High-Coverage, Unbounded Sound Predictive Race Detection*

Jake Roemer
Ohio State University, USA
roemer.37@osu.edu

Kaan Genç
Ohio State University, USA
genc.5@osu.edu

Michael D. Bond
Ohio State University, USA
mikebond@cse.ohio-state.edu

Extended technical report version of PLDI 2018 paper (adds Appendices B and C)

Abstract

Dynamic program analysis can *predict* data races knowable from an observed execution, but existing predictive analyses either miss races or cannot analyze full program executions. This paper presents *Vindicator*, a novel, sound (no false races) predictive approach that finds more data races than existing predictive approaches. Vindicator achieves high coverage by using a new, efficient analysis that finds all possible predictable races but may detect false races. Vindicator ensures soundness using a novel algorithm that checks each potential race to determine whether it is a true predictable race. An evaluation using large Java programs shows that Vindicator finds hard-to-detect predictable races that existing sound predictive analyses miss, at a comparable performance cost.

CCS Concepts • **Software and its engineering → Dynamic analysis**; *Software testing and debugging*;

Keywords Data race detection, dynamic predictive analysis

ACM Reference Format:
Jake Roemer, Kaan Genç, and Michael D. Bond. 2018. High-Coverage, Unbounded Sound Predictive Race Detection. In *Proceedings of 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18).* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3192366.3192385

1 Introduction

As parallel software becomes increasingly pervasive, *data races* represent a growing threat to system reliability. A shared-memory program has a data race if it is possible for two *conflicting* memory accesses (accesses, at least one of which is a write, to the same memory location by different threads) to execute *consecutively* (with no interleaving program operations). Data races lead to fatal crashes, data corruption, and other errors nondeterministically [9, 17, 19, 31, 41, 43, 48, 51, 63, 67, 77, 82, 88]. Modern shared-memory programming languages including C++ and Java provide undefined or ill-defined semantics for executions with data races [2, 10–13, 54, 82].

This paper focuses on detecting data races using *dynamic program analysis*, which observes a single execution's memory accesses and synchronization operations. Dynamic analysis can be *sound*, meaning that it reports only true races.[1] Soundness is an essential property because each reported race—whether true or false—takes substantial time for developers to investigate [4, 30, 35, 55, 63].

The most prevalent dynamic analysis for detecting data races is *happens-before (HB) analysis*, which soundly detects conflicting accesses unordered by the *HB relation*, a partial order that is the union of program and synchronization order [30, 46, 69, 71]. HB analysis does *not* detect all data races that are *predictable:* data races that, based on the observed execution alone, can definitely occur in *some* execution.

Sound predictive analysis detects more races than HB without reporting false races [20, 38, 39, 44, 50, 74, 79, 86]. Most existing predictive analyses cannot scale beyond analyzing bounded windows of execution, thus missing predictable races whose accesses are "far apart" in an observed execution (Section 7). An outlier is Kini et al.'s *weak-causally-precedes (WCP) analysis*, which computes the *WCP relation* efficiently for full program executions [44]. However, WCP analysis inherently misses predictable races.

Our approach. This paper introduces an approach called *Vindicator* that soundly predicts more races than WCP analysis and scales to full program executions. Vindicator consists of two novel components: (1) *doesn't-commute (DC) analysis*, an unsound analysis that detects *DC-races*, which include all

*This material is based upon work supported by the National Science Foundation under Grants CAREER-1253703, CCF-1421612, and XPS-1629126.

PLDI'18, June 18–22, 2018, Philadelphia, PA, USA

ACM ISBN 978-1-4503-5698-5/18/06…$15.00
https://doi.org/10.1145/3192366.3192385

[1] In this paper, an analysis is *sound* if it reports no false races, which follows the predictive data race detection literature (e.g., [38, 44, 86]).

predictable races[2] but may also include false races, and (2) *VindicateRace*, an algorithm that analyzes every DC-race to determine whether it is a true predictable race.

In our experiments, an implementation of Vindicator finds all predictable races in full executions of large Java programs. Vindicator finds hard-to-detect races and several statically distinct predictable races that WCP analysis cannot find, with comparable run-time overhead. Vindicator thus advances the state of the art in sound predictive race detection.

2 Background and Motivation

This section defines an execution trace and the properties that must hold to correctly reorder a trace, as well as other relevant definitions. We then describe predictive relations from prior work and show their limitations.

2.1 Execution Trace

An execution trace tr is a totally ordered list of events; tr represents a multithreaded execution without loss of generality.[3] We say $e <_{tr} e'$ if e occurs before e' in tr, and $e \leq_{tr} e'$ if $e <_{tr} e' \vee e = e'$. An event e is one of wr(x), rd(x), acq(m), or rel(m), where x is a variable and m is a lock.

We define a helper function $thr(e)$ that returns the thread identifier that executed event e. Function $A(r)$ returns the acquire event that starts the critical section ended by release event r, and $R(a)$ returns the release event that ends the critical section started by acquire event a. Function $CS(r)$ returns the set of events in the critical section ended by release event r, including r and $A(r)$. That is, $CS(r) \equiv \{e \mid thr(e) = thr(r) \wedge A(r) \leq_{tr} e \leq_{tr} r\}$.

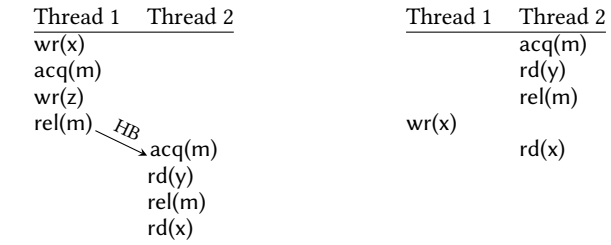
Events e and e' are *conflicting*, denoted $e \asymp e'$, if one is a write event and the other is a read or write event to the same variable and $thr(e) \neq thr(e')$.

Program-order (PO) is a relation that orders events executed by the same thread. For two events e and e', $e \prec_{\text{PO}} e'$ if $e <_{tr} e' \wedge thr(e) = thr(e')$.

Examples. Figures 1(a) and 2(a) show example execution traces, with $<_{tr}$ order from top to bottom and different threads in different columns. (The reader can ignore the arrows and Figures 1(b) and 2(b) for now.)

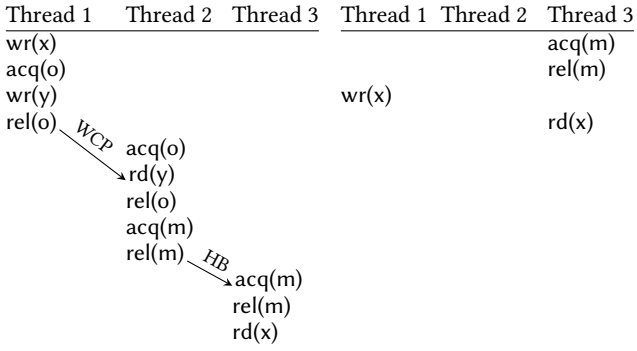
2.2 Reordered Execution Trace

Sound predictive race detection analyzes a trace tr and detects data races that occur in some other, unobserved execution. A *correctly reordered* trace tr' is any trace that must be feasible because the observed trace tr executed. Reasoning about feasible reordered traces can be tricky. Figures 1(b) and 2(b) show correct reorderings of Figures 1(a) and 2(a), respectively. However, suppose in Figure 1(a) we change Thread 1's wr(z) to wr(y). Then Figure 1(b) would no longer be a correct reordering because, in the original static program, *whether rd(x) occurs* may depend on the value that rd(y) reads. In addition to guaranteeing that read events see the same written values, a correctly reordered trace must respect program order and lock semantics:



(a) HB analysis detects no race **(b)** Correct reordering of (a)

Figure 1. The example execution in (a) has no HB-race (i.e., $\text{wr(x)} \prec_{\text{HB}} \text{rd(x)}$), but it has a predictable race, as the reordered execution in (b) demonstrates.



(a) WCP analysis detects no race **(b)** Correct reordering of (a)

Figure 2. The example execution in (a) has no WCP-race (i.e., $\text{wr(x)} \prec_{\text{WCP}} \text{rd(x)}$), but it has a predictable race, as the reordered execution in (b) demonstrates.

Definition 2.1 (Correct reordering)**.** A trace tr' is a correct reordering of tr if tr' contains only events in tr and the following properties hold:

Program-order (PO) rule: Two PO-ordered events must execute in the same order in tr' as in tr unless the second event is not in tr'. That is, $e \prec_{\text{PO}} e' \implies (e <_{tr'} e' \vee e' \notin tr')$.

Conflicting accesses (CA) rule: Two conflicting events must execute in the same order in tr' as in tr unless the second event is not in tr'. That is, $(e <_{tr} e' \wedge e \asymp e') \implies (e <_{tr'} e' \vee e' \notin tr')$.

Lock semantics (LS) rule: Critical sections on the same lock cannot overlap. That is, if a_1 and a_2 are acquire events on the same lock, $a_1 <_{tr'} a_2 \implies R(a_1) <_{tr'} a_2$.

[2] A caveat is that our definition of predictable race is slightly more strict than needed (Section 2.2).

[3] The total order $<_{tr}$ is a linearization of events in a sequentially consistent (SC) execution. We can safely assume SC, at least until the first race, because language memory models typically ensure SC for data-race-free executions [2, 11, 54].

Note that the CA rule, which prohibits reordering all pairs of conflicting accesses, is overly strict: it disallows some reordered traces in which every read has the same last write as in tr. For example, given rd(x) $\prec_{tr}$ wr(x), the CA rule disallows a tr' that contains wr(x) but not rd(x), although such a reordered trace is not necessarily invalid. However, we do not know how to encode a less-restrictive CA rule in a partial order such as this paper's DC relation (or prior work's CP and WCP relations [44, 86]), e.g., how to to encode that tr' can reorder a read–write conflict *only if* the *read* is not in the reordered execution.

Note also that the PO and CA rules constrain reordering based on ordering in tr, while the LS rule constrains tr' directly. This distinction is relevant to this paper's approach.

Definition 2.2 (Predictable race)**.** An execution trace tr has a *predictable race* if it has two conflicting events e_1 and e_2 ($e_1 \asymp e_2$) that in some correctly reordered trace tr' are *consecutive* ($e_1 \prec_{tr'} e_2 \wedge \nexists e \mid e_1 \prec_{tr'} e \prec_{tr'} e_2$).

Figures 1(b) and 2(b) demonstrate that Figures 1(a) and 2(a), respectively, each have a predictable race.

Definition 2.3 (Soundness)**.** A relation, analysis, or approach is *sound* if it finds no race for every execution trace tr that has no predictable race.

Definition 2.4 (Completeness)**.** A relation, analysis, or approach is *complete* if it finds a race for every execution trace tr that has a predictable race.

Note that completeness means detecting all predictable races knowable from an observed execution trace, not all of a program's data races.

These definitions of soundness and completeness, which follow the predictive data race detection literature (e.g., [38, 44, 86]), are swapped compared with most work on static and dynamic (non-predictive) race detection (cf. Section 7).

2.3 Sound Predictive Relations

Prior work introduces relations on events that are *predictive* because unordered conflicting accesses indicate a predictable race. The following presentation of predictive relations is similar to prior work's [44, 86].

Definition 2.5 (Happens-before)**.** Given a trace tr, $\prec_{\text{HB}}$ is the smallest relation that satisfies the following properties:

- Two events are ordered by HB if they are ordered by PO. That is, $e \prec_{\text{HB}} e'$ if $e \prec_{\text{PO}} e'$.
- Release and acquire events on the same lock are ordered by HB (a.k.a. synchronization order). That is, $r \prec_{\text{HB}} a$ if r and a are release and acquire events, respectively, on the same lock and $r \prec_{tr} a$.
- HB is transitively closed. That is, $e \prec_{\text{HB}} e'$ if $\exists e'' \mid e \prec_{\text{HB}} e'' \wedge e'' \prec_{\text{HB}} e'$.

An execution trace has an *HB-race* if it has two conflicting events that are unordered by the strict partial order HB.

Although the literature usually does not classify HB-race detection as "predictive," we (and others [44]) consider HB to be predictive because it predicts *consecutive* conflicting events in some reordered trace.

HB is incomplete: it misses predictable races. Figure 1(a)'s execution has no HB-races (wr(x) $\prec_{\text{HB}}$ rd(x)), but the execution has a predictable race (as Figure 1(b) demonstrates).

