López-Gómez, J., et al. Detecting semantic violations of lock-free data structures through C++ contracts, In: *The Journal of Supercomputing, 26 March 2019, 22 pp.*

DOI: https://doi.org/10.1007/s11227-019-02827-4

Detecting semantic violations of lock-free data structures through C++ contracts

Javier López-Gómez , David del Rio Astorga , Manuel F. Dolz , Javier Fernández , J. Daniel García

University Carlos III of Madrid, Spain
drio@pa.uc3m.es, {jalopezg,jfmunoz,jdgarcia}@inf.uc3m.es, dolzm@icc.uji.es

Abstract

The use of synchronization mechanisms in multithreaded applications is essential on shared-memory multi-core architectures. However, debugging parallel applications to avoid potential failures, such as data races or deadlocks, can be challenging. Race detectors are key to spot such concurrency bugs, nevertheless, if lock-free data structures are used, these may emit a significant number of false positives. In this paper, we present a framework for semantic violation detection of lock-free data structures which makes use of contracts, a novel feature of the upcoming C++20, and a customized version of the ThreadSanitizer race detector. We evaluate the detection accuracy of the framework in terms of false positives and false negatives. Thanks to this framework, we are able to check the correct use of lock-free data structures, thus reducing the number of false positives.

Keywords Parallel programming, Semantic violation detection, C++ contracts, Lock-free data structures

Formal publication `https://doi.org/10.1007/s11227-019-02827-4`

I. Introduction

Since the multi-/many core processors became mainstream in HPC platforms, the evolution of parallel programming models has allowed applications to progressively exploit their computational resources [3]. Parallel frameworks are designed to provide abstraction layers to applications in form of library interfaces, which avoid the direct use of low-level concurrency mechanisms. To achieve this goal, the usage of building blocks implementing core functionalities has been the *de facto* engineering methodology in many programming frameworks [5]. Such "software blocks" should be designed to ensure correctness and thread-safety in order to produce correct global results.

In this sense, parallel programming may entail unexpected concurrency errors, e.g. data races or deadlocks [4]. Discovering such concurrency bugs has been recognized as a complex task, given that these errors may only occur in low-probability event orderings and depend on external factors, e.g. machine load. These facts make data races extremely sensitive in terms of time, scheduling policies, compiler options, memory models, etc. Although data race detectors have alleviated the debugging task, there is still room for improvement [4, 1]. In particular, the use of lock-free data structures can generate false positives. This fact hinders developer's vision in finding harmful races and makes the debugging process even harder when tracing back the root cause of the problem. Furthermore, current race detectors are not able to detect semantic violations of lock-free data structures. In this line, the semantics of the Single-Producer/Single-Consumer (SPSC) lock-free queue have already been embedded into the ThreadSanitizer (`TSan`) race detector tool [6]. However, those semantic rules were hard-coded in the detector and could not be generalized to other lock-free structures.

In this paper, we extend the previous work with a novel semantic violation detection framework which makes use of C++ contracts to allow users to express semantic rules of other lock-free data structures. Specifically, this work contributes to the following:

- We present a syntactic mechanism based on C++ contracts to annotate the semantics of lock-free data structures.
- We extend both the Clang compiler and the `TSan` race detector tool in order to support the proposed syntax and to perform the semantic violation detection.
- We evaluate the detection accuracy of the proposed framework using the SPSC and MPMC lock-free queue structures from the Boost C++ library.

This paper is organized as follows: Section II revisits some related work in the area. Section III describes the main software components used in this work. Section IV describes the semantic violation detection framework based on C++ contracts. Section V formalizes the general semantics of the SPSC lock-free queue and describes a worked example of the framework using the Boost C++ SPSC and MPMC lock-free queues. Section VI evaluates the detection accuracy of the semantic violation detection using synthetic benchmarks of the C++ Boost library. Finally, Section VII provides some concluding remarks and future works.

II. Related work

Over the years, numerous solutions to detect data races have been proposed [16]. These, have been based on different well-known mechanisms: *i) happens-before* relations, *ii) locksets*, and *iii) hybrid* approaches, i.e., combining the previous two.

Basically, happens-before relations [12] are used as a mechanism to detect potential data races in concurrent memory accesses. For instance, Intel Inspector [10] and Acculock [27] are two well-known commercial and research tools making use of this mechanism. However, software implementations of *happens-before*-based detectors typically suffer from large run-time overheads, so hardware-based

solutions have also been proposed to overcome these issues [28]. Alternatively, the *locksets* approach reports a data race if there is no common lock held by two threads accessing the same memory address. This approach can be found in both static [8] and dynamic tools [23] from the state-of-the-art. Finally, *hybrid* approaches take advantage of *happens-before* mechanisms to reduce the false positives reported by *lockset*-based detectors and preserve the performance advantages of the *lockset* mechanisms. A race detector implementing this approach is ThreadSanitizer [25].

Although previous mentioned tools aid developers to find concurrency bugs, these can still miss ad-hoc synchronizations, and therefore generate false positive warning reports. To face these issues, Norris et al. [18, 19] provide an advanced dynamic partial-order reduction algorithm implemented within CDSchecker, a tool that detects potential data races by virtually executing a wide range of possible thread orderings. An advantage of this tool is that it is able to handle C++ atomics along with the different C++11 memory models.

Despite the wide literature on data race detection techniques, we only find the work by Ou et al. [21] to be able to directly handle the semantics of lock-free data structures within the CDSchecker with special annotations. In this work, we present a similar approach to detect semantic violations for this kind of data structures which makes use of the upcoming C++20 contracts programming features to express semantic constraints.

III. Background

In this section, we give an overview of the main software components that have been used to carry out the contributions in this paper. Specifically, we review C++ contracts, the LLVM infrastructure along with the TSan race detector, and some basic concepts about lock-free structures.

III.1 C++ contracts

Contracts are a new feature from the C++20 draft specification aiming to improve software correctness [22]. They allow users to state assertions as well as to specify predicates on functions interfaces in the form of preconditions and postconditions.

assertions may appear at any place where an executable statement is valid and they specify a predicate that is assumed to hold at its point in the computation. We use the attribute `assert` to state assertions.

preconditions are part of a function declaration and they express expectations on the function's arguments and/or the state of other objects using a predicate that is intended to hold upon entry into the function. We use the attribute `expects` to state preconditions.

postconditions are also part of a function declaration and they express conditions that the function should ensure for the return value and/or the state of objects using a predicate that is intended to hold upon exit from the function. We use the attribute `ensures` to state postconditions.

