Efficient, Near Complete and Often Sound Hybrid Dynamic Data Race Prediction

Martin Sulzmann
Karlsruhe University of Applied Sciences
Karlsruhe, Germany
martin.sulzmann@gmail.com

Kai Stadtmüller
Karlsruhe University of Applied Sciences
Karlsruhe, Germany
kai.stadtmueller@live.de

ABSTRACT

Dynamic data race prediction aims to identify races based on a single program run represented by a trace. The challenge is to remain efficient while being as sound and as complete as possible. Efficient means a linear run-time as otherwise the method unlikely scales for real-world programs. We introduce an efficient, near complete and often sound dynamic data race prediction method that combines the lockset method with several improvements made in the area of happens-before methods. By near complete we mean that the method is complete in theory but for efficiency reasons the implementation applies some optimizations that may result in incompleteness. The method can be shown to be sound for two threads but is unsound in general. Experiments show that our method works well in practice.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Concurrency, Data race prediction, Happens before, Lockset

ACM Reference Format:
Martin Sulzmann and Kai Stadtmüller. 2020. Efficient, Near Complete and Often Sound Hybrid Dynamic Data Race Prediction. In *Proceedings of* . ACM, New York, NY, USA, 23 pages.

1 INTRODUCTION

We consider verification methods in the context of concurrently executing programs that make use of multiple threads, shared reads and writes, and acquire/release operations to protect critical sections. Specifically, we are interested in data races. A data race arises if two unprotected, conflicting read/write operations from different threads happen at the same time.

Detection of data races via traditional run-time testing methods where we simply run the program and observe its behavior can be tricky. Due to the highly non-deterministic behavior of concurrent programs, a data race may only arise under a specific schedule. Even if we are able to force the program to follow a specific schedule, the two conflicting events many not not happen at the same time. Static verification methods, e.g. model checking, are able to explore the entire state space of different execution runs and their schedules. The issue is that static methods often do not scale for larger programs. To make them scale, the program's behavior typically needs to be approximated which then results in less precise analysis results.

The most popular verification method to detect data races combines idea from run-time testing and static verification. Like in case of run-time testing, a specific program run is considered. The operations that took place are represented as a program trace. A trace reflects the interleaved execution of the program run and forms the basis for further analysis. The challenge is to predict if two conflicting operations may happen at the same time even if these operations may not necessarily appear in the trace right next to each other. This approach is commonly referred to as *dynamic data race prediction*.

Run-Time Events and Traces. For example, consider the following trace

	$1\sharp$	$2\sharp$
1.	$w(x)$	
2.	$acq(y)$	
3.	$rel(y)$	
4.		$acq(y)$
5.		$w(x)$
6.		$rel(y)$

where for each thread we introduce a separate column and the trace position can be identified via the row number. Events $w(x)/r(x)$ refer to write/read events on the shared variable x. Events $acq(y)/rel(y)$ refer to acquire/release events on lock variable y. To identify an event, we often annotate the event with its thread id and row number. For example, $1\sharp w(x)_1$ refers to the write event in thread 1 at trace position 1. We sometimes omit the thread id as the trace position (row number) is sufficient to unambiguously identify an event.

Conflicting Events and Data Race Prediction. Let e, f be two read/write events on the same variable where at least one of them is a write event and both events result from different threads. Then, we say that e and f are two *conflicting events*. For the above trace, we find that $1\sharp w(x)_1$ and $2\sharp w(x)_5$ are two conflicting events. Based on the trace we wish to predict if two conflicting events can appear right next to each other. Such a situation represents a *data race*.

In the above trace, the two conflicting events $w(x)_1$ and $w(x)_5$ do not appear right next to each other in the trace. Hence, it seems that both events are not in a race. The point is that a trace represents *one* possible interleaving of concurrent events but there may be other *alternative* interleavings that result from scheduling the events slightly differently. The challenge of data race prediction is to find an alternative interleaving of the trace such two conflicting events appear right next to each other.

, ,
2020.

We could explore alternative interleavings by considering all trace reorderings, i.e. all permutations of events in the trace. In general, this is (a) too inefficient, and (b) leads to false results as the data race may not be reproducible by re-running the program. As we only consider the trace and not the program we impose the following assumptions on a *correctly reordered* trace. (1) The program order as found in each thread is respected. (2) Every read sees the same (last) write. (3) The lock semantics is respected so that execution will not get stuck.

For our running example, $[2\sharp acq(y)_4, 2\sharp w(x)_5, 1\sharp w(x)_1]$ is a correctly reordered prefix. We use here list notation to represent the trace. This reordered trace serves as a witness for the data race among the two conflicting events $w(x)_1$ and $w(x)_5$. We consider prefixes as we can 'stop' the trace as soon as the two conflicting events have appeared right next to each other.

First versus Subsequent Races. Earlier works [Kini et al. 2017; Smaragdakis et al. 2012] only consider the first race based on a total order of the occurrence of events in the original trace. One reason is that a subsequent race may only show itself due to an earlier race. As the program behavior may be undefined after the first race, the subsequent race many not be reproducible.

However, it is easy to fix the first race by making the events mutually exclusive. The former subsequent race becomes a first. To discover this race we would need to re-run the analysis. Hence, it is sensible to report all races and not only the first race.

Here is an example to illustrate this point.

	$1\sharp$	$2\sharp$
1.	$r(y)$	
2.	$r(x)$	
3.		$w(y)$
4.		$w(x)$

	$1\sharp$	$2\sharp$
1.	$acq(y')$	
2.	$r(y)$	
3.	$rel(y')$	
4.	$r(x)$	
5.		$acq(y')$
6.		$w(y)$
7.		$rel(y')$
4.		$w(x)$

For the trace on the left, $r(y)_1$ and $w(y)_3$ are in a race as shown by $[1\sharp r(y)_1, 2\sharp w(y)_3]$.

What about $r(x)_2$ and $w(x)_4$? For any reordering where $r(x)_2$ and $w(x)_4$ appear right next to each other we find that earlier in the trace $r(y)_1$ and $w(y)_3$ appear right next to each other. For instance, consider $[1\sharp r(y)_1, 2\sharp w(y)_3, 2\sharp r(x)_2, 1\sharp w(x)_4]$. Hence, $r(x)_2$ and $w(x)_4$ represent a subsequent race.

We can easily fix the first race by making the events involved mutually exclusive. See the trace on the right. The subsequent race becomes now a first race.

Our Goals and Contributions. For a given trace T, we wish to identify *a*ll predictable data races in T. This includes first and subsequent races as well. We write $\mathcal{P}^T$ to denote the set of all *p*redictable data race pairs (e, f) resulting from T where e, f are conflicting events in T and there exists a correctly reordered prefix of T under which e, f appear right next to each other.

The challenge is to be *efficient*, *sound* and *complete*. By efficient we mean a run-time that is linear in terms of the size of the trace. Sound means that races reported by the algorithm can be observed via some appropriate reordering of the trace. If unsound, we refer to wrongly a classified race as a *false positive*. Complete means that all valid reorderings that exhibit some race can be predicted by the algorithm. If incomplete, we refer to any not reported race as a *false negative*.

In this paper, we make the following contributions:

- We propose an efficient dynamic race prediction method that combines the lockset method with the happens-before method. Our method is novel and improves the state-of-the art. The method is shown to be complete in general and sound for the case of two threads (Section 3).
- We give a detailed description of how to implement our proposed method (Section 4). We present an algorithm that overall has quadratic run-time. This algorithm can be turned into a linear run-time algorithm by sacrificing completeness. For practical as well as contrived examples, incompleteness is rarely an issue.
- We carry out extensive experiments covering a large set of real-world programs as well as a collection of the many challenging examples that can be found in the literature. For experimentation, we have implemented our algorithm as well as its contenders in a common framework. We measure the performance, time and space behavior, as well as the precision, e.g. ratio of false positives/negatives etc. Measurements show that our algorithm performs well compared to state-of-the art algorithms such as ThreadSanitizer, FastTrack, SHB and WCP (Section 5).

The upcoming section gives an overview of our work and includes also a comparison against closely related works. Section 6 summarizes related work. Section 7 concludes. The appendix contains additional material such as proofs, extended examples, optimization details etc.

2 HAPPENS-BEFORE AND LOCKSET

We review earlier efficient data race prediction methods and discuss their limitations.

Happens-Before Methods. The idea is to is to derive from the trace a happens-before relation among events. If for two conflicting events, neither event happens before the other event, this is an indication that both events can appear next to each other. Happens-before methods can be implemented efficiently via the help of vector clocks [Fidge 1992; Mattern 1989]. However, none of the existing happens-before relations [Kini et al. 2017; Lamport 1978; Mathur et al. 2018] is sound and complete.

For example, Lamport's happens-before relation [Lamport 1978], referred to as the HB relation, is neither sound nor complete as shown by the following example.

Example 2.1. Consider the following two traces.

Trace A

	$1\sharp$	$2\sharp$
1.	$w(x)$	
2.	$acq(y)$	
3.	$rel(y)$	
4.	HB	$acq(y)$
5.		$w(x)$
6.		$rel(y)$

Trace B

	$1\sharp$	$2\sharp$
1.		$w(y)$
2.	$w(x)$	
3.	$w(y)$	
4.	SHB	$r(y)$
5.		$w(x)$

Consider trace A. The HB relation orders critical sections based on their position in the trace and therefore $rel(y)_3 <^{HB} acq(y)_4$ where $<^{HB}$ denotes the HB ordering relation. Hence, we find that $w(x)_1 <^{HB} w(x)_5$. This is a false negative. We are allowed to reorder the two critical sections and then two writes on x would appear right next to each other. Take $T' = [2\sharp acq(y)_4, 2\sharp w(x)_5, 1\sharp w(x)_1]$ where T' represents an alternative schedule.

Consider trace B. There are no critical sections. Hence, the conflicting events $w(x)_2$ and $w(x)_5$ are unordered under the HB relation. This is a false positive. We assume that programs are executed under the sequential consistency memory model [Adve and Gharachorloo 1996]. Hence, any reordering to exhibit the race among the writes on x violates the condition that each read must see the same (last) write. Consider the reordering

$$T' = [2\sharp w(y)_1, 2\sharp r(y)_4, 2\sharp w(x)_5, 1\sharp w(x)_2].$$

In the original trace, the last write for $r(y)_4$ is $w(y)_3$ but this does not apply to T'. As the read sees a different write, there is no guarantee that the events after the read on y would take place.

Mathur, Kini and Viswanathan [Mathur et al. 2018] show that the HB relation is only sound for the first race reported. They introduce the schedulable happens-before (SHB) relation $<^{SHB}$. The SHB relation additionally includes write-read dependencies and therefore the two writes on x in the above trace B are ordered under the SHB relation. The SHB relation is sound in general but still incomplete as critical sections are ordered by their position in the trace.

Kini, Mathur and Viswanathan [Kini et al. 2017] introduce the weak-causally precedes (WCP) relation. Unlike HB and SHB, WCP reorders critical sections under some conditions. Recall trace A from Example 2.1. Under WCP, events $w(x)_1$ and $w(x)_5$ are unordered. Hence, WCP is more complete compared to HB and SHB. Like HB, subsequent WCP races may be false positives. See trace B in Example 2.1 where $w(x)_2$ and $w(x)_5$ are not ordered under WCP but this represents a false positive.

The WCP relation improves over the HB and SHB relation by being more complete. However, WCP is still incomplete in general as shown by the following example.

Example 2.2. Consider

	$1\sharp$	$2\sharp$
1.	$w(x)$	
2.	$acq(y)$	
3.	$w(x)$	
4.	$rel(y)$	
5.		$acq(y)$
6.	WCP	$w(x)$
7.		$rel(y)$

Events $w(x)_1$ and $w(x)_6$ are in a predictable data race as witnessed by the following correctly reordered prefix $T' = [acq(y)_5, w(x)_1, w(x)_6]$. WCP is unable to predict this race.

The two critical sections contain conflicting events and therefore $rel(y)_4 <^{WCP} w(x)_6$. Then, we find that $w(x)_1 <^{WCP} w(x)_6$.

Lockset Method. A different method is based on the idea to compute the set of locks that are held when processing a read/write event [Dinning and Schonberg 1991]. We refer to this set as the *lockset*. For each event e we compute its lockset $LS(e)$ where $y \in LS(e)$ if $e \in CS(y)$ for some critical section $CS(y)$. Two conflicting events that are in a race if their locksets are disjoint.

The computation of locksets is efficient and it is straightforward to show that the lockset method is complete. However, on its own the lockset method produces many false positives as shown by our experiments later.

Hybrid Methods. The idea of Genç, Roemer, Xu and Bond [Genç et al. 2019] is to pair up the lockset method with happens-before. They introduce the strong-dependently precedes (SDP) and the weak-dependently precedes (WDP) relation. SDP and WDP are weaker compared to the earlier relations we have seen where WDP is even weaker compared to SDP. The lockset test is necessary to rule out (some) false positives.

Compared to WCP, SDP does not order critical sections if the conflicting events are only writes and there is no read that follows the write in the later critical section. Hence, under SDP the two writes on x in Example 2.2 are unordered. By weakening the WCP relation, the SDP relation on its own is no longer strong enough to rule out false positives (in case of of the first reported race).

Example 2.3. Consider

	$1\sharp$	$2\sharp$
1.	$acq(y)$	
2.	$w(x)$	
3.	$rel(y)$	
4.		$acq(y)$
5.		$w(x)$
6.		$rel(y)$

The two writes on x are unordered under SDP but there is obviously no race as both writes are part of a critical section that involves the same lock y. To deal with such cases, the SDP relation is paired with the lockset test.

SDP improves over WCP in case of write-write conflicting critical sections. But as all other WCP conditions are still in place SDP remains incomplete.

Example 2.4. Consider

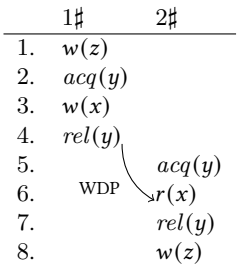
	$1\sharp$	$2\sharp$
1.	$w(x)$	
2.	$acq(z)$	
3.	$r(x)$	
4.	$w(y)$	
5.	$rel(z)$	
6.		$acq(z)$
7.	SDP	$w(x)$
8.		$rel(z)$
9.		$w(y)$

There is a read-write conflict on x within the two critical sections. Hence, under SDP we find that $w(y)_4 <^{SDP} w(y)_9$. This is a false negative as there is a correct reordering under which both events appear right next to each other.

To achieve completeness, Genç et al. [2019] introduce the WDP relation. WDP pretty much drops all of SDP's ordering conditions among critical sections. Two critical sections are ordered if one

contains a write and the other a conflicting read where the write is the read's last write.

Example 2.5. Consider

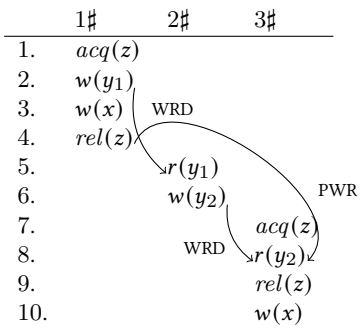


We find that $rel(y)_4 <^{WDP} r(x)_6$ and therefore the two writes on z are ordered under WDP.

The WDP ordering condition among critical section is a necessary condition. Genç et al. [2019] show that for any predictable race the events involved are unordered under WDP and their locksets are disjoint. That is, the WDP relation combined with the lockset test is complete.

