Enabling semantics to improve detection of data races and misuses of lock-free data structures

Manuel F. Dolz¹ | David Del Rio Astorga¹ | Javier Fernández¹ | Massimo Torquati² | José Daniel García¹ | Félix García-Carballeira¹ | Marco Danelutto²

¹Department of Computer Science, University Carlos III of Madrid, Leganés Madrid, 28911, Spain
²Department of Computer Science, University of Pisa, Pisa, 56127, Italy

Correspondence
Manuel F. Dolz, Department of Computer Science, University Carlos III of Madrid, Leganés, Madrid, Spain.
Email: mdolz@inf.uc3m.es

Summary
The rapid progress of multi/many-core architectures has caused data-intensive parallel applications not yet fully optimized to deliver the best performance. In the advent of concurrent programming, frameworks offering structured patterns have alleviated developers’ burden adapting such applications to multithreaded architectures. While some of these patterns are implemented using synchronization primitives, others avoid them by means of lock-free data mechanisms. However, lock-free programming is not straightforward, ensuring an appropriate use of their interfaces can be challenging, since different memory models plus instruction reordering at compiler/processor levels can interfere in the occurrence of data races. The benefits of race detectors are formidable in this sense; however, they may emit false positives if unaware of the underlying lock-free structure semantics. To mitigate this issue, this paper extends ThreadSanitizer, a race detection tool, with the semantics of 2 lock-free data structures: the single-producer/single-consumer and the multiple-producer/multiple-consumer queues. With it, we are able to drop false positives and detect potential semantic violations. The experimental evaluation, using different queue implementations on a set of μ benchmarks and real applications, demonstrates that it is possible to reduce, on average, 60% the number of data race warnings and detect wrong uses of these structures.

KEYWORDS
data race detectors, parallel programming, semantics, wait-/lock-free data structures

1 | INTRODUCTION

As we pave the way towards exascale computing, the use of multi and many-core architectures to efficiently solve scientific problems becomes a complex challenge that the High Performance Computing (HPC) community needs to face.¹,² The adoption of parallel programming frameworks executing multiple processes and/or threads simultaneously, drops developer’s burden to design and implement efficient parallel applications from scratch. Despite this, much of the current software is not yet fully accommodated to run on recent parallel platforms. In most cases, hardware design progresses faster than the parallelization and optimization of existing software. So as to deal with this issue, the use of building blocks implementing core functionalities has been a widely accepted approach in the HPC area.³ Indeed, many of scientific parallel applications leverage efficient parallel kernels from highly tuned libraries at the bottom of their food chain. However, these kernels must guarantee correctness and thread safety to generate correct global results.

While parallel programming techniques have been broadly adopted, concurrency bugs, especially data races, have become more frequent. The adversity in finding data races and deadlocks is a well-known problem.⁴ Indeed, it has been recognized as an arduous task, given that errors may occur only during low-probability sequences of events and may also depend on external factors such as the machine load. These facts make data races extremely sensitive of time, the presence of print statements, compiler options, or differences in memory models. Although data race detectors alleviate debugger’s task in finding these issues, they are still not perfect.⁵ In particular, if lock-free structures—implemented without high-level atomics—are used, they can still generate false positives. This fact blurs developer’s vision in finding harmful data 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 wrong uses of lock-free data structures, thus violating their semantics and possibly generating undefined results.

Given the foregoing, we benefit from semantics to improve the detection of data races and misuses of some lock-free data structures. Specifically, we contribute in this paper with the following:

- We formalize the semantics of the single-producer/single-consumer (SPSC) and multiple-producer/multiple-consumer (MPMC) lock-free queues.
- We describe the 2 main extensions to improve detection of data races: (1) to drop false positive data races and (2) to detect misuses even when data races are not encountered.
- We explain in detail how the aforementioned extensions have been implemented into ThreadSanitizer (TSan), a well-known race detector part of the low-level virtual machine (LLVM) infrastructure.
- We validate our extension for filtering false positive data races using bounded and unbounded SPSC queues along with different μ benchmarks and applications from the FastFlow framework. For MPMC queues, we leverage 4 well-known implementations.
- We analyze our extension for detecting misuses through a fault injection mechanism over a set of synthetic benchmarks using SPSC queues.

While some of these results were already presented in Dolz et al, the support for MPMC queues and the improved integration of the semantic extensions into TSan plus the detection of misuses are new contributions of this paper.

This paper is organized as follows: Section 2 revisits some related work and highlights the differences regarding this paper’s contribution. Section 3 describes the main software pieces used in our research. Section 4 describes the semantics of the SPSC and MPMC lock-free queues. Section 5 motivates the use of semantics to drop false positives and detect misuses, also when using more restrictive semantics. Section 6 details how our techniques for dropping false positives and detecting misuses have been implemented into the race detection tool. Section 7 evaluates such mechanisms with μ benchmarks and real applications that leverage SPSC and MPMC queues. Finally, Section 8 closes the paper with some concluding remarks and future works.

## 2 RELATED WORK

Over the years, numerous solutions to detect data races have been proposed. These have been basically based on different well-known mechanisms: (1) happens-before relations, (2) locksets, and (3) hybrid approaches, i.e., combining both happens-before and locksets mechanisms.

Basically, the happens-before relations are used to detect if 2 conflicting memory accesses are not ordered by synchronization operations, causing a potential data race. The Intel Inspector and Acculock are, respectively, a well-known commercial tool and a recent research tool implementing this algorithm. Nevertheless, software implementations of happens-before-based detectors typically suffer from large runtime overheads, so hardware-based solutions have also been proposed to overcome these issues. On the other hand, the locksets approach reports a data race if there is no common lock held by 2 threads accessing the same memory location. This approach can be found in both static and dynamic tools in the state-of-the-art. Finally, hybrid approaches take advantage of happens-before mechanisms to reduce the false positives reported by lockset-based race detectors and preserve the performance advantages of the lockset mechanisms. A race detector implementing this approach is TSan.

Although previously mentioned tools aid developers to find concurrency bugs, these can still miss ad hoc synchronizations and therefore generate benign race reports. So as to face these issues, 2 main approaches for improving the accuracy of data race detection have been adopted: (1) filtering out benign races and stick only with the harmful ones and (2) annotating custom synchronizations in the user code. For the first approach, the authors in Narayanasamy et al identify benign data races by comparing the execution result when swapping the execution order of a race pair. A TDetector performs a postmortem analysis to identify implicit synchronizations from address transfers and remove the related benign data races from the warning reports. However, these solutions are not aware of lock-free synchronizations nor data structure semantics. For the second approach, some race detectors provide users with a set of annotations that can be used to inform of synchronizations not automatically recognized by the tools. Examples of detectors using this approach are the Intel Inspector and TSan. However, the annotation task is very costly in time and users need additional knowledge about the custom synchronization points. In this sense, some solutions have been developed to automatically identify ad hoc synchronizations.

Despite the wide literature on data race detection techniques, it is difficult to analyze and compare them due to the use of various patterns, as for example, execution ordering (totally vs partially ordered), synchronization primitives (locks, wait/signal, atomics, etc.), memory model supported (Total Store Order, Sequential Consistency, Weak Ordering, etc.). To the best of our knowledge, previously mentioned tools are not able to correctly recognize data races in lock-free data structures when multiple target platforms and memory models are considered. Our proposal represents the first attempt towards the direction of extending tools for data race detection with semantics of lock-free data structures.

## 3 BACKGROUND

In this section, we give an overview of the 2 main software components that have been used to perform the contributions made in this paper. First, we review some basic concepts about lock-free structures and introduce the 2 main lock-free buffers used in this paper: SPSC and MPMC queues. Next, we revisit the LLVM infrastructure along with TSan data race detector to identify undefined and suspicious behavior of threads.