A contract attribute may include an assertion level (`default`, `audit`, or `axiom`) to express the assertion level of the contract. Those levels indicate relative costs of checks so that they can be enabled or disabled at build time.

To strengthen the understanding of subsequent explanations, we use a code example that uses contracts to ensure the correct behavior of the annotated functions. Listing 1 shows a simple C++ program annotated with contracts and its simplified LLVM Intermediate Representation (IR). Function `main` incorporates an assertion to check the `argc` value. In the IR, this contract is translated into the `icmp`–`br` instruction pair (lines 17-18), which causes a conditional jump to terminate execution if the predicate is evaluated to false. Afterwards, we find a call to the function `foo`, whose prototype has been annotated with a precondition and a postcondition. Basically, the precondition allows checking whether the value of `b` is greater than `a`, while the postcondition is intended to ensure that the return value is greater or equal than 1. Note that `ensures` attribute is accompanied by the `default` assertion level. Similar to the previous assertion contract, those conditions have been translated into conditional jumps in the IR.

In general, contracts have no observable effects in a correct program, beyond performance differences. Contracts are checked at run-time and its default action is to terminate execution when they are violated. In this paper, we use the `axiom` assertion level for annotating and formalizing semantics, as this level is intended to be used for expressing user-defined conditions that are not checked at run-time, but might be used by static analyzers or compiler plugins.

Listing 1: Example of C++ contracts.

(a) Contract-annotated code.

```
1  int foo(int a, int b)
2    [[expects: b > a]]
3    [[ensures default ret: ret
         >= 1]];
4
5
6  // the compiler shall
       generate code to
7  // evaluate the previous
       conditions (see
8  // the IR listing, function
       @_Z3fooii)
9  int foo(int a, int b)
10 {
11   return a + b;
12 }
13
14
15
16
17 int main(int argc, char *argv
       [])
18 {
19   // matches IR icmp/br at
         lines 17-18
20   [[assert: argc >= 1]];
21   foo(0, argc);
22
23   return 0;
24 }
```

(b) Simplified IR translation.

```
define i32 @_Z3fooii(i32 %a,
     i32 %b) {
  %cmp = icmp sgt i32 %b, %a
  br i1 %cmp, label %ok.0,
       label %fail.0
fail.0:
  tail call void @std::
       terminate()
ok.0:
  %add.i = add nsw i32 %b, %a
  %cmp1 = icmp sgt i32 %add.i,
       0
  br i1 %cmp1, label %ok.1,
       label %fail.1
fail.1:
  tail call void @std::
       terminate()
ok.1:
  ret i32 %add.i
}

define i32 @main(i32 %argc,
     i8** %argv) {
  %cmp = icmp sgt i32 %argc,
       0
  br i1 %cmp, label %ok.0,
       label %fail.0
  tail call void @_Z3fooii()
  ; ...
}
```

III.2 The LLVM infrastructure and the TSan data race detection tool

The LLVM (Low-Level Virtual Machine) is a compiler infrastructure designed to be a set of reusable libraries with well-designed interfaces [14]. Its Clang front-end generates intermediate code that is afterward converted into machine-dependent assembly code for a specific target platform. Thanks to its high-level API, LLVM provides the ability to develop and integrate new modules in order to perform compile-time analysis and instrumentation. Taking advantage of the latter feature, several run-time checks and tools have been developed to identify suspicious and undefined behavior of threads. One of them is ThreadSanitizer (TSan), a data race detector for applications written in C/C++ or Go that uses compile-time instrumentation to check for non-race-free memory accesses at run-time [25].

TSan instrumentation tracks synchronization primitives, thread routines from `libpthread`, memory allocation routines, dynamic annotations and other kinds of functions that lead to synchronizations. Its runtime library provides entry points for the instrumented code to keep all the information that is of interest for the race detector. With all these data, two race detection mechanisms based on *happens-before* and *locksets* relations are applied. As a summary of [20], these mechanisms develop the following strategies:

happens-before relations detect a potential data race when two events *a* and *b* access a shared memory location, where at least one of these accesses is a write, and neither *a happens-before b* nor *b happens-before a*. In other words, they are concurrent, so no causal relationship ordering exists between *a* and *b* [12].

locksets determine a data race when none of the locks held by a pair of events accessing to a shared memory location are the same, with at least one of these accesses being a write; i.e. when the intersection of their locksets is empty.

Contrary to other race detectors, the TSan detector can be switched to work only with the *happens-before* mechanism, also known as *pure happens-before*, or with a combination of both previous mechanisms, referred to as the *hybrid* mode [25]. While in the first mode the *concurrency* is only checked in terms of *happens-before* relations, in the *hybrid* mode both *happens-before* and *locksets* mechanisms are used to determine whether two events are concurrent.

In summary, the main reasons to extend TSan with high-level semantic violation detection are: *i)* it employs compile-time instrumentation, making it much faster than other solutions; *ii)* it is built on top of the LLVM infrastructure, being, therefore, an open-source software capable of accommodating new functionalities.

III.3 Lock-/Wait-free buffers

In general, concurrent data structures can be classified as either *blocking* or *non-blocking*. Non-blocking structures ensure thread-safety bypassing the use of traditional synchronization primitives, such as locks or mutexes. Lock-free is a level of progress guarantee for non-blocking data structures. A concurrent data structure is considered lock-free if there is guaranteed system-wide progress, i.e. at least a thread makes progress on its execution [7].

The absence of synchronization mechanisms allows better performance since no explicit waiting primitives are needed. However, some constructs may require atomic operations, so that no intermediate states can be seen by other executing threads. Internally, these atomic operations can be seen as a combination of both *load* and *store* instructions. Lock-free data structures, such as queues [17], hash tables [26] and SPSC buffers [9], are typically known to make use of these type of atomic instructions. Nevertheless, lock/wait-free data structures are significantly more complex to implement, and consequently to verify their correctness, with respect to lock-based structures. Apart from this fact, race detection tools are unable to properly determine whether a race is an actual error or a false positive due to the lack of synchronization mechanisms detectable by these tools.