Our Work. We further strengthen the WDP relation while maintaining completeness. Our approach is to strictly impose write-read dependencies (WRD) as employed by the SHB relation in Mathur et al. [2018]. This allows us to filter out more false positives and also improves the running time of the algorithm.

Example 2.6. Consider

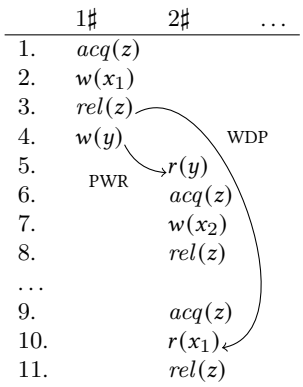


WDP reports that the two writes on x are in a race. This is a false positive.

Our PWR relation includes the WRD relations $w(y_1)_2 <^{WRD} r(y_1)_5$ and $w(y_2)_6 <^{WRD} r(y_2)_8$ and therefore $rel(z)_4 <^{PWR} r(y_2)_8$. PWR stands for program order, write-read dependency order and ordered critical sections (if events involved are ordered). Hence, we find that the two writes in x are ordered under PWR. PWR is stronger compared to WDP and therefore admits fewer false positives. We can show that PWR in combination with lockset is complete.

At the algorithmic level, PWR has performance benefits as shown by the following example.

Example 2.7. Consider



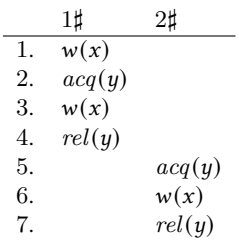
To check if two critical sections are ordered, the algorithm that implements the WDP relation needs to maintain a history of critical sections. For each critical section, we record (1) the writes for each variable, and (2) the happens-before time for the release. If there is a subsequent critical section (for the same lock) with a read where the last write is in some earlier critical section, then we need to enforce the WDP relation. See $rel(z)_3 <^{WDP} r(x_1)_{10}$.

The size of the history of critical sections as well as the writes per critical section can be significantly large. Our experiments show that this can have a significant impact on the performance. PWR improves over WDP as we do not maintain writes per critical section and can more aggressively remove critical sections.

For our example, due to the write-read dependency involving variable y, the critical sections in thread 1 and 2 are ordered under PWR. Hence, thread 2 does not need to record thread 1's critical section at all. Furthermore, we only need to record to the happens-before time of the acquire instead of all writes that are part of this critical section.

Another important contribution of our work is that we introduce a complete algorithm that computes all predictable data race pairs. The algorithm that implements the WDP relation is incomplete as shown by the following example.

Example 2.8. Consider



There is a predictable race among $w(x)_1$ and $w(x)_6$. Our algorithm that implements PWR reports this race but the algorithm that implements WDP, see Algorithm 2 in Genç et al. [2019], does not report a race here.

The issue is that $w(x)_1$ happens before $w(x)_3$ (due to program order). Algorithm 2 in Genç et al. [2019] only keeps the most 'recent' write per thread. Hence, we have forgotten about $w(x)_1$ as we only kept $w(x)_3$ by the time we reach $w(x)_6$. Events $w(x)_3$ and $w(x)_6$

are unordered under PWR but they share a common lockset. Hence, Algorithm 2 reports no race.

Our algorithm additionally records that $w(x)_1 <^{PWR} w(x)_3$. Via $w(x)_3$ we can derive that there is another potential race candidate $w(x)_1$ that might be in a race with $w(x)_6$. Their locksets are disjoint and thus we report the race.

Maintaining $w(x)_1 <^{PWR} w(x)_3$ and identifying additional race candidates requires extra time and space. Our algorithm requires a quadratic time and space. We apply some optimizations under which we obtain an efficient algorithm that runs in linear time and space. The optimization may lead to incompleteness. Our experiments show that this is mostly an issue in theory but not for practical examples.

The upcoming section formalizes the PWR relation. Section 4 covers the implementation. Experiments are presented in Section 5.

3 THE PWR RELATION

We formally define the PWR relation.

Definition 3.1 (PO + WRD + ROD). Let T be a trace. We define a relation $<^{PWR}$ among trace events as the smallest partial order that satisfies the following conditions:

- **Program order (PO):** Let $e, f \in T$ where $thread(e) = thread(f)$ and $pos(e) < pos(f)$. Then, we have that $e <^{PWR} f$.
- **Write-read dependency (WRD):** Let $w(x)_j, r(x)_k \in T$ where $w(x)_j$ is the last write of $r(x)_k$. That is, $j < k$ and there is no other $w(x)_l$ such that $j < l < k$. Then, we have that $w(x)_j <^{PWR} r(x)_k$.
- **Release-order dependency (ROD):** Let $e, f \in T$ be two events. Let $CS(y), CS(y)'$ be two critical sections where $e \in CS(y)$, $f \in CS(y)'$ and $e <^{PWR} f$. Then, we have that $rel(CS(y)) <^{PWR} f$.

We refer to $<^{PWR}$ as the *PO + WRD + ROD* (PWR) relation.

We distinguish between write-write, read-write and write-read race pair candidates. Write-write and read-write candidates are not ordered under PWR. For write-read candidates we assume that the write is the last write for the read under PWR.

Definition 3.2 (Lockset + PWR Write-Write and Read-Write Check). Let T be a trace where e, f are two conflicting events such that (1) $LS(e) \cap LS(f) = \emptyset$, (2) neither $e <^{PWR} f$ nor $f <^{PWR} e$, and (3) (e, f) is a write-write or read-write race pair. Then, we say that (e, f) is a *potential Lockset-PWR* data race pair.

Definition 3.3 (Lockset + PWR WRD Check). Let T be a trace. Let e, f be two conflicting events such that e is a write and f a read where $LS(e) \cap LS(f) = \emptyset$, $e <^{PWR} f$ and there is no g such that $e <^{PWR} g <^{PWR} f$. Then, we say that (e, f) is a *potential Lockset-PWR WRD* data race pair.

Definition 3.4 (Potential Race Pairs via Lockset + PWR). We write $\mathcal{R}^T_{<^{PWR}}$ to denote the set of all potential Lockset-PWR (and WRD) data race pairs as characterized by Definitions 3.2 and 3.3.

Proposition 3.5 (Lockset + PWR Completeness). *Let T be a trace. Let $e, f \in T$ such that $(e, f) \in \mathcal{P}^T$. Then, we find that $(e, f) \in \mathcal{R}^T_{<^{PWR}}$.*

Recall that $\mathcal{P}^T$ denotes the set of all predictable data race pairs (see the introduction).

We compare PWR against WDP.

Definition 3.6 (Weak-Dependently Precedes (WDP) [Genç et al. 2019]). Let T be a trace. We define a relation $<^{WDP}$ among trace events as the smallest partial order that satisfies condition PO as well as the following conditions:

- **Weak Release-Conflict Dependency (RCD):** Let $e, f \in T^{rw}_x$ be two conflicting events such that f is a read event and e is f''s last write event. Let $CS(y), CS(y)'$ be two critical sections where $f \in CS(y)$, $e \in CS(y)'$, $pos(rel(CS(y))) < pos(e)$. Then, $rel(CS(y)) <^{WDP} e$.
- **Release-Release Dependency (RRD):** Let $e, f \in T$ be two events. Let $CS(y), CS(y)'$ be two critical sections where $e \in CS(y)$, $f \in CS(y)'$ and $e <^{WDP} f$. Then, we have that $rel(CS(y)) <^{WDP} rel(CS(y'))$.

We refer to $<^{WDP}$ as the *weak-dependently precedes* (WDP) relation. [1]

Like PWR, the WDP relation in combination with the lockset check is complete. However, the PWR relation is stronger and therefore allows us to rule out more false positives.

Proposition 3.7 (PWR versus WDP). *We have that $<^{WDP} \subseteq <^{PWR}$ but the reverse direction does not hold in general.*

We can also state the that Lockset-PWR check is sound under certain conditions.

Proposition 3.8 (Lockset + PWR Soundness for Two Threads). *Let T be a trace that consists of at most two threads. Then, any potential Lockset-PWR data race pair with an empty lockset is also a predictable data race pair.*

Not every pair in $\mathcal{R}^T_{<^{PWR}}$ is predictable.

Example 3.9. Consider the following trace.

	1♯	2♯	3♯	4♯
1.	$acq(y)$			
2.	$w(z_1)$			
3.		$r(z_1)$		
4.		$\underline{w(x)}$		
5.		$w(z_2)$		
6.	$r(z_2)$			
7.	$rel(y)$			
8.			$acq(y)$	
9.			$w(z_3)$	
10.				$r(z_3)$
11.				$\underline{w(x)}$
12.				$w(z_4)$
13.			$r(z_4)$	
14.			$rel(y)$	

Due to the write-read dependencies involving variables z_1, z_2, z_3, z_4, the two writes on x are protected by the lock y. Hence, the pair

[1] Compared to the original WDP definition [Genç et al. 2019] we do not distinguish between branch-dependent and branch-independent reads. We assume that all reads are branch-dependent. That is, each read may affect the control flow. This is more conservative but requires a much simpler tracing scheme where we do not have to inspect the program text.

$(w(x)_4, w(x)_{11})$ is not a predictable data race pair. However, under PWR events $w(x)_4, w(x)_{11})$ are unordered and their lockset is empty. Hence, the Lockset-PWR method (falsely) reports the potential data race pair $(w(x)_4, w(x)_{11})$.

In the above example, $w(x)_4$ and $w(x)_{11})$ is not the first potential race. The WRDs on z_1, z_2, z_3 and z_4 are unprotected. Hence, the first potential race involves $w(z_1)_2$ and $r(z_1)_3$ (and this race is a predictable race). However, each WRD can be protected via their own private lock. Then, $w(x)_4$ and $w(x)_{11})$ becomes the first potential race reported but this race is still a false positive.

Soundness is certainly an important property. However, methods such as HB, WCP and SDP only guarantee that the first race reported is sound but subsequent races may be false positives. Algorithms/tools based on these methods commonly report as many (subsequent) races as possible. In this light, we argue that the potential unsoundness of the Lockset-PWR check is not a serious practical issue. The key advantage of PWR is that we can reduce the number of false positives compared to WDP. The upcoming section shows how to compute $\mathcal{R}^T_{<_{PWR}}$. Our experiments show that our method works well in practice.

4 THE PWR^{E+E} ALGORITHM

Algorithm PWR^{E+E} computes $\mathcal{R}^T_{<_{PWR}}$. We start with an overview.

4.1 Overview

To implement the PWR relation we combine ideas found in FastTrack [Flanagan and Freund 2010], SHB [Mathur et al. 2018] and WCP [Kini et al. 2017]. For example, we employ vector clocks and the more optimized epoch representation (FastTrack), we manage a history of critical sections (WCP) and track write-read dependencies (SHB). Like the above algorithms, our algorithm also processes events in a stream-based fashion and maintains a set $RW(x)$ of most recent reads/writes that are concurrent. By concurrent we mean that the events are unordered under PWR. Elements in $RW(x)$ are represented by their epoch where each epoch allows us to uniquely identify the corresponding event.

Recall Example 2.8 where we annotate the trace with $RW(x)$. For brevity, we omit vector clocks. Instead of epochs, we write w_i for a write at trace position i. A similar notation is used for reads.

	$1\sharp$	$2\sharp$	$RW(x)$
1.	$w(x)$		$\{w_1\}$
2.	$acq(y)$		
3.	$w(x)$		$\{w_3\}$
4.	$rel(y)$		
5.		$acq(y)$	
6.		$w(x)$	$\{w_3, w_6\}$
7.		$rel(y)$	

We consider the various states of $RW(x)$ while processing events. At trace position three, w_3 replaces w_1 due to the program order. At trace position six, we find $RW(x) = \{w_3, w_6\}$. Under PWR, w_3 and w_6 are concurrent but we do not report a race because their locksets share a common lock.

The issue is that there is a race among w_1 and w_6 but this race is not reported by standard single pass algorithms [Flanagan and Freund 2010; Genç et al. 2019; Kini et al. 2017; Mathur et al. 2018]. The reason is that $RW(x)$ maintains only the most recent concurrent reads/writes. See the above example where w_1 is replaced by w_3.

To cope with this issue we follow the SHB^{E+E} two-pass algorithm [Sulzmann and Stadtmüller 2019]. In a first pass, we (1) maintain a history of replaced events and (2) reads/writes that are concurrent. The second pass traverses the history to discover all conflicting concurrent events.

Here is our running example where this additional information has been annotated.

	$1\sharp$	$2\sharp$	$RW(x)$	$edges(x)$	$conc(x)$
1.	$w(x)$		$\{w_1\}$		
2.	$acq(y)$				
3.	$w(x)$		$\{w_3\}$	$w_1 \prec w_3$	
4.	$rel(y)$				
5.		$acq(y)$			
6.		$w(x)$	$\{w_3, w_6\}$		(w_3, w_6)
7.		$rel(y)$			

The history is represented as a set $edges(x)$ (E). Nodes connected via edges are reads/writes and can efficiently be represented via epochs (E). Concurrent events are stored in $conc(x)$. At trace position three, we record that w_3 replaces w_1. At trace position six, we record that w_3 and w_6 are concurrent under PWR.

Reporting of races is done in a second pass where we exploit the information recorded in $edges(x)$ and $conc(x)$. We consider all pairs in $conc(x)$. If their locksets are disjoint we report a race. This does not apply to the pair (w_3, w_6), however, this pair is crucial to discover further races. From (w_3, w_6) via $w_1 \prec w_3$ we obtain a further potential race candidate pair (w_1, w_6). Their locksets are disjoint and therefore we report the race pair (w_1, w_6). Thus, we are able to compute $\mathcal{R}^T_{<_{PWR}}$.

The first pass enjoys the same time complexity as earlier algorithms [Flanagan and Freund 2010; Genç et al. 2019; Kini et al. 2017; Mathur et al. 2018]. The second pass comes with an additional quadratic run-time. By limiting the size of elements in $edges(x)$, the second pass of traversing $edges(x)$ can be integrated into the first pass where we build up $edges(x)$. This might lead to incompleteness but yields an efficient, linear run-time algorithm. For practical examples it turns out that only maintaining a maximum of 25 edge constraints at a time is a good compromise.

4.2 First Pass

Algorithm 1 specifies the first pass of PWR^{E+E} and computes $edges(x)$ and $conc(x)$. Events are processed in a stream-based fashion. For each event we find a procedure that deals with this event. We immediately report write-read races. Reporting of write-write and read-write races takes place in a second pass.

We compute the lockset for read/write events and check if read-/write events are concurrent by establishing the PWR relation. To check if events are in PWR relation we make use of vector clocks and epochs. We first define vector clocks and epochs and introduce various state variables maintained by the algorithm that rely on these concepts.