### 3.1 Lock-/wait-free buffers

In general, concurrent data structures can be classified as either blocking or nonblocking. Nonblocking techniques bypass the use of traditional synchronization primitives, such as locks or mutexes, while ensuring thread safety. Lock and wait freedom are 2 important levels
of progress guarantees for nonblocking data structures. A concurrent data structure is lock free if there is guaranteed system-wide progress, i.e., at least a thread makes progress on its execution, while the structure is considered wait free if it is ensured that every thread can perform such a progress. In this sense, wait freedom is the strongest progress guarantee.

The absence of blocking synchronization mechanisms allows increasing 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. Some of the atomic operations used underneath are test and set, fetch and add (F AA), compare and swap (CAS), and load linked/store conditional.22 Basically, these operations atomically combine a load and a store operation. At hardware level, most of the current architectures from Intel and AMD already implement atomic operations and memory fences.23,24 At software level, atomic variables are natively implemented by programming languages (such as C++11) and used by some parallel frameworks and third-party libraries. Lock-free data structures, such as queues,25 hash tables,26 and SPSC buffers,27 are typically known to leverage these kind of atomic instructions internally. Nevertheless, lock-wait-free data structures are significantly more complex to implement and consequently to verify their correctness with respect to lock-based structures.

Focusing on concurrent lock-free data buffers, the most frequently used is the SPSC queue, as it can be used on shared cache multicore systems to implement 1-to-1 memory channels.27 This kind of queue, however, requires coordination between producer and consumer to ensure proper operation. While different kinds of shared data buffers provide different fairness levels, Lamport28 proposes a wait-free SPSC queue implementation that uses a circular buffer without any needing explicit synchronization mechanism. For our specific case, we leveraged the bounded and unbounded29 SPSC lock-free buffer implementations from the FastFlow programming framework,6 however, the methodology presented can be applied to any other implementation featuring these data structures.

Combinations of SPSC buffers can also be used to generate more complex communication channels, eg, N-to-1, 1-to-M, and N-to-M. These channels basically lead to MPMC queues. Several implementations of MPMC queues can be found in the literature. For example, the Michael and Scott’s lock-free queue25 is the most famous and widely used nonblocking queue that uses CAS operations. Other examples are the linked list cyclic ring (LCRQ)29 and the fast wait-free30 MPMC queues, both based on F AA operations.

3.2 The LLVM infrastructure and the TSan data race detection tool

The LLVM is a compiler infrastructure designed to be a set of reusable libraries with well-designed interfaces.31 The LLVM generates intermediate code, that is, afterwards converted into a 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 to perform compile-time analysis and instrumentation. Taking advantage of the latter feature, several runtime checks and tools have been developed to identify suspicious and undefined behavior of threads. One of them is 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 runtime.19

Specifically, TSan instrumentation tracks synchronization primitives, thread routines from libpthread, memory allocation routines, dynamic annotations, and other kinds of functions that lead to synchronization. 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, 2 race detection mechanisms based on happens-before and locksets relations are applied. As a summary of O’Callahan and Choi,15 these mechanisms develop the following strategies:

**Happens-before** relations detect a potential data race when 2 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. They are concurrent, so no causal relationship ordering exists between a and b.9

**Locksets** determine a data race when none of the locks held by a pair of events accessing to a shared memory location, where at least one of these accesses is a write, is the same, 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 as the hybrid mode.19 While in the first mode the concurrency is only checked in happens-before relations, in the hybrid mode, both happens-before and locksets mechanisms are used together to determine if 2 events are concurrent. The reason of having an hybrid mode is that maintaining vector clocks for every shared memory location and every lock, as it is the case for the pure happens-before mode, are too expensive in the practice. Also because the pure happens-before is less predictable and can miss data races, as too many bogus interthread messages are generated. The hybrid mode avoids this shortcomings by means of using happens-before relations in memory accesses and locksets in locking primitives.

In summary, the main reasons for having selected TSan as for the data race detector and improve it with lock-free data structure semantics are (1) it is the only tool that provides the most detailed output and includes a hybrid detection algorithm; (2) it uses compile-time instrumentation, making it much faster than other solutions; (3) it is build on the top of the LLVM infrastructure, being, therefore, an open-source software capable of accommodating semantic requirements.

4 SEMANTICS OF LOCK-FREE QUEUES

In this section, we describe formally the bounded and unbounded SPSC and the MPMC queues along with their semantics for the concurrent lock-free versions. These definitions allows us to proceed further with our rationale for developing rules that guarantee the proper use, among entities, of these lock-free parallel structures.

---

8 In FastFlow, an unbounded queue is implemented using a pool of bounded SPSC queues that grows or shrinks on demand.
4.1 | Formal definition

Consider a queue \( Q \) the tuple

\[
Q = (buf, pread, pwrite, M),
\]

where \( buf \), \( pread \) plus \( pwrite \) are internal read and write pointers for the buffer, respectively, and \( M \) is a set comprising the following methods:

- \text{init}: Initializes the buffer \( buf \), allocating space of possibly aligned memory and resetting the internal (\( pread \) and \( pwrite \)) pointers by placing them at the beginning of \( buf \). If \( buf \) has already been allocated, this method does nothing.
- \text{push}: Enqueues the item into the buffer \( buf \).
- \text{pop}: Removes and returns the first item in the buffer \( buf \).
- \text{empty}: Returns true if the buffer \( buf \) is empty.
- \text{register}: Registers a producer or a consumer for allocating internal variables. Note that this function is tied to the specific MPMC queue interface presented on this paper and should be called before the producers and consumers start pushing and popping data, respectively.

Note that, depending on the internal implementation of a particular queue, \( buf \) can be expressed in different ways. For example, in a SPSC bounded queue, \( buf \) can be declared as a circular buffer, while for an MPMC it can be an array or a list of pointers. Figure 1A depicts the internal working of the circular buffer \( buf \) from the first-in-first-out SPSC queue. Initially, \( pread \) and \( pwrite \) point to the initial position of the buffer, while afterwards, some elements have been added at the terminal position through \text{push} calls and others removed from the head position by means of \text{pop} calls. Additionally, Figure 1B shows an schema of a MPMC queue used by \( n \) and \( m \) producers and consumers.

4.2 | Semantics of the concurrent lock-free
SPSC queue

The correctness of parallel lock-free SPSC queues, such as the Lamport\(^{28}\) or FastForward implementation,\(^{27}\) is only ensured if several usage requirements are met. We define these requirements as the following semantics rules:

1. Roles: A lock-free concurrent SPSC queue instance can be shared by multiple entities acting as initializers, producers, and consumers. Note that a certain entity can perform any role, however, at any point in time, there must only exist a producer and a consumer performing operations on the same queue concurrently. Furthermore, an initializer cannot operate over the queue concurrently with any other entity. In any other case, we consider that the queue is misused thus having an undefined behavior due to the occurrence of potential data races.
2. Initialization methods: The initializers can call to methods belonging to

\[
\text{Init} = \{\text{init}\} : \quad \text{Init} \subseteq M.
\]

3. Producer methods: The producers entities should only invoke a subset of methods in \( M \),

\[
\text{Prod} = \{\text{push}\} : \quad \text{Prod} \subseteq M.
\]

4. Consumer methods: The consumers entities should only invoke a subset of methods in \( M \),

\[
\text{Cons} = \{\text{pop}, \text{empty}\} : \quad \text{Cons} \subseteq M.
\]

Particularly, all subsets allotted to different roles of the queue fulfill \( M = \text{Init} \cup \text{Prod} \cup \text{Cons} \). Note also that methods internally using the \( pwrite \) pointer are those assigned to the producer, while those using the \( pread \) pointer are related to the consumer.

To formalize the aforementioned semantics, we first make the following definitions. First, we define an event as invocation of a method at a certain point of time performed by an entity. In our particular case, we distinguish among 3 different event types: production, consumption, and initialization and denote them as \( p \), \( c \), and \( i \), respectively. Second, we define \( E \) as a set of events that is related to each of the preceding methods subsets in the queue \( Q \), which stores all past methods invocations.