In this paper, we contribute to improving the detection of semantic violations in lock-free data structures. To do so, we use C++ contracts to specify at user-code level the semantics of such lock-free structures and extend the TSan race detector with a module for semantic violation detection able to handle the contracts placed on the user code.

IV. The contract-based semantic violation detection framework

This section introduces the contract-based semantic violation detection framework (CSV) as the main contribution of this paper[5]. This framework is comprised of the contract-based interface and the TSan extension for semantic violation detection.

Figure 1 depicts the CSV framework on an application whose code has been annotated with the proposed contract interface for semantic violation detection. The user code implementing a lock-free structure has been annotated with the CSV interface defined in the `csv.h` header file, which provides the required declarations for specifying the semantic rules for such structures. Afterwards, the user code is compiled with a Clang frontend that includes support for C++ contracts and linked against a customized TSan version for semantic violation detection (using the command line option `-fsanitize=thread`). Finally, the build process yields an ELF executable statically linked against the modified TSan library, that includes additional instructions to evaluate the contract-based checks. Specifically, the TSan library has been extended with the implementation of functions declared in `csv.h`, which are invoked as part of semantic checking each time a contract-annotated function is called from the user code. Internally, violation detection is performed using Lamport clocks managed by TSan.

In the following sections, we explain in detail the two main components of CSV: the contract-based semantic interface and the extended TSan version for verifying the semantics stated in the contracts. Subsequently, we illustrate through a worked example of the Boost C++ Single-Producer/Single-Consumer lock-free queue how member functions of this data structure have been annotated using contracts and how the semantics are checked each time these functions are called.

IV.1 Contract-based semantics interface

The required declarations for contract-based checking of lock-free data structures is placed in the header file `csv.h`, which should be

[5]The Clang fork supporting C++ contracts has been open-sourced and accesible at `https://github.com/arcosuc3m/clang-contracts/`. CSV can be found in branch `CSV-src`.

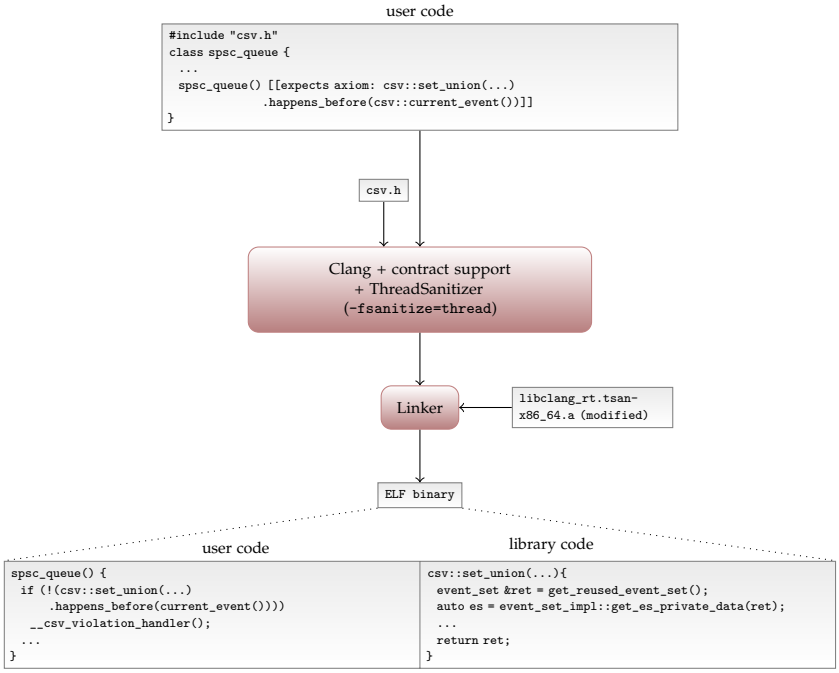

Figure 1: Workflow of the CSV framework.

included in the user code before referencing any of its functions. Listing 2 shows the proposed header file `csv.h`. Execution events of a given thread are represented by class `event`. They are used to establish temporal relations via Lamport clocks. Execution events are grouped in `event_set` objects. Both, event sets and events offer predicate operations to check *happens-before* relations: `happens_before()` and `concurrent()`. Two events are concurrent when none of them can be proven to *happen-before* the other.

Listing 2: CSV header file.

```
#include <contract>

namespace csv {
  class event_set;
  // Execution event
  class event {
   public:
    bool happens_before(const event_set &evs) const;
   private:
    // ... private members omitted
  };
  // Class for handling sets of events
  class event_set {
   public:
    event_set();
    ~event_set();

    std::size_t size() const;
    bool empty() const { return size() == 0; }
    void add_event(const event &ev);
    // Check if this event set happens-before another event or
         event set
    bool happens_before(const event &ev) const;
    bool happens_before(const event_set &evs) const;
    // Check if this event set is concurrent with another event
         of event set
    bool concurrent(const event &ev) const
      { return !happens_before(ev) && !ev.happens_before(*this);
           }
    bool concurrent(const event_set &evs) const
      { return !happens_before(evs) && !evs.happens_before(*this)
           ; }
   private:
    // ... private members omitted
  };
  // Get temporary reference to current event. It may be stored
       in a event_set.
  event &current_event();
  // Calculate set union and intersection
  template <class = void>
  event_set &set_union(const event_set &, const event_set &);
  template <class = void>
  event_set &set_intersection(const event_set &, const event_set
       &);

  template <typename... Ts>
  event_set &set_union(const event_set &a, const event_set &b,
       const Ts &...u)
    { return set_union(a, set_union(b, u...)); }
  template <typename... Ts>
  event_set &set_intersection(const event_set &a, const
       event_set &b, const Ts &...u)
    { return set_intersection(a, set_intersection(b, u...)); }
}
extern "C" { void __csv_violation_handler(const std::
     contract_violation &cv); }
```

Accessing the current event is a key feature required for the semantic checking. For this purpose, the `current_event()` function is used to get a reference to the current event that can be used as part of a semantic rule.