For each thread i we compute the current set $LS_t(i)$ of locks held by this thread. We use $LS_t(i)$ to avoid confusion with the earlier introduced set $LS(e)$ that represents the lockset for event e. We

Algorithm 1 PWR^{E+E} algorithm (first pass)

```
1:  function w3(V, LS_t)
2:      for y ∈ LS_t do
3:          for (j♯k, V') ∈ H(y) do
4:              if k < V[j] then
5:                  V = V ⊔ V'
6:              end if
7:          end for
8:      end for
        return V
9:  end function
1:  procedure acquire(i, y)
2:      Th(i) = w3(Th(i), LS_t(i))
3:      LS_t(i) = LS_t(i) ∪ {y}
4:      Acq(y) = i♯Th(i)[i]
5:      inc(Th(i), i)
6:  end procedure
1:  procedure release(i, y)
2:      Th(i) = w3(Th(i), LS_t(i))
3:      LS_t(i) = LS_t(i) − {x}
4:      H(y) = H(y) ∪ {(Acq(y), Th(i))}
5:      inc(Th(i), i)
6:  end procedure
1:  procedure write(i, x)
2:      Th(i) = w3(Th(i), LS_t(i))
3:      evt = {(i♯Th(i)[i], Th(i), LS_t(i))} ∪ evt
4:      edges(x) = {j♯k ≺ i♯Th(i)[i] | j♯k ∈ RW(x) ∧ k < Th(i)[j]} ∪ edges(x)
5:      conc(x) = {(j♯k, i♯Th(i)[i]) | j♯k ∈ RW(x) ∧ k > Th(i)[j]} ∪ conc(x)
6:      RW(x) = {i♯Th(i)[i]} ∪ {j♯k | j♯k ∈ RW(x) ∧ k > Th(i)[j]}
7:      L_W(x) = Th(i)
8:      L_{W_t}(x) = i
9:      L_{W_L}(x) = LS_t(i)
10:     inc(Th(i), i)
11: end procedure
1:  procedure read(i, x)
2:      j = L_{W_t}(x)
3:      if Th(i)[j] < L_W(x)[j] ∧ LS_t(i) ∩ L_{W_L}(x) = ∅ then
4:          reportPotentialRace(i♯Th(i)[i], j♯L_W(x)[j])
5:      end if
6:      Th(i) = Th(i) ⊔ L_W(x)
7:      Th(i) = w3(Th(i), LS_t(i))
8:      evt = {(i♯Th(i)[i], Th(i), LS_t(i))} ∪ evt
9:      edges(x) = {j♯k ≺ i♯Th(i)[i] | j♯k ∈ RW(x) ∧ k < Th(i)[j]} ∪ edges(x)
10:     conc(x) = {(j♯k, i♯Th(i)[i]) | j♯k ∈ RW(x) ∧ k > Th(i)[j]} ∪ conc(x)
11:     RW(x) = {i♯Th(i)[i]} ∪ {j♯k | j♯k ∈ RW(x) ∧ k > Th(i)[j]}
12:     inc(Th(i), i)
13: end procedure
```

have that $LS(e) = LS_t(i)$ where $LS_t(i)$ is the set at the time we process event e. Initially, $LS_t(i) = \emptyset$ for all threads i.

The algorithm also maintains several vector clocks.

Definition 4.1 (Vector Clocks). A *vector clock* V is a list of *time stamps* of the following form.

$$V \quad ::= \quad [i_1, \ldots, i_n]$$

We assume vector clocks are of a fixed size n. Time stamps are natural numbers and each time stamp position j corresponds to the thread with identifier j.

We define

$$[i_1, \ldots, i_n] \sqcup [j_1, \ldots, j_n] \quad = \quad [\mathsf{max}(i_1, j_1), \ldots, \mathsf{max}(i_n, j_n)]$$

to synchronize two vector clocks by building the point-wise maximum.

We write $V[j]$ to access the time stamp at position j. We write $\mathsf{inc}(V, j)$ as a short-hand for incrementing the vector clock V at position j by one.

We define vector clock V_1 to be smaller than vector clock V_2, written $V_1 < V_2$, if (1) for each thread i, i's time stamp in V_1 is smaller or equal compared to i's time stamp in V_2, and (2) there exists a thread i where i's time stamp in V_1 is strictly smaller compared to i's time stamp in V_2.

If the vector clock assigned to event e is smaller compared to the vector clock assigned to f, then we can argue that e happens before f. For $V_1 = V_2 \sqcup V_3$ we find that $V_1 \leq V_2$ and $V_1 \leq V_3$.

For each thread i we maintain a vector clock $Th(i)$. For each shared variable x we find vector clock $L_W(x)$ to maintain the last write access on x. Initially, for each vector clock $Th(i)$ all time stamps are set to 0 but position i where the time stamp is set to 1. For $L_W(x)$ all time stamps are set to 0.

To efficiently record read and write events we make use of epochs [Flanagan and Freund 2010].

Definition 4.2 (Epoch). Let j be a thread id and k be a time stamp. Then, we write $j\sharp k$ to denote an *epoch*.

Each event e can be uniquely associated to an epoch $j\sharp k$. Take its vector clock and extract the time stamp k for the thread j the event e belongs to. For each event this pair of information represents a unique key to locate the event. Hence, we sometimes abuse notation and write e when referring to the epoch of event e.

Via epochs we can also check if events are in a happens-before relation without having to take into account the events vector clocks.

Proposition 4.3 (FastTrack [Flanagan and Freund 2010] Epochs). *Let T be some trace. Let e, f be two events in T where (1) e appears before f in T, (2) e is in thread j, and (3) f is in thread i. Let V_1 be e's vector clock and V_2 be f's vector clock computed by the FastTrack algorithm. Then, we have that e and f are concurrent w.r.t. the $<^{HB}$ relation iff $V_2[j] < V_1[j]$.*

HB-concurrent holds when comparing vector clocks $V_2 < V_1$. If $V_2[j] < V_1[j]$ then the vector clocks of thread j and i have not been synchronized. Therefore, e and f must be concurrent. Similar argument applies for the direction from right to left. Our algorithm is an extension of FastTrack. Hence, the above property carries over to our algorithm and the PWR relation.

For each lock variable y, we find $Acq(y)$ to record the last entry point to the critical section guarded by lock y. $Acq(y)$ is represented by an epoch. The set $H(y)$ maintains the lock history for lock variables y. For each critical section we record the pair $(Acq(y), V)$ where $Acq(y)$ is the acquire's epoch and V is the vector clock of the corresponding release event. We refer to $(Acq(y), V)$ as a *lock history element* for a critical section represented by a matching acquire/release pair. Based on the information recorded in $H(y)$ we are able to efficiently apply the ROD rule as we will see shortly. The set $H(y)$ is initially empty. The initial definition of $Acq(y)$ can be left unspecified as by the time we access $Acq(y)$, $Acq(y)$ has been set.

For each shared variable x, the set $RW(x)$ maintains the current set of concurrent read/write events. Each event is represented the event's epoch. The set $RW(x)$ is initially empty.

The first-pass of PWR^{E+E} maintains three further sets that are important during the second pass. All sets are initially empty.

The set $edges(x)$ keeps track of the events from $RW(x)$ that will be replaced when processing reads/writes. If e replaces f this means that e happens-before f w.r.t. PWR. We record this information by adding the *edge constraint* $f \prec e$.

The set $conc(x)$ keeps track for each variable x of the set of potential race pairs where the events involved are concurrent to each other w.r.t. PWR. Such pairs represent potential write-write and read-write pairs. We *do not* enforce that their locksets must be disjoint because via a pair $(e, f) \in conc(x)$ where e, f share a common lock we may be able to reach a concurrent pair (g, f) where the locksets of g and f are disjoint. Recall the example from Section 4.1. For convenience, for all race pairs (e, f) collected by $conc(x)$ we maintain the property that $pos(e) < pos(f)$. For write-write pairs this property always holds. For read-write pairs the read is usually put first. Strictly following the trace position order makes the second pass easier to formalize as we will see shortly.

The set evt records for each read/write event its lockset and vector clock at the time of processing. We add the triple consisting of the event's epoch, lockset and vector clock to the set evt. The epoch serves as unique key for lookup. The information stored evt in will be used during the second pass. We traverse chains of edge constraints starting from candidates in $conc(x)$ to build new candidates. Each such found candidate must satisfy the Lockset-PWR check (see Definition 3.2). Based on the information stored in evt we can carry out this check easily.

Finally, we make use of $L_{W_t}(x)$ to record the thread id of the last write and $L_{W_L}(x)$ to record the last write's lockset. This information in combination with $L_W(x)$ is used to check for potential write-read race pairs.

In summary, the first pass of PWR^{E+E} maintains the following (global) variables:

- $LS_t(i)$, set of locks held by thread i.
- $Th(i)$, vector clock for thread i.
- $L_W(x)$, vector clock of last write on x.
- $L_{W_t}(x)$, thread id of last write on x.
- $L_{W_L}(x)$, lockset of last write on x.
- $RW(x)$, current set of concurrent reads/writes on x.
- $Acq(y)$, epoch of last acquire on y.
- $H(y)$, lock history for y.
- $L_W(x)$, last write access for x.
- $edges(x)$, set of edge constraints for x.
- $conc(x)$, accumulated set of concurrent reads/writes on x.
- evt, set of lockset and vector clock for each read/write.

We have now everything in place to consider the various cases covered by the first pass of PWR^{E+E}.

For each event we need to establish the PWR relation. In particular, we need to apply the ROD rule from Definition 3.1. Establishing the ROD rule is done via helper function w3.

Instead of some event $e \in CS(y)$ as formulated in the ROD rule, it suffices to consider the acquire event of $CS(y)$. In Appendix F we show that this is indeed sufficient.

The slightly revised ROD rule then reads as follows. If $CS(y)$ appears before $CS(y)'$ in the trace, $f \in CS(y)'$ and $acq(CS(y)) <^{PWR} f$, then $rel(CS(y)) <^{PWR} f$. Event f is represented by the two parameters V and LS_t. V is $f's$ vector clock and LS_t is the set of locks held when processing f. For each $y \in LS_t$ we check all prior critical sections on the same lock in the lock history $H(y)$. Each element is represented as a pair $(j\sharp k, V')$ where $j\sharp k$ is the epoch of the acquire and V' the vector clock of the matching release. The check $k < V[j]$ tests if the acquire happens-before f, i.e. $acq(CS(y)) <^{PWRA} f$. PWR then demands that $rel(CS(y)) <^{PWR} f$. This is guaranteed by $V = V \sqcup V'$.

In case of an acquire event in thread i on lock variable y, we first apply the ROD rule via helper function w3. Then, we extend the thread's lockset with y. In $Acq(y)$ we record the epoch of the acquire event. Finally, we increment the thread's time stamp to indicate that the event has been processed.

When processing the corresponding release event, we again apply first the ROD rule. Then, we remove y from the thread's lockset. We add the pair $(Acq(y), Th(i))$ to $H(y)$. $H(y)$ accumulates the *complete* lock history. There is no harm doing so but this can be of course inefficient. Optimizations to remove lock history elements are discussed later.

Next, we consider processing of write events. We apply first the ROD rule. Then we add the event's information to evt. We update $conc(x)$ by checking if the write is concurrent to any of the events in $RW(x)$. As discussed above, there is no need to compare vector clocks to check if two events are concurrent to each other. It suffices to compare epochs. Similarly, we update $RW(x)$ but only maintain the current set of concurrent reads/writes. Finally, we update the "last write" information and increment the thread's time stamp.

We consider processing of read events. We first check for a potential write-read race pair by checking if the read is concurrent to the last write and their locksets are disjoint. If the check is successful we immediately report the pair. Only after this check we impose the write-read dependency by synchronizing the last writes vector clock with the vector clock of the current thread. Then, we call w3 to apply the ROD rule. Updates for evt, $conc(x)$ and $RW(x)$ are the same as in case of write.

4.3 Second Pass

The first pass yields the set $edges(x)$ of edge constraints and the set $conc(x)$ of read/write pairs that are concurrent under PWR. In a second pass, we compute further concurrent pairs by systematically traversing $edges(x)$ starting with elements from $conc(x)$. The thus

obtained pairs are collected in some set $PC(x)$. Computation of $PC(x)$ is defined as follows.

Definition 4.4 (PWR^{E+E} Reporting Race Candidates). Let $conc(x)$ and $edges(x)$ be obtained by PWR^{E+E} for all shared variables x.

We define a total order among pairs in $conc(x)$ as follows. Let $(e, f) \in conc(x)$ and $(e', f') \in conc(x)$. Then, we define $(e, f) < (e', f')$ if $pos(e) < pos(e')$.

For each variable x, we compute the set $PC(x)$ by repeatedly performing the following steps. Initially, $PC(x) = \{\}$.

(1) If $conc(x) = \{\}$ stop.
(2) Otherwise, let (e, f) be the smallest element in $conc(x)$.
(3) Let $G = \{g_1, \ldots, g_n\}$ be maximal such that $g_1 \prec e, \ldots, g_n \prec e \in edges(x)$ and $pos(g_1) < \cdots < pos(g_n)$.
(4) $PC(x) := \{(e, f)\} \cup PC(x)$.
(5) $conc(x) := \{(g_1, f), \ldots, (g_n, f)\} \cup (conc(x) - \{(e, f)\})$.
(6) Repeat.

We can state that the set $PC(x)$ covers all concurrent reads/writes on x.

Proposition 4.5. *Let T be a trace of size n and x be some shared variable. Let $C^T(x) = \{(e, f) \mid e, f \in T^{rw}_x \wedge pos(e) < pos(f) \wedge e \not\prec^{PWR} f \wedge f \not\prec^{PWR} e\}$ Let x be a variable. Then, construction of $PC(x)$ takes time $O(n * n)$ and $C^T(x) \subseteq PC(x)$.*

We assume that the number of distinct (shared) variables x is a constant. Hence, construction of all sets $PC(x)$ takes time $O(n * n)$.

For each pair in $PC(x)$ we yet need to carry out the lockset check. We can retrieve the lockset for each event by consulting the set evt. The set evt records for each read/write event its lockset and vector clock at the time of processing. The vector clock is needed because the set $PC(x)$ overapproximates the set of concurrent reads/writes. Hence, we not only need to filter out pairs that share a common lock but also pairs that are not concurrent.

Example 4.6. Consider the following trace annotated with $RW(x)$, $edges(x)$ and $conc(x)$. We omit explicit vector clocks and epochs for brevity and write w_i (r_i) for a write (read) at trace position i.

	$1\sharp$	$2\sharp$	$3\sharp$	$RW(x)$	$edges(x)$	$conc(x)$
1.	$w(x)$			$\{w_1\}$		
2.	$w(y_1)$			$\{w_1\}$		
3.		$r(y_1)$		$\{w_1\}$		
4.		$w(y_2)$		$\{w_1\}$		
5.		$w(x)$		$\{w_5\}$	$w_1 \prec w_5$	
6.			$r(y_2)$	$\{w_5\}$		
7.			$w(x)$	$\{w_5, w_7\}$		(w_5, w_7)

Besides writes on x, we also find reads/writes on variables y_1 and y_2. We do not keep track of these events as their sole purpose is to enforce via some write-read dependencies that $w_1 <^{PWR} w_7$.

PWR^{E+E} yields $conc(x) = \{(w_5, w_7)\}$ and $edges(x) = \{w_1 \prec w_5\}$. The second pass then yields $PC(x) = \{(w_5, w_7), (w_1, w_7)\}$. However, $(w_1, w_7) \notin C^T(x)$ because $w_1 <^{PWR} w_7$.

The example shows that the set $PC(x)$ may contain some non-concurrent pairs. To filter out such pairs we apply the concurrency test specified in Proposition 4.3. We consult the vector clock of the event appearing later in the trace and check if the time stamp of the event appear first in the trace is greater. We also check that locksets are disjoint.

Lemma 4.7 (Lockset + PWR Filtering). *Let x be some variable. Let evt be obtained by PWR^{E+E} and $PC(x)$ via PWR^{E+E}'s second pass. Let $(i\sharp k, j\sharp l) \in PC(x)$ and $(j\sharp l, L_2, V_2) \in evt$. If $L_1 \cap L_2 = \emptyset$ and $k > V_2[j]$ then $(i\sharp k, j\sharp l)$ is either a write-write or read-write pair in $\mathcal{R}^T_{<^{PWR}}$ where we use the event's epoch as a unique identifier.*

We conclude that PWR^{E+E} (first pass) yields all write-read pairs in $\mathcal{R}^T_{<^{PWR}}$ and the second pass followed by filtering yields all write-write and read-write pairs in $\mathcal{R}^T_{<^{PWR}}$.