With these definitions, it is possible to control the proper use of the lock-free SPSC queue by checking 3 simple requirements depending on the type of a new incoming event. These requirements, defined in Equations 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, Equation 1 ensures that the new production event has a happens-before relation (\( \rightarrow \)) with all past initialization events and is not concurrent (\( \equiv \)) with all past production events. Similarly, Equation 2 performs the same verification but for incoming consumption events.

\[
p' , \quad \forall i \in \text{Init}.E , \quad \forall p \in \text{Prod}.E : \quad \text{Init}.E \neq \emptyset \land i \rightarrow p' \land p \equiv p'. \quad (1)
\]

\[
c' , \quad \forall i \in \text{Init}.E , \quad \forall c \in \text{Cons}.E : \quad \text{Init}.E \neq \emptyset \land i \rightarrow c' \land c \equiv c'. \quad (2)
\]

Additionally, Equation 3 ensures that all initialization events happened sequentially with any other event. That is, all past events of \( Q \) have
a happened-before relation 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 behaviors.

\[ i', \ ye \in \text{Init} \cup \text{Prod} \cup \text{Cons} : e \rightarrow i' (3) \]

Table 1 illustrates a correct execution sequence using a lock-free SPSC queue. This table is organized as follows: (1) column time represents the instant of time in which the events happen; (2) column event details the actions performed by the different threads using the queue; (3) columns \( \text{Init} \), \( \text{Prod} \), and \( \text{Cons} \) stand for the set of past events related to methods invocations; finally, (4) columns Equations 1 to 3 show how the semantic requirements are applied. 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 \( t_0 \), the thread with ID 1 acting as a producer, creates a new thread with ID 3 and, from this point on, the new thread can start producing elements in the queue. This fact guarantees that both threads 1 and 3 are synchronized before interchanging their roles, i.e., there exist happens-before relations between production events of both threads: \( p_1 \rightarrow p_2 \) and \( p_2 \rightarrow p_3 \). Therefore, Equations 1, 2, and 3 are met at any point in time.

Table 2 shows an execution sequence of 2 threads using a SPSC queue in a wrong way: Equations 1, 2, and 3 are violated. First, Equation 1 is not met, since the thread with ID 1 uses the queue before having initialized it. Next, Equation 1 is also violated in \( t_6 \), since the thread with ID 2 produces an element concurrently with the previous production event \( (p_1 \approx p_2) \). Besides, Equation 2 is violated; threads with ID 1 and 3 are performing pop operations concurrently, therefore leading to undefined behaviors. Finally, Equation 3 is as well not satisfied, considering that the initialization event \( i_2 \), performed by the thread with ID 1, has not happened after the previous production and consumption events, so the elements present in the queue at this point in time are inconsistent and may differ among executions.

4.3 Semantics of the concurrent lock-free MPMC queue

The requirements that guarantee the correctness of a parallel lock-free MPMC queues are slightly different to those for declared for the SPSC queues. These requirements are defined as the following semantics rules:

1. Roles: A lock-free concurrent MPMC queue instance can be shared among different entities that can act interchangeably as producers and consumers. However, they should have been registered themselves in the queue before producing and consuming on it. Also, an initializer cannot operate over the queue concurrently with any other entity, while entities registrations should happen after the last initialization. In any other case, we consider that the queue is misused thus having an undefined behavior due to the occurrence of potential data races.

2. Initialization methods: The initializers can call to methods belonging to

\[ \text{Init} = \{ \text{init} : \text{Init} \subseteq M \}. \]

### Table 1
Execution sequence of a SPSC queue

| Time | Event | \( \text{Init} \) | \( \text{Prod} \) | \( \text{Cons} \) | Equation 1 | Equation 2 | Equation 3 |
|------|-------|----------------|----------------|----------------|-------------|-------------|-------------|
| \( t_0 \) | \( i_1 = \text{T1 calls to init} \) | \{ \} | \{ \} | \{ \} | - | - | - |
| \( t_1 \) | \( \text{T1 creates T2} \) | | | | \( i_1 \rightarrow p_1 \) | - | - |
| \( t_2 \) | \( p_1 = \text{T1 calls to push} \) \{ \( i_1 \) \} | \{ \} | \{ \} | - | \( i_1 \rightarrow c_1 \) |
| \( t_3 \) | \( c_1 = \text{T2 calls to pop} \) \{ \( p_1 \) \} | \{ \} | \{ \} | - | \( i_1 \rightarrow c_1 \) |
| \( t_4 \) | \( p_2 = \text{T1 calls to push} \) \{ \( i_1 \) \} | \{ \( p_1 \) \} | \{ \( c_1 \) \} | - | \( i_1 \rightarrow p_2 \land p_1 \rightarrow p_2 \) |
| \( t_5 \) | \( c_2 = \text{T2 calls to pop} \) \{ \( i_1 \) \} | \{ \( p_1, p_2 \) \} | \{ \( c_1 \) \} | - | \( i_1 \rightarrow c_2 \land c_1 \rightarrow c_2 \) |
| \( t_6 \) | \( \text{T1 creates T3} \) | | | | \( i_1 \rightarrow c_3 \land c_1 \rightarrow c_3 \) |
| \( t_7 \) | \( p_3 = \text{T3 calls to push} \) \{ \( i_1 \) \} | \{ \( p_1, p_2 \) \} | \{ \( c_1, c_2 \) \} | \( i_1 \rightarrow p_3 \land (p_1, p_2) \rightarrow p_3 \) - |
| \( t_8 \) | \( c_3 = \text{T2 calls to pop} \) \{ \( i_1 \) \} | \{ \( p_1, p_2, p_3 \) \} | \{ \( c_1, c_2 \) \} | - | \( i_1 \rightarrow c_3 \land (c_1, c_2) \rightarrow c_3 \) |

Abbreviation: SPSC, single-producer/single-consumer.

The dash "-" indicates that the semantic rule does not proceed for the case.

### Table 2
Example of execution sequence of a misuse of the SPSC queue

| Time | Event | \( \text{Init} \) | \( \text{Prod} \) | \( \text{Cons} \) | Equation 1 | Equation 2 | Equation 3 |
|------|-------|----------------|----------------|----------------|-------------|-------------|-------------|
| \( t_0 \) | \( c_1 = \text{T1 calls to pop} \) | \{ \} | \{ \} | \{ \} | \( \text{Init} = \emptyset \) | - | - |
| \( t_1 \) | \( i_1 = \text{T1 calls to init} \) | \{ \} | \{ \} | \{ \( c_1 \) \} | - | - | \( c_1 \rightarrow i_1 \) |
| \( t_2 \) | \( \text{T1 creates T2} \) | | | | \( i_1 \rightarrow p_1 \) | - | - |
| \( t_3 \) | \( p_1 = \text{T1 calls to push} \) \{ \( i_1 \) \} | \{ \} | \{ \( c_1 \) \} | - | \( i_1 \rightarrow p_1 \land p_1 \rightarrow p_2 \) | - |
| \( t_4 \) | \( p_2 = \text{T2 calls to push} \) \{ \( i_1 \) \} | \{ \( p_1 \) \} | \{ \( c_1 \) \} | \( p_1 \approx p_2 \) | - |
| \( t_5 \) | \( c_2 = \text{T1 calls to pop} \) \{ \( i_1 \) \} | \{ \( p_1, p_2 \) \} | \{ \( c_1 \) \} | - | \( i_1 \rightarrow c_2 \land c_1 \rightarrow c_2 \) |
| \( t_6 \) | \( c_3 = \text{T2 calls to pop} \) \{ \( i_1 \) \} | \{ \( p_1, p_2 \) \} | \{ \( c_1, c_2 \) \} | - | \( c_2 \approx c_3 \) |
| \( t_7 \) | \( i_2 = \text{T1 calls to init} \) \{ \( i_1 \) \} | \{ \( p_1, p_2 \) \} | \{ \( c_1, c_2, c_3 \) \} | - | \( p_2 \approx i_2 \land c_3 \approx i_2 \) |

Abbreviation: SPSC, single-producer/single-consumer.