To operate with event sets, we define a series of functions that implement set operations (`set_union()` and `set_intersection()`). These functions take as argument a variadic list of sets. Because temporary sets returned by `csv::set_union()` and `csv::set_intersection()` are reused from a pool of objects in the Thread Local Storage (TLS) area, no additional memory management is required. Additionally, an `event_set` permits to check for emptiness (member function `empty()`). The previous functions are enough to write complex rules that involve temporal relations.

Finally, `event_set` provides an operation for adding an event to a specific set through the member function `add_event()`. All in all, the functions in this header file enable semantic rules in lock-free structures, while their implementation is part of the TSan static library.

IV.2 TSan extension

The counterpart of the CSV framework is the extension made in the TSan library for performing semantic violation detection. Basically, this extension implements the functions declared in the `csv.h` header file. These functions leverage the TSan internals in order to access the Lamport clocks managed by the detector. Thus, when a contract-annotated member function is processed by our modified Clang compiler, the contracts are transformed into statements including calls to the actual CSV functions. Besides, the TSan instrumentation is disabled as if the `__attribute__((no_sanitize("thread")))` was specified [25].

The semantic violation detection process makes use of the two aforementioned data structures of the CSV interface: the event and the set. The event structure contains the values of the Lamport clock related to the caller thread at a given point in time. On the other hand, the set stores copies of previous events of the same type. However, given that events occurring in the same thread are totally ordered, only the last needs to be stored in a set. This allows for faster *happens-before* comparisons among events from different threads. Moreover, these events are updated by means of the `current_event()` and the `add_event()` functions. The former returns a reference to the current event object, while the latter copies the current event into the specified set. Afterwards, the `happens_before()` function may be used to check whether the events in a set meet the necessary conditions by comparing the stored clocks.

V. Lock-free queues use cases

In this section, we employ the SPSC concurrent queue as a lock-free data structure to demonstrate the workings of the CSV framework. We have focused on this structure as it is an illustrative example commonly used on shared-memory architectures to implement 1-to-1 communication channels [9].

V.1 Formal definition and semantics of the SPSC queue

Consider a queue $\mathcal{Q}$ the tuple $\{buf, pread, pwrite\}$, where buf is the internal buffer and *pread* and *pwrite* are internal atomic read and write pointers to buf, respectively. This queue provides the following member functions:

`init` Initializes the buffer, allocating memory and setting the internal pointers to 0. If the buffer has already been allocated, `init` only resets the pointers.

`push` Enqueues an item to the buffer.

`pop` Dequeues an item from the buffer.

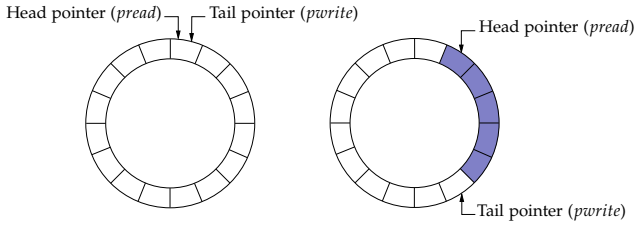

Figure 2: SPSC queue circular buffer.

Note that, depending on the internal implementation of a particular queue, the buffer can be represented in different ways. For instance, in an SPSC bounded queue, a circular buffer may be used. Figure 2 depicts the internal workings of the SPSC queue circular buffer. Initially *pread* and *pwrite* point to the initial position of the buffer. Afterwards, some elements have been added at the end of the buffer through `push` calls while others have been removed from the head by means of `pop` calls.

The correctness of parallel lock-free SPSC queues, such as the Lamport [13] or FastForward implementations [9], is only ensured if several usage requirements are met. Basically, we define these requirements as the following semantic rules:

1. A lock-free concurrent SPSC queue instance can be shared by multiple entities acting as initializers, producers, and consumers.

2. A certain entity can perform any role, however, at any point in time, there must only exist, at most, one producer and one consumer performing operations on the same queue concurrently.

3. An initializer cannot operate over the queue concurrently with any other entity.

In any other case, the semantics of the queue are violated, thus leading to undefined behavior due to the occurrence of potential data races.

To formalize the aforementioned semantics we define an event as the invocation of a function at a certain point in time performed by an entity. In our particular case, we distinguish among three different event types: production (`push`), consumption (`pop`), and initialization (`init`) and denote them as p, c and i, respectively. Also, each time these events occur they are accordingly stored in the sets *Prod*, *Cons* and *Init*.

With these definitions, it is possible to control the proper use of the lock-free SPSC queue by checking three simple requirements depending on the type of a new incoming event. These requirements, defined in Req. (1), (2) and (3), are checked each time a new production (p'), consumption (c') or initialization (i') event occurs, respectively. Assuming that there has been at least an initialization event, Req. (1) ensures that the new production event has a *happens-before*

relation ($\rightarrow$) with all past initialization events and is not concurrent ($\not\approx$) with all past production events. Similarly, Req. (2) performs the same violation detection, but for incoming consumption events.

$$p',\ \forall i \in Init,\ \forall p \in Prod:\ Init \neq \varnothing\ \wedge\ i\ \rightarrow\ p'\ \wedge\ p\ \not\approx\ p' \quad (1)$$

$$c',\ \forall i \in Init,\ \forall c \in Cons:\ Init \neq \varnothing\ \wedge\ i\ \rightarrow\ c'\ \wedge\ c\ \not\approx\ c' \quad (2)$$

Additionally, Req. (3) ensures that all initialization events happened sequentially with any other event. That is, all past events of $\mathcal{Q}$ are not concurrent with the new initialization event i'. If this requirement is not met at some point, it might be that the queue has not been properly initialized, and therefore it can lead to undefined behavior.

$$i',\ \forall e \in Init\ \cup\ Prod\ \cup\ Cons:\ e\ \not\approx\ i' \quad (3)$$

V.2 Practical use cases

In this section we leverage the SPSC and MPMC lock-free queues from the Boost C++ library to illustrate how their member functions are annotated with the proposed CSV contract interface according to the semantic rules of these structures.