Kini et al.'s *weak-causally-precedes* (WCP) relation is weaker than HB and thus predicts more races than HB. (WCP is likewise weaker than prior work's *causally-precedes* (CP) relation [73, 86] and thus predicts more races than CP.)

Definition 2.6 (Weak-causally-precedes)**.** Given a trace tr, $\prec_{\text{WCP}}$ is the smallest relation that satisfies the following properties:

(a) If two critical sections on the same lock contain conflicting events, then the first critical section is ordered by WCP to the second conflicting event. That is, $r_1 \prec_{\text{WCP}} e_2$ if r_1 and r_2 are release events on the same lock, $r_1 \prec_{tr} r_2$, $e_1 \in CS(r_1)$, $e_2 \in CS(r_2)$, and $e_1 \asymp e_2$.

(b) Release events on the same lock are ordered by WCP if their critical sections contain WCP-ordered events. Because of the next rule, this rule can be expressed simply as follows: $r_1 \prec_{\text{WCP}} r_2$ if r_1 and r_2 are release events on the same lock and $A(r_1) \prec_{\text{WCP}} r_2$.

(c) WCP is closed under left and right composition with HB. That is, $e \prec_{\text{WCP}} e'$ if $\exists e'' \mid e \prec_{\text{HB}} e'' \prec_{\text{WCP}} e' \vee e \prec_{\text{WCP}} e'' \prec_{\text{HB}} e'$.

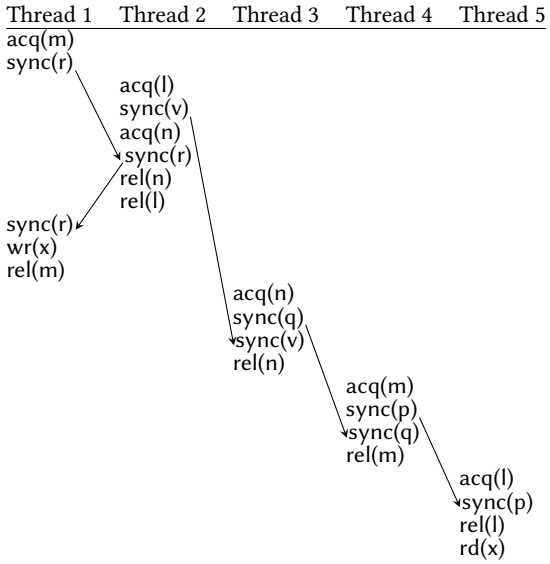
An execution trace has a *WCP-race* if it has two conflicting events unordered by the strict partial order $\prec_{\text{WCP}} \cup \prec_{\text{PO}}$. The execution in Figure 1(a) has a WCP-race on wr(x) and rd(x).

WCP is the weakest known partial order that is also sound.[4] Furthermore, WCP can be computed with a dynamic analysis that scales to whole execution traces [44]. However, WCP is incomplete. Figures 2(a) and 3(a) each show an execution that has no WCP-race but has a predictable race, as demonstrated by Figures 2(b) and 3(b), respectively.[5] Intuitively, WCP is incomplete because it composes with *synchronization order* (i.e., ordering between critical sections on the same lock; Definition 2.5). For example, in Figure 2(a), WCP orders Thread 1's rel(o) before Thread 3's acq(m), but these operations can be reordered in a correctly reordered execution, as Figure 2(b) shows.

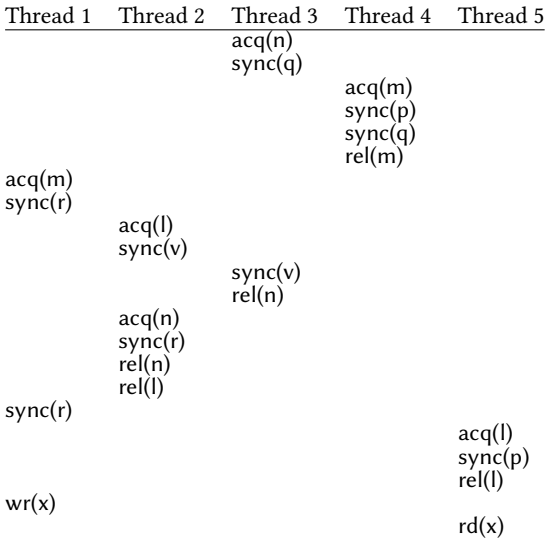
WCP's incompleteness leads to missing predictable races not only in theory but also in practice. This paper's goal is

[4] Technically, an execution with a WCP-race has a predictable race or a predictable deadlock [44].

[5] Although the arrows in Figure 3(a) show DC ordering (described later), for this execution DC ordering is the same as WCP ordering established by WCP rules (a) and (b). In the execution, wr(x) $\prec_{\text{WCP}}$ rd(x) because WCP composes with HB, which the figure does not show.



(a) Example execution. The arrows represent DC ordering (excluding PO ordering).



(b) Correctly reordered trace of (a) demonstrating a predictable race.

Figure 3. The execution in (a) has a predictable race and a DC-race (wr(x) $\not\prec_{\text{DC}}$ rd(x)) but no WCP-race (wr(x) $\prec_{\text{WCP}}$ rd(x)). sync(o) is an abbreviation for the sequence acq(o); rd(oVar); wr(oVar); rel(o).

to report the predictable races that WCP misses by detecting all predictable races in unbounded executions.

3 Vindicator Overview

This paper presents a novel approach called *Vindicator* for detecting all predictable races from an observed program execution. Vindicator consists of two main components, *doesn't-commute (DC) analysis* and the VindicateRace algorithm.

Section 4 introduces DC analysis, which tracks DC, a complete but unsound predictive relation. DC achieves completeness by forgoing WCP's composition with HB, effectively avoiding artificial constraints on the order of critical sections in a reordered trace. DC analysis identifies *DC-races*, which are *potential* predictable races (i.e., may be false races), and computes a *constraint graph* of executed events; the graph's transitive closure corresponds to DC ordering.

Section 5 presents VindicateRace, an algorithm for determining whether a DC-race is a true predictable race. The algorithm takes as input a single DC-race and constraint graph. VindicateRace adds additional constraints that are necessary for reordering critical sections to expose a potential predictable race. VindicateRace ensures soundness by constructing, for each reported predictable race, a correctly reordered trace that executes the conflicting accesses consecutively.

4 Doesn't-Commute Relation and Analysis

Doesn't-commute (DC) is a new relation that is unsound and complete, detecting all predictable races but also potential false races. Although DC is unsound, it detects few, if any, false races in practice.

Definition 4.1 (Doesn't-commute)**.** Given a trace tr, $\prec_{\text{DC}}$ is the smallest relation that satisfies the following properties:

(a) If two critical sections on the same lock contain conflicting events, then the first critical section is ordered by DC to the second conflicting event. That is, $r_1 \prec_{\text{DC}} e_2$ if r_1 and r_2 are release events on the same lock, $r_1 \prec_{tr} r_2$, $e_1 \in CS(r_1)$, $e_2 \in CS(r_2)$, and $e_1 \asymp e_2$.

(b) Release events on the same lock are ordered by DC if their critical sections contain DC-ordered events. Because of the next two rules, this rule can be expressed simply as follows: $r_1 \prec_{\text{DC}} r_2$ if r_1 and r_2 are release events on the same lock and $A(r_1) \prec_{\text{DC}} r_2$.

(c) Two events are ordered by DC if they are ordered by PO. That is, $e \prec_{\text{DC}} e'$ if $e \prec_{\text{PO}} e'$.

(d) DC is transitively closed. That is, $e \prec_{\text{DC}} e'$ if $\exists e'' \mid e \prec_{\text{DC}} e'' \wedge e'' \prec_{\text{DC}} e'$.

Note that DC's rules (a) and (b) are identical to WCP's rules (a) and (b), but with $\prec_{\text{WCP}}$ replaced by $\prec_{\text{DC}}$. DC differs from WCP by composing only with PO, not HB.

An execution trace has a *DC-race* if it has two conflicting events that are unordered by DC. The strict partial order $\prec_{\text{DC}}$ is strictly weaker than $\prec_{\text{WCP}} \cup \prec_{\text{PO}}$, and as a result, DC predicts races that WCP does not. For example, the executions in Figures 1(a), 2(a), and 3(a) contain DC-races. In fact, DC is complete (as defined in Definition 2.4).

Theorem 1 (DC completeness)**.** *If a trace tr has a predictable race (according to Definition 2.2), then tr has a DC-race.*

We prove this theorem in Appendix B.

However, DC is *unsound*: a DC-race may not be a true predictable race. Figures 4(a) and 4(b) show executions that each have conflicting events on x that are unordered by DC but cannot be consecutive in any correctly reordered trace.

DC analysis. We introduce *DC analysis*, a dynamic analysis that tracks the DC relation at every event and detects DC-races. The analysis, which uses vector clocks [59] to track the DC partial order, is analogous to prior work's WCP analysis [44]. The main difference with WCP analysis is that DC analysis does not track the HB relation and does not compose HB with DC, resulting in fewer orderings among events and thus increased race coverage. Like WCP analysis [44], DC analysis's running time is linear in trace length, and its space complexity is linear in the worst case. Appendix A presents DC analysis in detail.

5 Vindicating Predictable Races

Algorithm 1 presents VindicateRace, an algorithm that determines whether a DC-race is a true predictable race. VindicateRace takes as input a single DC-race and a constraint graph with nodes that are executed events and edges that correspond to DC order. Intuitively, the constraint graph's edges serve as constraints on a reordered trace, but the initial constraints (i.e., DC order) are not sufficient to satisfy the rules of a correctly reordered trace that exposes the input race. VindicateRace thus adds additional needed constraints, both on the order of the DC-race's events and on critical sections of the same lock. Ultimately, either the resulting constraint graph has a cycle that implies the DC-race is a false race; VindicateRace constructs a correctly reordered trace tr' in which the DC-race's events are consecutive; or VindicateRace fails to construct a correctly reordered trace from the constraint graph, which is inconclusive because the construction uses a greedy algorithm.

The rest of this section first defines the constraint graph and then explains VindicateRace and its helper procedures.

5.1 The Constraint Graph

The constraint graph G is a directed graph in which the nodes are the events in tr. G's edges, e.g., $(e, e') \in G$, intuitively represent constraints on any reordered trace. We use the notation $e \rightsquigarrow_G e'$ to indicate that e' is reachable from e in G:

$$e \rightsquigarrow_G e' \equiv (e, e') \in G \vee \exists e'' \mid e \rightsquigarrow_G e'' \wedge e'' \rightsquigarrow_G e'$$

When Vindicator calls VindicateRace, G initially has edges that represent DC ordering among events. That is, initially the following property holds:

$$\forall e, e' \in tr \left(e \prec_{\text{DC}} e' \iff e \rightsquigarrow_G e' \right)$$