4.4 Time and Space Complexity

We consider the time and space complexity of PWR^{E+E} including first, second pass and filtering. Let n be the size of trace T, k be the number of threads and c be the number of critical sections. We consider the number of variables as a constant.

We first consider the time complexity of PWR^{E+E} (first pass). The size of the vector clocks and the set $RW(x)$ is bounded by $O(k)$. Each processing step of PWR^{E+E} requires adjustments of a constant number of vector clocks. This takes time $O(k)$. Adjustment of sets $conc(x)$, $edges(x)$ and $RW(x)$ requires to consider $O(k)$ epochs where each comparison among epochs is constant. Altogether, this requires time $O(k)$. We consider evt as a map where adding a new element takes constant time. The Lockset-PWR WRD race check takes constant time as we assume lookup of time stamp is constant and the size of each lockset is a constant. Each call to w3 takes time $O(c)$. Overall, PWR^{E+E} takes time $O(n * k + n * c)$ to process trace T.

The space required by PWR^{E+E} is as follows. Sets evt, $conc(x)$ and $edges(x)$ require $O(n * k)$ space. This applies to evt because for each event the size of the vector clock is $O(k)$. The size of the lockset is assumed to be a constant. Each element in $conc(x)$ and $edges(x)$ requires constant space. In each step, we may add $O(k)$ new elements because the size of $RW(x)$ is bounded by $O(k)$. Set $H(y)$ requires space $O(c * k)$. Overall, PWR^{E+E} requires $O(n * k + c * k)$ space.

The time for the second pass is $O(n * n)$. There are $O(n * n)$ pairs where each pair requires constant space. Hence, $O(n * n)$ space is required. Filtering for each candidate takes constant time. The size of the lockset is constant, time stamp comparison is a constant and lookup of locksets and vector clocks in evt is assumed to take constant time.

Overall, the run-time of PWR^{E+E}, first and second pass, including filtering is $O(n * k + n * c + n * n)$. The space requirement is $O(n * k + c * k + n * n)$. Parameters k and c are bounded by $O(n)$. Hence, the run-time of PWR^{E+E} is $O(n * n)$.

4.5 Optimizations

There are a number optimizations, e.g. aggressive filtering and removal of critical sections, that can be carried. Details are discussed in Appendix G. These optimizations will not change the theoretical time complexity but are essential in a practical implementation.

We can turn PWR^{E+E} into a single-pass, linear run-time algorithm if we impose a limit on the history of critical sections and a

limit on the number of edge constraints. Then, we can merge the second pass into the first pass. We refer to this variant as PWR_L^{E+E}.

Imposing a limit on the number of edge constraints means that the second pass (traversal of edges) and filtering takes place during the first pass as well. Whenever candidates are added to $conc(x)$ we immediately apply the steps described in Definition 4.4 (but the number of edge constraints to consider is limited) and carry out the filtering check.

By imposing a limit on the number of edge constraints in $edges(x)$, we might miss out on some potential data race pairs. For example, consider the case of 27 subsequent writes in one thread followed by a write in another thread. We assume that each write is connected to a distinct code location. In our implementation, we treat events connected to the same code location as the same event. Each of the 27 subsequent writes is in a race with the write in the other thread. There are 27 race pairs overall but a standard single-pass algorithm would only report the *last* race pair. The 27 subsequent writes give rise to 26 edge constraints. As we only maintain 25 edge constraints, we fail to report the *first* data race. In our experience, limiting the size of $edges(x)$ to 25 turns out to be a good compromise.

Consider the history of critical sections $H(y)$. Instead of a global history, our implementation maintains thread-local histories. The number of thread-local histories (after applying optimizations) is only bounded by the number of threads and the number of distinct variables. This can still be a fairly high number and requires extra management effort. In our implementation, we simply impose a fixed limit on the size of thread-local histories. If the limit is exceeded, the newly added element simply overwrites the oldest element. This might have the consequence that two events may become unordered w.r.t. the *limited* PWR relation (where they should be ordered without limit). Completeness is unaffected but our method may produce more false positives. In our experience, limiting the size of thread-local histories to five turns out to be a good compromise.

5 EXPERIMENTS

Test Candidates and Benchmarks. The test candidates are FastTrack(FT), SHB_L^{E+E}, WCP, ThreadSanitizer (TSan), PWR_L, PWR_L^{E+E} and PWR^{E+E}. SHB_L^{E+E} and PWR_L^{E+E} limit the size of edge constrains to 25. The limit for histories is five. PWR_L is a variant of PWR_L^{E+E} where the limit for edge constraints is zero. PWR^{E+E} does not impose any limits and therefore requires two passes whereas all the other candidates run in a single pass. We have implemented all of them in a common framework for better comparability.

We have not implemented SDP and WDP. As WCP, SDP and WDP rely on effectively the same method, their performance in terms of time and space is similar. See Table 8 in Genç et al. [2019] where running times and space usage of WCP, SDP and WDP are compared. Hence, only including WCP allows for a fair comparison.

In terms of precision, the WDP algorithm has more false positives and false negatives compared to PWR_L^{E+E} and PWR^{E+E}. [Genç et al. 2019] make use of an additional Vindication phase to check for a witness to confirm that a reported race is not a false positive. Vindication requires extra time and there is no guarantee to filter out all false positives as Vindication is incomplete (if no witness is found after one try Vindication gives up).

Table 1: Benchmark results. The time is given in minutes:seconds, maximum memory consumption in megabytes.

	FT	SHB_L^{E+E}	WCP	TSan	PWR_L^{E+E}	PWR_L	PWR^{E+E}
Avrora							
Races:	20	20(0)		30	20(0)	20	20(0)
Time:	0:14	0:19	>30	0:22	0:21	0:17	0:22
Mem:	2125	3965	6385	2934	3886	1999	4048
Batik							
Races:	12	4(0)	12	12	4(0)	4	4(0)
Time:	0:01	0:01	0:02	0:01	0:01	0:01	0:01
Mem:	29	35	84	33	68	32	80
H2							
Races:	125	248(0)		672	252(2)	252	
Time:	1:35	2:22	> 30	4:52	2:48	1:55	3:56
Mem:	2154	13431	6350	4998	16393	3465	> 32gb
Lusearch							
Races:	15	15(0)	15	19	15(0)	15	15(0)
Time:	0:01	0:02	0:19	0:01	0:04	0:04	0:04
Mem:	14	14	8685	11	1848	1852	2243
Tomcat							
Races:	636	681(194)		1984	823(219)	623	823(219)
Time:	0:33	0:49	>30	0:37	0:51	0:36	23:19
Mem:	12245	13617	13268	7523	19919	14861	28452
Xalan							
Races:	41	44(0)	142	244	394(223)	185	
Time:	1:19	2:04	7:11	1:33	2:30	1:30	1:51
Mem:	7282	9591	14882	5342	24980	7284	> 32gb
Moldyn							
Races:	33	24(8)	33	56	24(8)	18	24(8)
Time:	0:32	0:54	0:37	0:46	0:55	0:33	1:23
Mem:	99	487	108	91	515	71	19833

We have carried out experiments that involve two benchmark suites. The first benchmark suite consists of test of the Java Grande benchmark suite [Smith et al. 2001] and the DaCapo benchmark suite (version 9.12, [Blackburn et al. 2006]). This is a standard set of real-world tests to measure the performance in terms of execution time and memory consumption. The second benchmark suite consists of small, tricky examples found in earlier works Kini et al. [2017]; Mathur et al. [2018]; Pavlogiannis [2019]; Roemer et al. [2019, 2018] and our own examples that we found while working with different race prediction algorithms. For these examples we know the exact number of predictable races and therefore we can measure the precision (false positives, false negatives) of our test candidates. In terms of precision, PWR_L^{E+E} performs the best among all test candidates. The limits employed by PWR_L^{E+E} yields the same results as for PWR^{E+E}. We refer to Appendix H for details.

Performance. For benchmarking we use an AMD Ryzen 7 3700X and 32 gb of RAM with Ubuntu 18.04 as operating system. We have evaluated the performance of all benchmarks from the Java Grande and DaCapo benchmark suites. For space reasons, we only discuss the results for some benchmarks. Other benchmarks not discussed have similar characteristics compared to the ones show in Table 1. The time is given in minutes and seconds (mm:ss). The memory consumption is also measured for the complete program and not only for the single algorithms. In row *Mem* the memory consumption is given in megabytes. We use the standard 'time' program in Ubuntu to measure the time and memory consumption.

For TSan, PWR_L^{E+E}, PWR_L and SHB_L^{E+E}, entry row *#Races* shows the number of reported data race pairs. We filter pairs connected to the same code locations. For PWR_L^{E+E} and SHB_L^{E+E} we write 24(8) if

24 data race pairs were reported which includes 8 that were found using edge constraints. Because we only count pairs connected to unique code locations, it is possible that PWR_L reports a pair that will be reported via edge constraints in case of PWR_L^{E+E} and PWR^{E+E}. Appendix I explains this point in more detail. For FastTrack(FT) and WCP the number of races are the number of data race connected to distinct code locations.

In terms of number of races reported, PWR_L^{E+E} performs best followed by PWR_L and SHB_L^{E+E}. SHB_L^{E+E} only reports races from trace-specific schedules. PWR_L lacks edge constraints which leads to missed races as shown by the test cases Tomcat, Xalan and Moldyn. FastTrack, TSan and WCP report significantly fewer races.

FastTrack has the best performance in terms of run-time and memory consumption. TSan also shows good run-time performance with the exception of the H2 test case. The reason is due to our use of vector clocks in our TSan implementation.

WCP has performance problems with the Avrora, H2 and Tomcat test cases. For all three cases we aborted the experiment after 30 minutes. The reason are several thousands of critical sections that seem to be checked for each read/write inside a critical section. Like PWR_L^{E+E}, WCP maintains a history of critical sections but (a) needs to track more information (all read/write accesses within a critical section), and (b) can remove critical section not as aggressively as PWR_L^{E+E} because write-read dependencies are not strictly enforced. As argued above, similar observations should apply to SDP and WCP.

Table 8 in Genç et al. [2019] shows reasonable performance for WCP, SDP and WDP for Avrora, H2 and Tomcat. We are a bit surprised here but the base time (first column in Table 8) seems to indicate that in our measurements the programs were running for much longer and then some performance issues seem to arise. Our time measurements do not show the base time nor the time it took to generate the trace. We only show the timings to carry out the analysis.

PWR^{E+E} has no performance problems for test cases Avrora and Tomcat. This shows that our approach of dealing with the history of critical sections appears to be superior in comparison to WCP and WDP. For H2 and Xalan, PWR^{E+E} runs out of memory. The timings indicate the point in time when out of memory occurred. The H2 test case consists of 100 million events and the Xalan test case consists of 80 million events. This leads a huge number of edge constraints which then leads to out of memory. Hence, limiting the number of edge constraints is crucial for achieving an acceptable performance.

The memory consumption of PWR_L^{E+E} is still high for some test cases such as H2, Tomcat and Xalan. Overall, the run-times of PWR_L^{E+E} are competitive compared to the fastest candidate FastTrack. In summary, PWR_L^{E+E} strives for a balance between good performance and high precision.

6 RELATED WORK

We review further works in the area of dynamic data race prediction.

Efficient methods. We have already covered the efficient (linear-time) data race prediction methods that found use in FastTrack [Flanagan and Freund 2010], SHB [Mathur et al. 2018], WCP [Kini et al. 2017], SDP/WDP [Genç et al. 2019] and TSan [Serebryany and Iskhodzhanov 2009]. TSan is also sometimes referred to as ThreadSanitizer v1.

The newer TSan version, ThreadSanitizer v2 (TSanV2) [ThreadSanitizer 2020], is an optimized version of the FastTrack algorithm in terms of performance. TSanV2 only keeps a limited history of write/read events. This improves the performance but results in a higher number of false negatives.

Acculock [Xie et al. 2013] optimizes the original TSan algorithm by employing a single lockset per variable. Acculock can be faster, but is less precise compared to TSan if a thread uses multiple locks at once.

SimpleLock [Yu and Bae 2016] uses a simplified lockset algorithm. A data race is only reported if at least one of the accesses is not protected by any lock. They show that they are faster compared to Acculock but miss more data races since they do not predict data races for events with different locks.

Semi-efficient methods. We consider semi-efficient methods that require polynomial run-time.

The SHB^{E+E} algorithm [Sulzmann and Stadtmüller 2019] requires quadratic run-time to compute all trace-specific data race pairs. Our PWR^{E+E} algorithm adopts ideas from SHB^{E+E} and achieves completeness while retaining a quadratic run-time. By limiting the history of edge constraints, the variant PWR_L^{E+E} runs in linear time. Due to this optimization we are only near complete. In practice, the performance gain outweighs the benefit of a higher precision.

The Vindicator algorithm [Roemer et al. 2018] improves the WCP algorithm and is sound for all reported data races. It can predict more data races compared to WCP, but requires three phases to do so. The first phase of Vindicator is a weakened WCP relation that removes the happens-before closure. For the second phase, it constructs a graph that contains all events from the processed trace. This phase is unsound and incomplete which is why a third phase is required. The third phase makes a single attempt to reconstruct a witness trace for the potential data race and reports a data race if successful. Vindicator has a much higher run-time compared to the "PWR" family of algorithms. We did not include Vindicator in our measurements as we experienced performance issues for a number of real world benchmarks (e.g. timeout due to lack of memory etc).

The M2 algorithm [Pavlogiannis 2019] can be seen as a further improvement of the Vindicator idea. Like Vindicator, multiple phases are required. M2 requires two phases. M2 has $O(n^4)$ run-time (where n is the size of the trace). M2 is sound and like PWR^{E+E} complete for two threads. The measurements by Pavlogiannis [2019] show that in terms of precision M2 improves over FastTrack, SHB, WCP and Vindicator for a subset of the real-world benchmarks that we also considered. We did not include M2 in our measurements as we are not aware of any publicly available implementation.

Exhaustive methods. We consider methods that are sound *and* complete to which we refer as exhaustive methods. Exhaustive methods come with a high degree of precision but generally are no longer efficient.

The works by Huang et al. [2014]; Luo et al. [2015]; Serbanuta et al. [2012] use SAT/SMT-solvers to derive alternative feasible traces from a recorded trace. These traces can be checked with an arbitrary race prediction algorithm for data races. This requires

multiple phases and is rather complimentary to the algorithms that we compare in this work as any of them could be used to check the derived traces for data races.

Kalhauge and Palsberg [Kalhauge and Palsberg 2018] present a data race prediction algorithm that is sound and complete. They also use an SMT-solver to derive alternative feasible traces. The algorithm inspects write-read dependencies in more detail, to determine at which point the control flow might be influenced by the observed write-read dependency. Deriving multiple traces and analyzing their write-read dependencies for their influence on the control flow is a very slow process that can take several hours according to their benchmarks.

Comparative studies. Previous works that compare multiple data race prediction algorithms use the Java Grande [Smith et al. 2001], Da Capo [Blackburn et al. 2006] and IBM Contest [Farchi et al. 2003] benchmark suits to do so. The DaCapo and Java Grande benchmark suite contain real world programs with an unknown amount of data races and other errors. The IBM Contest benchmark is a set of very small programs with known concurrency bugs like data races.