V.2.1 The SPSC lock-free queue

The first practical use case leverages the Boost C++ SPSC lock-free queue (`boost::lockfree::spsc_queue`) [2]. Before annotating member class functions, we first analyze the set of available functions in the data structure to check whether the general semantics proposed in the previous section can be used directly. According to the documentation, we find the presence of non-thread-safe functions (such as `reset`) which cannot be called concurrently to any other function. To handle such non-thread-safe functions in the general semantics, we add the set NTS which will contain events of type n related to such non-thread-safe function calls. With this, we can now classify the member functions of this particular implementation as follows:

$$Prod = \{\texttt{push}, \texttt{write\_available}\}$$
$$Cons = \{\texttt{pop}, \texttt{read\_available}, \texttt{front}, \texttt{consume\_one}, \texttt{consume\_all}\}$$
$$Init = \{\texttt{ctor}\}$$
$$NTS = \{\texttt{reset}\}$$

Next, we extend the general SPSC semantic rules to ensure that a non-thread safe event is not concurrent with a new production, consumption or initialization event. This is achieved by adding the constraint $\forall n \in NTS : n\ \not\approx\ e'$ in the requirements Req. (1), (2), and (3), being e' a new production, consumption or initialization event. For instance, the requirement Req. (1) will result as:

$$\begin{aligned} &p',\ \forall i \in Init,\ \forall p \in Prod,\ \forall n \in NTS: \\ &Init \neq \varnothing\ \wedge\ i\ \rightarrow\ p'\ \wedge\ p\ \not\approx\ p'\ \wedge\ n\ \not\approx\ p' \end{aligned} \quad (4)$$

To complete the semantic rules, it is also needed to ensure that all non-thread-safe events are not concurrent with other previous events. This constraint is defined with the requirement Req. (5):

$$n',\ \forall e \in Init\ \cup\ Prod\ \cup\ Cons\ \cup\ NTS:\ e\ \not\approx\ n' \quad (5)$$

Having the revised semantic rules for the Boost C++ SPSC queue implementation, it is possible to annotate the member functions with the CSV interface. Listing 3 shows SPSC queue class declaration with the member functions accordingly annotated with the revised semantic rules. Note that the class definition has been annotated with the new attribute `[[csv::checked]]` which automatically disables TSan instrumentation for member functions annotated with a CSV contract as if the attribute `__attribute__((no_sanitize("thread")))` was specified.

Listing 3: Boost C++ SPSC queue code with CSV annotations.

```
1  // In header: <boost/lockfree/spsc_queue.hpp>
2  #include "csv.h"
3
4  template<typename T, typename... Options>
5  class [[csv::checked]] spsc_queue {
6  private:
7    [[csv::event_sets(init_events, prod_events, cons_events,
         nts_events)]];
8  public:
9    ...
10   spsc_queue(void)
11     [[expects axiom:
12       csv::set_union(init_events, prod_events, cons_events,
             nts_events)
13         .happens_before(csv::current_event())]]
14     [[csv::add_current(init_events)]];
15
16   bool push(T const &)
17     [[expects axiom:
18       !init_events.empty() && init_events.happens_before(csv::
             current_event())]]
19     [[expects axiom:
20       !prod_events.concurrent(csv::current_event())]]
21     [[csv::add_current(prod_events)]];
22
23   bool pop()
24     [[expects axiom:
25       !init_events.empty() && init_events.happens_before(csv::
             current_event())]]
26     [[expects axiom:
27       !cons_events.concurrent(csv::current_event())]]
28     [[csv::add_current(cons_events)]];
29   ...
30   void reset(void)
31     [[expects axiom:
32       !csv::set_union(init_events, prod_events, cons_events,
             nts_events)
33         .concurrent(csv::current_event())]]
34     [[csv::add_current(nts_events)]];
35 };
```

The queue also includes the `[[csv::event_sets]]` attribute, which introduces a number of identifiers. For each identifier introduced, we add a new data member to the class of type `event_set`. In the case of this queue, we use four sets to keep track of `init`, `prod`, `cons` and `nts` events. New events are added to sets either using the attribute `[[csv::add_current]]` or calling the `csv::add_current()` function as part of a contract, which adds the current event to the `event_set` specified as argument.

As observed, the `push` function has been annotated with the contracts expressing the semantic requirement Req. (4). Similarly, the

pop, init and reset functions have also been annotated to express the corresponding Req. (2), (3) and (5), respectively. Once compiled, our modified Clang C++ compiler supporting contracts transforms this function into the corresponding IR (see Listing 4). Additionally, the above mentioned [[csv::add_current]] results in the insertion of invocations to the corresponding add_event() member function of the corresponding set, where the result of current_event() is passed.

Listing 4: Boost C++ SPSC queue simplified LLVM IR code.

```
define void @boost::lockfree::spsc_queue::ctor(%this) {
  %union = invoke @csv::set_union(%init_events, %prod_events, %
       cons_events, %nts_events)
  %current = invoke @csv::current_event()
  %HB = invoke @csv::event_set::happens_before(%union, %current)
  br i1 %HB, label %ok, label %fail
fail:
  invoke @__csv_contract_violation()
ok:
  invoke @csv::event_set::add_event(%init_events, %current)
  ; ...
}
define void @boost::lockfree::spsc_queue::push(%this) {
  %empty = invoke @csv::event_set::empty(%init_events)
  br i1 %empty, label %fail, label %l0
l0:
  %current = invoke @csv::current_event()
  %HB = invoke @csv::event_set::happens_before(%init_events, %
       current)
  br i1 %HB, label %l1, label %fail
l1:
  %concurr = invoke @csv::event_set::concurrent(%prod_events, %
       current)
  br i1 %concurr, label %fail, label %ok
fail:
  invoke @__csv_contract_violation()
ok:
  invoke @csv::event_set::add_event(%prod_events, %current)
  ; ...
}
define void @boost::lockfree::spsc_queue::pop(%this) {
  ; similar to 'boost::lockfree::spsc_queue::push()', but using
       cons_events
  ; ...
}
define void @boost::lockfree::spsc_queue::reset(%this) {
  %union = invoke @csv::set_union(%init_events, %prod_events, %
       cons_events, %nts_events)
  %current = invoke @csv::current_event()
  %concurr = invoke @csv::event_set::concurrent(%union, %current
       )
  br i1 %concurr, label %fail, label %ok
fail:
  invoke @__csv_contract_violation()
ok:
  invoke @csv::event_set::add_event(%nts_events, %current)
  ; ...
}
```

