Stream-based dynamic data race detection*

Svetlana Jakšić, Dan Li, Ka I Pun and Volker Stolz
Høgskulen på Vestlandet
{firstname.lastname}@hvl.no

Abstract

Detecting data races in modern code executing on multicore processors is challenging. Instrumentation-based techniques for race detection not only have a high performance impact, but also are not likely to be certified for safety-critical systems. This paper presents a data race detector based on the well-known lockset algorithm in the stream-based runtime verification language TeSSLa, using dynamic data structures to record lock operations and memory accesses. Such a specification can then be instantiated with particular parameters to make it suitable for the more limited planned monitoring using field-programmable gate arrays (FPGAs).

1 Introduction

Multicore processors have become commonplace. They offer speed boosts in two ways in modern IT: on the one hand, different tasks can run concurrently on multiple cores, sharing operating system resources. On the other hand, developers have the option of tailoring their applications to make use of multiple cores by explicitly introducing concurrency, usually in the form of threading. These threads by their very definition are not isolated from each other, but have the possibility to affect each other: memory can be shared between threads, and synchronization APIs allow for higher levels of abstraction. For programs, for example, written in the C programming language, a careful combination of modification of shared data and locking/synchronization is necessary (other programming languages like Go use message passing and leave such synchronization to the runtime library).

Using those concurrency constructs and their interactions incorrectly leads to widely discussed problems, such as deadlocks [1] and data races [8]. While the first type of problem (where a number of processes wait cyclically for a set of shared resources) is easily detectable at runtime, or through an obviously stuck application, the latter is more insidious: there may not be a definitive sign of misbehaviour, as read/write or write/write data races only subtly corrupt the data the application is working with: in a read/write race, one thread reads data, which is

*This work was partially supported by the European Horizon 2020 project COEMS under grant agreement no. 732016 (`https://www.coems.eu/`).

This paper was presented at the NIK-2018 conference; see http://www.nik.no/.

then subsequently overwritten by another thread, leading the first thread to proceed under the assumption that it has the current value. In a write/write race, another thread overwrites a previously written value, leading to a similar problem. We note that data races are not *per se* a fault in a program, but are usually unintended — certain lock-free algorithms are the exceptions.

Traditionally, these types of software defects have been approached from two sides: *static analysis* checks the source code and reports potential errors. To that end, over-approximations of the behaviour of the program 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. In contrast to analyzing the code before it actually runs, *dynamic analysis* looks at the actual behaviour of a program. Although this only gives a limited view on the behaviour of the actually executed code, it allows for the precise reporting of actual occurrences, which can be used to predict potential erroneous behaviour across different runs. As dynamic analyses 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 execution by a factor of 10 to 100, depending on the workload [13]. This is one reason why dynamic analysers are traditionally only employed during development and testing, but not included on the production system. Static analysis on the other hand frequently generate too many warnings of potential problems for developers to be useful.

The juxtaposition of static and dynamic analyses may make them seem like a choice developers can make. But in fact, dynamic analyses often rely on an additional static phase, which is usually required to instrument the additional code for the dynamic monitoring into the program. For example, ThreadSanitizer adds those during compilation, and contains analyses on where to put them, and to avoid them where they are unnecessary for performance reasons. We note that the technique in general relies on the availability of the source code and the possibility of recompilation.

In the EU Horizon 2020 project "COEMS – Continuous Observation of Embedded Multicore Systems", we are half-way on our journey to a hardware-based solution: even though the *field-programmable gate array* (*FPGA*), which will check the instruction trace in parallel to the running system, is still under development, we now have already in place the stream-based specification language, a software-based simulation of checks of events against the specifications, and the necessary static analyses that we later need to leverage the hardware-based monitoring to its full pontential. In our contribution, we show how to formulate the task of checking for potential data races with the stream-based specification, and use a software-based simulation of the monitoring hardware as a proof of concept.

The paper is organized as follows: Section 2 describes dynamic data race detection in general, and introduces the necessary events that we can monitor in the stream-based specification language TeSSLa. Section 3 presents two different views on the specification language that we use to encode our data race detector: a general approach that uses dynamic data structures like maps and sets, and a lower-level, more limited version that needs to be instantiated to monitor particular resources due to the limitations on the FPGA. Section 4 discusses the related work, and Section 5 concludes the paper.

```
void* f(void *arg) {
   for (int i = 0; i < 100; i++) {
      pthread_mutex_lock(&m);
      x++;
      pthread_mutex_unlock(&m);
   }
   return NULL;
}
```

```
void* g(void *arg) {
   for (int i = 0; i < 100; i++) {
      x++;
   }
   return NULL;
}
```

Figure 1: Example of incorrect locking.