Algorithm 1 Check if DC-race is a true predictable race

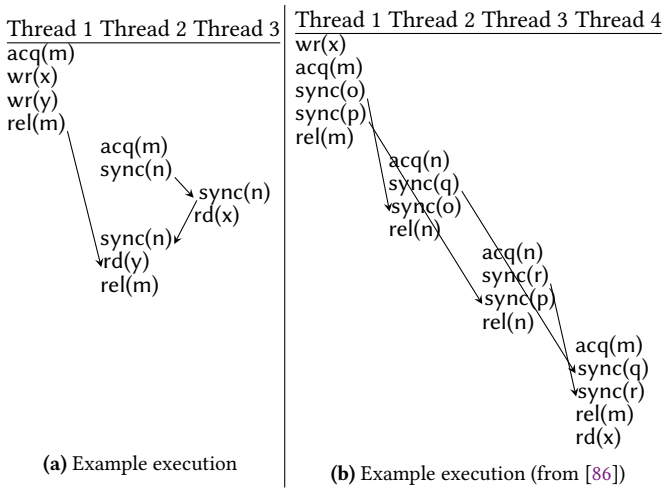
An execution trace is an ordered list of events: $\langle e, \ldots, e' \rangle$.
The operator $\oplus$ concatenates two traces:
$\langle e, \ldots, e' \rangle \oplus \langle e'', \ldots, e''' \rangle \equiv \langle e, \ldots, e', e'', \ldots, e''' \rangle$.

```
1:  procedure VindicateRace(G, e1, e2)
2:      G ← AddConstraints(G, e1, e2)
3:      if G = ∅ then
4:          return No predictable race
5:      else
6:          tr' ← ConstructReorderedTrace(G, e1, e2)
7:          if tr' ≠ ⟨ ⟩ then                ▷ Check for non-empty trace
8:              return Predictable race witnessed by tr'
9:          else
10:             return Don't know

11: procedure AddConstraints(G, e1, e2)
12:     C ← {(src, e2) | (src, e1) ∈ G} ∪ {(src, e1) | (src, e2) ∈ G}
13:     G ← G ∪ C
14:     do
15:         foreach (src, snk) ∈ C do
16:             foreach (a, r) | a is an acq ∧ a ⇝_G src ∧
                                 r is a rel ∧ snk ⇝_G r ∧
                                 L(a) = L(r) do
17:                 if (A(r) ⇝_G e1 ∨ A(r) ⇝_G e2) ∧
                       (a ⇝_G e1 ∨ a ⇝_G e2) then
18:                     C ← C ∪ {(R(a), A(r))}
19:                     G ← G ∪ {(R(a), A(r))}
20:         if ∃e | (e ⇝_G e1 ∨ e ⇝_G e2) ∧ e ⇝_G e then
21:             return ∅        ▷ Cycle detected; no predictable race
22:     while C has changed
23:     return G

24: procedure ConstructReorderedTrace(G, e1, e2)
25:     R ← {e | e ⇝_G e1 ∨ e ⇝_G e2}          ▷ Reachable events
26:     do
27:         tr' ← AttemptToConstructTrace(G, R, e1, e2)
28:         if tr' = ⟨r⟩ then ▷ tr' contains needed (release) event?
29:             R ← R ∪ {r} ∪ {e | e ⇝_G r}
30:     while R has changed
31:     return tr'

32: procedure AttemptToConstructTrace(G, R, e1, e2)
33:     tr' ← ⟨e1, e2⟩
34:     while R \ tr' ≠ ∅ do
35:         next ← {e ∈ R \ tr' | (∄e' | (e, e') ∈ G ∧ e' ∈ R \ tr')}
36:         legal ← {e ∈ next | ⟨e⟩ ⊕ tr' satisfies LS}
37:         if legal = ∅ then
38:             if ∃r | (∃e ∈ next | e ∈ CS(r) ∧
                         r ∉ R ∧ ⟨r⟩ ⊕ tr' satisfies LS) then
39:                 return ⟨r⟩              ▷ Return missing release
40:             return ⟨ ⟩                  ▷ Failed to construct trace
41:         else
                ▷ Select latest legal event in tr order
42:             let e ∈ legal s.t. ∄e' ∈ legal | e <_tr e'
43:             tr' ← ⟨e⟩ ⊕ tr'             ▷ Prepend event to trace
44:     return tr'                  ▷ Return correctly reordered trace
```



(a) Example execution

(b) Example execution (from [86])

Figure 4. Each execution has a DC-race, i.e., wr(x) $\not\prec_{DC}$ rd(x), but no predictable race. The arrows show DC ordering (excluding PO ordering). Figure 3 explains what sync(o) means.

Vindicator constructs the initial G during DC analysis (in addition to tracking the DC relation). Alternatively, DC analysis could record only the execution's events, and construct G on demand if and when DC analysis detects a DC-race.

Figures 2(a), 3(a), 4(a), and 4(b) show initial constraint graphs for three different executions (ignoring the "HB" edge in Figure 2(a)), each of which has a DC-race but no WCP-race. The arrows in each figure represent edges corresponding to DC rules (a) and (b).[6] The figures do *not* explicitly show the PO edges that exist between events by the same thread (DC rule (c)). Graph reachability provides transitivity (DC rule (d)). Note that Figure 2(a)'s "WCP" edge is equivalent to the "DC" edge that corresponds to $\text{rel(o)}^{T1} \prec_{DC} \text{rd(y)}$, where rel(o)^{T1} indicates the rel(o) by Thread 1. The rest of this section uses these four constraint graphs as running examples.

5.2 Adding Constraints to G

The initial constraint graph G lacks some of the constraints that must exist on a correctly reordered trace. For example, in Figure 2(a), a reordered trace that executes wr(x) and rd(x) consecutively must execute the critical sections on m in a different order from the original trace. Similarly, in Figure 3(a), a reordered trace that exposes the predictable race must execute the critical sections on m and n in reverse order.

AddConstraints (called at line 2 in Algorithm 1) adds constraints so that the DC-race's events execute consecutively in the reordered trace, and then discovers and adds constraints on the ordering of critical sections.

Making events consecutive. For e_1 and e_2 (the input DC-race) to be consecutive in a correctly reordered trace, every event that must execute before e_1 **or** e_2, must execute before e_1 **and** e_2. Lines 12–13 add *consecutive-event constraints* to G: for each predecessor event *src* of e_1 or e_2 in G, AddConstraints adds an edge from *src* to e_2 or e_1, respectively.

Figure 5 shows the updated constraint graphs for the four example executions. Each constraint graph represents consecutive-event constraints as dashed arrows. Note that edges from e_2 to e_1's predecessor (assuming $e_1 <_{tr} e_2$) will typically point *backward* relative to $<_{tr}$ order.

In Figure 5(a), AddConstraints adds only one consecutive-event constraint edge from rd(x)'s predecessor (rel(m)) to wr(x); wr(x) has no predecessors to add edges from. The same situation applies to Figure 5(d). In Figures 5(b) and 5(c), AddConstraints adds an edge from wr(x)'s lone predecessor to rd(x), and from rd(x)'s lone predecessor to wr(x).

AddConstraints adds the new constraint edges not only to G but also to a new set C whose edges are the starting point for discovering ordering constraints on critical sections.

Ordering critical sections. Using the added consecutive-event constraints, AddConstraints identifies and adds ordering constraints on critical sections, called *lock semantics (LS) constraints* (lines 14–19 in Algorithm 1). These constraints have the following form: if two critical sections on the same lock are ordered, at least in part, in G, and each critical section is ordered, at least in part, before e_1 or e_2, then the critical sections must be *fully* ordered in a correctly reordered trace.

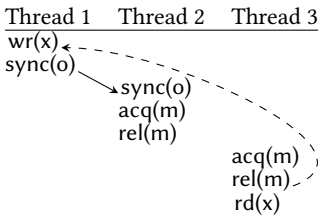
Consider the left half of Figure 5(b), in which there is a path from Thread 2's acq(m) to Thread 1's rel(m), i.e., $\text{acq(m)}^{T2} \rightsquigarrow_G \text{rel(m)}^{T1}$. Since both acquire events are ordered before at least one of rd(x) or wr(x) (in this case, both acquires reach both accesses), we know that at least part of each critical section must execute in tr'. AddConstraints identifies such critical sections in lines 16–17. Since these conditions hold, acq(m)^{T2}'s critical section must execute entirely before rel(m)^{T1}'s critical section, to enforce the LS rule of a correctly reordered trace on tr'. So AddConstraints adds an edge (dotted arrow) from rel(m)^{T2} to acq(m)^{T1} (lines 18–19). There is now a cycle that reaches wr(x) and rd(x) (detected at lines 20–21, as discussed below).

In general, the newly added edges may reveal new critical sections that must be fully ordered, and so AddConstraints continues looking for all ordered critical sections from edges in C until convergence.

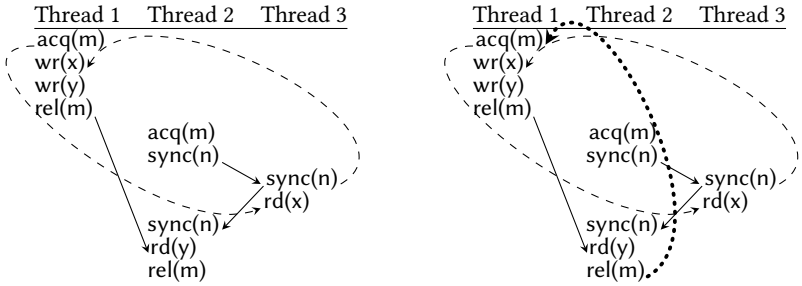
In Figure 5(c), after AddConstraints adds the consecutive-event edges (dashed arrows), it detects that $\text{acq(m)}^{T4} \rightsquigarrow_G \text{rel(m)}^{T1}$ and that both critical sections reach at least one access, so it adds an edge from rel(m)^{T4} to acq(m)^{T1}. This edge in turn creates the path $\text{acq(n)}^{T3} \rightsquigarrow_G \text{rel(n)}^{T2}$, and adds the edge rel(n)^{T3} to acq(n)^{T2}. After that, AddConstraints finds no new edges to add (convergence), and it returns.

In Figure 5(d), after AddConstraints adds consecutive-event edges, it detects that $\text{acq(n)}^{T3} \rightsquigarrow_G \text{rel(n)}^{T2}$ and that

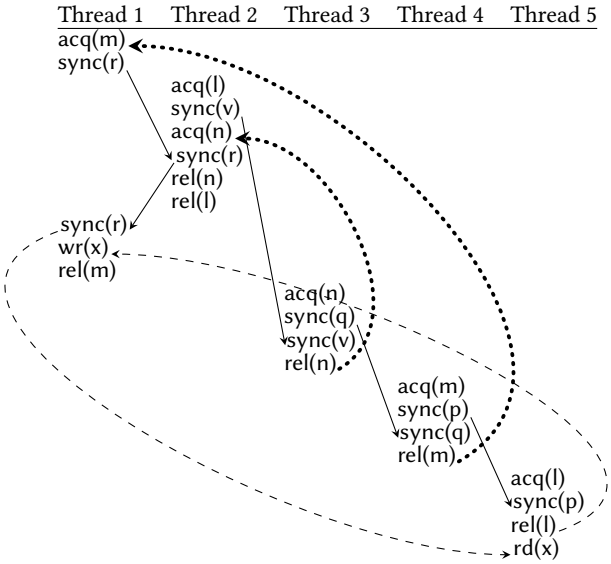
[6] The figures do not explicitly depict any rule (b) edges, since in these examples all rule (b) edges are already implied by other edges, e.g., Figure 4(a)'s rel(m) events are already ordered by a rule (a) edge composed with PO.



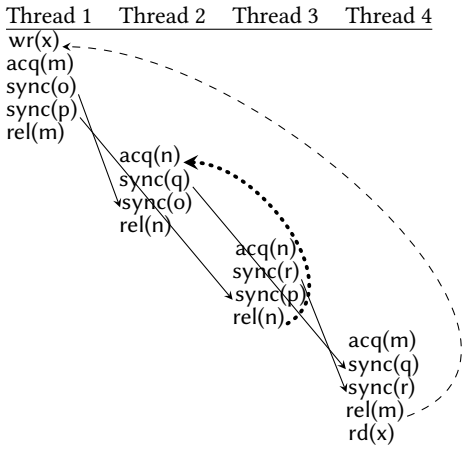
(a) After adding consecutive-event constraint. Execution has a predictable race. Figure 2(b) shows the reordered trace constructed by ConstructReorderedTrace.



(b) After adding consecutive-event constraints (left) and then LS constraints (right). The resulting constraint graph is cyclic, and the execution has no predictable race.



(c) After adding consecutive-event and LS constraints. Execution has a predictable race. Figure 3(b) shows the reordered trace constructed by ConstructReorderedTrace.



(d) After adding consecutive-event and LS constraints. The resulting constraint graph is cyclic, and the execution has no predictable race.

Figure 5. Constraint graphs for four example executions (example executions are from Figures 2(a), 3(a), 4(a), and 4(b)) after AddConstraints has added consecutive-event constraints (dashed arrows) and LS constraints (dotted arrows). In (a), AddConstraints adds only consecutive-event constraints; there are no LS constraints to add. For (b), we show adding of consecutive-event and LS constraints in two separate steps. For (c) and (d), we show all added constraints at once.

both critical sections reach at least one access, so it adds an edge from rel(n)^{T3} to acq(n)^{T2}. After adding this edge, the graph has a cycle that lines 20–21 detect (discussed below).

In Figure 5(a), AddConstraints does not identify any LS constraints to add (thus no dotted arrows in the figure). The consecutive-event edge is sufficient to constrain the reordered trace.

Detecting cycles. After AddConstraints adds consecutive-event and LS constraints to G, it may be possible to construct a correctly reordered trace that includes e_1 and e_2 and satisfies G's constraints—but only if G *does not have a cycle* of constraints ordered before e_1 or e_2. AddConstraints checks this condition (line 20) and returns an empty graph to indicate a cycle (line 21). (A cycle in G that does *not* reach e_1 or e_2 is not constraining since a correctly reordered trace does not need to contain any events *after* e_1 and e_2.)