The dash "-" indicates that the semantic rule does not proceed for the case.
3. Producer methods: The producers entities should invoke a subset of methods in \( M \),
   \[
   Prod = \{ \text{push} \} : \ Prod \subseteq M.
   \]

4. Consumer methods: The consumers entities should invoke a subset of methods in \( M \),
   \[
   Cons = \{ \text{pop, empty} \} : \ Cons \subseteq M.
   \]

5. Registration methods: The registration methods should be invoked before a producer or a consumer starts pushing or popping data from the queue, respectively:
   \[
   Regs = \{ \text{register} \} : \ Regs \subseteq M.
   \]

Specifically, all subsets allotted to the different queue roles fulfill \( M = Init \cup Prod \cup Cons \cup Comm \cup Regs \). In the same way that for SPSC queues, we formalize the aforementioned semantics making the following assumptions. For the MPMC queues, we add a new event type on the top of the previously defined SPSC queue-related events. This event type corresponds to registration operations and is denoted as \( r \). Furthermore, we specialize consumption, production, and registration events with the caller thread ID, i.e., given an arbitrary thread \( T \) generating an event \( e \), this event is represented as \( e_T \). Next, we assume that each of the preceding methods subsets includes the set \( E \) as an attribute including all past events related to their methods invocations.

In this case, the correct use of the lock-free MPMC queues can be controlled with the requirements Equations 4 to 7. These requirements are checked each time a new production (\( p' \)), consumption (\( c' \)), registration (\( r' \)), or initialization (\( i' \)) event occurs. Regarding Equation 4, a new production event \( p' \) and performed by the thread with ID \( T \) (ie, \( p'_T \)), should happen after the registration event \( r_T \) and the last initialization event \( i \), considering that \( i \) happened-before \( r_T \). Correspondingly, Equation 5 achieves the same goal but for incoming consumption events.

\[
\begin{align*}
\forall i \in Init.E : & \exists r_T \in Regs.E : \ Init.E \neq \emptyset \land i \rightarrow r_T \land r_T \rightarrow p'_T. & (4) \\
\forall i \in Init.E : & \exists r_T \in Regs.E : \ Init.E \neq \emptyset \land i \rightarrow r_T \land r_T \rightarrow c'_T. & (5)
\end{align*}
\]

Likewise, requirement 6 ensures that the queue \( Q \) has been initialized before a new registration event \( r' \) happens. Finally, Equation 7 guarantees that there are not concurrent events for a new initialization event \( i' \). If any of these requirements is not met, the correct use of a MPMC queue \( Q \) cannot be assured because of the occurrence of potential data races.

\[
\begin{align*}
\forall i \in Init.E : & Init.E \neq \emptyset \land i \rightarrow i'. & (6) \\
\forall e \in Init.E \cup Regs.E \cup Prod.E \cup Cons.E : & e \rightarrow i'. & (7)
\end{align*}
\]

5 | ENABLING SEMANTICS TO IMPROVE DETECTION OF DATA RACES

In this section, we motivate the use of semantics to improve the detection of data races related to lock-free data structures within TSan. Specifically, we find that TSan has some shortcomings when dealing with this kind of data types, since it is not capable of determining whether a data race is harmful or not determining if there have been misuses in a given shared lock-free data structure. In the following Sections 5.1 and 5.2, we expose these issues and our contributions to address them.

5.1 | Dropping false positives

Our first observation is that the implementation of TSan is completely semantic agnostic regarding lock-free data structures. The TSan detects all possible race conditions, regardless of whether they deal with lock-free structures or not. To illustrate this shortcoming, we leverage the execution sequence of Table 3, where a SPSC queue \( Q \) is being used by 2 different threads. This table is comprised of the following columns: (1) time represents the instant of time \( i \) in which the events occur; (2) event describes the actions performed by the entities; (3) data race column indicates if TSan has been able to identify a data race in time \( t \); and (4) false positive column asserts whether the data race detected by TSan is harmful or not. To support further explanations in Listing 1, we present an excerpt of the FastFlow-bounded SPSC queue implementation with the functions allotted to consumer and consumer threads.

The execution flow of sequence in Table 3 proceeds as follows. First, the thread with ID 1 calls to the \texttt{init} function to initialize the internal \( Q \) variables. Next, in \( t_2 \), the same thread creates a new thread with ID 2. In \( t_2 \) and \( t_3 \), both producer and consumer threads with ID 1 and 2 call concurrently to the \texttt{push} and \texttt{pop} functions, respectively. Finally, in \( t_4 \), the consumer thread invokes the \texttt{push} function. Assuming a multicore shared memory machine with a memory model supported by \( Q \), TSan would report data races in \( t_3 \) and \( t_4 \), since these functions read and write simultaneously from the same buffer entry, ie, when \texttt{write} and \texttt{read} are equal (see lines 8 and 20 of Listing 1). However, this is only true to some extent. While the data race reported in \( t_2 \) would have been harmful due to a misuse of \( Q \) (as there can only exist a producer at the same time, as stated in Equation 1), that reported in \( t_3 \) would have resulted in a false positive, as the semantics of \( Q \) have been accomplished at any time. Hence, being the SPSC queue a thread-safe concurrent lock-free data structure, no data race should occur in \( t_3 \).

In summary, not all race conditions detected by TSan are harmful when dealing with lock-free data structures. Current existing approaches within TSan to filter false positives are blacklists, used to
suppress data race reports in the specified source files or functions, and dynamic annotations, used to inform TSan of happens-before and happens-after relations in special synchronization points that cannot be handled by default.19 However, these solutions are not suitable for our purposes. First, the use of the TSan blacklists excludes functions from being instrumented at all: (1) harmful data races can be missed, given that multiple consumers or producers may operate concurrently on a same queue, and (2) false positives may appear, because of missed synchronization points, eg, via atomic operations. Second, the use of dynamic annotations requires the users to modify their code and to have specific knowledge about the lock-free structure so as to know where to introduce such explicit synchronization points. Our contribution follows a different path: we endow the TSan detector with the semantics of the SPSC and MPMC lock-free queues to drop those “benign” data races and reduce the amount of warning reports generated. With it, we preserve the original TSan instrumentation and prevent the users from introducing annotations in their codes. Thanks to this solution, we alleviate as well the debugging task as users can better focus on real failures.

### 5.2 Detecting misuses

Our second observation is that TSan only warns about potential data races; however, there might exist other semantic misuses in specific lock-free data structures that are not related at all to concurrent memory accesses. Taking into account that the hybrid mode of TSan determines a situation of data race when both conditions happens-before and locksets are not met, part of the conditions defined in our semantics (see the requirements in Section 4.2) only need to check for happens-before relations. Therefore, even if TSan does not detect any data race between a pair of events accessing a shared memory location, our semantics can detect misuses in SPSC/MPMC lock-free queues. As an example, Table 4 shows a situation where two threads call concurrently to the `init` and `push` functions, respectively, both wrapped into a critical section using the same lock. In this case, the TSan hybrid detector does not report a data race, as the locksetsintersection is not empty, but our semantic requirements determine an misuse of the SPSC queue, as there is not a happens before between the `push` and `init` events. This is given because the thread with ID 2 can acquire the lock before thus calling to `push` on an uninitialized queue and leading to an undefined behavior. Therefore, our contribution allows users to detect misuses of these kind of lock-free data structures.

### 5.3 Extending detection of misuses for FastFlow queues

While the previous 2 contributions aim at dropping false positives and detecting misuses in generic SPSC/MPMC queues using the semantics defined in Section 4, there exist other lock-free data structures that have, by definition, more restrictive semantics. To illustrate this situation, we focus on the FastFlow queues, as these structures have specific constraints that limit their use in a particular way. Specifically, the FastFlow lock-free SPSC queue implementation has been designed to be used by only 3 different entities during its lifetime, each of them acting as an initializer, a producer and a consumer.† In any other situation, FastFlow developers consider that the queue is misused, even if the user performs an external synchronization in the queue. The reason is that these structures, used as asynchronous communication channels, must only be shared between pairs of threads involved in the computation of a parallel pattern of the framework. Therefore, if a FastFlow developer uses a queue in a way that does not satisfy the aforementioned requirements, it implies that the parallel pattern implementation can contain potential errors, even if no data races occur.

