Dynamic Race Prediction in Linear Time

Dileep Kini Umang Mathur Mahesh Viswanathan
University of Illinois at Urbana-Champaign, USA
dileeprkini@gmail.com, umathur3@illinois.edu, vmahesh@illinois.edu

Abstract

Writing reliable concurrent software remains a huge challenge for today's programmers. Programmers rarely reason about their code by explicitly considering different possible inter-leavings of its execution. We consider the problem of detecting data races from individual executions in a sound manner. The classical approach to solving this problem has been to use Lamport's happens-before (HB) relation. Until now HB remains the only approach that runs in linear time. Previous efforts in improving over HB such as causally-precedes (CP) and maximal causal models fall short due to the fact that they are not implementable efficiently and hence have to compromise on their race detecting ability by limiting their techniques to bounded sized fragments of the execution. We present a new relation weak-causally-precedes (WCP) that is provably better than CP in terms of being able to detect more races, while still remaining sound. Moreover, it admits a linear time algorithm which works on the entire execution without having to fragment it.

CCS Concepts • **Software and its engineering** → **Software testing and debugging**; *Formal software verification*

Keywords Concurrency, Race Prediction, Online Algorithm

1. Introduction

Writing reliable concurrent program remains a huge challenge; depending on the order in which threads are scheduled, there are a large number of possible executions. Many of these executions remain unexplored despite extensive testing. The most common symptom of a programming error in multi-threaded programs is a data race. A data race is a pair of *conflicting* memory accesses such that in some execution of the program, these memory accesses are performed consecutively; here, by conflicting memory accesses, we mean, pair of read/write events to the same memory location performed by different threads, such that at least one of them is a write. The goal of dynamic race detectors is to discover the presence of a data race in a program by examining a *single* execution. Given its singular role in debugging multi-threaded programs, dynamic race detection has received robust attention from the research community since the seminal papers [23, 37] more than two decades ago.

All dynamic race detection algorithms can be broadly classified into three categories. First are the *lock-set* based approaches [9, 37] that detect *potential* data races by tracking the set of locks held during each data access. These methods are fast and have low overhead, but are *unsound* in that many potential races reported are spurious. The second class of techniques falls in the category of *predictive runtime analysis* techniques [18, 36]. Here the race detector explores all possible reorderings of the given trace, searching for a possible witness that demonstrates a data race. These techniques are precise — races detected are indeed data races, and they are likely to find all such races. The downside is that they are expensive. A single trace has potentially exponentially many reorderings. Therefore, these techniques are applied by slicing the trace in small-sized fragments, and searching for a race in these short fragments. The last class of techniques are what we call *partial order* based techniques. In these, one identifies a partial order P on the events in the trace such that events unordered by P correspond to "concurrent events". These algorithms are sound (presence of unordered conflicting events indicates a data race) and have low overhead (typically polynomial in the size of the trace). However, they are conservative and may miss anomalies detected by the predictive runtime analysis techniques. The approach presented in this paper falls in this last category.

Happens-before (HB) [23] is the simplest, and most commonly used partial order to detect races. It orders the events in a trace as follows: (i) two events performed by the same thread are ordered the way they appear in the trace, and (ii) synchronization events across threads are also HB-ordered in order of appearance in the trace if those events access the same synchronization objects. Rule (i) says that we cannot reorder events within a thread because we have no information about the underlying program which allows us to infer an alternate execution of the thread. Rule (ii) says that we cannot reorder synchronization events on the same objects (in our

PLDI'17, June 18–23, 2017, Barcelona, Spain
© 2017 ACM. 978-1-4503-4988-8/17/06...$15.00
http://dx.doi.org/10.1145/3062341.3062374

	t_1	t_2
1	acq(l)	
2	r(x)	
3	w(x)	
4	rel(l)	
5		acq(l)
6		r(x)
7		w(x)
8		rel(l)

(a) cannot swap critical sections

	t_1	t_2
1	w(y)	
2	acq(l)	
3	r(x)	
4	rel(l)	
5		acq(l)
6		r(x)
7		rel(l)
8		r(y)

(b) can swap critical sections

Figure 1. Example traces showing when critical sections can/cannot be swapped.

	t_1	t_2
1	w(y)	
2	acq(l)	
3	w(x)	
4	rel(l)	
5		acq(l)
6		r(x)
7		r(y)
8		rel(l)

(a) no predictable race

	t_1	t_2
1	w(y)	
2	acq(l)	
3	w(x)	
4	rel(l)	
5		acq(l)
6		r(y)
7		r(x)
8		rel(l)

(b) predictable race

Figure 2. Example traces showing how CP misses race due to small change.

case locks) as they would lead to violation of mutual exclusion (critical sections on same lock should not overlap—also referred to as *lock semantics*). Consider for example the trace shown in Figure 1a. (In all the example figures we follow the convention of representing events of the trace top-to-bottom, where temporally earlier events appear above the later ones. We also use the syntax of `acq(l)`/`rel(l)` for acquire/release events of lock `l` and `r(x)`/`w(x)` for read/write events on variable `x`.) The events on lines 4 and 5 cannot be interchanged temporally as mutual exclusion will be violated. We could consider circumventing this lock semantics violation by reordering the entire critical sections (which would also change relative positions of events on lines 4, 5), but we cannot infer such a move because the `r(x)` event of thread t_2 could see a different value which could cause alternative executions in the underlying program and hence the events following it might be different. So in this case the HB reasoning, of avoiding lock semantics violation correctly, though unwittingly, prevented the swapping of critical sections (the swapping does not violate lock semantics). Had the two `w(x)` events been absent, we could have actually swapped the two critical sections temporally to get a feasible alternate execution. For example, the trace in Figure 1b can be reordered to expose a race on the access of `y` by performing the critical section of t_2 before the other. Such a race is called *predictable* as the trace that exposes it can be obtained from the given trace by rearranging the temporal order of events across threads. That is, it can be predicted from the trace without having to look at the underlying program. For Figure 1b, HB will still not declare a race since events on lines 4 and 5 would be ordered.

The partial order Causally-Precedes (CP) [42] was introduced to detect races missed by HB such as those in Figure 1b while being sound. The CP relation is a subset of HB, which implies that CP can detect races above and beyond those detected by HB. Soundness of CP guarantees that a CP race is either an actual race or a deadlock. But there are two main drawbacks of CP. Firstly CP misses races that are predictable. Consider the traces shown in Figure 2; the only difference between the two traces is that lines 6 and 7 have been swapped. There is no predictable race in Figure 2a because `r(x)` in t_2 (line 6) must be performed after `w(x)` in t_1 (line 3), which prevents the critical sections from being reordered. On the other hand, Figure 2b has a predictable race on `y` — the sequence e_5, e_6, e_1 reveals the race (e_i refers to the event at line i). CP, however, does not detect a race in either trace, because it is agnostic to the ordering of events within a critical section. The second drawback of CP is that, while it can be detected in polynomial time [42], there is no known linear time algorithm [1]. This severely hampers its use on real-world examples with traces several gigabytes large. So any implementation of CP must resort to *windowing* where the trace is partitioned into small fragments. This means that it can only find races within bounded fragments of the trace, and so detects fewer races than what CP promises. Our experiments (Section 4) reveal windowing to be a serious impediment to race detection in large examples. A recent implementation of CP [34] performs an online analysis while avoiding windowing, even though, theoretically, its running time is not linear. Currently, it seems slower than our implementation of WCP, as it processes roughly a few million events in a few hours.

We address the two drawbacks of CP in one shot. We propose a partial order Weak Causally-Precedes (WCP) which, as the name suggests, is a weakening of the CP partial order. Thus WCP detects all races that CP does and even more (like the race in Figure 2b as explained in Section 2.3). We prove that WCP enjoys the same soundness guarantees as CP. Additionally, like HB, WCP admits a linear time Vector-Clock algorithm for race detection, thus solving the main open problem proposed in [42]. This is surprising because when HB was weakened to obtain CP it resulted in detecting more races but came at the cost of an expensive algorithm. But weakening CP to WCP not only allows for detecting more races but enables an efficient algorithm.

Our key contributions are the following:

1. We define a new *sound* relation weak-causally-precedes (WCP), which is weaker than causally-precedes. The

[1] We believe that there is quadratic time lower bound on *any* CP algorithm.

definition and its soundness proof, as claimed in [42], is challenging.

> "It is worth emphasizing that multiple researchers have fruitlessly pursued such a weakening of HB in the past. . . . (Both the definition of CP and our proof of soundness are results of multi-year collaborative work, with several intermediate failed attempts.)"

Even though we had the benefit of following the work on CP, our experience concurs with the above observation. The subtlety in these ideas is highlighted by the fact that the soundness proof for CP, presented in [42], is *incorrect*; informed readers can find an explanation of the errors in the CP soundness proof in our companion technical report [21]. Our attempts at fixing the CP proof led us to multiple years of fruitless labor until finally the results presented here. Our soundness proof for WCP (which also, by definition, applies to CP) requires significant extensions to the proof ideas outlined in [42] (see [21] for the full proof).