In Figure 5(b), we can see that a cycle exists that reaches e_1 (and e_2): $\text{acq(m)}^{T1} \rightsquigarrow_G \text{rel(m)}^{T2} \rightsquigarrow_G \text{acq(m)}^{T1} \rightsquigarrow_G \text{wr(x)}^{T1}$. Likewise, in Figure 5(d), a cycle exists that reaches e_2 (and e_1), e.g., $\text{acq(n)}^{T2} \rightsquigarrow_G \text{rel(m)}^{T4} \rightsquigarrow_G \text{wr(x)}^{T1} \rightsquigarrow_G \text{rel(n)}^{T3} \rightsquigarrow_G \text{acq(n)}^{T2} \rightsquigarrow_G \text{rd(x)}^{T4}$. AddConstraints detects these cycles, and VindicateRace reports the impossibility of constructing a correctly reordered trace.

In contrast, Figures 5(a) and 5(c) are acyclic, and AddConstraints returns an updated graph G.

Completeness. We informally argue, but have not formally proved, that the constraints computed by AddConstraints are *complete*, i.e., AddConstraints detects no cycle if a

predictable race exists. Our argument relies on showing that every constraint added by AddConstraints is a necessary constraint on any reordered trace tr' in which e_1 and e_2 execute consecutively:

- Each of AddConstraints's *consecutive-event constraints* is necessary for executing the conflicting events consecutively on tr'.
- Each of AddConstraints's *lock semantics (LS) constraints* is necessary, by the following argument. Suppose a_1 and a_2 are two acquire events that both must be in tr', and $a_2 \rightsquigarrow_G R(a_1)$ already exists from a prior step of AddConstraints. Since both a_1 and a_2 both must be in tr', at least one of the critical sections must be in tr' in its entirety, i.e., either $R(a_1) <_{tr'} a_2$ or $R(a_2) <_{tr'} a_1$. Inductively assuming that $a_2 \rightsquigarrow_G R(a_1)$ is a necessary constraint on tr', then $R(a_1) \not<_{tr'} a_2$ and therefore $R(a_2) <_{tr'} a_1$. Thus adding $(R(a_2), a_1)$ to G is a necessary constraint on tr'.

Since AddConstraints adds only necessary constraints, and the initial G has only necessary constraints (since DC is complete), all of G's edges are necessary constraints on tr'. Thus a cycle would contradict the existence of tr' that executes e_1 and e_2 consecutively.

Discussion. Although cyclic constraint graphs are possible (e.g., Figures 5(b) and 5(d)), we have not encountered a cyclic graph in our experiments. Our experiments not only encounter only acyclic constraint graphs, but each graph corresponds to a true predictable race.

However, it is possible for AddConstraints to return an acyclic graph even when no predictable race exists; Appendix C shows an example execution. Briefly, the example involves two pairs of critical sections on different locks whose implicit dependencies are cyclic.

Regardless, VindicateRace is sound overall because, prior to reporting a predictable race, it ensures it can construct a correctly reordered trace, as described next.

5.3 Constructing a Reordered Trace

Finally, if the computed constraints do not contain a cycle that reaches the input DC-race, VindicateRace tries to construct a correctly reordered trace tr' by calling ConstructReorderedTrace (line 6 in Algorithm 1). While G provides PO and CA ordering and some LS ordering (from Definition 2.1), it does *not* totally order *all* critical sections on the same lock. For example, Figure 5(c) contains neither the path $\text{rel(l)}^{T2} \rightsquigarrow_G \text{acq(l)}^{T5}$ nor the path $\text{rel(l)}^{T5} \rightsquigarrow_G \text{acq(l)}^{T2}$.

Thus an acyclic G is not sufficient to ensure that a correctly reordered trace exists. Furthermore, ConstructReorderedTrace is a greedy algorithm that does not backtrack to explore all possible traces, avoiding exponential complexity but risking failure when a correctly reordered trace exists.

Construction algorithm. ConstructReorderedTrace first computes the set of events R that reach e_1 or e_2 (line 25); these events (plus e_1 and e_2) must be in tr'. ConstructReorderedTrace then calls AttemptToConstructTrace (line 27), which builds tr' in *reverse* order. It first adds e_2 and e_1 to tr' (line 33). It then selects events from R, one at a time, and prepends them to tr' until tr' contains all events from R (lines 34–43). To prepend an event to tr', tr' prepended with the event must satisfy the constraints in G (line 35) and not violate lock semantics (line 36).

Algorithm 1 omits the detailed logic for checking lock semantics. Briefly, events in a critical section on m cannot be prepended if m is currently held by a different thread, and a critical section on m must be prepended in its entirety if tr' already contains events from another critical section on m.

AttemptToConstructTrace is a greedy algorithm that repeatedly chooses one event to prepend to tr' among multiple acceptable events. This choice affects the order of critical sections on the same lock in tr'. As line 42 shows, AttemptToConstructTrace always chooses the latest event in $<_{tr}$ order among acceptable events. Our insight here is that the original order of critical sections ($<_{tr}$ order) is most likely to avoid a failure to produce a correctly reordered trace. This insight turns out to be correct in practice: in our experiments, choosing the latest event in $<_{tr}$ order always succeeds, while choosing a different legal event can lead AttemptToConstructTrace to fail. Appendix C shows an execution for which choosing the latest event succeeds but choosing another event may fail.

Retrying construction. As mentioned above, if tr' already contains an acq(m) event, then in order to add an event $e \in CS(r)$ where r is a rel(m) event, AttemptToConstructTrace must add first add r to tr'. However, r may not be in R! If AttemptToConstructTrace encounters this case (line 38), it returns the missing event r (line 39). ConstructReorderedTrace then adds r and events that reach r to R (line 29) and calls AttemptToConstructTrace again. In the worst case, R might be missing release events for each critical section that contains a thread's last event in R, bounding the number of times that ConstructReorderedTrace can retry AttemptToConstructTrace.

Figure 5(c) helps to illustrate this case. After adding Thread 5's critical section on l to tr', AttemptToConstructTrace cannot legally add Thread 2's sync(r) to tr' without first adding Thread 2's rel(l)—which is not in R. AttemptToConstructTrace ultimately returns Thread 2's rel(l), and ConstructReorderedTrace adds Thread 2's rel(l) (and rel(n)) to R and again calls AttemptToConstructTrace, which returns the correctly reordered trace shown in Figure 3(b).

AttemptToConstructTrace eventually returns either a correctly reordered trace tr' that demonstrates a predictable race (line 44), or it fails if and when there are no missing release events and no legal events to add to tr', in which case it returns an empty trace (line 40).

Discussion. ConstructReorderedTrace is *sound*: if it returns a reordered trace tr', it is a correctly reordered trace in which e_1 and e_2 are consecutive. That is, it always fails if no predictable race exists.

ConstructReorderedTrace is *incomplete*: the greedy algorithm can fail even when a predictable race in fact exists. Appendix C shows an execution for which ConstructReorderedTrace fails by always choosing the latest event, yet a correctly reordered trace is feasible. ConstructReorderedTrace would be complete if it tried all (exponential in trace length) possible orders satisfying G.

A case we have not yet discussed is that an execution may have a predictable *deadlock* (i.e., if a correctly reordered execution has a deadlock) but not a predictable race. Note that *WCP* is sound with a deadlock caveat: a *WCP-race* indicates either a predictable race or a predictable deadlock [44]. In contrast, VindicateRace inherently reports only predictable races and will not report predictable deadlocks. Future work might be able to modify VindicateRace to detect sufficient conditions for a predictable deadlock.

5.4 Asymptotic Complexity

VindicateRace's time complexity is polynomial in N, the length of tr, because every loop's iteration count is bounded by G's size (nodes plus edges). The polynomial's degree depends on how VindicateRace is implemented. VindicateRace uses $\Omega(N)$ space for both G and tr'.

6 Evaluation

This section evaluates Vindicator's ability to detect predictable races and its run-time performance, compared with competing approaches.

6.1 Implementation

We implemented Vindicator in *RoadRunner*,[7] a dynamic analysis framework for concurrent Java programs [32] that is the implementation platform for the FastTrack race detector [30, 33, 34, 71]. RoadRunner provides analysis hooks at memory accesses and synchronization operations by instrumenting Java bytecode dynamically at class loading time.

Our implementation of Vindicator is publicly available.[8]

The implementation has two main components: DC analysis and VindicateRace. DC analysis constructs the constraint graph G as it executes, storing it in memory, and logs all detected DC-races. Vindicator performs WCP and HB analyses alongside DC analysis to determine if each DC-race is also a WCP-race and/or an HB-race. When the program execution ends, Vindicator calls VindicateRace on each *DC-only race*, which is a DC-race that is not also a WCP-race. Since every WCP-race is a DC-race and every WCP-race is a true predictable race or deadlock [44], only DC-races that are not WCP-races need to be vindicated to verify true predictable races. The implementation supports optionally calling VindicateRace on a WCP-race to obtain a correctly reordered trace or to distinguish a race from a deadlock.

DC analysis and constraint graph construction. Our implementation of DC analysis handles read, write, acquire, and release events according to Algorithm 2 in Appendix A. For thread fork (parent to child) and join (child to parent) edges, DC analysis directly DC-orders parent and child threads by joining vector clocks and updating G accordingly. The analysis similarly handles volatile write–read edges and static class initializers edges [49]. It handles Object.wait() by treating it as a lock release followed by an acquire. Two evaluated programs (pmd and tomcat) fork threads implicitly, using constructs from java.util.concurrent instead of explicitly invoking Thread.start(). Since RoadRunner does not currently support interposing on these constructs, DC analysis detects thread start and terminate in these cases and conservatively adds thread fork and join edges.

For each event e that DC analysis handles, the analysis creates an event node e in the constraint graph G and adds edge(s) to G for newly established DC ordering(s). DC analysis minimizes the number of edges added to G, adding (e_{src}, e) to G only if $e_{src} \rightsquigarrow_G e$ does not already hold, using DC analysis vector clocks to determine which ordering(s) have been newly established at e. To determine each e_{src}, DC analysis tracks, for each variable x and synchronization object m, the last event by each thread that accessed x or m.

DC analysis processes parallel events in parallel, using fine-grained synchronization on analysis metadata and nodes in G to provide analysis atomicity without serializing the analysis. Since the analysis does not observe a total order $<_{tr}$ of events, it assigns each event node a *Lamport timestamp* [46] such that $e \prec_{\text{HB}} e' \implies ts(e) < ts(e')$.

DC analysis uses an instrumentation "fast path" that identifies and skips redundant accesses (if there is a prior write, or if the prior and current events are reads) to the same variable by the same thread without interleaving synchronization. The fast path reduces run-time overhead and the size of G. Reducing G's size not only reduces space overhead, but it improves VindicateRace's run time. To further reduce the size of G, DC analysis *merges* adjacent PO-ordered nodes on the fly, i.e., $(e, e') \in G$ becomes a single node, if e and e' are both read and write events and e has no other outgoing edges and e' has no other incoming edges.

WCP and HB analyses. The implementation performs DC, WCP, and HB analyses on the *same observed trace*. We implemented WCP and HB analyses in RoadRunner (instead of using the WCP authors' available implementation [44] or the FastTrack implementation in RoadRunner [30, 34])

[7] We used RoadRunner version 0.5 (https://github.com/stephenfreund/RoadRunner/releases/tag/v0.5), released February 2017.
[8] https://github.com/PLaSSticity/Vindicator.git

to provide a fair comparison with DC analysis and to identify which DC-races are DC-only races. Each analysis's time complexity is linear in the size of the trace.

Handling DC-races. When DC analysis detects a DC-race between two events e_1 and e_2, it updates vector clocks and G so that $e_1 \prec_{DC} e_2$ and $e_1 \rightsquigarrow_G e_2$. This action avoids the possibility of a DC-race detected later in the run from being dependent on an earlier DC-race. Note that every HB-race is a WCP-race, and every WCP-race is a DC-race.

At a read or write event e, DC analysis may detect multiple DC-races with prior read or write events e' such that $e' \not\prec_{DC} e$. To avoid the complexity of determining whether some of these DC-races are dependent on each other, DC analysis records only a "shortest" DC-race, i.e., $e' \not\prec_{DC} e$ such that $ts(e')$ is maximal (potentially choosing arbitrarily among multiple concurrent events).

VindicateRace. Vindicator calls VindicateRace (Algorithm 1) on each DC-only race separately, using graph traversals to compute reachability between events. Before VindicateRace returns, it removes all edges that it added to G, in order to check each DC-race independently.