Yu, Park, Chun and Bae [Yu et al. 2017a] compare the performance of FastTrack [Flanagan and Freund 2010], SimpleLock+ [Yu and Bae 2016], Multilock-HB, Acculock [Xie et al. 2013] and Casually-Precedes (CP) [Smaragdakis et al. 2012] with a subset of the benchmarks found in the DaCapo, JavaGrande and IBM Contest suits. They reimplemented CP to use a sliding window of only 1000 shared memory events which does not affect the soundness but the amount of predicted data races. In our work we compare newer algorithms including Weak-Casually-Precedes which is the successor of CP.

The work by Liao et al. [2017] compares Helgrind, ThreadSanitizer Version 2, Archer and the Intel Inspector. They focus on programs that make use of OpenMP for parallelization. OpenMP uses synchronization primitives that are unknown to Helgrind, ThreadSanitizer v2 and the Intel Inspector. Only the Archer race predictors is optimized for OpenMP. For their comparison they use the Linpack and SPECOMP benchmark suits for which the number of concurrency errors is unknown. Most of their races are enforced by including OpenMP primitives to parallelize the code which are not part of the original implementation. Thus, they lack complex concurrency patterns. In some related work [Lin et al. 2018], the same authors test the four data race predictors from their previous work again with programs that make use of OpenMP and SIMD parallelism. Since SIMD is unsupported by all tested data race predictors, they encounter a high number of false positives. The data race predictors we tested would report many false positives for the same reasons.

The work by Alowibdi and Stenneth [2013] evaluates the static data race predictors RaceFuzzer, RacerAJ, Jchord, RCC and JavaRaceFinder. They only evaluate the performance and the number of data races that each algorithms predicts. Static data race prediction is known to report too many false positives since they need to over-approximate the program behavior. We only tested dynamic data race predictors that make use of a recorded trace to predict data races. In terms of accuracy we expect that our test candidates perform better compared to the static data race predictors.

Yu, Yang, Su and Ma [Yu et al. 2017b] test Eraser, Djit+, Helgrind+, ThreadSanitizer v1, FastTrack, Loft, Acculock, Multilock-HB, Simplelock and Simplelock+. It is the to the best of our knowledge the only previous work that includes ThreadSanitizer v1. In their work, they use the original implementations for testing. They test the performance and accuracy with the unit tests of ThreadSanitizer. The tested data race predictors ignore write-read dependency and are therefore only sound for the first predicted data race. We test current solutions that mostly include write-read dependencies. For accuracy testing we included a set of handwritten test cases to ensure that every algorithm sees the same order of events. All algorithms, except Vindicator, are reimplemented in a common framework to ensure that all algorithms use the same utilities and have the same parsing overhead.

7 CONCLUSION

We have introduced PWR^{E+E} and the practically inspired variant PWR_L^{E+E}. PWR_L^{E+E} is an efficient, near complete and often sound dynamic data race prediction algorithm that combines the lockset method with recent improvements made in the area of happens-before based methods. PWR^{E+E} is complete in theory. For the case of two threads we can show that PWR^{E+E} is also sound. Our experimental results show that PWR_L^{E+E} performs well compared to the state-of-the art efficient data race prediction algorithms. The implementation of PWR_L^{E+E} including all contenders as well as benchmarks can be found at

https://github.com/KaiSta/SpeedyGo. [2]

[2]The "PWR" algorithm is referred to as "W3PO" in the SpeedyGo framework.

REFERENCES

Sarita V. Adve and Kourosh Gharachorloo. 1996. Shared Memory Consistency Models: A Tutorial. Computer 29, 12 (Dec. 1996), 66–76. https://doi.org/10.1109/2.546611

Jalal S Alowibdi and Leon Stenneth. 2013. An empirical study of data race detector tools. In 2013 25th Chinese Control and Decision Conference (CCDC). IEEE, 3951–3955. https://doi.org/10.1109/CCDC.2013.6561640

Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In Proc. of OOPSLA '06. ACM, 169–190. https://doi.org/10.1145/1167515.1167488

Anne Dinning and Edith Schonberg. 1991. Detecting Access Anomalies in Programs with Critical Sections. SIGPLAN Not. 26, 12 (Dec. 1991), 85–96. https://doi.org/10.1145/127695.122767

Eitan Farchi, Yarden Nir, and Shmuel Ur. 2003. Concurrent bug patterns and how to test them. In Proceedings International Parallel and Distributed Processing Symposium. IEEE, 7–pp. https://doi.org/10.1109/IPDPS.2003.1213511

Colin J. Fidge. 1992. Process Algebra Traces Augmented with Causal Relationships. In Proc. of FORTE '91. North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands, 527–541.

Cormac Flanagan and Stephen N Freund. 2010. FastTrack: efficient and precise dynamic race detection. *Commun. ACM* 53, 11 (2010), 93–101. https://doi.org/10.1145/1543135.1542490

Kaan Genç, Jake Roemer, Yufan Xu, and Michael D. Bond. 2019. Dependence-Aware, Unbounded Sound Predictive Race Detection. Proc. ACM Program. Lang. 3, OOPSLA, Article 179 (Oct. 2019), 30 pages. https://doi.org/10.1145/3360605

Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. 2014. Maximal Sound Predictive Race Detection with Control Flow Abstraction. SIGPLAN Not. 49, 6 (June 2014), 337–348. https://doi.org/10.1145/2666356.2594315

Christian Gram Kalhauge and Jens Palsberg. 2018. Sound Deadlock Prediction. Proc. ACM Program. Lang. 2, OOPSLA, Article 146 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276516

Dileep Kini, Umang Mathur, and Mahesh Viswanathan. 2017. Dynamic Race Prediction in Linear Time. SIGPLAN Not. 52, 6 (June 2017), 157–170. https://doi.org/10.1145/3062341.3062374

Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (1978), 558–565. https://doi.org/10.1145/359545.359563

Chunhua Liao, Pei-Hung Lin, Joshua Asplund, Markus Schordan, and Ian Karlin. 2017. DataRaceBench: a benchmark suite for systematic evaluation of data race detection tools. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. ACM, 11. https://doi.org/10.1145/3126908.3126958

Pei-Hung Lin, Chunhua Liao, Markus Schordan, and Ian Karlin. 2018. Runtime and memory evaluation of data race detection tools. In International Symposium on Leveraging Applications of Formal Methods. Springer, 179–196. https://doi.org/10.1007/978-3-030-03421-4_13

Qingzhou Luo, Jeff Huang, and Grigore Rosu. 2015. Systematic Concurrency Testing with Maximal Causality. Technical Report.

Umang Mathur, Dileep Kini, and Mahesh Viswanathan. 2018. What Happens-after the First Race? Enhancing the Predictive Power of Happens-before Based Dynamic Race Detection. Proc. ACM Program. Lang. 2, OOPSLA, Article 145 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276515

Friedemann Mattern. 1989. Virtual Time and Global States of Distributed Systems. In Parallel and Distributed Algorithms. North-Holland, 215–226.

Andreas Pavlogiannis. 2019. Fast, Sound, and Effectively Complete Dynamic Race Prediction. Proc. ACM Program. Lang. 4, POPL, Article Article 17 (Dec. 2019), 29 pages. https://doi.org/10.1145/3371085

Jake Roemer, Kaan Genç, and Michael D Bond. 2019. Practical Predictive Race Detection. arXiv preprint arXiv:1905.00494 (2019).

Jake Roemer, Kaan Genç, and Michael D. Bond. 2018. High-coverage, Unbounded Sound Predictive Race Detection. SIGPLAN Not. 53, 4 (June 2018), 374–389. https://doi.org/10.1145/3192366.3192385

Traian-Florin Serbanuta, Feng Chen, and Grigore Rosu. 2012. Maximal Causal Models for Sequentially Consistent Systems. In Poc. of RV'12 (LNCS), Vol. 7687. Springer, 136–150. https://doi.org/10.1007/978-3-642-35632-2_16

Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: data race detection in practice. In Proc. of WBIA '09. ACM, New York, NY, USA, 62–71. https://doi.org/10.1145/1791194.1791203

Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. 2012. Sound Predictive Race Detection in Polynomial Time. SIGPLAN Not. 47, 1 (Jan. 2012), 387–400. https://doi.org/10.1145/2103656.2103702

Lorna A Smith, J Mark Bull, and J Obdrizalek. 2001. A Parallel Java Grande Benchmark Suite. In Proc. of SC'01. IEEE, 8–8. https://doi.org/10.1145/582034.582042

Martin Sulzmann and Kai Stadtmüller. 2019. Predicting all data race pairs for a specific schedule. In Proc. of MPLR'19. ACM, New York, NY, USA, 72–84. https://doi.org/10.1145/3357390.3361022

ThreadSanitizer 2020. ThreadSanitizer. https://github.com/google/sanitizers. (2020).

Xinwei Xie, Jingling Xue, and Jie Zhang. 2013. Acculock: Accurate and efficient detection of data races. Software: Practice and Experience 43, 5 (2013), 543–576. https://doi.org/10.1109/CGO.2011.5764688

Misun Yu and Doo-Hwan Bae. 2016. SimpleLock+: fast and accurate hybrid data race detection. *Comput. J.* 59, 6 (2016), 793–809. https://doi.org/10.1109/PDCAT.2013.15

Misun Yu, Seung-Min Park, Ingeol Chun, and Doo-Hwan Bae. 2017a. Experimental performance comparison of dynamic data race detection techniques. ETRI Journal 39, 1 (2017), 124–134. https://doi.org/10.4218/etrij.17.0115.1027

Zhen Yu, Zhen Yang, Xiaohong Su, and Peijun Ma. 2017b. Evaluation and comparison of ten data race detection techniques. International Journal of High Performance Computing and Networking 10, 4-5 (2017), 279–288. https://doi.org/10.1504/IJHPCN.2017.086532

A PREDICTABLE DATA RACES

We formalize our notion of predictable data races.

A.1 Run-Time Events and Traces

We assume concurrent programs making use of shared variables and acquire/release (a.k.a. lock/unlock) primitives. Further constructs such as fork and join are omitted for brevity. We assume that programs are executed under the sequential consistency memory model [Adve and Gharachorloo 1996]. This is a standard assumption made by most data race prediction algorithms. The upcoming program order condition (see Definition A.3) reflects this assumption.

Programs are instrumented to derive a trace of events when running the program. A trace is of the following form.

Definition A.1 (Run-Time Traces and Events).

$$\begin{array}{lcll} T & ::= & [] \mid i\sharp e : T & \text{Trace} \\ e & ::= & r(x)_j \mid w(x)_j \mid acq(y)_j \mid rel(y)_j & \text{Events} \end{array}$$

Besides e, we sometimes use symbols f and g to refer to events.

A trace T is a list of events. We use the notation a list of objects $[o_1, \ldots, o_n]$ is a shorthand for $o_1 : \cdots : o_n : []$. We write ++ to denote the concatenation operator among lists. For each event e, we record the thread id number i in which the event took place, written $i\sharp e$. We write $r(x)_j$ and $w(x)_j$ to denote a read and write event on shared variable x. We write $acq(y)_j$ and $rel(y)_j$ to denote a lock and unlock event on mutex y. The number j is distinct for each event and allows us to uniquely identify each event. For brevity, we sometimes omit the thread id i and the number j.

Example A.2. We often use a tabular notation for traces where we introduce for each thread a separate column and the trace position can be identified via the row number. Below, we find a trace specified as list of events and its corresponding tabular notation.

$$\begin{array}{rcl} T & = & [1\sharp w(x)_1, 1\sharp acq(y)_2, 1\sharp rel(y)_3, \\ & & 2\sharp acq(y)_4, 2\sharp w(x)_5, 2\sharp rel(y)_6] \end{array}$$

	$1\sharp$	$2\sharp$
1.	$w(x)$	
2.	$acq(y)$	
3.	$rel(y)$	
4.		$acq(y)$
5.		$w(x)$
6.		$rel(y)$

We define $thread_T(e) = j$ if $T = T_1$ ++ $[j\sharp e]$ ++ T_2 for some traces T_1, T_2. We define $pos_T(i\sharp e) = k$ if $i\sharp e$ is the k-th event in T.

We often drop the component T and write $thread(e)$ and $pos(e)$ for short.

We define $events(T) = \{e \mid \exists T_1, T_2, j. T = T_1 \mathbin{++} [j\sharp e] \mathbin{++} T_2\}$ to be the set of events in T. We write $e \in T$ if $e \in events(T)$.

We define $proj_{\sharp i}(T) = T'$ the projection of T onto thread i where (1) for each $e \in T$ where $thread_T(e) = i$ we have that $e \in T'$, and (2) for each $e, f \in T'$ where $pos_{T'}(e) < pos_{T'}(f)$ we have that $pos_T(e) < pos_T(f)$. That is, the projection onto a thread comprised of all events in that thread and the program order remains the same.

Traces must be well-formed: a thread may only acquire an unheld lock and may only release a lock it has acquired. Hence, for each release event $i\sharp rel(y)_l$ there exists an acquire event $i\sharp acq(y)_k$ where $k < l$ and there is no other acquire on y in between. We refer to $i\sharp acq(y)_k$ and $i\sharp rel(y)_l$ as a pair of *matching acquire-release* events. All events $e_1, ..., e_n$ in between trace positions k and l must be part of the thread i.

In such a situation, we write $i\sharp\langle acq(y)_k, e_1, \ldots, e_n, rel(y)_l\rangle$ to denote the events in the *critical section* represented by the pair $i\sharp acq(y)_k$ and $i\sharp rel(y)_l$ of matching acquire-release events.

We write $f \in i\sharp\langle acq(y)_k, e_1, \ldots, e_n, rel(y)_l\rangle$ if f is one of the events in the critical section. We often write $i\sharp CS(y)$ as a short-form for a critical section $i\sharp\langle acq(y)_k, e_1, \ldots, e_n, rel(y)_l\rangle$. We write $i\sharp CS(y) \in T$ to denote that the critical section is part of the trace T. We write $i\sharp acq(CS(y))$ to refer to $acq(y)_k$ and $i\sharp rel(CS(y))$ to refer to $rel(y)_l$. If the thread id does not matter, we write $CS(y)$ for short and so on. If the lock variable does not matter, we write CS for short and so on.

We define T^{rw}_x as the set of all read/write events in T on some variable x. We define T^{rw} as the union of T^{rw}_x for all variables x.

Let $e, f \in T^{rw}_x$ where e is a read event and f is a write event. We say that f is the *last write* for e w.r.t. T if (1) f appears before e in the trace, and (2) there is no other write event on x in between f and e in the trace.

A.2 Trace Reordering

A trace represents one possible interleaving of concurrent events. Based on this trace, we wish to explore alternative interleavings. In theory, there can be as many interleavings as there are permutations of the original trace. However, not all permutations are feasible in the sense that they could be reproduced by executing the program again.

We wish to characterize feasible alternative interleavings without having to take into account the program. For this purpose, we assume some idealistic execution scheme: (1) The program order as found in each thread is respected. (2) Every read sees the same (last) write. (3) The lock semantics is respected so that execution will not get stuck.