2. We achieve the holy grail for dynamic race detection algorithms — a *linear* running time. It is based on searching for conflicting events that are unordered by WCP. We prove that our algorithm is *correct*. Our algorithm uses linear space in the worst case, as opposed to the logarithmic space requirement of happens-before vector clock algorithm. However, in our experiments, we did not encounter these worst case bounds. Our algorithm scales to traces with hundreds of millions of events and the memory usage stays below 3% for most benchmarks (see Table 1). We further show that our algorithm is *optimal* in terms of its asymptotic running time and memory usage.
3. Our experiments show the benefits of our algorithm, in terms of the number of races detected and its efficiency when compared to state of the art tools such as RVPredict. They reveal the tremendous power of being able to analyze the entire trace as opposed to trace fragments that other sound race detection algorithms (other than those based on HB) are forced to be restricted to.

In Section 2 we describe the partial order WCP, and how it is a weakening of the CP relation. Section 3 describes the Vector-Clock algorithm that implements WCP faithfully and runs in linear time. In Section 4 we describe the implementation and the experimental results. We provide related work in Section 5 and give concluding remarks in Section 6.

2. Weak Causal Precedence

2.1 Preliminaries

In this paper we consider the sequential consistency model for assigning semantics to concurrent programs wherein the execution is viewed as an interleaving of operations performed by individual threads. The possible operations include lock acquire and release (`acq(l)`, `rel(l)`) and variable access which include read and write to variables (`r(x)`, `w(x)`).

Orderings: Let σ be a sequence of events. We say e_1 is *earlier than* e_2 according to σ, denoted by $e_1 <^{\sigma}_{tr} e_2$, when e_1 is performed before e_2 in the trace σ. We shall use $t(e)$ to denote the thread that performs e. We say e_1 is *thread ordered before* e_2, denoted by $e_1 <^{\sigma}_{TO} e_2$ to mean that $e_1 <^{\sigma}_{tr} e_2$ and $t(e_1) = t(e_2)$. We use $=^{\sigma}$ to denote the identity relation on events of σ and $\leq^{\sigma}_{tr}, \leq^{\sigma}_{TO}$ to denote the relations $(<^{\sigma}_{tr} \cup =^{\sigma})$ and $(<^{\sigma}_{TO} \cup =^{\sigma})$. We shall drop σ from the superscript of the relations when it is clear from context. We use $\sigma \restriction_t$ to denote the projection of σ onto the events by thread t.

Lock events: For a lock acquire/release event e, we use $l(e)$ to denote the lock on which it is operating. For an acquire event a we use $match(a)$ to denote the earliest release event r such that $l(a) = l(r)$ and $a <_{TO} r$. We similarly define the match of a release event r to be the latest acquire a such that $l(a) = l(r)$ and $a <_{TO} r$. A *critical section* is the set of events of a thread that are between (and including) an acquire and its matching release, or if the matching release is absent then all events that are thread order after an acquire. We use $\text{CS}(e)$ to denote a critical section starting/ending at an acquire/release event e. For event e and lock ℓ we use $e \in \ell$ to denote that e is contained in a critical section over ℓ.

Trace: For a sequence of events σ to be called a *trace* it needs to satisfy two properties:

1. *lock semantics:* for any two acquisition events a_1 and a_2 if $l(a_1) = l(a_2)$ and $a_1 <_{tr} a_2$ then $r_1 = match(a_1)$ exists in σ and $r_1 <_{tr} a_2$
2. *well nestedness:* for any critical section C, if there exists an acquire event a such that $a \in C$, then $r = match(a)$ exists and $r \in C$.

Race: Two events are said to be *conflicting* if they access the same variable and at least one of them is a write and the events are performed by different threads. We use $e_1 \asymp e_2$ to denote that e_1 and e_2 are conflicting. When one can execute a concurrent program such that conflicting events can be performed next to each other in the trace, we say that the trace has revealed a *race* in the program.

Deadlock: When a program is being executed to obtain a trace, the scheduler picks a thread and performs the "next event" in that thread. Note that when this event is performed the next event of the other threads is not going to be affected. This concept of next event is needed for understanding a deadlock. A trace is said to reveal a *deadlock* when a set of threads D cannot proceed because each of them is trying to acquire a lock that is held by another thread in that set. In other words, the next event in each thread in D is an `acq(l)` such that the lock `l` is acquired by another thread in D without having released it. In such a situation none of the threads in D can proceed because lock semantics will be violated, and

no matter how the rest of the threads proceed the next event of these threads will not change.

Predictability and Correct Reordering: In order to formalize the concept of predictable race/deadlock we need the notion of correct reordering. A trace σ' is said to be a *correct reordering* of another trace σ if for every thread t, $\sigma' \restriction_t$ is a prefix of $\sigma \restriction_t$, and the last `w(x)` event before any `r(x)` event is the same in both σ and σ'. This ensures that every read event in σ' sees/returns the same value as it did in σ. We say a trace σ has a *predictable race* (*predictable deadlock*) if there is a correct reordering of it which exhibits a race (deadlock).

2.2 Partial Orders

Given a trace σ, we consider various partial orders on its events. Formally, let $\mathcal{E}$ be the set of events in σ. A partial order P is a binary relation $\leq^\sigma_P$ on $\mathcal{E}$ (i.e. $\leq^\sigma_P \subseteq \mathcal{E}\times\mathcal{E}$) which is reflexive, antisymmetric and transitive. The relations $\leq^\sigma_{lr}, \leq^\sigma_{TO}$ defined earlier are examples of partial orders. We say two events e_1, e_2 are unordered by a partial order P, denoted by $e_1 \parallel_P e_2$, when neither $e_1 \leq^\sigma_P e_2$, nor $e_2 \leq^\sigma_P e_1$.

We say a trace σ exhibits a P-race between events e_1, e_2 when they are conflicting ($e_1 \asymp e_2$) and are unordered by P ($e_1 \parallel_P e_2$). The aim, in this race-detection paradigm of using partial orders, is to design partial orders P which can guarantee that for any σ, the presence of P-race in σ implies the presence of a predictable-race in σ. Such partial orders are said to be *strongly sound*. A partial order is said to be *weakly sound* if for any trace containing unordered conflicting events there is a correct reordering of the trace which reveals either a race or a deadlock. From a programmer's perspective deadlocks are as undesirable as races. Additionally, a weaker notion might allow for detection of additional races that are otherwise harder to detect. For programs which are guaranteed to be deadlock-free, the two notions coincide.

Before we delve into specific partial orders, we present some intuition regarding partial orders and alternative executions. Linearizations of a partial order $\leq^\sigma_P$ are possible executions of the trace with different interleavings of the threads. If two events are unordered by some partial order, then we can obtain a linearization in which the two events are placed next to one another (performed simultaneously). But this by itself does not imply soundness. This is because a linearization ρ of $\leq^\sigma_P$ (i) might not be a valid trace as it might violate lock semantics, or (ii) might not be a correct reordering of σ as there might be a read event `r(x)` whose corresponding last `w(x)` event in ρ does not match with that in σ. Therefore, two conflicting events unordered by a partial order only indicates the possibility of, and does not necessarily guarantee the existence of a predictable race. The cleverness lies in designing partial orders for which this possibility is indeed a guarantee.

We will now describe partial orders HB, CP (from literature) and WCP (our contribution).

Definition 1 (Happens-Before)**.** *Given a trace σ, $\leq^\sigma_{HB}$ is the smallest partial order on the events in σ with $<^\sigma_{TO} \subseteq \leq^\sigma_{HB}$ that satisfies the following rule: for a* `rel(l)` *event r and an* `acq(l)` *event a, if $r <^\sigma_{tr} a$, then $r \leq^\sigma_{HB} a$.*

Two salient features of HB are: (i) it is strongly sound, and (ii) it can be computed in linear time. Several race detection tools [14, 30] have been developed using this technique. However, as discussed in Section 1, HB can miss many races. The Causally-Precedes (CP) partial order was then introduced [42] as an improvement over HB. CP is a subset of HB i.e it has fewer orderings and hence more possible interleavings. It is thus able to detect more races than HB. CP is proved to be weakly sound. However, it is not known if CP can be computed in linear time. The race detection algorithm for CP is polynomial time but not linear.

Definition 2 (Causally-Precedes)**.** *Given a trace σ, $\prec^\sigma_{CP}$ is the smallest relation satisfying the following rules:*

(a) *for a* `rel(l)` *event r and an* `acq(l)` *event a with $r <^\sigma_{tr} a$, if the critical sections of r and a contain conflicting events ($e_1 \in \text{CS}(r)$, $e_2 \in \text{CS}(a)$, $e_1 \asymp e_2$), then $r \prec^\sigma_{CP} a$.*

(b) *for a* `rel(l)` *event r and an* `acq(l)` *event a with $r <^\sigma_{tr} a$, if the critical sections of r and a contain CP-ordered events ($e_1 \in \text{CS}(r), e_2 \in \text{CS}(a), e_1 \prec^\sigma_{CP} e_2$), then $r \prec^\sigma_{CP} a$.*

(c) *$\prec^\sigma_{CP} = (\prec^\sigma_{CP} \circ \leq^\sigma_{HB}) = (\leq^\sigma_{HB} \circ \prec^\sigma_{CP})$, i.e., $\prec^\sigma_{CP}$ is closed under composition with $\leq_{HB}$.*

As seen in the examples in Figure 2, CP is agnostic to the relative order of the events inside the same critical sections. That is, if we were to consider two read/write events that are enclosed within the same set of critical sections in the trace, and we interchanged their positions, then the resulting trace would have exactly the same CP orderings across threads. This is because Rule (a) of CP, that depends upon the position of read/write events, is only concerned with whether or not it occurs inside some critical section and not how they are relatively order within the critical section. This constraint prevents CP it from detecting races, as we saw in Figure 2b.