As a sanity check that is not required for Vindicator's correctness, the implementation optionally checks that tr' returned by ConstructReorderedTrace is a correctly reordered trace according to Definition 2.1.

VindicateRace uses two correctness-preserving optimizations. First, AddConstraints exploits PO ordering among events by the same thread to avoid considering many redundant acquire–release pairs. Second, AddConstraints only considers events within a window of events between e_1 and e_2, based on events' Lamport timestamps. To preserve correctness, AddConstraints expands the window on the fly to include each edge it adds to G.

6.2 Methodology

We evaluate Vindicator using benchmarked versions of real programs: the DaCapo benchmarks [7], version 9.12 Bach. We use RoadRunner's provided support for harnessing and running the DaCapo programs (e.g., for dynamic bytecode instrumentation in the presence of DaCapo's custom class loading); the provided workloads are close to DaCapo's default workload size. RoadRunner does not currently support eclipse, tradebeans, or tradesoap; our evaluation excludes those programs, as well as fop since it is single threaded.

The experiments run on a quiet system with an Intel Xeon E5-2683 14-core processor with hyperthreading disabled and 256GB of main memory, running Linux 3.10.0. We configure RoadRunner to tell programs that there are 8 available cores, which causes several DaCapo programs to create 8 worker threads. We run RoadRunner with the HotSpot 1.8.0 JVM and let it choose and adjust the heap size on the fly.

Program	Statically distinct races (dynamic races)					
	HB-races		WCP-races		DC-races	
avrora	5	(933)	5	(934)	5	(996)
batik	0	(0)	0	(0)	0	(0)
h2	10	(690)	11	(793)	11	(1,027)
jython	3	(3)	3	(4)	3	(4)
luindex	1	(1)	1	(1)	1	(1)
lusearch	0	(0)	0	(0)	0	(0)
pmd	4	(13)	4	(13)	5	(23)
sunflow	2	(8)	2	(10)	2	(14)
tomcat	109	(4,604)	110	(4,659)	110	(4,677)
xalan	4	(16)	63	(3,420)	67	(4,660)
Total	138	(6,268)	199	(9,834)	204	(11,402)

Table 1. HB-, WCP-, and DC-races detected by our implementation. *DC-races* include all WCP-races; *WCP-races* include all HB-races. **VindicateRace confirmed that every DC-race is a true predictable race.** Each result is the average of 10 trials, rounded to the nearest integer.

Program	Static DC-only race	Event distance
h2	StringCache.getNew():93 StringCache.get():48	11,288–248,799
h2	StringCache.getNew():83 StringCache.get():54	12,438–14,182
pmd	PMD.getSourceTypeOfFile():152 PMD.<init>():57	
pmd	PMD.setExcludeMarker():234 PMD.processFile():96	
xalan	LocPathIterator.setRoot():369 LocPathIterator.setRoot():370	24,870–71,922,359
xalan	AttributeIterator.getNextNode():56 LocPathIterator.setRoot():372	
xalan	FastStringBuffer.<init>():210 FastStringBuffer.append():653	23,540–106,725
xalan	FastStringBuffer.<init>():210 FastStringBuffer.append():488	2,146–2,629,775
xalan	OneStepIterator.setRoot():97 OneStepIterator.setRoot():97	

Table 2. Characteristics of the nine static DC-only races reported by Vindicator in our experiments. Each race is an unordered pair of static locations, which are represented as class, method, and line number. *Event distance* is the range of event distances (distance apart in $<_{tr}$ between the two conflicting events) across all dynamic instances of the race, from a separate experiment that totally orders events.

6.3 Detection Coverage of Predictable Races

Here we evaluate Vindicator's predictable race coverage compared with HB and WCP analyses.

Table 1 reports races detected by DC, WCP, and HB analyses on the same trace. For each kind of race, the table reports statically distinct races (a statically distinct race is an unordered pair of static program source locations), followed by dynamic races (a dynamic race is a pair of events in the trace;

multiple dynamic races may correspond to the same statically distinct race) in parentheses, averaged over 10 trials.

The table shows that on average DC analysis reports five *static DC-only races*, i.e., statically distinct DC-races that are not WCP-races. In addition, there are four static DC-only races (in h2, pmd, and xalan) that each occurs as a static DC-only race in 1 or 2 out of 10 trials, so they are not shown in the table due to rounding.

Table 2 shows details of the nine static DC-only races detected across the 10 trials. These static DC-only races did *not* manifest as a WCP-race in any of the trials, except for both of h2's races and xalan's AttributeIterator.getNextNode():56–LocPathIterator.setRoot():372 race, which manifest as a WCP-race in at least one trial. The table shows each statically distinct race's two static source locations and the range of event distances across all dynamic instances of the DC-only race across all trials in which it occurred. A dynamic race's *event distance* is the number of events apart, in the observed trace order $<_{tr}$, that the two conflicting events occurred. Since the implementation does not totally order events (Section 6.1), we collected event distances in a separate 10 trials that use global synchronization to assign totally ordered timestamps. In these 10 separate trials, five of Table 2's nine races occurred as static DC-only races.

Several programs have *dynamic* DC-only races, i.e., dynamic DC-races that are not also WCP-races. For example, the table reports 62 dynamic DC-only races on average for avrora. However, each of these races maps to the same statically distinct race as some dynamic WCP-race in the same trial, so avrora has no *static* DC-only races.

Event distances of all dynamic races. Figure 6 plots the cumulative distribution of event distances of all dynamic races from the 10 separate, globally synchronized trials described above. Each dynamic race is either a DC-only race, a WCP-only race (WCP-race that is not an HB-race), or an HB-race, and appears exactly once in the plot. For any event distance, the plot shows the percentage of dynamic races that have at least that event distance.

The plot shows that DC-only races have larger event distances than HB-races or WCP-only races by an order of magnitude or more. This result is notable for two reasons. First, prior work that is complete cannot scale beyond bounded windows of execution (Section 7) and would thus have difficulty finding many DC-only races. Second, VindicateRace successfully analyzes every dynamic DC-only race (Section 6.4) despite their large event distances.

Vindicating DC-races. By default, the Vindicator implementation invokes VindicateRace on each *dynamic* DC-only race, i.e., the difference between dynamic DC- and WCP-races in Table 1. In our experiments, **VindicateRace confirms that every dynamic DC-only race is a true predictable race.** That is, for every dynamic DC-race in Table 1 that is not also a WCP-race, VindicateRace verifies that it is a true predictable race: VindicateRace never encounters a cycle nor fails to construct a reordered trace, and it always reaches line 8 in Algorithm 1, returning a correctly reordered trace tr' that exposes the predictable race. (As a sanity check, our experiments also run VindicateRace on every *WCP-only and HB-race*, always producing a correctly reordered trace that exposes a race.)

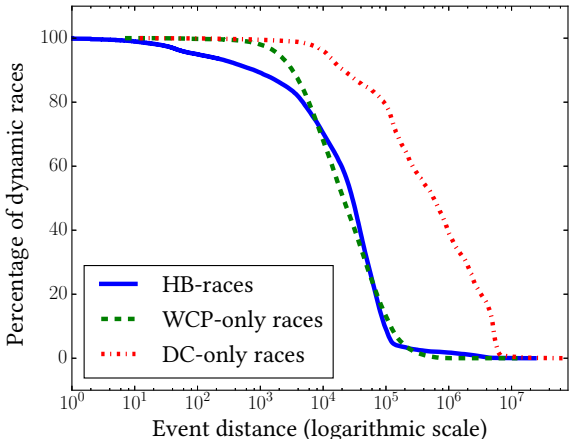


Figure 6. Cumulative distribution of the event distance of three kinds of dynamic races. For a given event distance, the plot shows the percentage of dynamic races with at least that event distance.

LS constraints added	0	1	2–3	4–7	8–15	16–135
DC-only races	14,398	553	325	212	149	32
Outer loop iterations	**1**	**2**	**3**	**4**	**5**	**6–14**
DC-only races	14,398	942	158	132	26	13

Table 3. Characteristics of VindicateRace (Algorithm 1) for all dynamic DC-only races across the 10 trials.

Table 3 shows characteristics of VindicateRace analyzing each of the dynamic DC-only races from the 10 trials. The table reports the distribution across all dynamic DC-only races, i.e., across all calls to VindicateRace. *LS constraints added* is the number of lock semantics (LS) constraints (edges) that AddConstraints adds (lines 18–19 in Algorithm 1). *Outer loop iterations* is the number of executed iterations of the do–while loop in AddConstraints (lines 14–22).

The table shows that for most dynamic DC-only races, AddConstraints adds no LS constraints and consequently does not repeat its outer loop. (AddConstraints always adds *consecutive-event* constraints; lines 12–13.) But for more than 1,000 DC-only races, AddConstraints performs multiple loop iterations and adds several LS constraints.

Program	Events		#Thr		Base time	Slowdown (normalized to Base time) Empty	WCP	DC	Vindicator			Memory usage (GB) WCP	DC	Vindicator
avrora	1,400M	(160M)	7	(7)	2.1 s	3.2×	23×	30×	33× (0 s)	+	9.7 s	< 2	< 2	< 2
batik	160M	(12M)	8	(7)	2.6 s	3.7×	13×	14×	16× (0 s)	+	0 s	4.4	4.5	5.9
h2	3,800M	(460M)	10	(9)	4.7 s	6.3×	70×	110×	110× (4.3 s)	+	790 s	39	68	64
jython	230M	(82M)	2	(2)	2.2 s	5.4×	27×	33×	37× (0 s)	+	0 s	< 2	< 2	< 2
luindex	400M	(45M)	3	(3)	1.1 s	5.1×	47×	58×	59× (0 s)	+	0 s	2.1	4.3	4.7
lusearch	1,400M	(190M)	10	(10)	1.1 s	8.7×	38×	45×	47× (0 s)	+	0 s	5.5	5.8	7.2
pmd	200M	(25M)	9	(9)	1.2 s	5.1×	19×	22×	22× (0.062 s)	+	0.13 s	< 2	2.9	2.6
sunflow	9,700M	(760M)	17	(9)	1.7 s	10×	88×	98×	120× (0 s)	+	0.016 s	< 2	< 2	< 2
tomcat	44M	(18M)	35	(22)	0.88 s	3.9×	13×	18×	21× (0 s)	+	7.1 s	2.6	3.1	4.3
xalan	610M	(260M)	9	(9)	2.1 s	2.9×	42×	64×	91× (11 s)	+	2,400 s	8.0	12	34

Table 4. Run time and memory usage for Vindicator and other analysis configurations. All values (except thread counts) are rounded to 2 significant figures. The table reports Vindicator's slowdown relative to uninstrumented execution as the first value in the *Vindicator* column (e.g., 33× for avrora). The text explains other values.

6.4 Run-Time Performance

Table 4 shows execution time, memory usage, and other dynamic characteristics of Vindicator, compared with configurations that perform a subset of Vindicator's functionality. Each value is the arithmetic mean from the 10 trials used in the rest of the evaluation. *Events* is total executed program events (memory accesses and synchronization operations); the subset of events that are not filtered out by fast-path instrumentation is in parentheses. *#Thr* is the total number of threads created and, in parentheses, the maximum number of threads active at any time. Reported execution times are wall-clock times. *Memory usage* is the maximum memory usage across all full-heap garbage collections (GCs) in the execution. Some executions perform no full-heap GCs because the JVM sets the initial heap size to 2 GB, so the table reports only values for full-heap GCs that exceed 2 GB. The Base and Empty configurations never exceed 2 GB, so we omit them from the table.

Base time reports execution time of an *uninstrumented* program. Other execution times are normalized to *Base time*. *Empty* executes RoadRunner's Empty tool, which instruments programs to generates events, but performs no analysis on them. The *WCP* configuration performs only WCP analysis, which includes HB analysis. *DC* performs DC analysis in addition to WCP analysis, but does not construct the constraint graph G. The table shows that *WCP* adds substantial run-time and memory overhead. On top of *WCP*, *DC* adds moderate run-time overhead but sometimes adds high memory overhead. The memory overhead is expected as even though DC is similar algorithmically to WCP, each analysis maintains separate data structures.