Our contribution in this case is to endow TSan with new capabilities for detecting misuses in specific data structures through a second high-level of semantic verification. For the FastFlow queues, a formalization of their semantics is possible if a set \( C \) of entity IDs is added as an attribute to each of the queue methods subsets: `init`, `Prod`, and `Cons`. By inserting the ID of the caller thread to the corresponding set \( C \) of the subsets each time a method belonging to it is invoked, it is possible to control the proper use of this kind of lock-free SPSC queue. This can be achieved by checking 3 new requirements. The first ensures that the cardinality of the \( C \) set of the initialization, producer, and consumer subsets should always be less or equal than 1.

\[
|\text{Init.} C| \leq 1 \quad \land \quad |\text{Prod.} C| \leq 1 \quad \land \quad |\text{Cons.} C| \leq 1;
\]

hence, only one and the same entity should use methods allotted to its role. The second guarantees that both producer and consumer are performing the right roles, ie,

\[
\text{Prod.} C \cap \text{Cons.} C = \emptyset.
\]

The third requirement guarantees that the queue has been initialized before being used by the producer and the consumer threads:

\[
\text{if } \text{Init.} C = \emptyset, \text{ then } \text{Prod.} C \cup \text{Cons.} C = \emptyset.
\]

As an example, Table 5 depicts an execution sequence a SPSC queue \( Q \) concurrently used by a constructor, producer and consumer entities. The arrangement of the table follows the same structure as Table 3: 

† In certain cases, the producer or the consumer can perform the role of the initializer, being only 2 different entities sharing the same queue.
However, in this case, we include the column misuse that displays whether there has been a violation of the previous semantics in t₁ or not. The execution is as follows. After the initialization of Q in t₀, the thread with ID 1 invokes push and creates a thread with ID 2. Then, this new thread mistakenly calls to push, a function not allotted to its role and thus violating Equation 9. Nevertheless, given that the consumer thread has been created right after the producer push call, TSan does not detect any data race in t₃.

### 6 | IMPLEMENTATION

In this section, we describe implementation details considered to integrate semantics of the bounded and unbounded SPSC and MPMC queues into the data race detector TSan. Particularly, we subdivide this section to explain the implementation details and required modifications in TSan runtime internals to perform the objectives introduced in Section 5.

#### 6.1 | Distinguishing between multiple queue instances

The first step to embed semantics into TSan is to distinguish between multiple queue instances given that a multithreaded application can use multiple lock-free SPSC/MPMC queues simultaneously. Therefore, it is necessary to univocally identify those in reports generated by TSan. To solve this issue, we have implemented a mechanism within TSan runtime internals to perform the objectives introduced in Section 5.

Listing 2: Excerpt of DWARF tree associated to the FastFlow SPSC queue class implementation obtained with the dwarfdump utility.

```plaintext
1) <1>`<0x00000176>`
2) `DW_TAG_subprogram`
3) `DW_AT_linkage_name` "_ZN2ff16uSWSR_Ptr_BufferPpushEPv"
4) `DW_AT_name` "push"
5) `DW_AT_decl_file` 0x00000007 ff/ubuffer.hpp
6) `DW_AT_decl_line` 0x0000016e
7) `DW_AT_type` <0x00000029>
8) `DW_AT_decl_type` yes
9) `DW_AT_external` yes
10) `DW_AT_accessibility` DW_ACCESS_public
11) <3>`<0x00000178>`
12) `DW_AT_type` <0x00003712>
13) `DW_AT_artificial` yes
14) <4>`<0x0000018b>`
15) `DW_AT_type` <0x000036a3>
16) ...<0x0000054a`
17) `DW_TAG_subprogram`
18) `DW_AT_low_pc` 0x0014e270
19) `DW_AT_high_pc` <offset-from-lowpc>451
20) `DW_AT_frame_base` len 0x0001: 56: DW_OP_reg6
21) `DW_AT_object_pointer` <0x000004c0`
22) `DW_AT_specification` <0x00000176>
23) <2>`<0x0000054c>`
24) `DW_AT_location` len 0x0002: 3170: DW_OP_freg -16
25) `DW_AT_name` "this"
26) `DW_AT_type` <0x000003a5c>
27) `DW_AT_artificial` yes
```
Implementing semantics

The second step is to implement the 2 levels of semantic verification described in the previous section for filtering false positives and to detect misuses of lock-free structures. In the following, we describe both approaches in detail.

Dropping false positives. Regarding the first level of semantic verification, we next describe how the requirements defined in Section 4 have been implemented. To do so, we need first to recapitulate the internal workings of TSan, briefly introduced in Section 3.2, to detect data races. As described in Serebryany et al., TSan uses shadow memory to track user space memory accesses and properly apply happens-before and hybrid mechanisms. Concretely, each 8-byte word of user-space memory is mapped into 4-shadow-memory 8-byte cells that are used to log accesses performed on such user-space memory address. Each of these cells stores the thread ID, the thread epoch (a scalar clock), the positions accessed in the user space 8-byte word and the access type: read or write. Afterwards, these cells are accordingly updated by TSan to track memory accesses performed in the user code thus allowing the detection of data races when 2 threads read and write from the same address. Note that each time a thread is synchronized with another via implicit barriers (thread creations or wait/signal operations), the first makes a local copy of the second thread pointers from queue objects along with their methods subsets to shift the memory addresses accessed in the user space memory to track user space memory accesses and properly apply happens-before and hybrid mechanisms. Concretely, each 8-byte word of user-space memory is mapped into 4-shadow-memory 8-byte cells that are used to log accesses performed on such user-space memory address. Each of these cells stores the thread ID, the thread epoch (a scalar clock), the positions accessed in the user space 8-byte word and the access type: read or write. Afterwards, these cells are accordingly updated by TSan to track memory accesses performed in the user code thus allowing the detection of data races when 2 threads read and write from the same address. Note that each time a thread is synchronized with another via implicit barriers (thread creations or wait/signal operations), the first makes a local copy of the second thread epoch. With the TSan hybrid detector, the locksets are also stored in the shadow memory, and their intersection is computed correspondingly each time TSan checks for a data race.

Our solution takes advantage of the internal TSan vector clock (epochs) structures and mutex sets (locksets) to shift the memory address granularity, in which the hybrid mode detects concurrency, to the data structure level, ie, within SPSC/MPMC queue objects. To support this functionality, we use several data structures to store this pointers from queue objects along with their methods subsets to collect events performed by the threads. So as to apply the semantic requirements, it is necessary to capture the events each time a thread calls to one of its members. To provide this feature, we offer a runtime support for inserting calls to our internal functions responsible for checking the semantics of the SPSC/MPMC queues.

To illustrate how TSan provides compile instrumentation, we leverage the example in Listing 3, in which the push and pop member functions of the FastFlow SPSC queue class have been instrumented. As can be seen, TSan instruments all read and write accesses to nonlocal variables plus the prologue and epilogue of the same function with __tsan_func_entry and __tsan_func_exit, respectively.

In our approach, we instrument as well the queue member function with the routine __tsan_register_event, which is responsible for registering the events occurred on a specific queue. This routine receives as a sole parameter the queue member routine name to properly unwind the stack and retrieve the this pointer location. Afterwards, the routine adds the pair key-value {ThrID, (ThrEpoch, ThrMutexSet)} into a event log structure (implemented as a hash table) that is related to the corresponding method subset of the queue, identified by the this pointer. If the key ThrID is already present in the log, only the value of such entry is updated with the new (ThrEpoch, ThrMutexSet).

With all above steps, when TSan detects a data race related to the SPSC/MPMC queues under study, we leverage the aforementioned event log structures to check the nonconcurrency condition (x), as stated in Equations 1 and 2, to determine whether a data race within a function member of the queue class is benign or not. To do so, we compare the epochs stored in the event log with the last synchronization epochs that TSan handles internally with respect to other threads. In case that the epoch stored in the event log is not greater than the last synchronization epoch, it means that there exist a happens-before relation between the current event and that stored in the log. Next, we compute the lockset intersection to check if it equals the empty set. After that, the semantics requirements stated for the SPSC/MPMC queues are checked correspondingly and, if any of them is not met, the data race detected by TSan is effectively harmful. Otherwise, it is considered to be a false positive and automatically filtered out.