Next we look at WCP, the partial order we introduce in this paper. It is obtained by weakening rules (a) and (b) of CP as follows:

Definition 3 (Weak-Causally-Precedes)**.** *Given a trace σ, $\prec^\sigma_{WCP}$ is the smallest relation that satisfies the following rules:*

(a) *for a* `rel(l)` *event r and a read/write event $e \in$* `l` *with $r <^\sigma_{tr} e$, if $\text{CS}(r)$ contains an event conflicting with e ($e' \in \text{CS}(r)$, $e \asymp e'$), then $r \prec^\sigma_{WCP} e$.*

(b) *for* `rel(l)` *events r_1, r_2 with $r_1 <^\sigma_{tr} r_2$, if the critical sections of r_1, r_2 contain WCP-ordered events ($e_1 \in \text{CS}(r_1)$, $e_2 \in \text{CS}(r_2)$, $e_1 \prec^\sigma_{WCP} e_2$), then $r_1 \prec^\sigma_{WCP} r_2$.*

(c) *$\prec^\sigma_{WCP} = (\prec^\sigma_{WCP} \circ \leq^\sigma_{HB}) = (\leq^\sigma_{HB} \circ \prec^\sigma_{WCP})$, i.e., $\prec^\sigma_{WCP}$ is closed under composition with $\leq^\sigma_{HB}$,*

Note that, Rule (a) of WCP orders the release event before the read/write event (and not the acquire as in CP). Thus, WCP makes a distinction between events of the same thread based on the relative order inside a critical section.

Note that $\prec^{\sigma}_{CP}$ and $\prec^{\sigma}_{WCP}$ are not partial orders as they are not reflexive. And unlike $\leq^{\sigma}_{HB}$ they do not contain thread order. But if we consider the relations $\leq^{\sigma}_{CP} = (\prec^{\sigma}_{CP} \cup \leq^{\sigma}_{TO})$ and $\leq^{\sigma}_{WCP} = (\prec^{\sigma}_{WCP} \cup \leq^{\sigma}_{TO})$, then both $\leq^{\sigma}_{CP}$ and $\leq^{\sigma}_{WCP}$ are partial orders. When defining races we use these partial orders. When clear from the context, we drop σ from superscript of the relations. Rules (a) and (b) in WCP are weaker versions of rules (a) and (b) in CP respectively. One can prove inductively that $\leq_{WCP} \subseteq \leq_{CP}$ hence any CP-race is also a WCP-race. Next we state the soundness theorem for WCP whose proof is provided in [21].

Theorem 1 (Soundness of WCP)**.** *WCP is weakly sound, i.e., given any trace σ, if σ exhibits a WCP-race then σ exhibits a predictable race or a predictable deadlock.*

2.3 Illustrations

Going back to the example in Figure 2b let us see how WCP is able to detect a race that CP cannot. Note that the `rel(l)` in t_1 is CP-ordered before the `acq(l)` in t_2 by rule (a) of CP. Further using rule (c) of CP we obtain that the operations on variable `y` are CP-ordered and therefore CP does not detect the predictable race as uncovered by the trace e_5, e_1, e_6. WCP on the other hand does not order the `rel(l)` and `acq(l)`. Rule (a) of WCP only orders the `r(x)` in t_2 after the `rel(l)` of t_1. If we look at the example to its left in Figure 2a the same reasoning can be used to obtain that CP does not detect any race, and indeed there is no predictable race. In the case of WCP since the `r(x)` in t_2 appears above the `r(y)`, it ends up ordering `w(y)` and `r(y)` thus not declaring any race either.

The intuitive reason for formulating rule (a) in CP is that "*two conflicting events have to occur in the same order in every correctly reordered execution if it is the case that they do not constitute a race. Consequently the critical sections containing them need to be ordered in their entirety*" [42]. This intuition is correct when we know for sure that the conflicting events *do occur* in the correctly reordered execution (a correctly reordered execution need not include all the events in the given execution, but only prefixes for each thread). In the example in Figure 2b the two operations on `x` do not appear in the set of events that need to be scheduled in order to reveal a race, and hence any orderings derived from them need not be considered. But then if the `r(x)` is indeed included in some (other) reordering then the `rel(l)` to `acq(l)` ordering should be respected. Rule (a) of WCP does not enforce this ordering completely, it only makes sure that the later of the conflicting events is ordered after the earlier release. It would seem that this ordering is not sufficient to enforce lock semantics of the traces we would be interested in, but it turns out that this weaker version is sufficient to guarantee soundness.

We also weaken rule (b) in the same spirit as above. When considering critical sections over the same lock containing ordered events, instead of ordering the critical sections entirely (as CP does by ordering `rel(l)` of the earlier CS and the `acq(l)` of the latter) WCP only orders the earlier `rel(l)` before the latter `rel(l)`.

	t_1	t_2	t_3
1	`acq(l)`		
2	`sync(x)`		
3	`r(z)`		
4	`rel(l)`		
5		`sync(x)`	
6		`acq(l)`	
7		`acq(n)`	
8		`rel(n)`	
9		`rel(l)`	
10			`acq(n)`
11			`rel(n)`
12			`w(z)`

Figure 3. Example to demonstrate how weakening rule (b) is useful. CP:"No race". WCP:"Race"

In Figure 3 we show a trace in which there is a predictable race which is not detected by CP but is detected by WCP owing to weakening of rule (b). We use e_i to refer to event at line number i. Borrowing notation from [42], the event `sync(x)` is a shorthand for `acq(x) r(xVar)w(xVar) rel(x)`, where `xVar` is the unique variable associated with lock `x`. Any two `sync` events over the same lock are ordered by CP/WCP using Rule (a) (for WCP the ordering is from the `rel(x)` to the latter `r(xVar)` but for our example this is not important). Now we describe the example in Figure 3 in detail. First note that there is a predictable race between e_3 and e_{12} as revealed by the correctly reordered trace $e_1, e_2, e_{10}, e_{11}, e_3, e_{12}$. This race is not detected by CP as the conflicting events e_3 and e_{12} are CP related as follows: Firstly $e_2 \prec_{CP} e_5$. We have $e_1 \leq_{HB} e_2$ and $e_5 \leq_{HB} e_6$ (thread ordering). Applying Rule (c) of HB/CP composition on $e_1 \leq_{HB} e_2 \prec_{CP} e_5 \leq_{HB} e_6$ we get $e_1 \prec_{CP} e_6$. Next we apply Rule (b) of CP on the critical sections over lock `l` to get $e_4 \prec_{CP} e_6$. We also have $e_8 \leq_{HB} e_{10}$ (critical sections over same lock `n`). Applying Rule (c) of CP ($e_3 <_{TO} e_4 \prec_{CP} e_6 <_{TO} e_8 \leq_{HB} e_{10} <_{TO} e_{12}$) we obtain $e_3 \prec_{CP} e_{12}$. We see that this line of reasoning fails if you try to prove $e_3 \prec_{WCP} e_{12}$ because e_4 and e_6 are not $\prec_{WCP}$ related. Rule (b) of WCP would give us that the releases are ordered i.e., $e_4 \prec_{WCP} e_9$ but this cannot be composed with HB edges to get a path to e_{12}. It turns out e_3 and e_{12} are not ordered by $\prec_{WCP}$, and therefore the race that exists between them is correctly detected by WCP.

We present a more involved example in Figure 4 to once again see how a predictable race is revealed by WCP but not CP. Starting with the two `sync(x)` events that are CP related we get enclosing critical section on lock `n` to be CP related by Rule (b) i.e., $e_9 \prec_{CP} e_{11}$. We then use $e_3 \leq_{HB} e_6$ along

	t_1	t_2	t_3
1	acq(l)		
2	acq(m)		
3	rel(m)		
4	r(z)		
5	rel(l)		
6		acq(m)	
7		acq(n)	
8		sync(x)	
9		rel(n)	
10		rel(m)	
11			acq(n)
12			acq(l)
13			rel(l)
14			sync(x)
15			w(z)
16			rel(n)

Figure 4. Example trace exhibiting a predictable race that is detected by WCP but not CP

with thread orderings and Rule (c) to get $e_1 \prec_{CP} e_{12}$. Applying Rule (b) we obtain $e_5 \prec_{CP} e_{12}$ which then gives $e_4 \prec_{CP} e_{15}$ from Rule (c), hence CP declares that the two conflicting events r(z)/w(z) are not in race. But they are indeed in race as revealed by the correctly reordered trace $e_6, e_7, e_8, e_9, e_{10}, e_{11}, e_{12}, e_{13}, e_{14}, e_1, e_2, e_3, e_4, e_{15}$. Unlike CP, WCP does not order e_4 and e_{15}. This is because WCP does not order e_9 and e_{11} since the weaker Rule (b) only gives $e_9 \prec_{WCP} e_{14}$ which does not compose with the HB edge on the critical sections over lock l.

	t_1	t_2	t_3
1	acq(l)		
2	acq(m)		
3	rel(m)		
4	r(z)		
5	rel(l)		
6		acq(m)	
7		acq(n)	
8		sync(x)	
9		rel(n)	
10			acq(n)
11			acq(l)
12			rel(l)
13			sync(x)
14			w(z)
15			rel(n)
16			sync(y)
17		sync(y)	
18		rel(m)	

Figure 5. Example trace exhibiting a predictable deadlock but no predictable race

In Figure 5 we show a trace that is only slightly different from the previous example but the subtlety involved results in the absence of a predictable race but presence of a predictable deadlock. The reordered trace e_1, e_6, e_{10} exhibits the deadlock. Using identical reasoning as in the previous example we can derive that the two conflicting events r(z)/w(z) are CP ordered and WCP unordered. The proof of correctness of CP shows that it cannot detect deadlocks involving more than 2 threads as in this example. This shows that WCP can detect deadlocks that CP cannot.