The *Vindicator* configuration computes all relations while also building the constraint graph; when the program terminates, Vindicator calls VindicateRace on each DC-race. Each cell reports three numbers: Vindicator's run time normalized to *Base time*, which includes calling VindicateRace on a single dynamic instance of each static DC-only race (shown in parentheses as non-normalized run time), followed by the additional non-normalized run time for checking all remaining dynamic DC-only races (i.e., every dynamic DC-only race not already checked as a static DC-only race). For example, Vindicator takes 91× longer than 2.1 s to analyze xalan, including 11 s to run VindicateRace, on average, on 4 static DC-only races. It takes an additional 2,400 s to run VindicateRace on 1,236 *additional* dynamic DC-races. See Table 1 for corresponding static and dynamic race counts.

Building the constraint graph G adds relatively low run-time overhead over *DC*, and often adds low memory overhead because merging event nodes (Section 6.1) reduces G's size significantly. Vindicator adds high memory overhead over *DC* for xalan, which we have confirmed is due to building the constraint graph, not from running VindicateRace.

In a few cases, building G results in lower memory overhead than *not* building G. This counterintuitive result is due to the way we measure memory overhead: by recording the maximum of reported live memory across all full-heap GCs, which is an imperfect estimate of maximum live memory size. As a result, the reported memory is affected by how often GC happens and by the JVM's automatic adjustments to the heap size between GCs. RoadRunner periodically cleans up analysis resources that are no longer in use, further affecting the measurements.

The table shows that a small fraction of Vindicator's run time is from calling VindicateRace on static DC-only races (the times in parentheses). Vindicator incurs this overhead only for the three programs that have static DC-only races. Vindicator takes 11 seconds on average to analyze 4 static DC-only races for xalan, and a fraction of a second for 1 static DC-only race for pmd. For h2, only 3 out of 10 trials have a static DC-only race, which VindicateRace takes 14 seconds on average to analyze (or 4.3 seconds on average across all 10 trials, as the table reports). VindicateRace's relatively high cost for h2 is mainly due to ConstructReorderedTrace: the races' event distances are not large, but the events occur

late in the execution, so the size of the reordered trace is large, and in fact most of the execution time is GC time. The additional cost of analyzing every dynamic DC-only race (the time values right of the '+' symbol) is nontrivial—a few programs have many dynamic DC-only race—but the average cost of vindicating each race is low.

These results show that, in practice, Vindicator is efficient enough for testing-time use, including for finding and vindicating DC-races whose accesses are millions of events apart.

6.5 Summary and Discussion

Our prototype implementation of Vindicator adds significant performance costs, but the overheads are likely acceptable for heavyweight testing and worth the cost to expose new, hard-to-detect data races. The implementation of VindicateRace is efficient enough to produce reordered traces that verify races between accesses separated by millions of events.

Regarding coverage, Vindicator detects more predictable races than existing approaches. It finds more races than WCP analysis [44], which has the highest coverage among existing sound predictive analyses that scale to full execution traces. Existing approaches that predict more races than WCP rely on constraint solving and cannot scale beyond bounded windows of execution [38, 74, 79] (Section 7). Our results show that many DC-only races' accesses are millions of events apart—outside the range of windowed approaches.

Furthermore, in our experiments, Vindicator detects *all* predictable races (according to Definition 2.2) in real program executions. This work thus helps answer an open question of just how many predictable races exist in real programs.

7 Related Work

Prior work introduces a variety of approaches for detecting data races.

Static analysis. Static race detection can detect all feasible data races across every execution [26, 61, 62, 70, 92], but it is inherently unsound (reports false races). In practice, existing techniques report thousands of false races (e.g., [5, 47]). Static analysis results can optimize dynamic analysis, but the high rate of false positives limits the benefits [21, 25, 47, 91].

Dynamic analysis. Dynamic analysis analyzes a single execution and is typically sound. (An exception is lockset analysis, which detects false races [21, 23, 64, 65, 75, 90].) *Happens-before analysis* detects conflicting events unordered by the happens-before relation [25, 30, 46, 69, 80, 81]. Several other analyses detect a similar set of races, based on conflicting regions or making conflicting events to happen simultaneously [5, 6, 24, 27, 77, 89].

Data races manifest nondeterministically under specific thread interleavings, program inputs, and execution environments so that they may stay hidden even for extensively tested programs [88]. This nondeterminism can require tens or hundreds of runs or more to manifest a race [95] and takes weeks to reproduce, diagnose, and fix in production systems [35, 51]. A *production-time* analysis finds data races that occur in production settings, using sampling to trade coverage for performance [5, 14, 27, 42, 55, 83, 94] or requiring custom hardware support [22, 68, 76, 93, 95].

Predictive analysis. *Sound predictive analysis* detects data races that are possible in an execution *other than* the observed execution [20, 38, 39, 44, 50, 74, 79, 86]. Most existing sound predictive analyses cannot scale to full program executions; they instead analyze bounded windows of execution (e.g., 500-10,000 events), so they cannot predict data races between accesses that are "far apart" in the observed execution [20, 38, 39, 50, 74, 79, 86]. The exception is *weak-causally-precedes (WCP) analysis*, which can analyze whole program executions [44]. However, WCP analysis is incomplete, missing races that are knowable from a dynamic execution, not only in theory but also in practice (Section 6.3).

In concurrent work with ours, *DigHR* [53] uses a new *afterward-confirm (AC)* relation that relaxes *causally-precedes (CP)* [86] for critical sections with conflicting writes. The paper claims that AC is sound, but to our understanding, AC is unsound because it does not compose with HB. The paper's evaluation claims efficient online analyses for AC and CP, but does not explain how the implementation overcomes the known challenges of developing an online CP (or AC) analysis [73, 86], nor how DigHR outperforms CP and even FastTrack [30]. Furthermore, the evaluation does not report how many events the evaluated programs execute.

Some predictive approaches detect races beyond those knowable from an observed dynamic execution alone, by encoding control-flow constraints in addition to dynamic execution constraints [38–40]. These approaches can thus predict some races that Vindicator cannot find. However, these approaches can only scale to analyzing bounded windows of execution, so they cannot find predictable races whose accesses are "far apart," including many of the DC-only races detected by Vindicator (Section 6.3).

Schedule exploration. Other approaches explore multiple thread interleavings, either systematically or based on heuristics for exposing new behaviors [16, 18, 28, 36, 60, 77]. In contrast, predictive analysis detects data races from a single observed execution, making it complementary with schedule exploration. *Maximal causality reduction* combines predictive analysis with schedule exploration [37].

Alternatives to detecting data races. Researchers have introduced language, type, and system support for avoiding data races or providing well-defined behavior for them [1, 3, 8, 15, 29, 52, 56–58, 66, 72, 76, 78, 84, 85, 87]. Existing solutions have significant drawbacks that have limited their adoption, such as requiring writing code in new languages

or adding type annotations, impacting production-time performance, or requiring significant hardware changes.

8 Conclusion

Vindicator advances the state of the art in sound predictive race detection by detecting all predictable races (under the assumption that conflicting accesses cannot be reordered) from full program executions in time and space that are reasonable for heavyweight in-house testing. Vindicator detects and verifies hard-to-detect races between accesses that are millions of events apart—outside the range of windowed approaches—and also detects and verifies races that the prior state-of-the-art unbounded approach (WCP) cannot find.

A DC Analysis Details

The following notation and terminology follow the WCP paper's [44]. A vector clock $C : Tid \mapsto Val$ maps each thread to a nonnegative integer [59]. Operations on vector clocks are point-wise comparison ($\sqsubseteq$) and point-wise join ($\sqcup$):

$$C_1 \sqsubseteq C_2 \iff \forall t. C_1(t) \leq C_2(t)$$
$$C_1 \sqcup C_2 \equiv \lambda t. max(C_1(t), C_2(t))$$

DC analysis, detailed in Algorithm 2, handles each kind of event in a trace and maintains the following analysis state:

- a vector clock C_t for each thread t;
- vector clocks for each program variable, R_x and W_x, representing the last read and write to x by each thread;
- vector clocks $L^r_{m,x}$ and $L^w_{m,x}$ that represent joined critical sections on lock m that have read and written x;
- a set of variables read and written by each lock m's ongoing critical section (if any), R_m and W_m; and
- for each lock, two queues for each pair of threads, $Acq_{m,t}(t')$ and $Rel_{m,t}(t')$, explained below.

Initially, every vector clock maps all threads to 0, and every set and queue is empty.

At each release event r executed by t, the analysis increments t's logical time $C_t(t)$ (line 12). This action discerns events that occur before and after r, since later the analysis may order r to an event by another thread.

The analysis orders events by DC rule (a) as follows. At a read or write to x by t in a critical section on lock m, the analysis joins C_t with all prior critical sections on m that have performed conflicting events to x (lines 15 and 21). The analysis updates $L^r_{m,x}$ and $L^w_{m,x}$ at each release of m based on which variables the latest critical section on m accessed (lines 8–9).

To order events by rule (b) of DC, the analysis uses the queues of vector clocks, $Acq_{m,t}(t')$ and $Rel_{m,t}(t')$. $Acq_{m,t}(t')$ is a queue of vector clocks, each of which corresponds to an acquire of lock m by t' that has *not* been determined to be DC ordered to the most recent release of m by t. $Rel_{m,t}(t')$ maintains a queue of vector clocks for the release events corresponding to each acquire event represented in $Acq_{m,t}(t')$.

Algorithm 2 DC analysis at each event type

```
1:  procedure Acquire(t, m)
2:      foreach t' ≠ t do Acq_{m,t'}(t).Enque(C_t)
3:  procedure Release(t, m)
4:      foreach t' ≠ t do
5:          while Acq_{m,t}(t').Front() ⊑ C_t do
6:              Acq_{m,t}(t').Deque()
7:              C_t ← C_t ⊔ Rel_{m,t}(t').Deque()
8:      foreach x ∈ R_m do L^r_{m,x} ← L^r_{m,x} ⊔ C_t
9:      foreach x ∈ W_m do L^w_{m,x} ← L^w_{m,x} ⊔ C_t
10:     R_m ← W_m ← ∅
11:     foreach t' ≠ t do Rel_{m,t'}(t).Enque(C_t)
12:     C_t(t) ← C_t(t) + 1
13: procedure Read(t, x)
14:     foreach m ∈ HeldLocks(t) do
15:         C_t ← C_t ⊔ L^w_{m,x}
16:         R_m ← R_m ∪ {x}
17:     if W_x ⋢ C_t then DC-race
18:     R_x(t) ← C_t(t)
19: procedure Write(t, x)
20:     foreach m ∈ HeldLocks(t) do
21:         C_t ← C_t ⊔ (L^r_{m,x} ⊔ L^w_{m,x})
22:         W_m ← W_m ∪ {x}
23:     if W_x ⋢ C_t then DC-race
24:     if R_x ⋢ C_t then DC-race
25:     W_x(t) ← C_t(t)
```

These queues of vector clocks help to order events by DC rule (b): at t's release of m, the analysis checks whether the release is ordered to a prior acquire of m by each thread t', by checking $Acq_{m,t}(t')$.Front() $\sqsubseteq C_t$ (line 5). If so, the algorithm orders the release corresponding to $Acq_{m,t}(t')$.Front() to t's release of m (line 7). It is sufficient to use a queue and check only the head (line 5): other vector clocks in the queue will pass the check only if the head also passes the check (since PO implies DC).

At each read and write, DC analysis maintains the logical time of each thread's last read and write to x (lines 18 and 25). The analysis checks for DC-races by checking for DC ordering with prior conflicting events to x; a failed check indicates a DC-race (lines 17, 23, and 24).

Asymptotic complexity. DC analysis has the same time complexity as WCP analysis [44]: $O(N \times (L \times T + T^2))$, where N, L, and T are the numbers of events, locks, and threads, respectively.[9] Like WCP analysis, DC analysis uses $\Omega(N)$ space in the worst case because there exist executions for which the number of *Acq* and *Rel* queue elements is equal to the number of acq and rel events executed so far [44].

[9] We have confirmed with the authors that because each read and write event takes $O(L \times T)$ time, time complexity is in fact $O(N \times (L \times T + T^2))$ [45], not $O(N \times (L + T^2))$ as originally stated in the paper [44].