V.2.2 The MPMC lock-free queue

The second practical use case employs the Boost C++ MPMC lock-free queue (boost::lockfree::queue) [2]. For the sake of limited space, we depart from the general semantics of the MPMC queue described in our previous work [6]. To adapt the MPMC queue semantics to this specific use case, we define a single set for storing events from thread-safe function calls, namely TS. According to the definition of MPMC queue, this structure can be used indistinguishably by multiple producers and consumers concurrently; thus, the single set TS is enough to store such thread-safe events. Similar to the SPSC queue, we also encounter member functions that are non-thread-safe. To store these events, we declare the set NTS. Finally, we define the $Init$ set for events related to the class constructor. This set will allow to check whether the queue is properly initialized before its use. With this, we can classify the MPMC queue member functions as follows:

$$TS = \{\texttt{pop}, \texttt{reserve}, \texttt{consume\_one}, \texttt{consume\_all}, \texttt{push}, \texttt{bounded\_push}, \texttt{empty}\}$$
$$NTS = \{\texttt{unsynchronized\_pop}, \texttt{unsynchronized\_push}, \texttt{reserve\_unsafe}\}$$
$$Init = \{\texttt{ctor}\}$$

Next, we define the following semantic requirements that are checked each time a new thread-safe, non-thread-safe or initialization event occurs. Req. (6) ensures that a new thread-safe event (s') is not executed concurrently with previous non-thread-safe event. Similarly, Req. 7 checks whether a new non-thread-safe event (n') is not executed concurrently with any other events. Both Req. (6) and (7) also guarantee that the queue is properly initialized before its use.

$$s',\ \forall i \in Init,\ \forall n \in NTS:\ Init \neq \varnothing \ \wedge\ i \ \rightarrow\ s' \ \wedge\ n \not\approx s' \tag{6}$$

$$n',\ \forall i \in Init,\ \forall e \in TS \cup NTS:\ Init \neq \varnothing \ \wedge\ i \ \rightarrow\ n' \ \wedge\ e \not\approx n' \tag{7}$$

Finally, Req. 8 checks whether initialization events are not executed concurrently with any other previous operation performed over the same queue.

$$i',\ \forall e \in Init \ \cup\ NTS \ \cup\ TS:\ e \not\approx i' \tag{8}$$

Similar to the CSV annotations performed on the SPSC queue code in Section V.2.1, the Boost C++ MPMC queue member functions have been accordingly annotated following the requirements stated in Req. (6), (7) and (8) for experimental evaluation. However, we do not show the MPMC queue code as the contract annotations result straightforward having defined the formal semantic rules.

VI. Evaluation

In this section, we perform an evaluation of the proposed semantic violation detection framework using the previous lock-free SPSC and MPMC queues as a use case. In the following, we describe in detail the lock-free structure, software and target platform used for the evaluation.

Lock-free structures. We leverage the SPSC queue, the Multiple-Producer/Multiple-Consumer (MPMC) queue and the Stack lock-free data structures from the Boost C++ library v1.54.0 [9].

Software. The compiler used is Clang v6.0.0 (part of the LLVM project) together with the runtime libraries (compiler-rt) in its Subversion revision 314968, which support the ThreadSanitizer data race detector.

Target platform. The evaluation has been carried out on a server platform comprised of 2× Intel Xeon Ivy Bridge E5-2695 v2 with a total of 24 cores running at 2.40 GHz, 30 MB of L3 cache and 128 GB of DDR3 RAM. The OS is a Ubuntu 14.04.2 LTS with the Linux kernel 3.13.0-57.

The methodology used to evaluate the CSV framework consists in analyzing the relative decrease in false positives/negatives using the semantics handled by CSV with respect to the original TSan race detection mechanisms. In the following two sections we perform the analysis using a series of synthetic parallel benchmarks which make use of the member functions of the aforementioned lock-free structures.

VI.1 Analysis of representative benchmark executions

In this section, we analyse with three representative execution sequences of SPSC queues in order to gain insight into the violation detection process during correct and invalid situations under two different scenarios: *i) lock-protected*, in which both producers and the consumer wrap the respective `push` and `pop` calls into a common lock for protecting queue accesses; and *ii) producer-protected*, in which only the producer threads protect the corresponding `push` calls within the same lock.

Table 1a illustrates a correct execution sequence using a lock-free SPSC queue in the lock-free scenario. This table is organized as follows: *i)* column *Time* represents the instant of time i in which the events happen; *ii)* column *Event* detail the actions performed by the different threads using the queue; *iii)* columns *Init*, *Prod* and *Cons* stand for the set of past events related to methods invocations; *iv)* columns Req. (1)–(3) show how the semantic requirements are applied; and *v)* columns TSan and CSV indicate if these tools produced false positive (FP) or missed a potential error (FN). In this example, 3 different threads alternate their roles during the queue lifetime, however, at a given point in time, only a producer and a consumer coexist. For instance, in t_3, t_4, t_5 and t_8 TSan reports data race warnings when in fact the semantic rules are satisfied at any time. Also, in t_7 TSan emits a false positive, while the *happens-before* relation between `T1` and `T3` in t_6 prevents this warning from being an actual race. On the contrary, using CSV, no data race warnings are emitted, as the semantics of the queue are always met.

On the other hand, Table 1b shows another execution sequence using a SPSC queue under the same scenario but in a wrong way: Req. (1), (2), and (3) are violated. First, Req. (1) is not met in t_0, since `T1` uses the queue before having initialized it. Next, Req. (1) is also violated in t_4, as `T2` produces an element concurrently with the previous production event. Besides, Req. (2) is violated in t_6: `T2` and `T3` are performing `pop` operations concurrently, leading to undefined behavior. Req. (3) is as well not satisfied in t_7, considering that the initialization event i_2, performed by `T1`, has not happened after the previous production and consumption events, so the state of the queue at this point is inconsistent and may differ among executions. Specifically, TSan is not able to encounter potential errors in t_0 and t_7 and reports wrong race warnings in t_5 and t_6, given the pair of calls at these time points. In contrast, CSV correctly reports errors in t_0, t_4, t_6 and t_7.

The last execution sequence in Table 1c is also a wrong use of the SPSC queue under the lock-protected scenario, where all the calls are made after a shared lock has been acquired. In this case, TSan does not emit any warning as all accesses are serialized, i.e. at t_1, t_3 and t_5 the proposed semantics may be violated, since locks do not guarantee any ordering and `push`/`pop` calls could be executed before the first call to `init`. For the same reason, CSV may detect errors in t_1, t_3 and t_5[1].

