 2020)

21. Leucker, M., Sánchez, C., Scheffel, T., Schmitz, M., Schramm, A.: TeSSLa: Runtime verification of non-synchronized real-time streams. In: ACM Symposium on Applied Computing (SAC). pp. 1925–1933. ACM (2018)
22. Lucia, B., Ceze, L., Strauss, K., Qadeer, S., Boehm, H.: Conflict exceptions: simplifying concurrent language semantics with precise hardware exceptions for data-races. In: Seznec, A., Weiser, U.C., Ronen, R. (eds.) 37th Intl. Symp. on Computer Architecture (ISCA 2010). pp. 210–221. ACM (2010)
23. Matar, H.S., Tasiran, S., Unat, D.: EmbedSanitizer: Runtime race detection tool for 32-bit embedded ARM. In: Lahiri, S.K., Reger, G. (eds.) Runtime Verification - 17th International Conference, RV 2017. Lecture Notes in Computer Science, vol. 10548, pp. 380–389. Springer (2017), https://doi.org/10.1007/978-3-319-67531-2_24
24. MIRA Ltd: MISRA C:2012 Guidelines for the use of the C language in critical systems (2013)
25. Moosbrugger, P., Rozier, K.Y., Schumann, J.: R2U2: monitoring and diagnosis of security threats for unmanned aerial systems. Formal Methods in System Design **51**(1), 31–61 (2017), https://doi.org/10.1007/s10703-017-0275-x
26. Preußer, T., Weiss, A.: The CEDARtools platform - massive external memory with high bandwidth and low latency under fine-granular random access patterns. In: Sourdis, I., Bouganis, C., Álvarez, C., Díaz, L.A.T., Valero-Lara, P., Martorell, X. (eds.) 29th Intl. Conf. on Field Programmable Logic and Applications, FPL 2019. pp. 426–427. IEEE (2019)
27. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.E.: Eraser: A dynamic data race detector for multithreaded programs. ACM Trans. Comput. Syst. **15**(4), 391–411 (1997)
28. Serebryany, K., Potapenko, A., Iskhodzhanov, T., Vyukov, D.: Dynamic race detection with LLVM compiler - compile-time instrumentation for ThreadSanitizer. In: Khurshid, S., Sen, K. (eds.) 2nd Intl. Conf. on Runtime Verification, RV 2011. Lecture Notes in Computer Science, vol. 7186, pp. 110–114. Springer (2011)
29. Watterson, C., Heffernan, D.: Runtime verification and monitoring of embedded systems. IET software **1**(5), 172–179 (2007)
30. Yu, Z., Yang, Z., Su, X., Ma, P.: Evaluation and comparison of ten data race detection techniques. International Journal of High Performance Computing and Networking **10**(4-5), 279–288 (2017)