2 Dynamic data race detection

The standard definition of a data race is as follows [11]: a data race in the multi-threaded program exists when two or more threads access the same memory location concurrently, at least one of the accesses is a write and the threads are not using any synchronization mechanism to control their accesses to that memory. In general, such bugs are hard to detect because they are highly timing dependent, often related to interaction of two different modules, and their manifestation may be postponed.

For example, let us consider the code in Figure 1 and assume that functions `f` and `g` are executed by two different threads, `p1` and `p2`, resp. In order to execute statement `x++`, a thread needs to perform the following three operations:

1. loading the value of `x` to a register;
2. incrementing the value of the register;
3. storing the new value of the register to `x`.

Both threads are reading from and writing to the address of the shared variable `x`. Thread `p1` is accessing `x` using lock `m`, while `p2` is accessing without any lock. Therefore, other threads executing function `f` may be blocked at the lock statement, and mutual exclusion around the update is guaranteed, while any thread calling function `g` like thread `p2` is not doing so. This may lead to the situation in which some of the increments get lost and the final value of `x` varies in each run. For instance, the following interleaving between the threads may happen:

1. `p1` loads the current value of `x` (e.g., 5) to a register `r1`;
2. `p2` loads the same value of `x` to a register `r2`;
3. `p1` increments `r1` to 6 and writes this value to `x`;
4. `p2` increments `r2` to 6 and also writes 6 to `x`.

We see that the intention is that the code increments the variable twice 100 times. Due to the observed effect (both threads writing the same value), half of the updates may actually get lost. On the other hand, it may happen that one or even several runs of the program do not reveal any flaws in the code, because none such interleaving happens.

One of the most frequently used algorithms for finding possible data races is *lockset algorithm*, introduced and implemented in Eraser [11]. The lockset algorithm finds data races by checking if a program follows a certain locking discipline. In particular, it enforces that every shared variable is always protected by some set of locks and this set of locks must be held by any thread accessing the variable. In the following, we give the basic version of the algorithm. For each shared variable:

1. we identify the accesses to the variable;
2. on each access to the variable, we identify the set of locks held by the thread which is accessing the variable;
3. we initialize the set of locks guarding the variable with the set of all locks;
4. on each access to the variable, we update the set of locks guarding the variable by intersecting it with the set of locks held by the thread which is accessing the variable.

The lockset algorithm is able to detect the error in our example in a single run, regardless of the final value of `x` (as would be other tools like ThreadSanitizer).

In summary, we see that the lockset algorithm needs to know about relevant events with respect to *locking* (which thread is taking which lock), and *memory accesses* (which thread is reading from or writing to a memory location). In the next section, we introduce the specification language that we are using, and discuss how we encode the events above in the TeSSLa language with the help of the events generated by our dynamic monitoring platform

3 Specifying data races with TeSSLa

We use the temporal stream-based specification language TeSSLa [5] to define the property we are monitoring, which is a particular form of data race freedom in this work. TeSSLa is designed specifically to generate monitors, e.g., for real time properties, that can be synthesised and executed in FPGAs such that the monitors can exploit the high parallelism of the hardware to process large amounts of data online. TeSSLa makes use of asynchronous streams as underlaying model, which allows reasoning about program traces derived from embedded trace units in multicore processors. This leads to a very specific set of basic functions, but these functions can be efficiently executed in hardware.

The TeSSLa specification language

In the following, we give an overview of the TeSSLa language with the focus on the features used in our lockset specification. Informally, an *event* in TeSSLa is a pair where the first component, called *timestamp*, determines the time and second the value of the event. A *stream* is a sequence of events which appear ascending ordered by the timestamps. A *trace* is a set of streams. A TeSSLa *specification* takes as input a trace, transforms the streams from the trace using expressions, built-in operations, predefined and newly defined macros (functions), and outputs interesting intermediate or final streams.

In a specification, streams from the input trace are declared by the keyword `in`, followed by the name of the stream and the type of the events on that stream. Currently supported types of events on the input are integer, boolean, string and unit. The limited data types are a deliberate restriction placed on TeSSLa to be able to synthesize efficient hardware monitors.