VII. Conclusions

Data race detectors aid, to a great extent, to identify races in parallel applications. However, few of them are aware of the semantics of lock-free data structures and may emit false positive warnings when these are used. Similarly, an application free of warning race reports does not entail that its internal data structures have been properly used. In this paper, we present CSV, a tool built on top of TSan that enables the annotation and run time checking of semantics of lock-free data structures using C++ contracts. To implement this framework, we extended both Clang compiler and TSan race detector tool in order to support a contract-based syntax that allows checking of user-defined semantics at run time. Though this framework has been implemented within the TSan detector, it would be possible to port it to other data race detectors from the state-of-the-art, such as CDSchecker [18] or RCMC [11].

Throughout the evaluation, we demonstrated the benefits of CSV with respect to the stand-alone TSan detector. Basically, CSV provides two main features: *i)* to filter misleading data race reports (false positives); and *ii)* to detect violations of the semantics of lock-free data structures. Assuming that the implementation of the data structure is correct, CSV replaces TSan low-level reports by meaningful traces. Additionally, given the high-level syntax of CSV for annotating semantics, other lock-free structures may benefit from this framework, as long as a formalization of their semantics is feasible.

As future work, we aim at extending the syntax to support other types of event relations and detection of other potential failures, e.g. deadlocks, livelocks and lock starvation. An ultimate goal is to evaluate the framework with concurrent applications making use of high-level parallel programming frameworks, e.g., FastFlow or Raftlib.

References

[1] Artho, C., Havelund, K., Biere, A.: High-level data races. Software Testing, Verification and Reliability **13**(4), 207–227 (2003)

[2] Blechmann, T.: Chapter 17. Boost.Lockfree. `https://www.boost.org/doc/libs/1_54_0/doc/html/lockfree.html` (2011)

[3] Borkar, S.: The exascale challenge. In: VLSI Design Automation and Test (VLSI-DAT), 2010 International Symposium on, pp. 2–3 (2010). DOI 10.1109/VDAT.2010.5496640

[4] Boyapati, C., Lee, R., Rinard, M.: Ownership types for safe programming: Preventing data races and deadlocks. In: ACM SIGPLAN Notices, vol. 37:11, pp. 211–230. ACM (2002)

[5] Choi, J., Dukhan, M., Liu, X., Vuduc, R.: Algorithmic Time, Energy, and Power on Candidate HPC Compute Building

[1] It may fail detecting a violation (depending on the execution order).

Table 1: Representative execution sequences of SPSC queues with semantic violation detection.

(a) Correct use of a SPSC queue under the lock-free scenario.

Time	**Event**	*Init*	*Prod*	*Cons*	**Req. (1)**	**Req. (2)**	**Req. (3)**	TSan	CSV
t_0	$i_1 =$ `T1 calls to init`	$\{\}$	$\{\}$	$\{\}$	-	-	-		
t_1	`T1 creates T2`								
t_2	$p_1 =$ `T1 calls to push`	$\{i_1\}$	$\{\}$	$\{\}$	$i_1 \rightarrow p_1$	-	-		
t_3	$c_1 =$ `T2 calls to pop`	$\{i_1\}$	$\{p_1\}$	$\{\}$	-	$i_1 \rightarrow c_1$	-	FP	
t_4	$p_2 =$ `T1 calls to push`	$\{i_1\}$	$\{p_1\}$	$\{c_1\}$	$i_1 \rightarrow p_2 \wedge p_1 \not\approx p_2$	-	-	FP	
t_5	$c_2 =$ `T2 calls to pop`	$\{i_1\}$	$\{p_1, p_2\}$	$\{c_1\}$	-	$i_1 \rightarrow c_2 \wedge c_1 \not\approx c_2$	-	FP	
t_6	`T1 creates T3`								
t_7	$p_3 =$ `T3 calls to push`	$\{i_1\}$	$\{p_1, p_2\}$	$\{c_1, c_2\}$	$i_1 \rightarrow p_3 \wedge \{p_1, p_2\} \not\approx p_3$	-	-	FP	
t_8	$c_3 =$ `T2 calls to pop`	$\{i_1\}$	$\{p_1, p_2, p_3\}$	$\{c_1, c_2\}$	-	$i_1 \rightarrow c_3 \wedge \{c_1, c_2\} \not\approx c_3$	-	FP	

(b) Invalid use of a SPSC queue under the lock-free scenario.

Time	**Event**	*Init*	*Prod*	*Cons*	**Req. (1)**	**Req. (2)**	**Req. (3)**	TSan	CSV
t_0	$c_1 =$ `T1 calls to pop`	$\{\}$	$\{\}$	$\{\}$	$Init = \varnothing$	-	-	FN	
t_1	$i_1 =$ `T1 calls to init`	$\{\}$	$\{\}$	$\{c_1\}$	-	-	$c_1 \not\approx i_1$		
t_2	`T1 creates T2`								
t_3	$p_1 =$ `T1 calls to push`	$\{i_1\}$	$\{\}$	$\{c_1\}$	$i_1 \rightarrow p_1$	-	-		
t_4	$p_2 =$ `T2 calls to push`	$\{i_1\}$	$\{p_1\}$	$\{c_1\}$	$p_1 \approx p_2$	-	-		
t_5	$c_2 =$ `T1 calls to pop`	$\{i_1\}$	$\{p_1, p_2\}$	$\{c_1\}$	-	$i_1 \rightarrow c_2 \wedge c_1 \not\approx c_2$	-	FP	
t_6	$c_3 =$ `T2 calls to pop`	$\{i_1\}$	$\{p_1, p_2\}$	$\{c_1, c_2\}$	-	$c_2 \approx c_3$	-	FP*	
t_7	$i_2 =$ `T1 calls to init`	$\{i_1\}$	$\{p_1, p_2\}$	$\{c_1, c_2, c_3\}$	-	-	$p_2 \approx i_2 \wedge c_3 \approx i_2$	FN	

(c) Invalid use of a SPSC queue under the lock-protected scenario.