3. Vector-Clock Algorithm for WCP

In this section we describe a vector clock algorithm that *implements* WCP. The algorithm assigns a "timestamp" C_e to each event e. These timestamps are vector times which can be compared to each other. The key property about the timestamps is that it preserves the WCP relation ($\leq_{WCP}$). This means in order to find out if two events are WCP ordered we can simply compare their timestamps. Therefore we also refer to these timestamps as WCP time or simply time.

3.1 Vector Clocks and Times

Let us first recall some basic notions pertaining to vector times. A vector time *VT : Tid* $\rightarrow$ *Nat*, is a function that maps each thread in a trace to a non-negative integer. It can also be viewed as $\mathcal{T}$-tuple, where $\mathcal{T}$ is the number of threads in the given trace. Vector times support comparison operation $\sqsubseteq$ for point-wise comparison, join operation ($\sqcup$) for point-wise maximum, and component assignment of the form $V[t := n]$ which assigns the time $n \in$ *Nat* to component $t \in$ *Tid* of vector time V. Vector time $\bot$ maps all threads to 0.

$$\begin{array}{rcl} V_1 \sqsubseteq V_2 & \text{iff} & \forall t : V_1(t) \leq V_2(t) \\ V_1 \sqcup V_2 & = & \lambda t : max(V_1(t), V_2(t)) \\ V[u := n] & = & \lambda t : \texttt{if } (t = u) \texttt{ then } n \texttt{ else } V(t) \\ \bot & = & \lambda t : 0 \end{array}$$

Before we describe the algorithm we would like to point out the distinction between *clocks* and *times*. Clocks are to be thought of as variables, they are place holders for times which are the values taken up by the clock. The time of a clock will change as the trace is processed. Events will be assigned different kinds of times based on the value of different clocks right after the event is processed. We use double struck font for denoting clocks (e.g., $\mathbb{C}, \mathbb{P}, \mathbb{H}, \mathbb{N}$) and normal font for denoting vector times (e.g., C, P, H, N).

3.2 Algorithm

Our algorithm works in a streaming fashion and processes the trace by looking at its events one-by-one from beginning to end. As it handles each event it updates its *state*. The state captures all the information required to assign a timestamp (a vector time) to the last event of the trace. At each step, an event is processed and the state is updated, which allows us to

compute a timestamp of that event from the updated state. For each thread t the state is going to consist of a vector clock $\mathbb{C}_t$ (among other things) that reflects the time of the last event in the thread t so far. The timestamp/time of an event e denoted by C_e is simply the value of $\mathbb{C}_{t(e)}$ just after having processed e. To ensure that the assigned times preserve WCP ordering, the algorithm needs to ensure that the time of an event a is *"communicated"* to b if there is a WCP edge from a to b. Thus, b can be assigned a time C_b such that $C_a \sqsubseteq C_b$. In order to achieve this communication, we refer to auxiliary times P_e (***WCP-predecessor time***) and H_e (***HB time***) associated with every event e. Formally $P_e = \bigsqcup\{C_{e'} \mid e' \prec_{WCP} e\}$ and $H_e = \bigsqcup\{C_{e'} \mid e' \leq_{HB} e\}$. We use vector clocks $\mathbb{P}_t$ and $\mathbb{H}_t$ in the state to record times P_e and H_e of the last events e of thread t.

In the following paragraphs we motivate different components of the state, and explain how these components are updated as part of the algorithm. The state update procedure is described in Algorithm 1. It consists of procedures that are prompted depending on the type of event being processed. Each procedure has an argument t denoting the thread that is performing the event. The argument ℓ for acquire/release events denotes the lock being operated. The argument x for read/write denotes the variable being accessed in the event. The parameter L for read/write events denotes the set of locks corresponding to the enclosing critical sections of the event. The parameters R/W for the release event correspond to the set of variables that have been read/written inside the critical section corresponding to the release in question. The purpose of these parameters will become clear as we move forward. Next we describe different components of the state along with how these procedures manipulate them.

Local Clock $\mathbb{N}_t$ ***:*** With each thread t we associate an integer counter $\mathbb{N}_t$ in the state. It represents the *local clock* of thread t. The *local time* of an event e, denoted by N_e, is the value of $\mathbb{N}_t$ after e is processed. For any t the value $\mathbb{C}_t(t)$ will always refer to $\mathbb{N}_t$.

Local Clock Increment : The local clock $\mathbb{N}_t$ is incremented just before an event of t is processed iff the previous event in t was a release. Since this increment is common to all events, we omit it from the pseudocode.

WCP clocks $\mathbb{P}_t, \mathbb{P}_\ell$ ***:*** With each thread we associate a vector clock $\mathbb{P}_t$. The *WCP-predecessor time* P_b of an event b is the value of the clock $\mathbb{P}_{t(b)}$ after b has been processed. As defined earlier, P_b represents the "knowledge" that b has about other events with respect to the $\prec_{WCP}$ relation. Note that, for any event a, if $N_a \leq P_b(t(a))$ (i.e., P_b "knows" a), then it is the case that $a \prec_{WCP} b$ (Lemma C.8 in [21]). This invariant is maintained by making sure that whenever $a \prec_{WCP} b$ then the WCP-time of a, C_a (which also has local time of a), is made known to b so that it can update $\mathbb{P}_{t(b)}$ appropriately. Sending the entire vector time C_a (rather than just N_a) to b ensures that P_b also gets to "know" the events in other threads that transitively precede it via the event a.

We also maintain, for each lock ℓ, a vector clock $\mathbb{P}_\ell$ that remembers the WCP-predecessor time P_r of the last $\texttt{rel}(\ell)$ event r seen until then (Line 9). $\mathbb{P}_\ell$ is needed so that the clock $\mathbb{P}_t$ can be maintained correctly. Consider the case when an $\texttt{acq}(\ell)$ event a by thread t is processed. Now, an event b such that $b \prec_{WCP} a$ but not $b \prec_{WCP} prev(a)$ ($prev$ refers to the previous event in the thread) has to be such that $b \prec_{WCP} r$, where r is the last $\texttt{rel}(\ell)$ event before a, in some thread other than $t(a)$. This is because WCP edges entering an acquire event from another thread are due to WCP-HB composition (Rule (c) of WCP). Now C_b is already known to P_r since $b \prec_{WCP} r$. So the event a can obtain time of events such as b through clock $\mathbb{P}_\ell$. This is achieved on Line 2 of the algorithm.

We shall come back to how $\mathbb{P}_t$ is updated on other events (release/read/write) as it deals with other components of the state which are described below. For now, note that $\mathbb{P}_t$ corresponds to the relation $\prec_{WCP}$ and $\mathbb{C}_t$ corresponds to $\leq_{WCP} = (\prec_{WCP} \cup \leq_{TO})$. Therefore $\mathbb{C}_t$ is simply obtained by incorporating thread order information obtained from the local clock $\mathbb{N}_t$ into $\mathbb{P}_t$ as: $\mathbb{C}_t = \mathbb{P}_t[t := \mathbb{N}_t]$. Since $\mathbb{C}_t$ can be derived this way from the components $\mathbb{P}_t$ and $\mathbb{N}_t$, we choose not to feature how $\mathbb{C}_t$ is updated in the algorithm.

HB clocks $\mathbb{H}_t, \mathbb{H}_\ell$ ***:*** For events a, b if $a \prec_{WCP} b$, then P_b should not only receive a's time C_a, but also the time C_c of every event c such that $c \leq_{HB} a$ (Rule (c)). To account for this, we maintain a clock $\mathbb{H}_t$ for each thread t in the state. Right after an event e is performed, the clock $\mathbb{H}_{t(e)}$ holds the value of the ***HB time*** H_e, defined earlier. So going back to $a \prec_{WCP} b$, instead of passing C_a to P_b, we pass H_a so that the times of all events c (including a) with $c \leq_{HB} a$, is received by b. Again, the clock $\mathbb{H}_t(t)$ is made to refer to $\mathbb{N}_t$ so that we do not have to specify when $\mathbb{H}_t(t)$ has to be changed.

We also maintain a vector clock $\mathbb{H}_\ell$ for each lock ℓ that stores the HB time of the last $\texttt{rel}(\ell)$ event seen till then (Line 9). Consider, once again, the case when an $\texttt{acq}(\ell)$ event a is processed. If there is any event b where $b \leq_{HB} a$ but not $b \leq_{HB} prev(a)$, then $b \leq_{HB} r$ where r is the last $\texttt{rel}(\ell)$ event before a. To ensure that b's time C_b reaches a we pass H_r to a using $\mathbb{H}_\ell$. $\mathbb{H}_\ell$ contains the time of all events b such that $b \leq_{HB} r$. This is achieved in Line 1 of the algorithm.