Expressions are, essentially, `if`–`then` and `if`–`then`–`else` constructs, unary and binary operators, and macro (function) calls. Unary and binary operators are logical (conjunction, disjunction and negation) and integer (addition, subtraction, multiplication and comparison). The semantics of expressions is defined as expected. For example in Figure 2 , the expression `if writevar == "x" then ()` selects events on

Input

```
909556974571: threadid = 140217848600320
909556974571: readvar = "x"
909556976871: threadid = 140217848600320
909556976871: writevar = "x"
909556981348: threadid = 140217840207616
909556981348: readvar = "y"
909556988517: threadid = 140217840207616
909556988517: readvar = "x"
```

Specification

```
in threadid: Events[Int]
in readvar: Events[String]
in writevar: Events[String]
def read_x := if readvar=="x" then ()
def write_x := if writevar=="x" then ()
def access_x := merge(read_x, write_x)
def thread_accessing_x := on(access_x,threadid)
out thread_accessing_x
```

Output

```
909556974571: thread_accessing_x = 140217848600320
909556976871: thread_accessing_x = 140217848600320
909556988517: thread_accessing_x = 140217840207616
```

Figure 2: A TeSSLa input, specification and output.

the stream `writevar` which match the string `"x"`, and emits the unit value on stream `write_x`.

Built-in operations are the core of TeSSLa language and are used in order to define further functions. Some of the built-in operations used in our example below are `default`, `time` and `last`. The first parameter `a` of the operation `default(a,b)` is a stream, while the second `b` can be either a value or another stream. In the case where `b` is a value and `a` has no value at time 0, the operation adds value `b` at time 0 to the stream `a`. If stream `a` has a value at time 0, the `default` operation returns stream `a` unchanged. In the case where `b` is a stream, the operation `default` returns a stream which is obtained from `a` by prepending the first event from the stream `b`. If the first event of the stream `b` occurs later than the first event of the stream `a`, then the operation returns stream `a` unchanged. The operation `time(a)` returns a stream which has the timestamps of `a` as both timestamps and values. The operation `last(a,b)` returns a stream containing the previous value of the first stream at the time stamps of the second.

In addition to this basic functionality, the current version of TeSSLa is able to generate streams with sets, maps and queues as event values, but can only be executed in a software simulator[1]. Operations on these data structures are defined as expected and built-in. TeSSLa comes with a library of predefined macros (functions), some examples are `on`, `merge`, `mergeValues` and `collectSetWithRemove`. The expression `on(a,b)` is a stream of values from the stream `b` which appear at the same timestamps as the values on the stream `a`. The expression `merge(a,b)` is a stream that has an event of unit type whenever a value on `a` or `b` appears. The expression `mergeValues(a,b)` is a stream containing events from both `a` and `b`. The expression `collectSetWithRemove(a,b)` returns a stream of sets as event values, which are obtained by adding values from `a` and removing values from `b`.

New macros (functions) can be defined by combining basic expressions and built-in macros (functions). New streams are obtained by specifying parameters and combining expressions, built-in operations or newly defined macros. Both, new macros and new streams, are defined by using the keyword `def` followed by an expression. The streams which are about to be a part of the output are declared with the keyword `out` followed by the name of the stream.

Figure 2 shows an example of a TeSSLa trace consisting of streams `threadid`,

[1]Cf. the web-based IDE at `http://tessla.isp.uni-luebeck.de`.

`readvar` and `writevar`. The events are presented in the form of

$$\textit{timestamp: stream name} = \textit{value} \ .$$

The event values on the stream `threadid` are of integer type, while on streams `readvar` and `writevar` we have string values. Notice that events on different streams can have the same timestamps. The first three lines of the specification given in Figure 2, starting with the keyword `in`, declare three input streams. The next two lines of the specification filter read and write accesses to the variable `x`. If we run this specification on the input trace given in Figure 2 , the value of the condition `readvar == "x"` is `true` at the timestamps 909556974571 and 909556988517, so the stream `read_x` would have events at these two timestamps with values `()`, which are the values of unit type. Stream `access_x` is obtained from `read_x` and `write_x` by using a macro `merge` from the standard library. It has events at three timestamps with unit values. Finally, by using the `on` macro, we define an output stream `thread_accessing_x` that has events at the same timestamps as `access_x`, but the values from the stream `threadid`. The output obtained by running the specification on the trace is shown in Figure 2 as well.