B DC Completeness Proof

This section proves the following theorem from Section 4:

Theorem 1 (DC completeness). *If a trace tr has a predictable race (according to Definition 2.2), then tr has a DC-race.*

To prove completeness, we use the following lemma:

Lemma 1. *For any two events e_1 and e_2 in tr, if $e_1 \prec_{DC} e_2$ then e_1 cannot be correctly reordered after e_2.*

To prove the lemma, we introduce a concept called *DC-distance* that defines a finite minimum number of DC rules that must be applied to establish $e \prec_{DC} e'$:

Definition (DC-distance). *The* DC-distance *of events e and e', $d(e, e')$, is defined as follows (the rules refer to the DC rules in Definition 4.1):*

$$d(e,e') = \min \begin{cases} 0 \textbf{ if } e \prec_{DC} e' \text{ by rule (a)} \\ 1 + d(A(e), e') \textbf{ if } e \prec_{DC} e' \text{ by rule (b)} \\ 0 \textbf{ if } e \prec_{PO} e' \text{ by rule (c)} \\ 1 + \min_{e''}(\max(d(e,e''), d(e'',e'))) \\ \quad \textbf{if } \exists e'' \mid e \prec_{DC} e'' \wedge e'' \prec_{DC} e' \text{ by rule (d)} \\ \infty \textbf{ otherwise} \end{cases}$$

Note that $d(e, e') = \infty$ if and only if $e \not\prec_{DC} e'$. Like DC, the DC-distance rules feed into each other recursively, e.g., rule (d) satisfies rule (b), which satisfies rule (d), etc. DC-distance converges (provides finite distance for DC-ordered events) by being defined as taking the minimum distance over all choices.

Proof of Lemma 1. Let e_1 and e_2 be events in tr such that $e_1 \prec_{DC} e_2$. We prove the lemma by induction on the DC-distance $d(e_1, e_2)$.

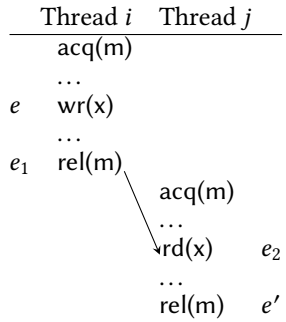
Base case: $d(e_1, e_2) = 0$

According to the definition of DC-distance, $e_1 \prec_{DC} e_2$ by DC rule (a) or (c).

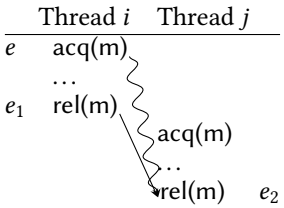
Case 1: $e_1 \prec_{DC} e_2$ by rule (a).

Then e_1 is a rel(m) event and e_2 is a rd(x) or wr(x) event. According to DC rule (a), there must exist e that is a rd(x) or wr(x) event and e' that is a rel(m) event, such that $e \in CS(e_1)$, $e_2 \in CS(e')$, and $e \asymp e_2$. Figure 7(a) depicts this case (with e as a write and e_2 as a read). Since $e \asymp e_2$, by the CA rule of correct reordering (Definition 2.1) e cannot be reordered after e_2.

Since $e_2 \in CS(e')$ and no two acquire events of the same lock may be totally ordered without an interleaved release event of the same lock (LS rule of Definition 2.1), e_1 and all events in $CS(e_1)$ must be reordered after e' in order for e_1 to be reordered after e_2. Since $e \in CS(e_1)$ and e cannot be reordered after e_2, e_1 cannot be reordered after e'. Therefore, e_1 cannot be reordered after e_2.



(a) *Base case, Case 1.* The arrow shows ordering established by DC rule (a).



(b) *Inductive step, Case 1.* Establishing DC ordering between e_1 and e_2 (indicated by a solid arrow) by DC rule (b) is preconditioned on the DC ordering indicated by the squiggly arrow.

Figure 7. Depictions of two cases of the Lemma 1 proof.

Case 2: $e_1 \prec_{DC} e_2$ by rule (c).

Then $e_1 \prec_{PO} e_2$, so e_1 cannot be reordered after e_2 by the PO rule of correct reordering (Definition 2.1).

Inductive step: $d(e_1, e_2) > 0$

Suppose the lemma statement holds true for all events e and e' in tr such that $d(e, e') < d(e_1, e_2)$.

Since $d(e_1, e_2) > 0$, by the definition of DC-distance, at least one of the following cases applies.

Case 1: $e_1 \prec_{DC} e_2$ by rule (b) and $d(e_1, e_2) = 1 + d(e, e_2)$ where $e = A(e_1)$.

Then e_1 and e_2 are rel(m) events on the same lock, and e is the acq(m) corresponding to e_1. According to rule (b) of the DC relation, $e_1 \prec_{DC} e_2$ holds because $e \prec_{DC} e_2$. Figure 7(b) depicts this case.

By the inductive hypothesis, since $d(e, e_2) < d(e_1, e_2)$ and $e \prec_{DC} e_2$, e cannot be reordered after e_2. Since no two acquire events of the same lock may be totally ordered without an interleaved release event of the same lock (LS rule of Definition 2.1), therefore e_1 and all events in $CS(e_1)$, including e, must be reordered after e_2 in order for e_1 to be reordered after e_2. Since e cannot be reordered after e_2, then e_1 cannot be reordered after e_2.

Case 2: $e_1 \prec_{DC} e_2$ by rule (d) and $\exists e \mid d(e_1, e_2) = 1 + \max(d(e_1, e), d(e, e_2))$.

Let e be such that $e_1 \prec_{DC} e \prec_{DC} e_2$ and $d(e_1, e_2) = 1 + \max(d(e_1, e), d(e, e_2))$. Therefore, $d(e_1, e) < d(e_1, e_2)$ and $d(e, e_2) < d(e_1, e_2)$. So by the inductive hypothesis, e_1 cannot be reordered after e and e cannot be reordered after e_2. Therefore, e_1 cannot be reordered after e_2.

Thus, in all cases e_1 cannot be reordered after e_2. □

We can now prove that DC is complete.

Proof of Theorem 1. The proof proceeds by contradiction. Suppose tr has a predictable race (Definition 2.2) but no DC-race. Let e_1 and e_2 be two events in tr such that there is a predictable race between e_1 and e_2 and $e_1 \prec_{DC} e_2$. By Lemma 1, e_1 cannot be reordered after e_2. Note that e_1 and e_2 cannot be directly ordered by rule (a) or (b) of the DC relation (Definition 4.1), both of which require e_1, which is a read or write access, to be a release event. Thus one or both of the following hold:

Case 1: e_1 and e_2 are directly ordered by DC rule (c).

So $e_1 \prec_{PO} e_2$ in tr. Thus e_1 and e_2 cannot be conflicting accesses and therefore cannot be a predictable race, a contradiction.

Case 2: e_1 and e_2 are directly ordered by DC rule (d).

So $\exists e \mid e_1 \prec_{DC} e \prec_{DC} e_2$ in tr. By Lemma 1, e cannot be reordered before e_1 or after e_2, so $e_1 <_{tr'} e <_{tr'} e_2$ for any reordered trace tr' that includes e_2. Thus e_1 and e_2 cannot be consecutive in any reordered trace, which contradicts the initial assumption of a predictable race between e_1 and e_2.

Thus all applicable cases lead to a contradiction. □

C Limitations of VindicateRace

This section provides four example execution traces that collectively demonstrate that ConstructReorderedTrace is incomplete and AddConstraints is unsound (both procedures are from Algorithm 1). For all four examples, there is a DC-race, and the constraint graph is acyclic after AddConstraints adds its constraints.

ConstructReorderedTrace

We found that for our evaluated programs, ConstructReorderedTrace always succeeded if it chose the latest legal event, but it sometimes failed if allowed to choose any legal event (Section 5.3). We first show examples where choosing an event *other than* the latest event can fail (i.e., if we change line 42 of ConstructReorderedTrace to "**let** $e \in$ *legalEvents*" so the algorithm may choose any legal event to prepend at each step), then show an example where choosing the latest event can fail.

Choosing arbitrary events can fail. Figure 8 shows an execution for which a correctly reordered execution exists, but ConstructReorderedTrace can fail if it chooses a certain legal event at each step. The figure marks six legal events that ConstructReorderedTrace has so far prepended to tr'. However, there is no seventh legal event to prepend to tr'. Since $\text{acq(m)}^{T1} <_{tr'} \text{rel(p)}^{T2}$ and $\text{acq(p)}^{T3} <_{tr'} \text{rel(m)}^{T4}$, the remaining mandatory constraints $\text{rel(m)}^{T4} <_{tr'} \text{acq(m)}^{T1}$ and $\text{rel(p)}^{T2} <_{tr'} \text{acq(p)}^{T3}$ are impossible to satisfy since $\text{rel(m)}^{T4} <_{tr'} \text{acq(m)}^{T1} <_{tr'} \text{rel(p)}^{T2} <_{tr'} \text{acq(p)}^{T3} <_{tr'} \text{rel(m)}^{T4}$.

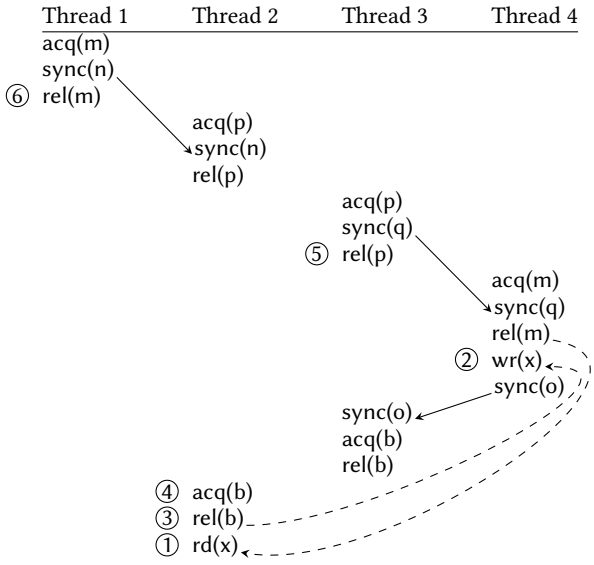


Figure 8. An execution with a predictable race and a DC-race ($\text{wr(x)} \not\prec_{DC} \text{rd(x)}$). Solid arrows are the initial DC constraints, and dashed arrows are constraints added by AddConstraints. The circled numbers (e.g., ①) represent the order of events prepended to a reordered trace by ConstructReorderedTrace. A seventh event cannot be prepended to the trace without violating the graph constraints or lock semantics.

We see that ConstructReorderedTrace is a greedy algorithm that chooses events without considering remaining implicit constraints, resulting in failure in this case.

Similarly, Figure 9(a) (i.e., Figure 9 *excluding* the sync(s6) events) shows an execution for which ConstructReorderedTrace can fail if it does *not* choose the latest event. In particular, choosing rel(p)^{T3}, or both rel(p)^{T5} and rel(m)^{T2}, prematurely (before a later legal event) will fail to construct a correctly reordered trace.

We note that for both executions, if ConstructReorderedTrace always chooses the latest event, it constructs a reordered trace successfully.

Choosing the latest event can fail. Consider Figure 9(b) (i.e., Figure 9 *excluding* the sync(s4) events). (Unmodified) ConstructReorderedTrace chooses the latest legal event at each step, which ultimately leads to failure. In particular, prepending rel(p)^{T5} and rel(m)^{T4} before the other critical sections on p and m makes failure inevitable. Although choosing the latest legal event fails, there does exist a set of legal event choices that ConstructReorderedTrace can make to construct a correctly reordered trace.

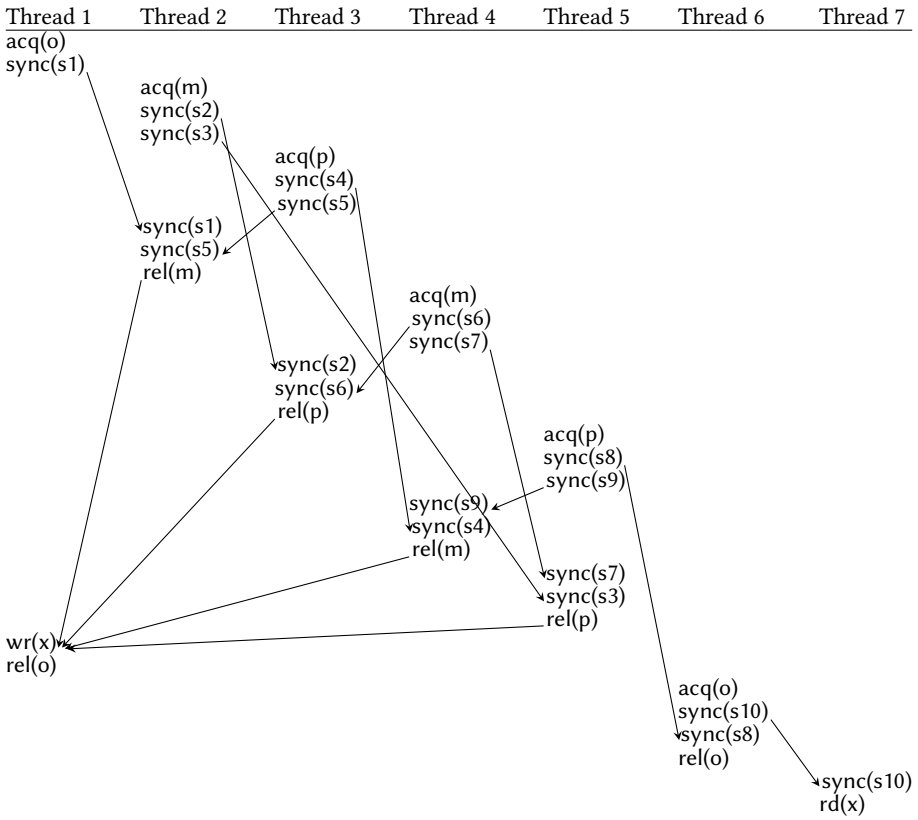


Figure 9. We use this figure to represent three different example executions: **(a)** an execution *omitting* the sync(s6) events, which has a predictable race; **(b)** an execution *omitting* the sync(s4) events, which has a predictable race; and **(c)** the unmodified execution, which has no predictable race. The arrows represent DC ordering.