In order to motivate the remaining components of the state, we look at an example trace in Figure 6. We use the event $\texttt{acrl(y)}$ as a short hand for the events $\texttt{acq(y)rel(y)}$ performed in succession. This way two $\texttt{acrl(y)}$s are HB related. An edge between two events in the trace indicates a WCP order between them that can be deduced using Rules (a) or (b); the other ordering edges are omitted for clarity. As before, we will use e_i to denote the event on Line i.

Release Times $\mathbb{L}^r_{\ell,x}, \mathbb{L}^w_{\ell,x}$ ***:*** Consider the edge $e_6 \prec_{WCP} e_{17}$ derived from Rule (a). Here e_{17} should receive the HB time of e_6, and more generally, the HB time of any other $\texttt{rel}(\ell)$ event r on lock ℓ enclosing e_{17} ($e_{17} \in \ell$) such that the critical

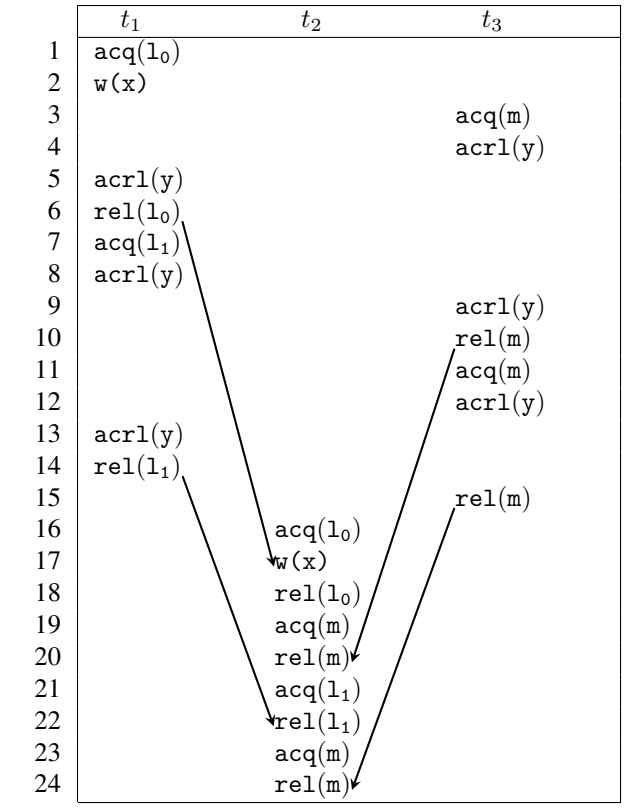

Figure 6. Example trace to motivate the Algorithm 1

section CS(r) contains an event conflicting with e_{17}. For this purpose, we maintain, for every lock ℓ and variable x, a vector clock $\mathbb{L}^r_{\ell,x}$ that records the join of the HB times of all $\texttt{rel}(\ell)$ events (seen so far) whose critical sections contain a $\texttt{r}(x)$ event. Similarly, for each lock ℓ and variable x, $\mathbb{L}^w_{\ell,x}$ maintains the join of the HB times of all the $\texttt{rel}(\ell)$ events (seen so far) whose critical sections contain a $\texttt{w}(x)$ event. Lines 7 and 8 of Algorithm 1 maintain these invariants. Hence, when e_{17} is processed it can look up the time of e_6 using $\mathbb{L}^w_{\texttt{l}_0,\texttt{x}}$. This is achieved in Line 11 of the algorithm. Similarly when a $\texttt{w}(x)$ event is encountered within a critical section of lock ℓ, the times of the relevant releases (those that contain either read or write of x) can be accessed using $\mathbb{L}^r_{\ell,x}, \mathbb{L}^w_{\ell,x}$. This is achieved in Line 12 of Algorithm 1.

FIFO Queues $Acq_\ell(t), Rel_\ell(t)$ **:** Consider the WCP edge $e_{10} \prec_{WCP} e_{20}$ introduced because of Rule (b) — the critical sections of e_{10} and e_{20} contain WCP ordered events (because $e_3 <_{TO} e_4 \leq_{HB} e_5 <_{TO} e_6 \prec_{WCP} e_{17} <_{TO} e_{20}$ giving $e_3 \prec_{WCP} e_{20}$ by Rule (c)). Note that two events in two critical sections are WCP ordered iff the acquire of the first is WCP ordered to the release of the second (this follows from thread order and Rule (c)). When processing e_{20}, if we somehow knew that $e_3 \prec_{WCP} e_{20}$ then we would need the HB time of $e_{10} = match(e_3)$ to be communicated to e_{20} because $e_{10} \prec_{WCP} e_{20}$ by Rule (b). Therefore the appropriate times of the critical section e_3, e_{10} (and possibly of other critical sections on the same lock performed before e_{20}) need to be stored in the state so they can be used for future reference by events such as e_{20}. Note that, once e_{20} receives the time of e_{10}, no future critical section over lock $\texttt{m}$, if any, in thread t_2 (for example, e_{24} in this case) is required explicitly receive the time of e_3, because a previous release in the same thread (e_{20}) has already received the time of the appropriate release event (e_{10}). Thus, as far as t_2 is concerned, once the critical section e_3, e_{10} is processed at e_{20}, it can be discarded. Therefore, for each lock ℓ and thread t (in this case $\texttt{m}$ and t_2), we need to accumulate, in a *queue*, appropriates times for critical sections on lock ℓ, to be later used when a $\texttt{rel}(\ell)$ events of thread t is encountered. With this in mind, we maintain, in the state of our algorithm, two FIFO queues $Acq_\ell(t), Rel_\ell(t)$ for every lock ℓ and every thread t. These queues will store the times of $\texttt{acq}(\ell)/\texttt{rel}(\ell)$ events (in chronological order), performed by other threads t' ($\neq t$) (Lines 3 & 10). When processing a $\texttt{rel}(\ell)$ event performed by thread t, the algorithm looks up the times of the critical section in the front of the queue and removes it from the queue if Rule (b) is applicable, and further, updates its own time (Lines 4-6 in Algorithm 1).

Initialization : The vector clocks $\mathbb{P}_t, \mathbb{P}_\ell, \mathbb{H}_\ell, \mathbb{L}^r_{\ell,x}, \mathbb{L}^w_{\ell,x}$ for any thread t, lock ℓ and variable x are initialized to $\bot$. For every thread t , the local clock $\mathbb{N}_t$ is initialized to 1 and the vector clock $\mathbb{H}_t$ is initialized to $\bot[t := \mathbb{N}_t]$. Each of the queues $Acq_\ell(t), Rel_\ell(t)$ is empty to begin with.

Algorithm 1: *Updating vector clocks on different events*

procedure `acquire`(t, ℓ)
1 $\quad \mathbb{H}_t := \mathbb{H}_t \sqcup \mathbb{H}_\ell$;
2 $\quad \mathbb{P}_t := \mathbb{P}_t \sqcup \mathbb{P}_\ell$;
3 $\quad$ **foreach** $t' \neq t$ **do** $Acq_\ell(t')$.Enque($\mathbb{C}_t$) ;

procedure `release`(t, ℓ, R, W)
4 $\quad$ **while** $Acq_\ell(t)$.Front() $\sqsubseteq \mathbb{C}_t$ **do**
5 $\quad\quad Acq_\ell(t)$.Deque();
6 $\quad\quad \mathbb{P}_t := \mathbb{P}_t \sqcup Rel_\ell(t)$.Deque();
7 $\quad$ **foreach** $x \in R$ **do** $\mathbb{L}^r_{\ell,x} := \mathbb{L}^r_{\ell,x} \sqcup \mathbb{H}_t$;
8 $\quad$ **foreach** $x \in W$ **do** $\mathbb{L}^w_{\ell,x} := \mathbb{L}^w_{\ell,x} \sqcup \mathbb{H}_t$;
9 $\quad \mathbb{H}_\ell := \mathbb{H}_t$; $\mathbb{P}_\ell := \mathbb{P}_t$;
10 $\quad$ **foreach** $t' \neq t$ **do** $Rel_\ell(t')$.Enque($\mathbb{H}_t$) ;

procedure `read`(t, x, L)
11 $\quad \mathbb{P}_t := \mathbb{P}_t \sqcup_{\ell \in L} \mathbb{L}^w_{\ell,x}$

procedure `write`(t, x, L)
12 $\quad \mathbb{P}_t := \mathbb{P}_t \sqcup_{\ell \in L} (\mathbb{L}^r_{\ell,x} \sqcup \mathbb{L}^w_{\ell,x})$

Next we state the correctness of the algorithm which states the correspondence between the ordering of timestamps assigned to events (C_e for event e) and the WCP ordering. The proof the theorem is provided in [21].

Theorem 2 (Correctness of Algorithm 1)**.** *For a trace σ and events a, b with $a <^{\sigma}_{tr} b$, we have $a \leq^{\sigma}_{WCP} b \iff C_a \sqsubseteq C_b$*

Theorem 2 tells us that two events a and b where $a <_{tr} b$ are in WCP-race exactly when C_a and C_b are incomparable. This yields an algorithm for checking all the WCP-races for the trace. For each variable x, we maintain vector clocks $\mathbb{R}_x$ and $\mathbb{W}_x$ that record the join of the C_e times of all the read and write events e on the variable x that have been seen for the prefix of the trace that is processed. On encountering a read event $e = \mathtt{r}(x)$, we check if $\mathbb{W}_x \sqsubseteq C_e$ to confirm that all earlier events conflicting with e are indeed ordered before e. If this check fails, then e is in WCP-race with some earlier conflicting write event, and thus we can declare a warning. Similarly for a write event $e = \mathtt{w}(x)$, we check if $\mathbb{R}_x \sqcup \mathbb{W}_x \sqsubseteq C_e$ holds and declare a warning otherwise. Note that our soundness theorem only guarantees that the first race pair is an actual race. But in practice we have observed that subsequent pairs that are in WCP-race also happen to be in race. Note that this methodology only gives us the second component e_2 of a pair (e_1, e_2) of events in race. In order to determine the first part, we would have to go over the trace once more and individually compare the WCP times of the events against those conflicting events appearing later that were flagged to be in race in the initial analysis.