Detecting misuses of lock-free data structures. In the same way that we supported semantics in TSan for dropping false positives, we add capabilities for controlling the semantics each time a function of the data structure under study is called. To support this functionality, we instrument at LLVM level the SPSC/MPMC queue member function with the routine __tsan_check_semantics that verifies specific parts of the semantic requirements. In this case, we check the part of the semantics that does not involve the nonconcurrency condition, ie, cases in which TSan does not detect any data race. For instance, the condition init.E ≠ ∅ of Equation 1 is checked regardless of whether there exist a data race or not. This is also the case for pure happens-before relations, eg, condition i → p' of Equation 1.
Focusing on the extended semantics of the FastFlow queues, as explained in Section 5.3, these are controlled by inserting the caller thread ID into a set structure related to the methods subsets of the corresponding queue and verified in the same \texttt{__tsan\_check\_semantics} routine. In this special case, we also instrument the main function with the routine \texttt{__tsan\_print\_semantics\_report} at its end. With it, we print a summary report of all the FastFlow-related queues used within the applications along with information of the semantic rules violated, if any. Thanks to this report, we let users know which of those lock-free queues have violated any of their semantics.

### 7 | EXPERIMENTAL RESULTS

In this section, we perform an experimental evaluation of the semantics implemented for the SPSC and MPMC lock-free queues in TSan. In the following, we describe in detail the target platform, software, and benchmarks sets used for the evaluation.

- **Target platform.** The evaluation has been performed on a server platform comprised of 2x 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 Linux Ubuntu 14.04.2 LTS with the kernel 3.13.0-57. We denote this platform as IVY.

- **Queue implementations.** For testing the SPSC queues, we leverage both bounded and unbounded variants of the SPSC lock-free queue\textsuperscript{35} from the FastFlow v2.0.4 parallel programming framework. These queues have been basically inspired by Lamport’s wait-free buffers and FastForward queues.\textsuperscript{27} On the other hand, to experiment with MPMC queues, we leverage 4 implementations from the state-of-the-art: the Michael and Scott’s (MS),\textsuperscript{25} the Morrison and Afek’s (LCRQ),\textsuperscript{29} Fatourou and Kallimanis’s (CC)\textsuperscript{36} and Yang Mellor-Crummey’s (wait free),\textsuperscript{30} as introduced in Section 3. The reference implementation of these MPMC queues is the one provided as supplemental material in the Yang Mellor-Crummey’s work.\textsuperscript{37}

- **Data race detector.** As for the compiler we used the LLVM compiler infrastructure v3.7.0 together with the compiler runtime libraries (compiler-rt) that supports the data race detector TSan. We installed the respective LLVM \texttt{libc++} and \texttt{libc++ ABI} to support the C++11 standard library for the FastFlow examples.

Featuring the aforementioned hardware and software, we perform the evaluation using a set of \( \mu \) benchmarks using individually unbounded and bounded SPSC queues and MPMC queues:

- SPSC \( \mu \) benchmarks. They primarily focus on testing internal structures and performance of specific FastFlow features. These tests can be effectively used as FastFlow \( \mu \) benchmarks leveraging both unbounded (uSPSC) and bounded (SPSC) versions of the SPSC lock-free queues. With this, we aim at testing internal workings of FastFlow and therefore analyzing all possible ways in which these SPSC queues are used. We run these applications with the default parameters. Note that we have manually added three synthetic tests using an uSPSC queue: \texttt{uspsc}, \texttt{uspsc\_r1}, and \texttt{uspsc\_r2}, being the last 2 faulty on purpose to evaluate the extension for detecting misuses.

- MPMC \( \mu \) benchmarks. This set focuses on testing the MPMC queue semantics. For that, it provides 4 synthetic benchmarks, each of them testing one of the aforementioned MPMC lock-free queue implementations. Concretely, each of the multithreaded benchmarks creates 12 producer and 12 consumer threads, pushing and popping 10,000 elements to/from the queue, respectively.

Furthermore, we leverage a set of real applications internally leveraging different FastFlow parallel patterns. Throughout the execution of these examples, we aim at internally enforcing the use of multiple bounded and unbounded SPSC queue instances in multiple ways. Table 6 provides more details about these applications. Note that all the aforementioned benchmarks were compiled using Clang compiler with the following additional flags \texttt{-std=\texttt{c++11}}, \texttt{-stdlib=\texttt{libc++}}, \texttt{-fsanitize=thread}, \texttt{-lc++abi}, \texttt{-O0-g}, \texttt{-fno-inline}, \texttt{-fnoomit-frame-pointer}, and \texttt{-fno-builtins}. (Note that we configured TSan to use the hybrid runtime.) Afterwards, we executed them using a fixed pool of 24 worker threads, ie, to fully populate the cores of IVY. To analyze the amount of false positives filtered out, we gathered data races reports generated by TSan, while to evaluate detection of misuses, we collected the results of the reports generated.

#### 7.1 | Analysis of global data races

Our first experiment analyzes statistically data races occurred during the execution of both SPSC and MPMC \( \mu \) benchmarks plus the application set with special focus on those related to the both queue types, respectively. Specially, we evaluate how the implementation of semantics aids to drop false positives, as the first contribution of this paper.

SPSC queues. We start by studying the impact of SPSC queue data races on the execution of the SPSC \( \mu \) benchmarks and applications. Figure 2 shows (percentagewise) the portion of SPSC-bounded and uSPSC queue-related data races with respect to the others for
both sets. Note that we consider part of the SPSC/uSPSC races those in which only one side was related to a function member of the SPSC/uSPSC queue classes. As observed for the \( \mu \)-benchmark set, roughly 71% of the data races, on average, were due to SPSC queues. A similar percentage can be appreciated for the application set (69%). Generally, these percentages give a notion of the importance of this kind of data races occurring in SPSC queue-related functions. In this concrete case, we also observe that uSPSC queues have little impact on the data races detected. This is due to uSPSC queues are internally implemented using multiple SPSC queues, and most of the conflicting functions only belong to SPSC queues.

As stated in Section 5, a contribution of the paper is to filter, whenever possible, those benign SPSC/uSPSC queue-related data races according to the Equation 1 and 2 stated in Section 4.2. Taking advantage of our implementation, in Figure 3, we classify data races into 3 different groups: benign, undefined, and harmful. These type of races are defined as follows:

- **Benign** data races represent those complying both requirements.
- **Undefined** data races stand for those in which TSan failed to restore the stack of one of the threads involved in the data race, and thus, the semantic requirements could not be checked.
- **Harmful** data races stand for those in which, at least, one of the requirements was violated.

Analyzing the percentage breakdown of the different groups, we observe small percentages of data races (about 5%) were classified as undefined. Since we are not aware of the specific cause that prevented TSan from restoring the stack, we are not confident to classify these data races as benign or harmful. A deep understanding of the TSan implementation is needed to understand the nature of such undefined races. This step will be considered part of the future work.

To gain insights into this issue, we performed an extra experiment considering the FastFlow implementation of the bounded and unbounded SPSC queues plus the Lamport version. These tests, buf_spsec, buf_uspsec, and buf_Lamport, corroborate that percentages of the undefined data races are independent of the queue version. Considering that all implementations are semantically correct but data races are still detected by TSan, we assume that they are all false positives.