Lockset algorithm in TeSSLa

We choose the lockset algorithm since it is commonly used by the state-of-the-art static [12] and dynamic tools [13] and it is suitable for implementation in TeSSLa. In order to implement the basic version of the algorithm in TeSSLa, we have to identify the variables which we will check and the actions taken by the threads. Input to our specification consists of streams which identify thread read/write accesses to shared memory and their lock/unlock actions. In case our specification detects an error, it is useful to issue an informative message about it. Therefore, streams which identify function and line of code being executed are also parts of the input.

Limitations of the TeSSLa language impose some restrictions on the specification of the lockset algorithm as well. In particular, we need to identify the set of locks held by a thread. The proper data structure for that would be one which maps thread identifiers to sets of locks, while TeSSLa is currently equipped only with maps where both key and value have to be of integer type. A similar issue is raised when we need to identify sets of locks guarding variables. One possibility to overcome this is to explicitly specify the variables and number of different threads which we want to check. Another one would be to track a fixed number of variables and threads. The first one is more precise while requiring less interaction with the user. In this work, we have opted to specify exact variables and the upper limit to the number of created threads, while small modifications would allow us to support the second option as well.

We have created a template which requires the variable names and number of threads as parameters. Figure 3 shows the most informative fragment of a specification generated from the template with variable `x` and two threads as parameter values. The number of threads we could derive from inspection of the source code, but in fact we have also developed a static analysis tool (not presented here), which will derive an upper bound on the number of created threads through `pthread_create()`. To this analysis result we add one more thread which represents the main thread of the program. Thus, our specification will actually do the check for three threads, namely `T0`, `T1` and `T2`.

```
-------------------------------------------------------------------------------------------
-------SHARED VARIABLES----------------------------------------------------------------------
-------------------------------------------------------------------------------------------
def read_x := if readvar=="x" then ()
def write_x := if writevar=="x" then ()
def access_x := merge(read_x, write_x)
def thread_accessing_x := on(access_x,threadid)
out thread_accessing_x
-------------------------------------------------------------------------------------------
-------HOLDING LOCKS-------------------------------------------------------------------------
-------------------------------------------------------------------------------------------
def T0_holding_locks := collectSetWithRemove(on(T0,mutexlockaddr),on(T0,mutexunlockaddr))
def T1_holding_locks := collectSetWithRemove(on(T1,mutexlockaddr),on(T1,mutexunlockaddr))
def T2_holding_locks := collectSetWithRemove(on(T2,mutexlockaddr),on(T2,mutexunlockaddr))
-------------------------------------------------------------------------------------------
-------CHECKING x----------------------------------------------------------------------------
-------------------------------------------------------------------------------------------
def T0_holding_locks_accessing_x := last(T0_holding_locks,on(T0,access_x))
def T1_holding_locks_accessing_x := last(T1_holding_locks,on(T1,access_x))
def T2_holding_locks_accessing_x := last(T2_holding_locks,on(T2,access_x))
def holding_x:=mergeValues(T0_holding_locks_accessing_x,T1_holding_locks_accessing_x,
                           T2_holding_locks_accessing_x)
out holding_x
def guarding_x:=default(Set_intersection(last(guarding_x,holding_x), holding_x),holding_x)
out guarding_x
-------------------------------------------------------------------------------------------
def error_x := if guarding_x==Set_empty then ()
-------------------------------------------------------------------------------------------
def error_x_in_line := on(error_x, line)
out error_x_in_line
-------------------------------------------------------------------------------------------
def error_x_in_function := on(error_x,function)
out error_x_in_function
```

Figure 3: A fragment of lockset algorithm implementation in TeSSLa.