3.3 Linear Running Time

In order to analyze the running time of Algorithm 1, we fix the following parameters for a given trace σ. Let $\mathcal{T}$ be the number of threads, $\mathcal{L}$ be the number of locks used in σ, and $\mathcal{N}$ be the total number of events in σ. We state the running time in Theorem 3 and provide the proof in [21]. Note that we assume arithmetic operations take constant time.

Theorem 3. *Given a trace σ with parameters $\mathcal{N}, \mathcal{T}$ and $\mathcal{L}$ as defined above, the total running time of the WCP vector-clock algorithm over σ is proportional to $\mathcal{N} \cdot (\mathcal{L} + \mathcal{T}^2)$.*

Note that for most applications the parameter $\mathcal{T}$ is usually small (< 25) and parameter $\mathcal{L}$ is at most a few thousand. Typically, the bottleneck for any online race-detection technique is the length of the trace $\mathcal{N}$ which can be of the order of hundreds of millions or even billions (10^8,10^9) especially for industrial scenarios. Our proposed algorithm is ***linear*** in the size of the trace $\mathcal{N}$ (unlike CP/RVPredict) and therefore truly scales to large traces without having to rely on any windowing strategy that would restrict the scope of the races detected. Our experimental evaluation (Section 4) supports this claim by showing that the algorithm scales well in practice.

3.4 Lower Bounds

Our algorithm is clearly optimal in terms of running time (in terms of the length of the trace), since one has to spend linear time just looking at the entire trace. However, it can take up linear space in the worst case owing to the queues that hold the times of the previous acquires/releases. In this section, we state lower bounds result that states that any algorithm that implements WCP in linear time takes linear space. Hence our algorithm is also optimal in terms of the space requirements.

The goal is to show that one needs linear space to recognize WCP. But we want to do this when the number of threads $\mathcal{T}$ is constant, and the number of variables $\mathcal{V}$ and locks $\mathcal{L}$ is sublinear in the size of the trace ($O(\frac{n}{logn})$). If the number of variables and locks is linear then it is easy to get a linear space lower bound, and even HB takes linear space in that case. The following theorem is proved in [21].

Theorem 4. *Any algorithm that implements WCP by doing a single pass over the trace takes $\Omega(n)$ space.*

In Theorem 5, we prove a lower bound result that applies to any algorithm (not just single pass), with details in [21].

Theorem 5. *For any algorithm that computes the WCP relation in time $T(n)$ and space $S(n)$ it is the case that $T(n)S(n) \in \Omega(n^2)$*

Therefore, our WCP algorithm is optimal in terms of time/space trade-off as well.

4. Experimental Evaluation

We implemented the vector clock algorithm (Algorithm 1) in our tool RAPID (RAce PreDiction) written in Java, available at [1]. RAPID uses the logging functionality of the tool RVPredict [35], to generate program traces which can be used for race detection. RAPID only considers events corresponding to read/write to memory locations, lock acquire/release, and thread fork/join events, and ignores other events (such as *branch* events) generated by RVPredict. RAPID also implements a vector algorithm for detecting HB races.

Algorithm 1 runs in linear time. Also, since the sizes of data structures involved in the algorithm do not grow very fast, even for very large benchmarks, the memory requirement for the algorithm did not seem to be a huge bottleneck. Hence, we did not have to split our analysis into small windows, to handle large traces in RAPID. This is in stark contrast to most predictive dynamic race detection techniques [18, 42].

Our experiments were conducted on an 8-core 2.6GHz 46-bit Intel Xeon(R) Linux machine, with HotSpot 1.8.0 64-Bit Server as the JVM and 50 GB heap space. We set a time limit of 4 hours for evaluating each of the techniques on each of the benchmarks. Our results are summarized in Table 1.

We compare the performance of RAPID only against sound techniques for race detection. For each of the techniques compared, as in [18], we attempt to analyze the following two characteristics of our algorithm:

1. *Race detection capability*, measured by the number of distinct race pairs detected. A WCP (HB) race pair is an unordered tuple of *program locations* corresponding to some pair of events in the trace that are unordered by the partial order WCP (HB). We compare the race detection capability of RAPID's WCP vector clock algorithm with

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Program	**LOC**	**#Events**	**#Thrd**	**#Locks**	**#Races**					**WCP Queue Length (%)**	**Time**			
					WCP	**HB**	**RV w=1K s=60s**	**RV w=10K s=240s**	**Max**		**WCP**	**HB**	**RV w=1K s=60s**	**RV w=10K s=240s**
account	87	130	4	3	4	4	4	4	4	0.0	0.2s	0.3s	1s	1s
airline	83	128	2	0	4	4	4	4	4	0.0	0.2s	0.2s	0.8s	2s
array	36	47	3	2	0	0	0	0	0	4.3	0.2s	0.2s	1.1s	0.8s
boundedbuffer	334	333	2	2	2	2	2	2	2	0.0	0.3s	0.2s	1s	0.8s
bubblesort	274	4K	10	2	6	6	6	0	6	2.4	0.7s	0.5s	3.6s	7m3s
bufwriter	199	11.7M	6	1	2	2	2	2	2	10	47s	22.4s	4.1s	4.5s
critical	63	55	4	0	8	8	8	8	8	0.0	0.2s	0.2s	1.7s	0.9s
mergesort	298	3K	5	3	3	3	1	2	2	1.3	0.4s	0.4s	1.1s	1.4s
pingpong	124	146	4	0	7	7	7	7	7	0.0	0.5s	0.3s	1.2s	1.3s
moldyn	2.9K	164K	3	2	44	44	2	2	2	0.0	7.1s	2.4s	1.4s	17.4s
montecarlo	2.9K	7.2M	3	3	5	5	1	1	1	0.0	23.4s	16.2s	7.1s	5.7s
raytracer	2.9K	16K	3	8	3	3	2	3	3	0.0	2.4s	1s	1s	14.7s
derby	302K	1.3M	4	1112	23	23	11	-	14	0.6	16.2s	7s	31.2s	TO
eclipse	560K	87M	14	8263	**66**	64	5	0	8	0.4	6m51s	4m18s	26.2s	15m10s
ftpserver	32K	49K	11	304	36	36	10	12	12	2.2	5.7s	2.1s	3.8s	3m
jigsaw	101K	3M	13	280	**14**	11	6	6	6	0.0	18s	11.8s	2.8s	14.7s
lusearch	410K	216M	7	118	160	160	0	0	0	0.0	10m13s	6m48s	57.3s	46.7s
xalan	180K	122M	6	2494	**18**	15	7	8	8	0.1	7m22s	4m46s	43.1s	7m11s

Table 1. Experimental results : Columns 1-2 describe the benchmarks (name and lines of source code respectively). Columns 3, 4 and 5 denote the number of events, threads and locks in the generated trace. Columns 6, 7 denote the number of distinct race pairs detected by RAPID by running WCP and HB analysis respectively. Columns 8 and 9 denote the number of races detected by RVPredict when the window sizes are respectively 1K and 10K and the solver timeouts are respectively 60 and 240 seconds. For programs for which WCP detects more races than HB, the corresponding entries in Column 6 are boldfaced. Column 10 represents the maximum number of races detected by RVPredict with all combinations of windows sizes (1K, 2K, 5K, 10K) and solver timeouts (60s, 120s, 240s). Column 11 reports the maximum value of the sum (over locks ℓ and threads t) of lengths of the queues $Acq_\ell(t)$ and $Rel_\ell(t)$ (Algorithm 1) attained at any point while performing WCP analysis on the generated trace, as a fraction of #events. Columns 12 and 13 respectively denote the time taken by RAPID for WCP and HB analysis. Columns 14 and 15 respectively denote the time taken by RVPredict for window sizes 1K and 10K, and solver timeouts 60s and 240s respectively. A '-' in Column 9 and 'TO' in Column 15 represents timeout which we set to be 4 hours.

RVPredict (version 1.8.2), which, in theory, detects at least as many races as any sound dynamic race detection technique.

2. *Scalability*, measured by the time taken to analyze the entire trace. We compare our analysis time against HB vector clock algorithm for race detection (also implemented in RAPID), since HB is the simplest sound technique, and admits a fast linear time algorithm.

As stated earlier, WCP detects all the races detected using CP. Further, it is not clear if an algorithm based on CP relation can scale without windowing strategy. Therefore, we omit any comparison with CP [42].

We run all the three techniques (HB, WCP, RVPredict) on the same set of traces. This was possible because both RVPredict and RAPID can analyze a logged trace produced by RVPredict's logging feature. We have tried to ensure that the comparison is fair; we implement the linear time vector clock algorithm for detecting HB [26], and do not restrict the HB analysis to small windows, unlike in [18] and [42].