Similarly, Table 7 combines the breakdown for the different types of data races at SPSC and application levels for both benchmark sets. Additionally, it incorporates figures representing the total number of data races, average of data races per test, and the corresponding percentages over the total data races detected on the application, regardless of their source. Note that the analysis of the SPSC

---

\( \text{ buf_spsec, buf_uspsec, and buf_Lamport, corroborate that percentages of the undefined data races are independent of the queue version. Considering that all implementations are semantically correct but data races are still detected by TSan, we assume that they are all false positives.} \)

---

\( ^{\text{The codes of these structures can be found in the FastFlow SVN repository https://sourceforge.net/projects/mc-fastflow, more specifically in file ff/buffer.hpp.}} \)
breakdown for the \( \mu \) benchmarks and applications sets has already been performed through Figures 3 and 2, respectively; however, SPSC and uSPSC queues appear merged in a single figure. In this table, we added 2 more subdivisions of data races: those related only to FastFlow but not to SPSC\( \times \)SPSC queues and others not related at all to any of these structures. As observed, these percentages represent approximately a third of the total reports.

Finally, the last 2 columns of Table 7 present figures without and with the data race filtering technique, respectively. As can be seen, we reduce about two-thirds of the number of warnings of data races for both sets tested. Being aware that in this case the filtering technique was performed with the SPSC and uSPSC queues, we are confident that more false positives would have been reduced if semantics for other parallel lock-free data structures had been taken into account.

MPMC queues. We perform a similar study for the MPMC queues data races on the execution of the MPMC \( \mu \) benchmarks. Figure 4 combines the percentage of MPMC queue races with respect to the total number of data races detected and the breakdown of MPMC data races for its specific \( \mu \) benchmarks. As can be seen in the first plot, the average percentage of MPMC data races represents about 65 % of the total data races detected. To gain insights into the root cause of these false data races, we review individually the different MPMC queue implementations of the MPMC \( \mu \) benchmark set.

- For the MS lock-free queue, we observe that most of false data races occurred mainly when a thread was enqueuing an item while other was executing an atomic CAS operation on a pop operation. We also detect that some data races occurred when a thread was performing another CAS operation when other thread was handling a hazard pointer. Note that hazard pointers are an approach to solve dynamic memory management in lock-free data structures dealing with the ABA problem. Obviously, all these data races were considered as false positives since the Michael and Scott’s implementation has been proven to be a correct lock-free structure.
- For the LCRQ lock-free queue, we detect a similar behavior regarding the occurrence of data races in the MS implementation. Basically, we note that a large part of these false positives were caused by a thread pushing an element to the queue while other thread was dealing with a CAS or FAA atomic operation. Although in these cases, TSan is, indeed, instrumenting atomic operations; it still reports data races. We believe that these aspects are not fully supported by TSan, as on the contrary these races would not have been reported. Regardless of these issues and thanks to the implemented semantics we are able to drop them in a safe way.
- The CC queue presents a different behavior. Taking into account that this structure is a blocking queue that leverages a coarse-grain lock; false data races detect are mainly given by synchronization primitives between threads pushing and popping data to/from the queue. In this case, TSan is also not able to discard them automatically. Furthermore, we note an important portion of data races occurring between 2 threads calling respectively to a serial push operation and a POSIX memalign routine.
- The wait-free queue presents a similar nature with respect to the MS and LCRQ queue implementations. In this case, the implementation leverages CAS and FAA operations, thus incurring in false TSan data race reports within these atomic operations. Specifically, we find data races occurring on a spinning thread while other thread was executing an atomic operation. We believe this is also an issue of the TSan race detector, as it is not capable to fully handle this kind of atomic operations.

Focusing on right-hand side plot of Figure 4, we appreciate a small percentage of undefined data races. As mentioned in the study of SPSC queues data races, undefined data races occurred when TSan was not
able to unwind the stack of the thread that wrote in the same memory address while the second was reading or writing on it. In the same way it was done for SPSC data races, Table 8 combines figures for the total, per test and percentage of data race warning messages for the aforementioned MPMC queue implementations. As can be seen, the average of benign data races detected is about 56%, while the undefined were roughly 8%. Other data races, not related to MPMC queues, occupied a third part of the total warnings generated. Overall, with the semantic verification we are able to detect benign data races and to drop about 56% of the total data race warnings reported to the end user.

### Evaluation of misuses of lock-free queues

In this section, we evaluate how a set of high-level semantics can improve the detection of specific misuses in lock-free data queues, even when TSan does not report any data race. Specifically, we test this feature using the special semantics for the FastFlow queues introduced in Section 5.2, as the second main contribution of this paper. To this end, we use a fault injection mechanism to assess whether misuses of SPSC queues can be detected in the form of data races or not, while we use semantics to really determine if they are real misuses. Therefore, we use the synthetic tests `uspac`, `uspac_r1`, `uspac_r2` and `uspac_lock`, being the last 3 faulty on purpose. Each of these tests uses an uSPSC queue in the following ways:

- **uspac**: comprised of a producer and a consumer thread, enqueuing and dequeuing 10 000 elements, respectively. Thus, this test makes good use of the uSPSC queue.
- **uspac_r1**: is composed of a producer and 2 consumer threads. While the producer pushes 10 000 elements, each of the concurrent consumers pop only 5000 items. Of course, it is a faulty test, as there are 2 consumers using the same uSPSC queue. Thus, it violates Equation 8.
- **uspac_r2**: contains a producer and a consumer threads handling 10 000 elements. In this case, the producer calls to `push`, but it also eventually calls to `pop`, so it violates Equation 9. Implicitly it also violates Equation 8, as there are exist 2 different threads calling to methods allotted to the consumer role.
- **uspac_lock**: ptest involves 3 threads: 2 producers, enqueuing 5000 elements each, and a consumer dequeuing 10 000 elements in the uSPSC queue. For this case, the calls to the `push` function in body loops of the producers are wrapped with corresponding calls to `pthread_mutex_lock` and `pthread_mutex_unlock`, sharing the same mutex variables. Therefore, the `push` calls are serialized, as they belong to a critical section. Obviously, the fact of having 2 producers using the same uSPSC queue violates Equation 8.

Figure 5 combines data race breakdown percentages presented in Figure 3 along with the percentage of queues that have been well and wrongly used for the aforementioned faulty tests. Looking at the test `uspac`, we note that all data races detected, even for SPSC and uSPSC are false positives; however, roughly 20% of the SPSC queues internally used by the main uSPSC queue have violated some semantic requirement. Since these are unexpected results, we deeply analyzed application traces to find out the root cause of the misuses. Particularly, the way that a FastFlow uSPSC queue dynamically manages internal SPSC

---

### Table 8: Statistics of data races for the MPMC benchmarks

| Benchmark set | Metrics | Total | Per test | Percentage, % |
|---------------|---------|-------|----------|---------------|
| MPMC μ benchmarks | Benign | 14.25 | 57 | 56.44 |
| | Undefined | 2.00 | 8 | 7.92 |
| | Harmful | 0.00 | 0 | 0.00 |
| | MPMC | 16.25 | 65 | 64.36 |
| | Others | 9.00 | 36 | 35.64 |
| | W/o MPMC | 25.25 | 101 | 100.00 |
| | W/ MPMC | 11.00 | 44 | 43.56 |

Abbreviation: MPMC, multiple-producer/multiple-consumer.
DOLZ ET AL.

8 CONCLUSIONS

Data race detectors aid to a great extent developers to easily identify data races in parallel applications. Several postmortem and dynamical approaches for data race detection have been implemented among a range of tools and plug-ins for compilation infrastructures. However, none of them is aware of the semantics behind the data races detected. The concurrent use of a shared resource within a correct lock-free parallel structure should not always imply a data race, unless its semantics are violated. In the same way, an application free of warning data reports does not entail that its internal data structures have been properly used. In this paper, we focused on the general of the SPSC and MPMC queues as for the lock-free parallel structures and leveraged, as a use cases, μ benchmarks and applications from FastFlow and several state-of-the-art MPMC queue implementations.

Being aware of the importance of these structures, we formalize the semantics of the SPSC and MPMC queues and build a set of requirements to determine whether a queue has been properly used or not. Afterwards, we implement the formalization of these semantics into TSan, a well-known dynamic data race detector among the LLVM Clang compiler. With it, we provide 2 novel features: (1) filtering data race warnings classified as false positives, and (2) detecting misuses via semantics of such lock-free data structures. The ability of detecting benign data races at runtime is a very helpful feature to prevent overwhelming users due to excessive false race reports. Also, it allows detecting misuses in data structures through a second level of verification semantics, even when data races are not detected. Through these extensions we demonstrate that we are able to discard, on average 60% of data races classified as false positives. We also observe that some wrong uses of lock-free data structures cannot be detected with a race detector but via high-level semantics.