Definition A.3 (Correct Reordering). Let T be a well-formed trace. Then, trace T' is a *correctly reordered prefix* of T iff the following conditions hold:

- Program order: For each thread id i we have that $proj_{\sharp i}(T')$ is a subtrace of $proj_{\sharp i}(T)$.
- Last writer: For each read event e in T' where f is the last write for e w.r.t. T, we have that f is in T' and f is also the last write for e w.r.t. T'
- Lock semantics: For e_1, e_2 be two acquire events on the same lock where $pos_{T'}(e_1) < pos_{T'}(e_2)$ we have that $pos_{T'}(e_1) < pos_{T'}(f_1) < pos_{T'}(e_2)$ where f_1 is e_1's matching release event.

A correctly reordered trace is a permutation of the original trace that respects the idealistic execution scheme. As we will see, a data race may only reveal itself for some prefix.

Critical sections represent atomic units and the events within cannot be reordered. However, critical sections themselves may be reordered. We distinguish between schedules that leave the order of critical sections unchanged (trace-specific schedule), and schedules that reorder critical sections (alternative schedule).

Definition A.4 (Schedule). Let T be a well-formed trace and T' some correctly reordered prefix of T.

We say T' represents the *trace-specific schedule* in T if the relative position of (common) critical sections (for the same lock variable) in T' and T is the same. For lock variable y and critical sections $CS(y)_1, CS(y)_2 \in T$ where $CS(y)_1$ appears before $CS(y)_2$ in T we have that $CS(y)_1, CS(y)_2 \in T'$ and $CS(y)_1$ appears before $CS(y)_2$ in T'. Otherwise, we say T' that represents some *alternative schedule*.

Example A.5. Consider the well-formed trace

$$\begin{aligned} T \quad = \quad & [1\sharp w(x)_1, 1\sharp acq(y)_2, 1\sharp rel(y)_3, \\ & 2\sharp acq(y)_4, 2\sharp w(x)_5, 2\sharp rel(y)_6]. \end{aligned}$$

Then,

$$\begin{aligned} T' \quad = \quad & [2\sharp acq(y)_4, 2\sharp w(x)_5, 1\sharp w(x)_1, \\ & 2\sharp rel(y)_6, 1\sharp acq(y)_2, 1\sharp rel(y)_3] \end{aligned}$$

is a correctly reordered prefix of T where T' represents an alternative schedule.

A.3 Data Race

A data race is represented as a pair (e, f) of events where e and f are in conflict and we find a correctly reordered prefix (schedule) where e appears right before f in the trace.

The condition that e appears right before f is useful to clearly distinguish between write-read and read-write races. We generally assume that for each read there is an initial write. Write-read race pairs are linked to write-read dependencies where a write immediately precedes a read. Read-write race pairs indicate situations where a read might interfere with some other write, not the read's last write. For write-write race pairs (e, f) it turns out that if e appears right before f for some reordered trace then f can also appear right before e by using a slightly different reordering. Hence, write-write pairs (e, f) and (f, e) are equivalent and we only report the representative (e, f) where e appears before f in the original trace.

Definition A.6 (Initial Writes). We say a trace T satisfies the *initial write* property if for each read event e on variable x in T there exists a write event f on variable x in T where $pos_T(f) < pos_T(e)$.

The initial write of a read does not necessarily need to occur within the same thread. It is sufficient that the write occurs before the read in the trace. From now on we assume that all traces satisfy the initial write assumption, as well as the well-formed property.

Definition A.7 (Predictable Data Race Pairs). Let T be a trace. Let T' be a correctly reordered prefix of T'. Let $e, f \in T$. We refer to (e, f) as a *predictable data race pair* if (a) e, f are two conflicting events in T, and (b) e appears right before f in the trace T'. We refer to T' as *witness*.

We say (e, f) is a *write-read* race pair if e is a write and f is a read. We say (e, f) is a *read-write* race pair if e is a read and f is a write. We say (e, f) is a *write-write* race pair if both events are writes.

We write $e \overset{T \rhd T'}{\asymp} f$ for predictable write-read, read-write and write-write race pairs and traces T and T' as specified above. For write-write pairs (e, f) we demand that $pos_T(e) < pos_T(f)$.

We define $\mathcal{P}^T = \{(e, f) \mid e, f \in T \wedge \exists T'.T \rhd T' \wedge e \overset{T \rhd T'}{\asymp} f\}$. We refer to $\mathcal{P}^T$ as the set of *all predictable data pairs* derivable from T.

We define $\mathcal{S}^T = \{(e, f) \mid e, f \in T \wedge \exists T'.T \rhd T' \wedge e \overset{T \rhd T'}{\asymp} f \wedge T'$ trace-specific schedule$\}$. We refer to $\mathcal{S}^T$ as the set of *all trace-specific predictable data race pairs* derivable from T.

Our definition of predictable races follows [Genç et al. 2019; Mathur et al. 2018]. and is more general compared to earlier definitions as found in [Kini et al. 2017; Smaragdakis et al. 2012]. The difference is that [Kini et al. 2017; Smaragdakis et al. 2012] only consider the 'first' race as a predictable race whereas [Genç et al. 2019; Mathur et al. 2018] also consider 'subsequent' races as predictable races. Identifying races beyond the first race is useful as we explain via the following example.

Example A.8. Consider the following trace T where we use the tabular notation.

	1♯	2♯	3♯
1.	$w(x)$		
2.		$w(x)$	
3.		$r(x)$	
4.			$r(x)$
5.			$w(x)$

For each event e we consider the possible candidates f for which (e, f) forms a predictable race pair. We start with event $w(x)_1$.

For $w(x)_1$ we immediately find (1) $(w(x)_1, w(x)_2)$. We also find (2) $(w(x)_1, w(x)_5)$ by putting $w(x)_1$ in between $r(x)_4$ and $w(x)_5$. There are no further combinations $(w(x)_1, f)$ where $w(x)_1$ can appear right before some f. For instance, $(w(x)_1, r(x)_3)$ is not valid because otherwise the last writer condition in Definition A.3 is violated.

Consider $w(x)_2$. We find (3) $(w(x)_2, w(x)_1)$ because

$$T' = [w(x)_2, w(x)_1]$$

is a correctly reordered prefix of T. It is crucial that we only consider prefixes. Any extension of T' that involves $r(x)_3$ would violate the last writer condition in Definition A.3. For $w(x)_2$ there is another pair (4) $(w(x)_2, r(x)_4)$. The pair $(w(x)_2, r(x)_3)$ is not a valid write-read race pair because $w(x)_2$ and $r(x)_3$ result from the same thread and therefore are not in conflict.

Consider $r(x)_3$. We find pairs (5) $(r(x)_3, w(x)_1)$ and (6) $(r(x)_3, w(x)_5)$. For instance (5) is due to the prefix

$$[w(x)_2, r(x)_3, w(x)_1].$$

The remaining race pairs are (7) $(r(x)_4, w(x)_1)$ and (8) $(w(x)_5, w(x)_1)$.

Pairs (1) and (3) as well as pairs (2) and (8) are equivalent write-write race pairs. When collecting all predictable race pairs we only keep the representatives (1) and (2). Hence, we find $\mathcal{P}^T = \{(1), (2), (4), (5), (6), (7)\}$ where each race pair is represented by the numbering schemed introduced above. There are no critical sections and therefore no alternative schedules. Hence, $\mathcal{P}^T = \mathcal{S}^T$.

Kini et al. [2017]; Smaragdakis et al. [2012] identify race pair (1) as the first race pair. All race pairs (1-7) are schedulable races according to Mathur et al. [2018]. For example, consider (4) $(w(x)_2, r(x)_4)$. and (6) $(r(x)_3, w(x)_5)$. A witness for (6) is $T' = [w(x)_2, r(x)_4, r(x)_3, w(x)_5]$. In T' there is the 'earlier' race (4) and there is no other witness for (6) that does not contain (4). So it seems that (6) is not a 'real' race because after (4) the program's behavior may become undefined.

However, it is easy to fix earlier races. We make the conflicting events mutually exclusive by introducing a fresh lock variable. In terms of the original trace, we replace subtrace $[2\sharp w(x)_2]$ by

$$[1\sharp acq(y), 2\sharp w(x)_2, 1\sharp rel(y)]$$

and subtrace $[3\sharp r(x)_4]$ by

$$[2\sharp acq(y), 3\sharp w(x)_4, 2\sharp rel(y)]$$

where y is a fresh lock variable. Race (6) becomes then a real race. Hence, the motivation to consider all races as we otherwise require multiple execute-report-fix cycles.

The next example highlights the fact that a race may only reveal itself for some prefix.

Example A.9. Consider

	1♯	2♯
1.	$w(y)$	
2.	$acq(z)$	
3.	$w(x)$	
4.	$rel(z)$	
5.		$acq(z)$
6.		$w(y)$
7.		$r(x)$
8.		$rel(z)$

There is one predictable race $(w(y)_1, w(y)_6)$ that results from some alternative schedule. Consider

$$T' = [2\sharp acq(z)_5, 1\sharp w(y)_1, 2\sharp w(y)_6].$$

There is no extension of T' that covers all events in T as otherwise we would violate the last writer condition.

We summarize. For each race pair (e, f) there is a reordering where e appears right before f in the reordered trace. Each write-write race pair (e, f) is also a write-write race pair (f, e). We choose the representative (e, f) where e appears before f in the original trace. For each write-read race pair (e, f) we have that e is f's last write. Each read-write race pair (e, f) represents a situation where the read e can interfere with some other write f. Formal statements see below.

Lemma A.10. *Let T be some trace and (e, f) be some write-write race pair for T. Then, we have that (f, e) is also a write-write race pair for T.*

Proof. By assumption T' is some correctly reordered prefix where $T' = [\ldots, e, f]$. We can reorder e and f in T' while maintaining the conditions in Definition A.3. Thus, we are done. □

Lemma A.11. *Let T be some trace and (e, f) be some write-read race pair for T. Then, (f, e) cannot be a read-write race pair for T.*

Proof. By construction e must be f's 'last write'. Hence, (f, e) is not valid as otherwise the 'last write' property is violated. □

Lemma A.12. *Let T be some trace and (e, f) be some read-write race pair for T. Then, (f, e) cannot be a write-read race pair for T.*

Proof. For this result we rely on the initial writes assumption. For the read-write race pair (e, f) we know that f is *not e's* 'last write'. Then, (f, e) is not valid. If it would then f is *e's* 'last write'. Contradiction. □

From above we conclude that for each write-read race pair (e, f) we have that e appears before f in the original trace T. For read-write race pairs (e, f), e can appear before or after f in the original trace. See cases (5) and (6) in Example A.8.

B FORK AND JOIN

Algorithm 2 extends Algorithm 1 with fork and join.

C ADDITIONAL EXAMPLES

Example C.1. We consider a run of the first pass of PWR^{E+E} for the following trace. Instead of epochs, we write w_i for a write at trace position i. A similar notation is used for reads. We annotate the trace with $RW(x)$, $edges(x)$ and $conc(x)$. For $edges(x)$ and $conc(x)$ we only show incremental updates. For brevity, we omit the set evt because locksets and vector clocks of events do not matter here.

	$1\sharp$	$2\sharp$	$RW(x)$	$edges(x)$	$conc(x)$
1.	$w(x)$		$\{w_1\}$		
2.	$w(x)$		$\{w_2\}$	$w_1 \prec w_2$	
3.		$w(x)$	$\{w_2, w_3\}$		(w_2, w_3)
4.		$r(x)$	$\{w_2, r_4\}$	$w_3 \prec r_4$	(w_2, r_4)

The potential races covered by $conc(x)$ are (w_2, w_3) and (r_4, w_2). These are also predictable races. As said, the set $conc(x)$ follows the trace position order. Hence, we find $(w_2, r_4) \in conc(x)$. Overall, there are four predictable races. The first pass of PWR^{E+E}, i.e. the set $conc(x)$, fails to capture the predictable races (w_1, w_3) and (r_4, w_1).

The missing pairs can be obtained via the second pass as follows. Starting from $(w_2, w_3) \in conc(x)$ via $w_1 \prec w_2 \in edges(x)$ we can reach (w_1, w_3). From $(w_2, r_4) \in conc(x)$ via $w_1 \prec w_2 \in edges(x)$ we reach (w_1, r_4). The pair (w_1, r_4) represents a read-write pair. When reporting this pair we simply switch the order of events.

D PROOFS OF RESULTS IN MAIN TEXT

D.1 Auxiliary Results

Lemma D.1. $<^{SHB} \not\subseteq <^{WCP}$.

Proof. Consider Example 2.1. □

Algorithm 2 PWR^{E+E} algorithm (first pass) with fork and join

```
1:  function w3(V, LS_t)
2:      for y ∈ LS_t do
3:          for (j♯k, V') ∈ H(y) do
4:              if k < V[j] then
5:                  V = V ⊔ V'
6:              end if
7:          end for
8:      end for
        return V
9:  end function
1:  procedure ACQUIRE(i, y)
2:      Th(i) = w3(Th(i), LS_t(i))
3:      LS_t(i) = LS_t(i) ∪ {y}
4:      Acq(y) = i♯Th(i)[i]
5:      inc(Th(i), i)
6:  end procedure
1:  procedure RELEASE(i, y)
2:      Th(i) = w3(Th(i), LS_t(i))
3:      LS_t(i) = LS_t(i) − {x}
4:      H(y) = H(y) ∪ {(Acq(y), Th(i))}
5:      inc(Th(i), i)
6:  end procedure
1:  procedure WRITE(i, x)
2:      Th(i) = w3(Th(i), LS_t(i))
3:      evt = {(i♯Th(i)[i], Th(i), LS_t(i))} ∪ evt
4:      edges(x) = {j♯k ≺ i♯Th(i)[i] | j♯k ∈ RW(x) ∧ k < Th(i)[j]} ∪ edges(x)
5:      conc(x) = {(j♯k, i♯Th(i)[i]) | j♯k ∈ RW(x) ∧ k > Th(i)[j]} ∪ conc(x)
6:      RW(x) = {i♯Th(i)[i]} ∪ {j♯k | j♯k ∈ RW(x) ∧ k > Th(i)[j]}
7:      L_W(x) = Th(i)
8:      L_{W_t}(x) = i
9:      L_{W_L}(x) = LS_t(i)
10:     inc(Th(i), i)
11: end procedure
1:  procedure FORK(i, j)
2:      Th(j) = Th(i)
3:      inc(Th(i), i)
4:  end procedure
1:  procedure JOIN(i, j)
2:      Th(i) = Th(j) ⊔ Th(i)
3:      inc(Th(i), i)
4:  end procedure
```

Lemma D.2. $<^{SHB} \subseteq <^{WCP}$.

Proof. Both relations apply the PO condition.

Consider the 'extra' WCP conditions. These conditions relax the RAD condition. Hence, if any of these WCP conditions apply, the RAD condition applies as well. □

Lemma D.3. *Let T be a trace. Let $<$ denote some strict partial order among elements in T. Let $e, f \in T$, $CS(y)_1$ and $CS(y)_2$ be two critical*

sections for the same lock variable y such that (1) $acq(CS(y)_1) < e < rel(CS(y)_1)$, (2) $acq(CS(y)_2) < f < rel(CS(y)_2)$, and (3) $e < f$. Then, we have that $\neg(rel(CS(y)_2) < acq(CS(y)_1))$.

Proof. Suppose, $rel(CS(y)_2) < acq(CS(y)_1)$. Then, we find that $acq(CS(y)_1) < e < f < rel(CS(y)_2) < acq(CS(y)_1)$. This is a contradiction and we are done. □

D.2 Proof of Proposition 3.7

Proof. We first show that $<^{WDP} \subseteq <^{PWR}$. PWR applies PO like WDP. The WDP rule RCD is an instance of the PWR rule ROD in combination with the WRD rule. Similarly, the WDP rule RRD is an instance of the PWR rule ROD in combination with the PO rule.