First we identify accesses to the shared variable `x` by merging read and write events obtained at the input streams `readvar` and `writevar`. On stream `thread_accessing_x`, we output the identification number of the thread accessing the variable. Next, we identify sets of locks held by each individual thread. These streams can be used for checking not only variable `x`, but also other memory accesses and therefore, we compute them before checking specific variables. The first step in checking variable `x` is to compute the sets of locks held by threads `T0`, `T1` and `T2` while accessing variable `x`. Then we merge values from these three streams into stream `holding_x`. Now, this stream indicates the set of locks held by threads on each access to the variable `x`. We proceed by computing the stream of sets of locks guarding accesses to the variable `x`. The stream `guarding_x` is initialized with the set of locks held by the first thread accessing `x`, by using the `default` operation with the stream `holding_x` as the second parameter. That is slightly different from the original version of the lockset algorithm where the guarding set is initialized to the set of all locks, but does not affect the result. We choose this set as the initializer because identifying the set of all locks would impose additional limitation, namely, it would require us to specify the number of locks to the template, in addition to the variable names and number of threads. The set of guarding locks is being updated on each access to the variable `x` by intersecting the old value with the set of locks held by the thread currently accessing. If the set of locks guarding `x` becomes empty, we issue an event on the stream `error_x`. This event is present on every further access to the variable `x`. We use this stream as a trigger for more informative error messages,

such as which line of the code and which function caused unprotected accesses. Besides streams `thread_accessing_x`, `error_x_in_line` and `error_x_in_functions`, the output of our specification also contains streams `holding_x` and `guarding_x` which carry information about locks involved.

A lockset algorithm without sets. As the overall goal is to execute the stream-based monitoring on the COEMS hardware-based technology, for future integration, we have another hurdle to take: On the FPGAs, dynamic memory that could be used to keep information in data structures, such as the sets and maps in the previous specification, is scarce. To overcome this limitation, we need to instantiate the template even further and essentially also unfold it with regard to the locks.

Specifying the lockset algorithm without advanced data structures imposes further restrictions. In the absence of streams of sets and related operations, locking and unlocking events have to be identified on separate streams for each different lock and the intersection of sets has to be simulated by a more restrictive check. Since it is impossible to build the set of guarding locks, this check confirms if one or more of the identified locks are used by each thread on every access to the observed variable. Therefore, for each lock, it is necessary to define a stream that checks if the lock always protects the variable. Event on the error stream is issued when the values on all these streams indicate that none of the locks is used on each access to the variable. The template for such specification, in addition to the variables and the number of threads, would need to have the number of locks as a parameter as well.

Detecting data races with TeSSLa

To detect data races with our specification, we need to obtain traces from a running program. These traces should include the relevant information that we used in the specification above, such as information about memory accesses and lock operations. While in the future these events will be generated by the hardware monitor developed in the COEMS project, here we use a software-based instrumentation, which works very much along the lines of other runtime checkers, such as ThreadSanitizer: We specifically instrument the code using the LLVM Compiler Infrastructure, carefully choosing the locations where we have to emit events at runtime, so as to avoid redundant events.

Every event that the software instrumentation generates, contains at least the information about source code location, type of access (variable read/write, functioncall, lock or thread operation), and the identity of the executing thread. Depending on the type, the event is augmented with information such as the memory location for a read or write access, any offset and size of the access; lock operations additionally carry the lock identity on a separate stream.

We show a sample usage of the COEMS tools to check for data races in our program, with the help of the `coemseu/d41demo` Docker image that we prepared for this purpose:

```
$ docker run --rm -it -v $PWD:/coems -w /coems coemseu/d41demo /bin/bash
coems# coems_instrument_and_compile.sh example.c
x= 171
coems# /usr/bin/template_sets.tessla -t 2 -s x >data-race-sets.tessla
coems# java -jar /usr/lib/tessla.jar data-race-sets.tessla traces.log
909551503651: holding_x = Set(6308096)
909551503651: guarding_x = Set(6308096)
```

```
909551503651: thread_accessing_x = 140217848600320
909551506178: holding_x = Set(6308096)
909551506178: guarding_x = Set(6308096)
909551506178: thread_accessing_x = 140217848600320
909551553371: holding_x = Set(6308096)
909551553371: guarding_x = Set(6308096)
909551553371: thread_accessing_x = 140217848600320
909551615613: holding_x = Set()
909551615613: guarding_x = Set()
909551615613: error_x_in_line = 19
909551615613: error_x_in_function = "g"
...
```

This interactive session first shows how to enter the Docker container, which we provide with the necessary software packages installed. Inside the container, we next run the software-based instrumentation, which will introduce event generation into the program, compile and run it. The recorded events are in file `traces.log`, subsequent invocations of the binary `./example` would overwrite it with a new trace. Then, we instantiate the general TeSSLa template for the number of threads and the variables that we are interested in.