Time	**Event**	*Init*	*Prod*	*Cons*	**Req. (1)**	**Req. (2)**	**Req. (3)**	TSan	CSV
t_0	`T1 creates T2`								
t_1	$c_1 =$ `T1 locked call to pop`	$\{\}$	$\{\}$	$\{\}$	$Init = \varnothing$	-	-	FN	FN*
t_2	$i_1 =$ `T2 locked call to init`	$\{\}$	$\{\}$	$\{c_1\}$	-	-	$c_1 \not\approx i_1$		
t_3	$p_1 =$ `T1 locked call to push`	$\{i_1\}$	$\{\}$	$\{c_1\}$	$i_1 \rightarrow p_1$	-	-	FN	FN*
t_4	$c_2 =$ `T2 locked call to pop`	$\{i_1\}$	$\{p_1\}$	$\{c_1\}$	-	$i_1 \rightarrow c_2 \wedge c_1 \not\approx c_2$	-		
t_5	$c_3 =$ `T1 locked call to pop`	$\{i_1\}$	$\{p_1\}$	$\{c_1, c_2\}$	-	$i_1 \rightarrow c_3 \wedge \{c_1, c_2\} \not\approx c_3$	-	FN	FN*
t_6	$i_2 =$ `T2 locked call to init`	$\{i_1\}$	$\{p_1\}$	$\{c_1, c_2, c_3\}$	-	-	$\{i_1, p_1, c_1, c_2, c_3\} \not\approx i_2$		

Blocks. In: Parallel and Distributed Processing Symposium, 2014 IEEE 28th International, pp. 447–457 (2014). DOI 10.1109/IPDPS.2014.54

[6] Dolz, M.F., Del Rio Astorga, D., Fernández, J., Torquati, M., García, J.D., García-Carballeira, F., Danelutto, M.: Enabling semantics to improve detection of data races and misuses of lock-free data structures. Concurrency and Computation: Practice and Experience **29**(15) (2017). DOI 10.1002/cpe.4114. URL `http://dx.doi.org/10.1002/cpe.4114`

[7] Dongol, B.: Formalising progress properties of non-blocking programs. In: Z. Liu, J. He (eds.) Formal Methods and Software Engineering, pp. 284–303. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)

[8] Engler, D., Ashcraft, K.: RacerX: Effective, Static Detection of Race Conditions and Deadlocks. SIGOPS Oper. Syst. Rev. **37**(5), 237–252 (2003). DOI 10.1145/1165389.945468

[9] Giacomoni, J., Moseley, T., Vachharajani, M.: Fastforward for efficient pipeline parallelism: A cache-optimized concurrent lock-free queue. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '08, pp. 43–52. ACM, NY, USA (2008). DOI 10.1145/1345206.1345215

[10] Intel: Inspector XE 2017. `https://software.intel.com/en-us/intel-inspector-xe`

[11] Kokologiannakis, M., Lahav, O., Sagonas, K., Vafeiadis, V.: Effective stateless model checking for c/c++ concurrency. Proc. ACM Program. Lang. **2**(POPL), 17:1–17:32 (2017). DOI 10.1145/3158105. URL `http://doi.acm.org/10.1145/3158105`

[12] Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978). DOI 10.1145/359545.359563

[13] Lamport, L.: How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. IEEE Trans. Comput. **28**(9), 690–691 (1979). DOI 10.1109/TC.1979.1675439

[14] Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization, CGO '04, pp. 75–. IEEE Computer Society, Washington, DC, USA (2004)

[15] Lidbury, C., Donaldson, A.F.: Dynamic race detection for c++11. SIGPLAN Not. **52**(1), 443–457 (2017). DOI 10.1145/3093333.3009857. URL `http://doi.acm.org/10.1145/3093333.3009857`

[16] Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XIII, pp. 329–339. ACM, New York, NY, USA (2008). DOI 10.1145/1346281.1346323

[17] Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, PODC '96, pp. 267–275. ACM, New York, NY, USA (1996)

[18] Norris, B., Demsky, B.: CDSchecker: Checking Concurrent Data Structures Written with C/C++ Atomics. SIGPLAN Not. **48**(10), 131–150 (2013). DOI 10.1145/2544173.2509514. URL `http://doi.acm.org/10.1145/2544173.2509514`

[19] Norris, B., Demsky, B.: A Practical Approach for Model Checking C/C++11 Code. ACM Trans. Program. Lang. Syst. **38**(3), 10:1–10:51 (2016). DOI 10.1145/2806886. URL `http://doi.acm.org/10.1145/2806886`

[20] O'Callahan, R., Choi, J.D.: Hybrid dynamic data race detection. SIGPLAN Not. **38**(10), 167–178 (2003)

[21] Ou, P., Demsky, B.: Checking Concurrent Data Structures Under the C/C++11 Memory Model. SIGPLAN Not. **52**(8), 45–59 (2017). DOI 10.1145/3155284.3018749. URL `http://doi.acm.org/10.1145/3155284.3018749`

[22] Reis, G.D., Garcia, J.D., Lakos, J., Meredith, A., Myers, N., Stroustrup, B.: Support for contract based programming in C++. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0542r5.html` (2018)

[23] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: A dynamic data race detector for multithreaded programs. ACM Transactions on Computer Systems (TOCS) **15**(4), 391–411 (1997)

[24] Schweizer, H., Besta, M., Hoefler, T.: Evaluating the Cost of Atomic Operations on Modern Architectures. In: 24th International Conference on Parallel Architectures and Compilation (PACT'15). ACM (2015)

[25] Serebryany, K., Potapenko, A., Iskhodzhanov, T., Vyukov, D.: Dynamic Race Detection with LLVM Compiler. In: S. Khurshid, K. Sen (eds.) Runtime Verification, pp. 110–114. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)

[26] Shalev, O., Shavit, N.: Split-ordered lists: Lock-free extensible hash tables. J. ACM **53**(3), 379–405 (2006)

[27] Xie, X., Xue, J.: Acculock: Accurate and efficient detection of data races. In: Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '11, pp. 201–212. IEEE Computer Society, Washington, DC, USA (2011)

[28] Zhou, P., Teodorescu, R., Zhou, Y.: Hard: Hardware-assisted lockset-based race detection. In: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture, HPCA '07, pp. 121–132. IEEE Computer Society, Washington, DC, USA (2007). DOI 10.1109/HPCA.2007.346191