Example 2.6 shows that the reverse direction $<^{PWR} \subseteq <^{WDP}$ does not hold. □

D.3 Proof of Proposition 3.5

Proof. We need to show that the $<^{PWR}$ relation does not rule out any predictable data race pairs. For this to hold we show that any correctly reordered prefix satisfies the $<^{PWR}$ relation. Clearly, this is the case for the PO and WRD.

What other happens-before conditions need to hold for correctly reordered prefixes? For critical sections we demand that they must follow a proper acquire/release order. We also cannot arbitrarily reorder critical sections as write-read dependencies must be respected. See Lemma D.3. Condition ROD catches such cases.

We have $e \in CS(y)$, $f \in CS(y)'$ and $e <^{PWR} f$. Critical section $CS(y)'$ appears after $CS(y)$ (otherwise $e <^{PWR} f$ would not hold). Considering the entire trace, $CS(y)'$ cannot be put in front of $CS(y)$ via some reordering (see Lemma D.3).

As we may only consider a prefix, it is legitimate to apply some reordering that only affects parts of $CS(y)'$. Due to $e <^{PWR} f$ we may only reorder the part of $CS(y)'$ that is above of f in the trace. This requirement is captured via $rel(CS(y)) <^{PWR} f$.

We find that the $<^{PWR}$ relation does not rule out any of the correctly reordered prefixes. This concludes the proof. □

D.4 Proof of Proposition 3.8

Proof. We need to show that some correctly reordered prefix of T exists for which the potential Lockset-PWR race pair (e, f) appear right next to each other in the reordered trace. W.l.o.g. we assume that e appears before f in T and $thread(e) = 1$ and $thread(f) = 2$.

By assumption $LS(e) = LS(f) = \{\}$. The layout of the trace is as follows.

$1\sharp$	$2\sharp$
$\vdots$	$\vdots$
e	
T_1	
	T_1'
T_2	
	T_2'
$\vdots$	$\vdots$
T_n	
	T_n'
	f

Clearly, none of the parts $T_1, \ldots, T_n$ can happen before any of the parts $T_1', \ldots, T_n'$ w.r.t. the $<^{PWR}$ relation. Otherwise, $e <^{PWR} f$ which contradicts the assumption.

Hence, $T_1', \ldots, T_n'$ are independent of $T_1, \ldots, T_n$ and the trace can be correctly reordered as follows.

$1\sharp$	$2\sharp$
$\vdots$	$\vdots$
	T_1'
	$\vdots$
	T_n'
e	
	f
T_1	
$\vdots$	
T_n	

Hence, we are done. □

The result does not extend to more than two threads. The condition that the lockset is empty is also critical.

Example D.4. Consider

	$1\sharp$	$2\sharp$
1.		$w(x)$
2.	$acq(z)$	
3.	$r(x)$	
4.	$w(y)$	
5.	$rel(z)$	
6.		$acq(z)$
7.		$w(x)$
8.		$rel(z)$
9.		$w(y)$

Events $w(y)_4$ and $w(y)_9$ are not ordered under PWR. The lockset of $w(y)_4$ contains z. Both events are a potential lockset-PWR race pair but this is not a predictable data race pair.

D.5 Proof of Proposition 4.5

We first state some auxiliary results.

In general, we can reach all missing pairs by using pairs in $conc(x)$ as a start and by following edge constraints. This property is guaranteed by the following statement. We slightly abuse notation and identify events e, f, g via their epochs and vice versa.

Lemma D.5. *Let T be a trace and x be some variable. Let $edges(x)$ and $conc(x)$ be obtained by PWR^{E+E}. Let (e, f) be two conflicting events involving variable x where $(e, f) \notin conc(x), pos(e) < pos(f)$ and e, f are concurrent to each other w.r.t. PWR. Then, there exists $g_1, \ldots g_n \in edges(x)$ such that $e \prec g_1 \prec \ldots \prec g_n$ and $(g_n, f) \in conc(x)$.*

Proof. We consider the point in time event e is added to $RW(x)$ when running PWR^{E+E}. By the time we reach f, event e has been removed from $RW(x)$. Otherwise, $(e, f) \in conc(x)$ which contradicts the assumption.

Hence, there must be some g_1 in $RW(x)$ where

$$pos(e) < pos(g_1) < pos(f).$$

As g_1 has removed e, there must exist $e \prec g_1 \in edges(x)$ (1).

By the time we reach f, either g_1 is still in $RW(x)$, or g_1 has been removed by some g_2 where $g_1 \prec g_2 \in edges(x)$ and $g_2 \in RW(x)$. As between e and f there can only be a finite number of events, we must reach some $g_n \in RW(x)$ where $g_1 \prec \ldots \prec g_n$ (2). Event g_n must be concurrent to f.

Suppose g_n is not concurrent to f. Then, $g_n <^{PWR} f$ (3). The case $f <^{PWR} g_n$ does not apply because g_n appears before f in the trace. Edges imply PWR relations. From (2), we conclude that $g_1 <^{PWR} \ldots <^{PWR} g_n$ (4). (1), (2) and (4) combined yields $e <^{PWR} f$. This contradicts the assumption that e and f are concurrent.

Hence, g_n is concurrent to f. Hence, $(g_n, f) \in conc(x)$. Furthermore, we have that $e \prec g_1 \prec \ldots \prec g_n \in edges(x)$. □

Example 4.6 does not contradict the above Lemma D.5. The lemma states that all concurrent pairs can be identified.

The next property characterizes a sufficient condition under which a pair is added to $conc(x)$.

Lemma D.6. *Let T be a well-formed trace. Let $e, f \in T^{rw}_x$ for some variable x such that (1) e and f are concurrent to each other w.r.t. PWR, (2) $pos(f) > pos(e)$, and (3) $\neg\exists g \in T^{rw}_x$ where g and f are concurrent to each other w.r.t. PWR and $pos(f) > pos(g) > pos(e)$. Let $conc(x)$ be the set obtained by PWR^{E+E}. Then, we find that $(e, f) \in conc(x)$.*

Proof. By induction on T. Consider the point where e is added to $RW(x)$. We assume that e's epoch is of the form $j\sharp k$. We show that e is still in $RW(x)$ at the point in time we process f.

Assume the contrary. So, e has been removed from $RW(x)$. This implies that there is some g such that $e <^{PWR} g$ and $pos(f) > pos(g) > pos(e)$. We show that g must be concurrent to f.

Assume the contrary. Suppose $g <^{PWR} f$. But then $e <^{PWR} f$ which contradicts the assumption that e and f are concurrent to each other. Suppose $f <^{PWR} g$. This contradicts the fact that $pos(f) > pos(g)$.

We conclude that g must be concurrent to f. This is a contradiction to (3). Hence, e has not been removed from $RW(x)$.

By assumption e and f are concurrent to each other. Then, we can argue that $k > Th(i)[j]$ where by assumption $Th(i)$ is f's vector clock and e has the epoch $j\sharp k$. Hence, (e, f) is added to $conc(x)$. □

We are now ready to verify Proposition 4.5.

Proof. We first show that the construction of $PC(x)$ terminates by showing that no pair is added twice. Consider $(e, f) \in conc(x)$ where $g \prec e$. We remove (e, f) and add (g, f).

Do we ever encounter (f, e)? This is impossible as the position of first component is always smaller than the position of the second component.

Do we re-encounter (e, f)? This implies that there must exist g such that $e \prec g$ where $(g, f) \in conc(x)$. By Lemma D.6 this is in contradiction to the assumption that (e, f) appeared in $conc(x)$. We conclude that the construction of $PC(x)$ terminates.

Pairs are kept in a total order imposed by the position of the first component. As shown above we never revisit pairs. For each e any predecessor g where $g \prec e \in edges(x)$ can be found in constant time (by using a graph-based data structure). Then, a new pair is built in constant time.

There are $O(n * n)$ pairs overall to consider. We conclude that the construction of $PC(x)$ takes time $O(n * n)$. By Lemma D.5 we can guarantee that all pairs in $C^T(x)$ will be reached. Then, $C^T(x) \subseteq PC(x)$. □

D.6 Proof of Lemma 4.7

Proof. Follows from the fact that PWR^{E+E} computes the event's lockset and vector clock. To check if two events are concurrent it suffices to compare the earlier in the trace events time stamp against the time stamp of the later in the trace event. Recall that for pairs in $conc(x)$ and therefore also $PC(x)$, the left component event occurs earlier in the trace than the right component event. □

E WRD RACE PAIRS

Lockset-PWR WRD race pairs characterize write-read races resulting from the trace-specific or alternative schedules.

Example E.1. Consider the following trace (on the left) and the set of predictable and trace-specific race pairs (on the right).

	$1\sharp$	$2\sharp$
1.		$w(x)$
2.	$w(x)$	
3.	$acq(y)$	
4.	$rel(y)$	
5.		$acq(y)$
6.		$rel(y)$
7.		$r(x)$

where

$$\mathcal{P}^T = \{(w(x)_1, w(x)_2), (w(x)_2, r(x)_7)\}$$

$$\mathcal{S}^T = \{(w(x)_1, w(x)_2)\}$$

There are no read-write races in this case. The pair $(w(x)_2, r(x)_7)$ results from the correctly reordered prefix (alternative schedule) $T' = [2\sharp w(x)_1, 2\sharp acq(y)_5, 2\sharp rel(y)_6, 1\sharp w(x)_2, 2\sharp r(x)_7]$. The pair $(w(x)_2, r(x)_7)$ is not in $\mathcal{S}^T$ because T' represents some alternative schedule and there is no trace-specific schedule where the write and read appear right next to each other.

The pair $(w(x)_1, r(x)_7)$ is Lockset-PWR WRD race pair. However, this pair is not a SHB WRD race pair because the write-read race results from some alternative schedule.

F PWR VARIANTS

We consider the following variant of PWR where we impose a slightly different ROD rule.

Definition F.1 (WRD + ROD with Acquire). Let T be a trace. We define a relation $<^{PWRA}$ among trace events as the smallest partial order that satisfies conditions PO and WRD as well as the following condition:

ROD with Acquire: Let $f \in T$ be an event. Let $CS(y), CS(y)'$ be two critical sections where $CS(y)$ appears before $CS(y)'$ in the trace, $f \in CS(y)'$ and $acq(CS(y)) <^{PWRA} f$. Then, $rel(CS(y)) <^{PWR} f$.

We refer to $<^{PWRA}$ as the *WRD + ROD with Acquire* (PWRA) relation.

The ROD rule in Definition 3.1 is more general compared to the ROD with Acquire rule. The ROD rule says that if $e \in CS(y)$, $f \in CS(y)'$ and $e <^{PWR} f$. then $rel(CS(y)) <^{PWR} f$. Hence, the ROD with Acquire rule is an instance of this rule. Take $e = acq(CS(y))$. Hence, $<^{PWRA} \subseteq <^{PWR}$. We can even show that all PWR relations are already covered by PWRA.

Lemma F.2. $<^{PWR} = <^{PWRA}$.

Proof. Case $<^{PWRA} \subseteq <^{PWR}$: Follows from the fact that PWRA is an instance of PWR.

Case $<^{PWR} \subseteq <^{PWRA}$: We verify this case by induction over the number of ROD rule applications.

The base cases of the induction proof hold as both PWR and PWRA assume PO and WRD. Consider the induction step. We must find the following situation. We have that $rel(CS(y)) <^{PWR} f$ where (1) $e \in CS(y)$, (2) $f \in CS(y)'$ and (3) $e <^{PWR} f$. We need to show that $rel(CS(y)) <^{PWRA} f$.

From (1), (3) and PO we conclude that $acq(CS(y)) <^{PWR} f$. By induction we find that $acq(CS(y)) <^{PWRA} f$. We are in the position to apply the ROD with Acquire rule and conclude that $rel(CS(y)) <^{PWRA} f$ and we are done. □

We consider yet another variant of PWR.

Definition F.3 (WRD + ROD for Read). Let T be a trace. We define a relation $<^{PWRR}$ among trace events as the smallest partial order that satisfies conditions PO and WRD as well as the following condition:

ROD for Read: Let $e, f \in T$ be two events where f is a read event. Let $CS(y), CS(y)'$ be two critical sections where $e \in CS(y)$, $f \in CS(y)'$ and $e <^{PWRR} f$. Then, $rel(CS(y)) <^{PWRR} f$.

We refer to $<^{PWRR}$ as the *WRD + ROD for Read* (PWRR) relation.

The difference to PWR is that the ROD for Read rule only applies to read events. Again, we find that $<^{PWRR} \subseteq <^{PWR}$ because PWRR is an instance of PWR. However, the other direction does not hold because some PWR relations do not apply for PWRR as the following example shows.

Example F.4. Consider the trace

	$1\sharp$	$2\sharp$
1.	$acq(y)$	
2.	$w(x)$	
3.	$w(z)$	
4.	$rel(y)$	
5.		$r(x)$
6.		$acq(y)$
7.		$w(z)$
8.		$rel(y)$
9.		$w(z)$

Between $w(x)_2$ and $r(x)_5$ there is a WRD. In combination with PO, we find that $acq(y)_1 <^{PWR} w(z)_7$. Via the ROD rule we conclude that $rel(y)_4 <^{PWR} w(z)_7$. As there is no read event in the (second) critical section $(acq(y)_6, rel(y)_8)$, we do not impose $rel(y)_4 <^{PWR} w(z)_7$ under PWRR.

We summarize. PWR and PWRA are equivalent. PWRR is weaker. In the context of data race prediction this means that by using PWRR we may encounter more false positives.

Consider again Example F.4. Under PWRR, conflicting events $w(z)_3$ and $w(z)_9$ are not synchronized and their lockset is disjoint. Hence, $(w(z)_3, w(z)_9)$ form a potential data race pair under PWRR. This is a false positive because due to the WRD the critical sections cannot be reordered such that $w(z)_3$ and $w(z)_9$ appear right next to each other.

G PWR^{E+E} OPTIMIZATIONS

G.1 Application of ROD Rule

Function w3 enforces the ROD rule. In general, this needs to be done for each event to be processed. For events in thread i, we can skip w3 if w3 has been called for some earlier event in thread i and no new critical sections from some other thread are added to the history.

G.2 Read-Read Pair Removal

Algorithm 3 PWR^{E+E} Read-Read Optimizations

1: **procedure** READ(i, x)
2: $j = L_{W_t}(x)$
3: **if** $Th(i)[j] > L_W(x)[j] \wedge LS_t(i) \cap L_{W_L}(x) = \emptyset$ **then**
4: $reportPotentialRace(i\sharp Th(i)[i], j\sharp L_W(x)[j])$
5: **end if**
6: $Th(i) = Th(i) \sqcup L_W(x)$
7: $Th(i) = \text{w3A}(Th(i), LS_t(i))$
8: $evt = \{(i\sharp Th(i)[i], Th(i), LS_t(i))\} \cup evt$
9: $edges(x) = \{j\sharp k \prec i\sharp Th(i)[i] \mid j\sharp k \in RW(x) \wedge k < Th(i)[j]\} \cup edges(x)$
10: $conc(x) = \{(j\sharp k, i\sharp Th(i)[i]) \mid j\sharp k \in RW(x) \wedge k > Th(i)[j] \wedge j\sharp k \text{ is a write}\} \cup conc(x)$
11: $RW(x) = \{i\sharp Th(i)[i]\} \cup \{j\sharp k \mid j\sharp k \in RW(x) \wedge (k > Th(i)[j] \vee j\sharp k \text{ is a write})\}$
12: $\mathsf{inc}(Th(i), i)$
13: **end procedure**

The set $conc(x)$ also maintains concurrent read-read pairs. This is necessary as we otherwise might miss to detect some read-write

race pairs. We give an example shortly. In practice there are many more reads compared to writes. Hence, we might have to manage a high number of concurrent read-read pairs.