We use the TeSSLa interpreter with the instantiated set-based specification, and obtain output on intermediate streams, prefixed by the timestamps. Eventually, the specification will uncover the inconsistent locksets (at timestamp 909551615613 above), and print the involved variable, and in which source code location the access occurred (if debugging symbols are available in the compiled binary).

To complete the demonstration, we next instantiate the template that does not use high-level data structures at all, and which is ultimately intended to be run on the processor trace received by the hardware monitor:

```
coems# /usr/bin/template_sets.tessla -t 2 -s x >data-race-nosets.tessla
coems# java -jar /usr/lib/tessla.jar data-race-nosets.tessla traces.log
909551503651: thread_accessing_x = 140217848600320
909551506178: thread_accessing_x = 140217848600320
909551553371: thread_accessing_x = 140217848600320
909551615613: error_x_in_function = "g"
909551615613: error_x_in_line = 19
...
```

Again, we identify the potential race and give information about the location. We conclude by pointing out that while in principle the trace contains a lot of information that can now be used to aid the developer in pinpointing the interfering threads, the final hardware-based solution has only the last *two* values on a stream available. This means that additional recording will be necessary to provide information for debugging, or a separate way of preserving the trace outside of the tracing hardware for offline-debugging.

4 Related Work

Using instrumentation (modifying a program during or after compilation) to introduce additional runtime checks is not a new idea. Valgrind [7] and ThreadSanitizer [13] are commonly used by software engineers in the development process to check for erroneous behaviour, despite their relatively high runtime costs [10]. RVPredict [3] follows a similar approach, and inserts calls to a Java-based verifier during compilation with the help of LLVM and the CLANG compiler. These

purely software-based solutions work with arbitrary dynamic resource creation, i.e., they are not limited in the number of monitored threads, locks and locations, as we are in our resource-constrained setting. On the other hand, they will encounter a growing overhead for runtime checking as the number of tracked elements grows.

Other runtime-based techniques that need to instrument the program include for example coverage analysis, where detailed information about conditionals needs to be recorded during execution. Such information cannot be seen from the outside, or requires heavy intervention through a debugger.

The runtime based approach can naturally only make statements about actual observations. Alternatives to runtime checks are of course static analyses, which inspect the source code only. For instance, Goblint [12] is a path- and context-sensitive analysis, which uses global invariants to calculate locksets and thread interleavings to detect data races in multithreaded C programs. Another static approach, by Kidd et al. [4], generates an abstraction of a program to check for data races in concurrent Java programs, by abstracting an unlimited number of Java objects into a finite set of abstract ones whose locks are binary.

The earlier work of the authors [9] presented a static analysis in the form of an inference algorithm to the end of static deadlock detection. The context-sensitive analysis summarizes locks based on their creation-site. This approach could be seen as an improvement to the underlying static analysis that we have implemented here to derive the number of threads and locks.

An important general difference between our approach and the dynamic and static approaches above is that we use an intermediate specification languages. On the one hand this increases flexibility, as we should be easily able to cover different properties within the same framework. On the other hand, this flexibility comes at the cost of less optimization for a particular purpose. Some of the known optimizations related to event generation, we should be able to put into use in the instrumentation-phase, though.

On the side of theory, static checkers aim to be sound: they (should) faithfully report *potential* problems in the code, without missing real races. In practice, especially for the C language, many tools are not sound [2]. The reason for this is a deliberate tradeoff between soundness and precision, so as to avoid drowning developers in a flood of false positives that stem from defensive assumptions in the abstraction. There is a growing consensus that soundness may not practical in real-world software engineering, and that instead analyses should aim for *soundy* [6], giving *useful* results and insights to developers.

We are facing similar challenges, for example in our interpretation of calls to library functions, which may or may not have side effects that could effect the outcome. Unfortunately, very often these libraries are taken as given, and no source code is available for analysis or instrumentation.

5 Conclusions and Future Work

We have presented our approach to checking for dynamic data races using a lockset algorithm. Unlike other dynamic approaches, the algorithm has been encoded in the stream-based specification language TeSSLa, and an augmented instruction trace of the running application is checked against the lockset specification. As the ultimate goal of the project is checking such traces in hardware (due to the high number of events, and the performance impact of dynamic monitoring), we presented the

specification in two variations: one using data structures such as maps and sets, and another, unfolded, specification which does not use dynamically growing data structures, but places an upper limit on the number of threads and locks that can be observed.