For future work, we aim at supporting other kinds of lock-free data structures, such as hash tables, sets, red-black trees, etc. In general, we advocate that other lock-free data races can be supported as long as a formalization of their semantics is feasible. Also, we plan to use semantics to detect other types of catastrophic failures, eg, deadlocks, livelocks and lock starvation, and provide support for other architectures, such as PowerPC and ARM.

ACKNOWLEDGMENTS

This work was supported by the EU Project644235 “RePhrase: Refactoring Parallel Heterogeneous Resource-Aware Applications” under the programme ICT-09-2014.

REFERENCES

1. Borkar S. The exascale challenge. 2010 International Symposium on VLSI Design Automation and Test (VLSI-DAT), Hsin Chu, Taiwan; 2010:2–3. doi: 10.1109/VDAT.2010.5496640
2. Ashby S et al. The opportunities and challenges of Exascale computing. Summary Report of the Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee; 2010
3. Choi J, Dukhan M, Liu X, Vuduc R. Algorithmic time, energy, and power on candidate HPC compute building blocks. 2014 IEEE 28th International Parallel and Distributed Processing Symposium Phoenix, Arizona, USA; 2014:447–457. doi: 10.1109/IPDPS.2014.54
4. Boyapati C, Lee R, Rinard M. Ownership types for safe programming: preventing data races and deadlocks. OOPSLA ‘02 Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, Seattle, Washington, USA, vol. 37:11, ACM; 2002:211–230
5. Artho C, Havelund K, Biere A. High-level data races. Softw Test Verif Behav. 2003;13(4):207–227.
6. Danielutto M, Torquati M. Structured Parallel Programming with “core” FastFlow. In: Zsók V, Horváth Z, Csató L, eds. In Proceedings of the 5th Summer School CEPF 2013, Cluj-Napoca, Romania, LNCS, vol. 8606: Springer International Publishing, Switzerland; 2015:29–75.
7. Dolz MF, del Rio Astorga D, Fernández J, et al. Embedding semantics of the single-producer/single-consumer lock-free queue into
a race detection tool. Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM’16. ACM, New York, NY, USA; 2016:20–29. doi: 10.1145/2883404.2883406

8. Lu S, Park S, Seo E, Zhou Y. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XIII. ACM, New York, NY, USA; 2008:329–339. doi: 10.1145/1346281.1346323

9. Lampert L. Time, clocks, and the ordering of events in a distributed system. Commun ACM. July 1978;21(7):558–565. doi: 10.1145/359545.359563

10. Intel. Inspector xe 2017. https://software.intel.com/en-us/intel-inspector-xe. Accessed November 1, 2016.

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

12. Zhou P, Teodorescu R, Zhou Y. Hard: hardware-assisted lockset-based race detection. Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture, HPCA ’07. IEEE Computer Society, Washington, DC, USA; 2007:121–132. doi: 10.1109/HPCA.2007.346191

13. Engler D, Ashcraft K. RacerX: effective, static detection of race conditions and deadlocks. SIGOPS Oper Syst Rev. October 2003;37(5):237–252. doi: 10.1145/1165389.945468

14. Savage S, Burrows M, Nelson G, Sobalvarro P, Anderson T. Eraser: a dynamic data race detector for multithreaded programs. ACM T Comput Syst. 1997;15(4):391–411.

15. O’Callahan R, Choi J-D. Hybrid dynamic data race detection. SIGPLAN Not. June 2003;38(10):167–178.

16. Serebryany K, Ishkodzhanozov T. ThreadSanitizer: data race detection in practice. Proceedings of the Workshop on Binary Instrumentation and Applications, ACM, New York, NY, USA; 2009:62–71.

17. Narayanasamy S, Wang Z, Tigani J, Edwards A, Calder B. Automatically classifying benign and harmful data races using replay analysis. SIGPLAN Not. 2007;42(6):22–31.

18. Zhang J, Xiong W, Liu Y, Park S, Zhou Y, Ma Z. At-detector: improving the accuracy of a commercial data race detector by identifying address transfer. Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-44. ACM, New York, NY, USA; 2011:206–215. doi: 10.1145/2155620.2155645

19. Serebryany K, Potapenko A, Ishkodzhanozov T, Vyukov D. Dynamic Race Detection with LLVM Compiler. Berlin, Heidelberg: Springer Berlin Heidelberg; 2012:110–114.

20. Jannesari A, Tichy WF. Identifying ad-hoc synchronization for enhanced race detection. 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS), Atlanta, GA, USA; 2010:1–10. doi: 10.1109/IPDPS.2010.5470343

21. Xiong W, Park S, Zhang J, Zhou Y, Ma Z. Ad hoc synchronization considered harmful. OSDI; vol. 10; 2010:163–176.

22. Schweizer H, Besta M, Hoefer T. Evaluating the cost of atomic operations on modern architectures. 24th International Conference on Parallel Architectures and Compilation (PACT’15). ACM, San Francisco (CA), USA; 2015.

23. Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer’s Manual. 253669-033US; 2009.

24. AMD Corporation. Revision Guide for AMD Athlon 64 and AMD Opteron Processors. 25759; 2009.

25. Michael MM, Scott ML. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, PODC ’96. ACM, New York, NY, USA; 1996:267–275.

26. Shalev O, Shavit N. Split-ordered lists: lock-free extensible hash tables. JACM. May 2006;53(3):379–405.

27. Giacomoni J, Moseley T, Vachharajani M. Fastforward for efficient pipeline parallelism: A cache-optimized concurrent lock-free queue. Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP ’08. ACM, New York, NY, USA; 2008:43–52. doi: 10.1145/1345206.1345215

28. Lampert L. How to make a multiprocessor computer that correctly executes multithreaded programs. IEEE Trans Comput. September 1979;28(9):690–691. doi: 10.1109/TC.1979.1675439

29. Morrison A, Afek Y. Fast concurrent queues for x86 processors. Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP ’13. ACM, New York, NY, USA; 2013:103–112.

30. Yang C, Mellor-Crummey J. A wait-free queue as fast as fetch-and-add. Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP ’16. ACM, New York, NY, USA; 2016:16:1–16:13.

31. Lattner C, Adve V. LLVM: a compilation framework for lifelong program analysis & transformation. Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization, CGO ’04. IEEE Computer Society, Washington, DC, USA; 2004:75–.

32. Project S. The libunwind project. http://www.nongnu.org/libunwind/. Accessed September 1, 2016.

33. David A. DWARF Page. https://www.prevanders.net/dwarf.html. Accessed April 15, 2016.

34. Eager M. The DWARF Debugging Standard. http://dwarfstd.org/. Accessed April 15, 2016.

35. Aldinucci M, Danelutto M, Kilpatrick P, Meneghin M, Torquati M. An efficient unbounded lock-free queue for multi-core systems.. In: Kaklamakis C, Papathodorou T, Spirakis P, eds. Euro-Par 2012 Parallel Processing, LNCS, vol. 7484: Springer: Berlin Heidelberg; 2012:662–673. doi: 10.1007/978-3-642-32820-6 65

36. Fatourou P, Kallimanis ND. Revisiting the combining synchronization technique. SIGPLAN Not. February 2012;47(8):257–266. doi: 10.1145/2370036.2145849

37. Yang C, Mellor-Crummey J. Fast Wait Free Queue. https://github.com/chaoran/fast-wait-free-queue. Online; Accessed April 15, 2016.

How to cite this article: Dolz MF, Del Rio Astorga D, Fernández J, et al. Enabling semantics to improve detection of data races and misuses of lock-free data structures. Concurrency Computat: Pract Exper. 2017:29:e4114. https://doi.org/10.1002/cpe.4114