We can remove all read-read pairs from $conc(x)$ if we relax the assumptions on $RW(x)$. Usually, all events in $RW(x)$ must be concurrent to each other. We relax this condition as follows:

- All writes considered on their own and all reads considered on their own are concurrent to each.
- A write may happen before a read.

Based on the relaxed condition, set $conc(x)$ no longer needs to keep track of read-read pairs.

Algorithm 3 shows the necessary changes that only affect the processing of reads. The additional side condition "$j\sharp k$ is a write" ensures that no read-read pairs will be added to $conc(x)$. For $RW(x)$ the additional side condition guarantees that a write can only be removed by a subsequent write (in happens-before PWR relation).

Example G.1. Consider the trace in Figure 1. We write $RW(x)'$ and $conc(x)'$ to refer to the sets as calculated by Algorithm 1 whereas $RW(x)$ and $conc(x)$ refer to the sets as calculated by Algorithm 3.

The race pair (w_1, r_4) is detected in the first pass of Algorithm 3. Based on Algorithm 1 we require some second pass to detect (w_1, r_4) based on $w_1 \prec r_2$ and (r_2, r_4).

We conclude. All read-read pairs can be eliminated from $conc(x)$ by making the adjustments described by Algorithm 3. By relaxing the conditions on $RW(x)$ any write-read pair that is detectable by the second pass via a read-read pair and some write-read edges is immediately detectable via the set $RW(x)$. Recall that a write in $RW(x)$ will only be removed from $RW(x)$ if there is a subsequent write in happens-before PWR relation. Hence, Algorithms 1 and 3 and their respective second passes yield the same number of potential race pairs.

The time and space complexities are also the same. The set $RW(x)$ under the relaxed conditions is still bounded by $O(k)$. We demand that that all writes considered on their own and all reads considered on their own are concurrent to each. Hence, there can be a maximum of $O(k)$ writes and $O(k)$ reads.

The above example suggests that we may also remove write-read edges. The edge $w_1 \prec r_2$ plays no role for the second pass based on Algorithm 3. This assumption does not hold in general. The construction of $edges(x)$ for Algorithms 1 and 3 must remain the same.

Example G.2. Consider the following trace.

	$1\sharp$	$2\sharp$	$3\sharp$
1.	$w(x)$		
2.	$r(x)$		
3.	$w(y)$		
4.		$r(y)$	
5.		$r(x)$	
6.		$w(x)$	
7.			$w(x)$

Due to the write-read dependency involving variable y, Algorithm 3 only reports a single write-write pair, namely (w_6, w_7). The additional pair (w_1, w_7) is detected during the second pass where write-read and read-write edges such as $w_1 \prec r_2$ and $r_5 \prec w_6$ are necessary.

G.3 Aggressive Filtering

We aggressively apply the filtering check (Lemma 4.7) during the second pass. A pair $(e, f) \in conc(x)$ (step (2) in Definition 4.4) that fails the Lockset + PWR Filtering check will not be added to $PC(x)$ (step (4)). But we have to consider the candidates (g_i, f) and add them to $conc(x)$ (step (5)) as we otherwise might miss some potential race candidates.

Example G.3. Consider the following trace.

	$1\sharp$	$2\sharp$
1.	$w(x)$	
2.	$acq(y)$	
3.	$w(x)$	
4.	$rel(y)$	
5.	$w(x)$	
6.		$acq(y)$
7.		$w(x)$
8.		$rel(y)$

In the first pass we obtain $conc(x) = \{(w_5, w_7)\}$ and $edges(x) = \{w_1 \prec w_3 \prec w_5\}$. The second pass proceeds as follows. Via (w_5, w_7) we obtain the next candidate (w_3, w_7). This candidate is not added to $PC(x)$ because locksets of w_3 and w_7 are not disjoint. Hence, the filtering check fails.

We remove (w_5, w_7) from $conc(x)$ but add (w_3, w_7) to $conc(x)$. Adding (w_3, w_7) is crucial. Via (w_3, w_7) we obtain candidate (w_1, w_7). This candidate is added to $PC(x)$ (and represents an actual write-write race pair).

There are cases where we can completely ignore candidates. If the filtering check fails because e and f are in happens-before PWR relation, then we can completely ignore (e, f) and add (e, f) *not* to $conc(x)$. This is safe because all further candidates reachable via edge constraints will also be in PWR relation. Hence, such candidates would fail the filtering check as well.

G.4 Removal of Critical Sections

The history of critical sections for lock y is maintained by $H(y)$. We currently only add critical sections without ever removing them. From the view of thread i and its to be processed events, we can safely remove a critical section if (a) thread i has already synchronized with this critical section (see function w3 in Algorithm 1), and (b) the release event happens-before the yet to be processed events.

Removing of critical sections is specific to a certain thread. Hence, we use thread-local histories $H(i, y)$ instead of a global history $H(y)$. Both removal conditions can be integrated into function w3. See the updated function w3 in Algorithm 4. Function w3 additionally expects the thread id (and therefore all calls must include now this additional parameter).

We always remove after synchronization. Hence, removal checks (a) and (b) boil down to the same check which is carried out within line numbers 4-6. If the time stamp of the release is smaller compared to thread's time stamp (for the thread the release is in), the

	1♯	2♯	3♯	$RW(x)'$	$RW(x)$	$conc(x)'$	$conc(x)$	$edges(x)$
1.	$w(x)$			$\{w_1\}$	$\{w_1\}$			
2.	$r(x)$			$\{r_2\}$	$\{w_1, r_2\}$			$w_1 \prec r_2$
3.		$w(x)$		$\{r_2, w_3\}$	$\{w_1, r_2, w_3\}$	(r_2, w_3)	(w_1, w_3)	
							(r_2, w_3)	
4.		$r(x)$		$\{r_2, r_4\}$	$\{w_1, r_2, w_3, r_4\}$	(r_2, r_4)	(w_1, r_4)	$w_2 \prec r_4$
5.			$r(x)$	$\{r_2, r_4, r_5\}$	$\{w_1, r_2, w_3, r_4, r_5\}$	(r_2, r_5)	(w_1, r_5)	
						(r_4, r_5)	(w_3, r_5)	

Figure 1: Read-Read Optimization

Algorithm 4 Thread-local history and removal

```
1:  function w3(i, V, LS_t)
2:      for y ∈ LS_t do
3:          for (j♯k, V') ∈ H(i, y) do
4:              if V'[ j ] < V[ j ] then
5:                  H(i, y) = H(i, y) − {(j♯k, V')}
6:              else
7:                  if k < V[j] then
8:                      V = V ⊔ V'
9:                  end if
10:             end if
11:         end for
12:     end for
          return V
13: end function
1:  procedure RELEASE(i, y)
2:      Th(i) = w3(i, Th(i), LS_t(i))
3:      LS_t(i) = LS_t(i) − {y}
4:      for i' ≠ i do
5:          H(i', y) = H(i', y) ∪ {(Acq(()x), Th(i))}
6:      end for
7:      inc(Th(i), i)
8:  end procedure
```

release happens-before and therefore the critical section can be removed.

In case of a release event, we add the critical section to all other thread-local histories. Processing of all other events as well as the second pass remains unchanged.

In theory, the size of histories can still grow considerably.

Example G.4. Consider the following trace.

	1♯	2♯
1.	$acq(y)$	
2.	$w(x_1)$	
3.	$acq(y)$	
...		
	$acq(y)$	
	$w(x_n)$	
	$rel(y)$	
		$acq(y)$
		$r(x_i)$
		$rel(y)$

In thread 2's thread-local history we would find all n critical sections of thread 1. This shows that size of thread-local histories may grow linearly in the size of the trace.

As we assume the number of distinct variables is a constant, some of the variables x_j might be repeats. Hence, we could truncate thread 2's thread-local history by only keeping the most recent critical section that contains a write access to x_j. Hence, the number of distinct variable imposes a bound on the size of thread-local histories.

Similarly, we can argue that the number of thread imposes a bound on the size of thread-local histories. Hence, we claim that the size of thread-local histories be limited to the size $O(v * k)$ without compromising the correctness of the resulting PWR relation. We assume that k is the number of threads and v the number of distinct variables.

Maintaining the size $O(v * k)$ for thread-local histories would require additional management effort. Tracking thread id's of critical sections and the variable accesses that occur within critical sections etc. In our practical experience, it suffices to simply impose a fixed limit for thread-local histories. For the examples we have encountered, it suffices to only keep the five most recent critical sections. That is, when adding a critical section to a thread-local history and the limit is exceeded, the to be added critical section simply overwrites the oldest critical section in the thread-local history.

H PRECISION BENCHMARKS

The precision benchmark suite consists of 28 tests cases that give rise to 45 predictable races. For 6 out of the 28 test cases there are no data races. Many test cases require alternative schedules to be explored to predict the the data race.

Recall that PWR_L^{E+E} employs a limited number of edge constraints which may result in incompleteness (false negatives) and also limits the history of critical sections which may lead to more false positives. The limits we employ PWR_L^{E+E} have no impact on the number of false negatives and false positives compared to PWR^{E+E}. We introduce the additional candidates SHB and TSanWRD. SHB is a variant of SHB_L^{E+E} where the limit of edge constraints is zero. TSanWRD is an extension of TSan that includes write-read dependencies.

Table 2 shows the precision measurements for each algorithm. Column #Race Candidates / False Positives reports the overall number of race candidates reported and the number of false positives among candidates. TSan reports the highest number of race

	#Race Candidates / False Positives	$FP_{\not\exists}$	$FP_{\forall}$	$FN_{\not\exists}$	$FN_{\forall}$
FastTrack	23 / 5	25	0	4	15
SHB	14 / 0	28	0	4	15
SHB_L^{E+E}	19 / 0	28	0	5	15
TSan	54 / 16	17	5	20	0
TSanWRD	46 / 8	20	4	20	0
PWR_L	45 / 7	21	3	20	0
PWR_L^{E+E}	52 / 7	21	3	22	0
WCP	31 / 7	23	1	10	9

Table 2: Precision results (28 test cases with 45 predictable races)

candidates (54) but includes a large number of false positives (16). Hence, only 38 (=54-16) are (actual) data races. TSanWRD catches like TSan 38 (=46-8) data races but reports fewer race candidates (46) out of which eight are false positives. The precision of PWR_L is similar to TSanWRD. 38 data races are caught out of 45 candidates that include seven false positives. PWR_L^{E+E} catches all 45 (=52-7) races and reports 52 race candidates out of which seven are false positives. WCP reports 31 race candidates out of which 24 (=31-7) are data races due to seven false positives. SHB and SHB^{E+E} report the fewest number of race candidates but come with the guarantee that no false positives are reported. FastTrack catches 18 (=23-5) data races. Recall that FastTrack ignores write-read dependencies.

Based on the overall precision measured in terms of number of race candidates and false positives, we draw the following conclusions. When it comes to zero false positives, SHB and SHB^{E+E} perform best. TSan yields many false positives. When aiming for many data races with a manageable number of false positives, TSanWRD, PWR_L and WCP are good choices. PWR_L^{E+E} is the best choice when the aim is to catch all data races with a manageable number of false positives. FastTrack yields also a manageable number of false positives but catches considerably fewer data races.

We examine in more the detail the issue of false positives and false negatives. For this purpose, we measure the number of tests for which an algorithm yields no false positives among candidates reported (column $FP_{\not\exists}$), only false positives among candidates reported (column $FP_{\forall}$), no false negatives (column $FN_{\not\exists}$), only false negatives (column $FN_{\forall}$). By no false negatives we mean that all races for that test are reported. By only false negatives we mean that no races are reported although the test has a race.

FastTrack does not report any false positives for 25 out of the 28 tests cases. See column $FP_{\not\exists}$. On the other hand, there are 15 tests cases with races for which no race is reported (column $FN_{\forall}$) and there are only 4 test cases for which all races are reported (column $FN_{\not\exists}$). Any case listed in $FN_{\forall}$ also contributes to $FP_{\not\exists}$. Hence, the number 25 in column $FP_{\not\exists}$ results from the fact that FastTrack reports considerably fewer race candidates compared to some of the other algorithms. SHB and SHB^{E+E} have the same number of "false negative" cases as FastTrack. Their advantage is that both come with the guarantee of not having any false positives.

WCP is able to detect races resulting from alternative schedules. This is the reason that WCP performs better than FastTrack, SHB and SHB^{E+E} when comparing the numbers in columns $FN_{\not\exists}$ and $FN_{\forall}$. However, WCP appears to be inferior compared to the family of "TSan" and "PWR" algorithms. TSan, TSanWRD and PWR_L are able to report all races for 20 test cases. For PWR_L^{E+E} we find 22 test cases. See column $FN_{\not\exists}$. Recall that there are 28 test cases overall out of which 22 test cases have races and six test cases have no races. Hence, 22 test cases is the maximum number to achieve in column $FN_{\not\exists}$.

In summary, the performance and precision benchmark suites show that PWR_L^{E+E} offers competitive performance while achieving high precision.

I COUNTING DATA RACES

In our measurements, we report 252(2) to indicate that 252 races are found overall of which two races are found via edge constraints. Because we only count races modulo their code locations, it is possible that a race can be found directly *and* via edge constraints. Instead of "direct" races and "edge constraint" races, we introduce the notation of first pass and second pass race.

We refer to a *second pass* race as a race that is obtained via the help of edge constraints $edges(x)$ and the set $conc(x)$ of concurrent reads/writes. We refer to a *first pass* race as a race that is either a write-read race or a race pair from $conc(x)$ that is not a second pass race pair. We assume that race pairs are reported following the order as defined by the construction in Definition 4.4.

The side condition "not a second pass race pair" for a first pass race pair seems strange as there should not be any mix up between first and second pass races. However, this is possible for two reasons. We only report race pairs modulo their code locations and we compare variants of "PWR" that impose different limits on edge constraints.

Consider the trace

	$1\sharp$	$2\sharp$
1.	$w(x)^a$	
2.	$w(x)^b$	
3.		$w(x)^c$
4.	$w(x)^a$	

where we attach superscripts to indicate the code locations where the events result from. We find that w_1 and w_4 result from the same code location.

We find that $conc(x) = \{(w_2, w_3), (w_3, w_4)\}$ and $edges(x) = \{w_1 \prec w_2, w_2 \prec w_4\}$. We first report the first pass race (w_2, w_3). Via edge constraints we report the second pass race (w_1, w_3). The race pair (w_3, w_4) is not reported because we have already reported (w_1, w_3). Recall that these race pairs refer to the same code locations. In terms of code locations, we find the first pass race (b, c) and the second pass race (a, c).

If we impose a limit on edge constraints, say zero, we only consider $conc(x)$. Then, first pass races reported are (w_2, w_3) and (w_3, w_4). In terms of code locations, we report (b, c) and (a, c). But this means that (a, c) can either be reported as a first pass or second pass race depending on the limit imposed on edge constraints.

Such cases arise in our measurements in Table 1. For H2, PWR_L reports 252 first pass races whereas PWR_L^{E+E} reports 252(2) races. That is, two of the second pass races are already reported by PWR_L as first pass races due to the fact the filter races modulo their code locations.