Future Work. In the COEMS project, the overall goal is to avoid costly instrumentations and runtime penalties. The FPGA-based monitoring hardware for ARM-based SoCs has been prototyped, and partner are currently working on the translation of TeSSLa specifications that will parametrize the verification logic on the FPGA. Due to novel observation features such as ARM's CoreSight and Intel's upcoming IntelPT processor trace technology, developers will soon have the chance of monitoring running programs with minimal interference. As these observation techniques literally observe execution steps by the CPU, the high number of generated events must at least be preprocessed in hardware: the trace must be reconstructed from a highly compressed format, and does not include additional arguments or values in addition to the program counter. Some instrumentation will be necessary to introduce this additionally required data into the trace, although care must be taken not to introduce additional computation which would affect the execution time of the program.

The techniques that we have used here aim exactly for this minimal instrumentation, which generates just the events necessary to sound*ly* monitor the given property. While development of the hardware is ongoing in the project, we have demonstrated the feasibility of our approach with a software-based monitoring, which uses a simulator to implement monitoring of the specification. Due to the limited memory for dynamic data such as the locksets on the FPGA, in practice a more refined specification is necessary that limits the observation to the relevant code locations. We are currently developing such an analysis which aims to narrow down the instrumentation, effectively leading to a fruitful interplay of static and dynamic analysis.

A future task will also be providing further additional help to developers to interpret the results. Again, in practice the challenge will be in using the limited amount of data that can be stored in the hardware-based solution. We will consider implementation of improvements and optimizations, present already in Eraser [11], as well as the hybrid algorithm of ThreadSanitizer [13].

The experiment described above can be reproduced with the Docker image `coemseu/d41demo`. Further description and necessary files are available from `https://www.coems.eu/set-based-data-race-detection/`.

References

[1] E. G. Coffman Jr., M. Elphick, and A. Shoshani. System deadlocks. *Computing Surveys*, 3(2):67–78, June 1971.

[2] Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. Machine-learning-guided selectively unsound static analysis. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 519–529, Piscataway, NJ, USA, 2017. IEEE Press.

[3] Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. Maximal sound predictive race detection with control flow abstraction. In *Proc. of the 35th*

ACM SIGPLAN Conf. on Programming Language Design and Implementation, PLDI '14, pages 337–348, New York, NY, USA, 2014. ACM. `https://runtimeverification.com/predict/`.

[4] Nicholas Kidd, Thomas W. Reps, Julian Dolby, and Mandana Vaziri. Finding concurrency-related bugs using random isolation. *STTT*, 13(6):495–518, 2011.

[5] Martin Leucker, César Sánchez, Torben Scheffel, Malte Schmitz, and Alexander Schramm. TeSSLa: Runtime verification of non-synchronized real-time streams. In *ACM Symposium on Applied Computing (SAC)*. ACM, 2018.

[6] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: A manifesto. *Commun. ACM*, 58(2):44–46, January 2015.

[7] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In Jeanne Ferrante and Kathryn S. McKinley, editors, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 89–100. ACM, 2007.

[8] Robert H. B. Netzer and Barton P. Miller. What are race conditions? *ACM Letters on Programming Languages and Systems*, 1(1), March 1992.

[9] Ka I Pun, Martin Steffen, and Volker Stolz. Effect-polymorphic behaviour inference for deadlock checking. *Journal of Logical and Algebraic Methods in Programming*, 85(6):1234 – 1267, 2016.

[10] Caitlin Sadowski and Jaeheon Yi. How developers use data race detection tools. In Joshua Sunshine, Thomas D. LaToza, and Craig Anslow, editors, *Proc. of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools*, pages 43–51. ACM, 2014.

[11] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[12] Helmut Seidl and Vesal Vojdani. Region analysis for race detection. In Jens Palsberg and Zhendong Su, editors, *Proceedings of 16th International Symposium on Static Analysis (SAS)*, volume 5673 of *Lecture Notes in Computer Science*, pages 171–187. Springer, 2009.

[13] Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitriy Vyukov. Dynamic race detection with LLVM compiler - compile-time instrumentation for threadsanitizer. In Sarfraz Khurshid and Koushik Sen, editors, *2nd. Intl. Conf. on Runtime Verification , RV 2011*, volume 7186 of *Lecture Notes in Computer Science*, pages 110–114. Springer, 2011.