RVPredict supports tuning of parameters like window sizes and timeout for its backend SMT solver. The tight interplay between window sizes and solver timeout in RVPredict makes it difficult to estimate the best combination of these parameters. Small windows result in a low number of reported races because every occurrence of most of the races occur across multiple windows. On the other hand, a large window implies that the logical formulae generated in RVPredict are too large to be solved for the SMT solver, within the timeout, as a result of which, most of the windows do not report any races. In Figure 7, we depict how the number of races vary for different values of solver timeouts and window sizes, for three benchmark examples, and as can be seen, there is no clear pattern. We run RVPredict on each of the benchmarks with several parameter combinations; we vary window sizes as (1K, 2K, 5K, 10K) and solver timeout values as (60s, 120s, 240s). An attempt of testing RVPredict beyond these parameter values often led to very large running times or excessive memory requirements. In Table 1, we report the observations only for two of these combinations.

4.1 Benchmarks

Our evaluation benchmarks (Column 1) have primarily been derived from [18]. The benchmarks are designed for a comprehensive performance evaluation : the lines of code range from 60 to 0.5M, and the number of events vary from an

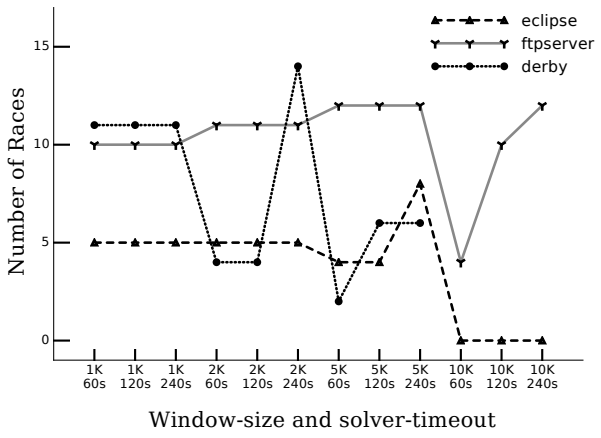

Figure 7. Number of races detected by RVPredict for different values of window size and solver timeout parameters

order of 10 to 200M. The first set of small-sized benchmarks (account to pingpong) and are originally derived from IBM Contest benchmark suite [11]. The second set of medium sized benchmarks are derived from the Java Grande Forum benchmark suite [43]. The third set of benchmarks come from large real world software - Apache FTPServer, W3C Jigsaw web server, Apache Derby, and some applications from the DaCaPo benchmark suite (version 9.12) [3].

4.2 Scalability

Columns 12-15 report the times taken by WCP, HB and RVPredict. WCP analysis times are comparable to HB analysis for all the examples.

For the small set of examples, all the three techniques finish their analysis in a reasonably small time, roughly proportional to the size of the traces. However, for large examples, both WCP and HB outperform the running times of RVPredict. For 'derby', RVPredict exceeds the time limit of 4 hours for large window sizes. In general, it is difficult to gauge the running times of RVPredict from the length of the trace and window size, because the actual size of the internal logical formulation generated by RVPredict for every window, and the running times of its backend SMT solver crucially depend on how complex these windows are.

We highlight that the worst case linear space complexity of Algorithm 1 was not observed in our experiments. In Column 11, we report the maximum value of the total lengths of the FIFO queues (Section 3), attained at any point while performing WCP analysis on the traces, as a percentage of the number of events. As can be seen, this fraction stays below 3% for almost all examples, and is 10% for 'bufwriter' benchmark.

4.3 Bug Detection Capability

Columns 6 and 7 report the number of distinct HB and WCP race pairs. Columns 8-10 report the number of race pairs detected by RVPredict, with different parameters. The extra races discovered by WCP and not by HB (boldfaced entries in Column 6) were either found to be RVPredict races, or were manually inspected to be valid race pairs.

For the smaller benchmarks, when the number of events is relatively small, the number of race pairs detected by the three techniques is almost the same, with WCP and HB detecting the maximum number of races for each of these benchmark examples, despite the fact that RVPredict employs a theoretically more comprehensive technique than WCP.

For the larger benchmark examples, the number of races detected by RVPredict are much lower than those predicted by WCP or HB. For the 'derby' benchmark, RVPredict could not finish its analysis within 4 hours. In fact, for the benchmark 'bubblesort', RVPredict runs out of memory (50GB) for a window size of 5K. This is primarily because of the tight interplay between the window sizes and timeout parameter for the backend SMT solver. We conjecture that most of the races that are not reported by RVPredict either occur across windows, or are missed because the solver did not return with an answer within the specified time limit. On the other hand, the theoretical guarantee provided by the techniques used in RVPredict [18] would have ideally resulted in it detecting possibly more data races than both HB and WCP. This indicates that a windowing strategy for analyzing large traces can potentially result in significant loss in the bug detection capability of any dynamic race detection technique. In fact, on careful analysis of the predicted races, we found that both HB and WCP detect races having a *distance* of millions of events — the *distance* of a race between program locations (pc_1, pc_2) is the minimum separation (in terms of the number of events in the trace) between any pair of events (e_1, e_2) exhibiting the (pc_1, pc_2) race. Specifically, both HB and WCP expose more than 25 races in 'eclipse' having a distance of at least 4.8 million events, with the maximum distance being 53 million. Clearly, any windowing based analysis will be incapable of catching these races.

In all the examples, the *set* of races detected by HB are a subset of the *set* of races detected by WCP. This is expected as WCP is a weakening of CP (and hence of HB). For the large benchmarks 'eclipse', 'jigsaw' and 'xalan', the number of races detected by WCP are more than those detected by HB. In all examples, the *set* of races detected by RVPredict are a subset of the races reported by WCP and HB, except in 'mergesort' and 'ftpserver', where RVPredict reports one extra race each.

Note that, our WCP based race detection algorithm does not report drastically more races than the simpler HB based algorithm. While this is surprising, given the optimistic comparison of CP and RVPredict versus HB, as reported in [18, 42], the apparent disparity can be explained by the fact that both [42] and [18] compare their techniques against a windowed implementation of HB based vector clock algorithm, which can potentially miss HB races between pairs of events (unordered by HB) that occur far apart in the

trace, and thus, possibly missing out on program location pairs corresponding to these event pairs. As stated before, our implementation of HB vector clock algorithm is not *windowed* and catches these far-away event pairs unordered by HB. That being said, the few extra races missed by (our implementation of) HB, but predicted by WCP and/or RVPredict, are quite subtle.

5. Related Work

Our work generalizes the causally-precedes (CP) relation proposed by Smaragdakis et. al [42]. WCP is a weaker relation than CP (any data race detected by CP will also be detected by WCP) and can be implemented using a linear time vector clock algorithm. WCP based race detection can be viewed as belonging to the class of predictive analysis techniques, similar to CP [42], Said et. al [36] RVPredict [18], IPA [24], [47], which essentially reason about correct reorderings of a given trace for estimating concurrency errors in other possible executions. RVPredict [18] and Rosu et. al., [40] both show how maximal and sound causal models can be used to resolve concurrency bugs. These methods explore all possible interleavings that can be deduced from the given trace, but such complete explorations are known to be intractable. RVPredict [35] has also been extended for analyzing traces with missing events [16]. Predictive analysis techniques have also been used for checking atomicity violations and synchronization errors in tools like jPredictor [6], [39] and TAME [17]. GPredict [19] uses predictive analysis for resolving higher level concurrency issues like authentication-before-use in Java. Our experimental evaluation suggests that heavy weight predictive techniques such as SMT based search do not scale well in practical scenarios, and often forego predictive power for scalability.

Lockset based techniques such as ERASER [37], which assign sets of locks to program locations and variable accesses are known to be unsound. Methods such as random testing [38] and static escape analysis [45] aim to improve the efficiency of, and reduce the number of false alarms raised by lockset based analysis.

Other dynamic race detection techniques use Lamport's happens-before (HB) relation. HB admits a linear time vector clock algorithm [26], and is adopted by various techniques including [8, 14, 30]. The DJIT$^+$ [30] algorithm uses the epoch optimization for performance improvement in the traditional vector clock algorithm. This was further enhanced by FASTTRACK [14]. Both WCP and CP are weaker relations than HB, and thus, in principle, detect all the races that any happens-before based race detection algorithm.

Other techniques include combinations of HB and lockset approach [9, 29, 41, 50], statistical techniques [4, 25], and crowd-sourced inference [20]. Tools such as ROADRUNNER [15] and Sofya [22] provide frameworks for implementing dynamic analysis tools. Techniques such as [7, 12, 33, 44, 49] leverage structured parallelism to optimize memory overhead for dynamic race detection.

Static race detection techniques [10, 28, 31, 32, 46, 51] suffer from the undecidability problem, and raise many false alarms, but still remain popular amongst developers. Type systems for detecting concurrency errors [2, 5, 13], aim to help programmers write safer programs.

Model checking techniques like [27] and [48] aim to exhaustively explore all possible concurrent executions to detect data races. However, due to the state explosion problem, explicit state model checking encounters a huge slowdown.

6. Conclusion

In this paper, we presented a sound technique for detecting data races through a new partial order called Weak-Causally-Precedes (WCP). Since WCP is a weakening of CP, it provably detects more races than CP. We showed how WCP races can be detected in linear time using a vector clock algorithm and we proved its optimality. We implemented our techniques in a prototype tool RAPID which shows promising results when evaluated on industrial sized benchmarks.

There are several avenues for future work. These include use of epoch based optimizations for improving memory requirements of the implementation, further weakening of the WCP relation while preserving soundness, and incorporating control flow information for enhanced race detection capability.

Acknowledgments

We thank Grigore Rosu and Yilong Li for their help with setting up RVPredict, and Jeff Huang for providing many of the benchmark examples. We gratefully acknowledge the support of the following grants — Dileep Kini was partially supported by NSF TWC 1314485; Umang Mathur was partially supported by NSF CSR 1422798; and Mahesh Viswanathan was partially supported by NSF CPS 1329991 and AFOSR FA9950-15-1-0059.

References

[1] `https://publish.illinois.edu/race-prediction/`.

[2] M. Abadi, C. Flanagan, and S. N. Freund. Types for Safe Locking: Static Race Detection for Java. *ACM Trans. Program. Lang. Syst.*, 28(2):207–255, Mar. 2006.

[3] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. *SIGPLAN Not.*, 41(10):169–190, Oct. 2006.

[4] M. D. Bond, K. E. Coons, and K. S. McKinley. PACER: Proportional Detection of Data Races. *SIGPLAN Not.*, 45(6): 255–268, June 2010.

[5] C. Boyapati, R. Lee, and M. Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. *SIGPLAN Not.*, 37(11):211–230, Nov. 2002.

[6] F. Chen, T. F. Şerbănuţă, and G. Roşu. jPredictor: a predictive runtime analysis tool for Java. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pages 221–230, New York, NY, USA, 2008. ACM.

[7] G.-I. Cheng, M. Feng, C. E. Leiserson, K. H. Randall, and A. F. Stark. Detecting Data Races in Cilk Programs That Use Locks. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '98, pages 298–309, New York, NY, USA, 1998. ACM.

[8] M. Christiaens and K. D. Bosschere. TRaDe: Data Race Detection for Java. In *Proceedings of the International Conference on Computational Science-Part II*, ICCS '01, pages 761–770, London, UK, UK, 2001. Springer-Verlag.

[9] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: A Race and Transaction-aware Java Runtime. *SIGPLAN Not.*, 42(6):245–255, June 2007.

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

[11] E. Farchi, Y. Nir, and S. Ur. Concurrent Bug Patterns and How to Test Them. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, IPDPS '03, pages 286.2–, Washington, DC, USA, 2003. IEEE Computer Society.

[12] M. Feng and C. E. Leiserson. Efficient Detection of Determinacy Races in Cilk Programs. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '97, pages 1–11, New York, NY, USA, 1997. ACM.

[13] C. Flanagan and S. N. Freund. Type-based Race Detection for Java. *SIGPLAN Not.*, 35(5):219–232, May 2000.

[14] C. Flanagan and S. N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. *SIGPLAN Not.*, 44(6):121–133, June 2009.

[15] C. Flanagan and S. N. Freund. The RoadRunner Dynamic Analysis Framework for Concurrent Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '10, pages 1–8, New York, NY, USA, 2010. ACM.

[16] J. Huang and A. K. Rajagopalan. Precise and Maximal Race Detection from Incomplete Traces. *SIGPLAN Not.*, 51(10): 462–476, Oct. 2016.

[17] J. Huang and L. Rauchwerger. Finding Schedule-sensitive Branches. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 439–449, New York, NY, USA, 2015. ACM.

[18] J. Huang, P. O. Meredith, and G. Rosu. Maximal Sound Predictive Race Detection with Control Flow Abstraction. *SIGPLAN Not.*, 49(6):337–348, June 2014.

[19] J. Huang, Q. Luo, and G. Rosu. GPredict: Generic Predictive Concurrency Analysis. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 847–857, Piscataway, NJ, USA, 2015. IEEE Press.

[20] B. Kasikci, C. Zamfir, and G. Candea. RaceMob: Crowdsourced Data Race Detection. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 406–422, New York, NY, USA, 2013. ACM.

[21] D. Kini, U. Mathur, and M. Viswanathan. Dynamic Race Prediction in Linear Time. *CoRR*, abs/1704.02432, 2017. URL `http://arxiv.org/abs/1704.02432`.

[22] A. Kinneer, M. B. Dwyer, and G. Rothermel. Sofya: Supporting Rapid Development of Dynamic Program Analyses for Java. In *Companion to the Proceedings of the 29th International Conference on Software Engineering*, ICSE COMPANION '07, pages 51–52, Washington, DC, USA, 2007. IEEE Computer Society.

[23] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, July 1978.

[24] P. Liu, O. Tripp, and X. Zhang. Ipa: Improving predictive analysis with pointer analysis. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 59–69, New York, NY, USA, 2016. ACM.

[25] D. Marino, M. Musuvathi, and S. Narayanasamy. LiteRace: Effective Sampling for Lightweight Data-race Detection. *SIGPLAN Not.*, 44(6):134–143, June 2009.

[26] F. Mattern. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1988.

[27] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 267–280, Berkeley, CA, USA, 2008. USENIX Association.

[28] M. Naik, A. Aiken, and J. Whaley. Effective Static Race Detection for Java. *SIGPLAN Not.*, 41(6):308–319, June 2006.

[29] R. O'Callahan and J.-D. Choi. Hybrid Dynamic Data Race Detection. *SIGPLAN Not.*, 38(10):167–178, June 2003.

[30] E. Pozniansky and A. Schuster. Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs. *SIGPLAN Not.*, 38 (10):179–190, June 2003.

[31] P. Pratikakis, J. S. Foster, and M. Hicks. LOCKSMITH: Practical Static Race Detection for C. *ACM Trans. Program. Lang. Syst.*, 33(1):3:1–3:55, Jan. 2011.

[32] C. Radoi and D. Dig. Practical Static Race Detection for Java Parallel Loops. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 178–190, New York, NY, USA, 2013. ACM.

[33] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav. Scalable and Precise Dynamic Datarace Detection for Structured Parallelism. *SIGPLAN Not.*, 47(6):531–542, June 2012.

[34] J. Roemer and B. M. D. An Online Dynamic Analysis for Sound Predictive Data Race Detection. Technical Report OSU-CISRC-11/16-TR05, 2016. URL `http://web.cse.ohio-state.edu/~bond.213/raptor-tr.pdf`.

[35] G. Rosu. RV-Predict, Runtime Verification. `https://runtimeverification.com/predict/`. Accessed: 2016-

11-15.

[36] M. Said, C. Wang, Z. Yang, and K. Sakallah. Generating Data Race Witnesses by an SMT-based Analysis. In *Proceedings of the Third International Conference on NASA Formal Methods*, NFM'11, pages 313–327, Berlin, Heidelberg, 2011. Springer-Verlag.

[37] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multi-threaded Programs. *SIGOPS Oper. Syst. Rev.*, 31(5):27–37, Oct. 1997.

[38] K. Sen. Race Directed Random Testing of Concurrent Programs. *SIGPLAN Not.*, 43(6):11–21, June 2008.

[39] K. Sen, G. Roşu, and G. Agha. Detecting Errors in Multithreaded Programs by Generalized Predictive Analysis of Executions. In *Proceedings of the 7th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems*, FMOODS'05, pages 211–226, Berlin, Heidelberg, 2005. Springer-Verlag.

[40] T. F. Şerbănuţă, F. Chen, and G. Roşu. Maximal causal models for sequentially consistent systems. In *International Conference on Runtime Verification*, pages 136–150. Springer, 2012.

[41] K. Serebryany and T. Iskhodzhanov. ThreadSanitizer: Data Race Detection in Practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, pages 62–71, New York, NY, USA, 2009. ACM.

[42] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan. Sound Predictive Race Detection in Polynomial Time. *SIGPLAN Not.*, 47(1):387–400, Jan. 2012.

[43] L. A. Smith and J. M. Bull. A multithreaded java grande benchmark suite. In *Proceedings of the third workshop on Java for high performance computing*, 2001.

[44] R. Surendran and V. Sarkar. Dynamic determinacy race detection for task parallelism with futures. In *International Conference on Runtime Verification*, pages 368–385. Springer, 2016.

[45] C. von Praun and T. R. Gross. Object Race Detection. *SIGPLAN Not.*, 36(11):70–82, Oct. 2001.

[46] J. W. Voung, R. Jhala, and S. Lerner. RELAY: Static Race Detection on Millions of Lines of Code. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 205–214, New York, NY, USA, 2007. ACM.

[47] C. Wang, S. Kundu, M. Ganai, and A. Gupta. Symbolic predictive analysis for concurrent programs. In *Proceedings of the 2Nd World Congress on Formal Methods*, FM '09, pages 256–272, Berlin, Heidelberg, 2009. Springer-Verlag.

[48] E. Yahav. Verifying Safety Properties of Concurrent Java Programs Using 3-valued Logic. *SIGPLAN Not.*, 36(3):27–40, Jan. 2001.

[49] A. Yoga, S. Nagarakatte, and A. Gupta. Parallel Data Race Detection for Task Parallel Programs with Locks. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 833–845, New York, NY, USA, 2016. ACM.

[50] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. *SIGOPS Oper. Syst. Rev.*, 39(5):221–234, Oct. 2005.

[51] S. Zhan and J. Huang. ECHO: Instantaneous in Situ Race Detection in the IDE. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 775–786, New York, NY, USA, 2016. ACM.