AddConstraints

The execution in Figure 9(c) (i.e., the figure exactly as shown) has no predictable race. However, AddConstraints produces an acyclic graph for this trace. Intuitively, there exist implicit, mutually incompatible ordering constraints on two pairs of critical sections on different locks, for which AddConstraints does not produce a cycle of dependencies.

Although AddConstraints is unsound, ConstructReorderedTrace is sound and thus VindicateRace is sound. ConstructReorderedTrace fails to construct a correctly reordered trace for Figure 9(c), and VindicateRace returns "Don't know" (Algorithm 1).

Acknowledgments

Steve Freund provided valuable help with RoadRunner; Nima Mirzaei contributed to the implementation; Yannis Smaragdakis and Jaeheon Yi provided resources from their CP paper. The anonymous reviewers and the paper's shepherd, Yannis Smaragdakis, provided insightful feedback that improved the paper. Scott Lavigne, Miheer Vaidya, and Yufan Xu gave feedback on an early draft.

References

[1] M. Abadi, C. Flanagan, and S. N. Freund. Types for Safe Locking: Static Race Detection for Java. *TOPLAS*, 28(2):207–255, 2006.

[2] S. V. Adve and H.-J. Boehm. Memory Models: A Case for Rethinking Parallel Languages and Hardware. *CACM*, 53:90–101, 2010.

[3] W. Ahn, S. Qi, M. Nicolaides, J. Torrellas, J.-W. Lee, X. Fang, S. Midkiff, and D. Wong. BulkCompiler: High-Performance Sequential Consistency through Cooperative Compiler and Hardware Support. In *MICRO*, pages 133–144, 2009.

[4] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *CACM*, 53(2):66–75, 2010.

[5] S. Biswas, M. Cao, M. Zhang, M. D. Bond, and B. P. Wood. Lightweight Data Race Detection for Production Runs. In *CC*, pages 11–21, 2017.

[6] S. Biswas, M. Zhang, M. D. Bond, and B. Lucia. Valor: Efficient, Software-Only Region Conflict Exceptions. In *OOPSLA*, pages 241–259, 2015.

[7] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, 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. In *OOPSLA*, pages 169–190, 2006.

[8] R. L. Bocchino, Jr., V. S. Adve, S. V. Adve, and M. Snir. Parallel Programming Must Be Deterministic by Default. In *HotPar*, pages 4–9, 2009.

[9] H.-J. Boehm. How to miscompile programs with "benign" data races. In *HotPar*, 2011.

[10] H.-J. Boehm. Position paper: Nondeterminism is Unavoidable, but Data Races are Pure Evil. In *RACES*, pages 9–14, 2012.

[11] H.-J. Boehm and S. V. Adve. Foundations of the C++ Concurrency Memory Model. In *PLDI*, pages 68–78, 2008.

[12] H.-J. Boehm and S. V. Adve. You Don't Know Jack about Shared Variables or Memory Models. *CACM*, 55(2):48–54, 2012.

[13] H.-J. Boehm and B. Demsky. Outlawing Ghosts: Avoiding Out-of-Thin-Air Results. In *MSPC*, pages 7:1–7:6, 2014.

[14] M. D. Bond, K. E. Coons, and K. S. McKinley. Pacer: Proportional Detection of Data Races. In *PLDI*, pages 255–268, 2010.

[15] C. Boyapati, R. Lee, and M. Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *OOPSLA*, pages 211–230, 2002.

[16] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs. In *ASPLOS*, pages 167–178, 2010.

[17] J. Burnim, K. Sen, and C. Stergiou. Testing Concurrent Programs on Relaxed Memory Models. In *ISSTA*, pages 122–132, 2011.

[18] Y. Cai and L. Cao. Effective and Precise Dynamic Detection of Hidden Races for Java Programs. In *ESEC/FSE*, pages 450–461, 2015.

[19] M. Cao, J. Roemer, A. Sengupta, and M. D. Bond. Prescient Memory: Exposing Weak Memory Model Behavior by Looking into the Future. In *ISMM*, pages 99–110, 2016.

[20] F. Chen, T. F. Şerbănuţă, and G. Roşu. jPredictor: A Predictive Runtime Analysis Tool for Java. In *ICSE*, pages 221–230, 2008.

[21] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs. In *PLDI*, pages 258–269, 2002.

[22] J. Devietti, B. P. Wood, K. Strauss, L. Ceze, D. Grossman, and S. Qadeer. RADISH: Always-On Sound and Complete Race Detection in Software and Hardware. In *ISCA*, pages 201–212, 2012.

[23] A. Dinning and E. Schonberg. Detecting Access Anomalies in Programs with Critical Sections. In *PADD*, pages 85–96, 1991.

[24] L. Effinger-Dean, B. Lucia, L. Ceze, D. Grossman, and H.-J. Boehm. IFRit: Interference-Free Regions for Dynamic Data-Race Detection. In *OOPSLA*, pages 467–484, 2012.

[25] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: A Race and Transaction-Aware Java Runtime. In *PLDI*, pages 245–255, 2007.

[26] D. Engler and K. Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *SOSP*, pages 237–252, 2003.

[27] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective Data-Race Detection for the Kernel. In *OSDI*, pages 1–16, 2010.

[28] M. Eslamimehr and J. Palsberg. Race Directed Scheduling of Concurrent Programs. In *PPoPP*, pages 301–314, 2014.

[29] C. Flanagan and S. N. Freund. Type Inference Against Races. *SCP*, 64(1):140–165, 2007.

[30] C. Flanagan and S. N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *PLDI*, pages 121–133, 2009.

[31] C. Flanagan and S. N. Freund. Adversarial Memory for Detecting Destructive Races. In *PLDI*, pages 244–254, 2010.

[32] C. Flanagan and S. N. Freund. The RoadRunner Dynamic Analysis Framework for Concurrent Programs. In *PASTE*, pages 1–8, 2010.

[33] C. Flanagan and S. N. Freund. RedCard: Redundant Check Elimination for Dynamic Race Detectors. In *ECOOP*, pages 255–280, 2013.

[34] C. Flanagan and S. N. Freund. The FastTrack2 Race Detector. Technical report, Williams College, 2017.

[35] P. Godefroid and N. Nagappan. Concurrency at Microsoft – An Exploratory Survey. In *EC*^{2}, 2008.

[36] T. A. Henzinger, R. Jhala, and R. Majumdar. Race Checking by Context Inference. In *PLDI*, pages 1–13, 2004.

[37] J. Huang. Stateless Model Checking Concurrent Programs with Maximal Causality Reduction. In *PLDI*, pages 165–174, 2015.

[38] J. Huang, P. O. Meredith, and G. Roşu. Maximal Sound Predictive Race Detection with Control Flow Abstraction. In *PLDI*, pages 337–348, 2014.

[39] J. Huang and A. K. Rajagopalan. Precise and Maximal Race Detection from Incomplete Traces. In *OOPSLA*, pages 462–476, 2016.

[40] S. Huang and J. Huang. Speeding Up Maximal Causality Reduction with Static Dependency Analysis. In *ECOOP*, pages 16:1–16:22, 2017.

[41] B. Kasikci, C. Zamfir, and G. Candea. Data Races vs. Data Race Bugs: Telling the Difference with Portend. In *ASPLOS*, pages 185–198, 2012.

[42] B. Kasikci, C. Zamfir, and G. Candea. RaceMob: Crowdsourced Data Race Detection. In *SOSP*, pages 406–422, 2013.

[43] B. Kasikci, C. Zamfir, and G. Candea. Automated Classification of Data Races Under Both Strong and Weak Memory Models. *TOPLAS*, 37(3):8:1–8:44, May 2015.

[44] D. Kini, U. Mathur, and M. Viswanathan. Dynamic Race Prediction in Linear Time. In *PLDI*, pages 157–170, 2017.

[45] D. Kini, U. Mathur, and M. Viswanathan, 2018. Personal communication.

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

[47] D. Lee, P. M. Chen, J. Flinn, and S. Narayanasamy. Chimera: Hybrid Program Analysis for Determinism. In *PLDI*, pages 463–474, 2012.

[48] N. G. Leveson and C. S. Turner. An Investigation of the Therac-25 Accidents. *IEEE Computer*, 26(7):18–41, 1993.

[49] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Prentice Hall PTR, 2nd edition, 1999.

[50] P. Liu, O. Tripp, and X. Zhang. IPA: Improving Predictive Analysis with Pointer Analysis. In *ISSTA*, pages 59–69, 2016.

[51] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *ASPLOS*, pages 329–339, 2008.

[52] B. Lucia, L. Ceze, K. Strauss, S. Qadeer, and H.-J. Boehm. Conflict Exceptions: Simplifying Concurrent Language Semantics with Precise Hardware Exceptions for Data-Races. In *ISCA*, pages 210–221, 2010.

[53] P. Luo, D. Zou, H. Jin, Y. Du, L. Zheng, and J. Shen. DigHR: precise dynamic detection of hidden races with weak causal relation analysis. *J. Supercomputing*, 2018.

[54] J. Manson, W. Pugh, and S. V. Adve. The Java Memory Model. In *POPL*, pages 378–391, 2005.

[55] D. Marino, M. Musuvathi, and S. Narayanasamy. LiteRace: Effective Sampling for Lightweight Data-Race Detection. In *PLDI*, pages 134–143, 2009.

[56] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy. DRFx: A Simple and Efficient Memory Model for Concurrent Programming Languages. In *PLDI*, pages 351–362, 2010.

[57] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy. A Case for an SC-Preserving Compiler. In *PLDI*, pages 199–210, 2011.

[58] N. D. Matsakis and F. S. Klock, II. The Rust Language. In *HILT*, pages 103–104, 2014.

[59] F. Mattern. Virtual Time and Global States of Distributed Systems. In *Workshop on Parallel and Distributed Algorithms*, pages 215–226, 1988.

[60] M. Musuvathi and S. Qadeer. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. In *PLDI*, pages 446–455, 2007.

[61] M. Naik and A. Aiken. Conditional Must Not Aliasing for Static Race Detection. In *POPL*, pages 327–338, 2007.

[62] M. Naik, A. Aiken, and J. Whaley. Effective Static Race Detection for Java. In *PLDI*, pages 308–319, 2006.

[63] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically Classifying Benign and Harmful Data Races Using Replay Analysis. In *PLDI*, pages 22–31, 2007.

[64] H. Nishiyama. Detecting Data Races using Dynamic Escape Analysis based on Read Barrier. In *VMRT*, pages 127–138, 2004.

[65] R. O'Callahan and J.-D. Choi. Hybrid Dynamic Data Race Detection. In *PPoPP*, pages 167–178, 2003.

[66] J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. ...and region serializability for all. In *HotPar*, 2013.

[67] PCWorld. Nasdaq's Facebook Glitch Came From Race Conditions, 2012. http://www.pcworld.com/article/255911/nasdaqs_facebook_glitch_came_from_race_conditions.html.

[68] Y. Peng, B. P. Wood, and J. Devietti. PARSNIP: Performant Architecture for Race Safety with No Impact on Precision. In *MICRO*, pages 490–502, 2017.

[69] E. Pozniansky and A. Schuster. MultiRace: Efficient On-the-Fly Data Race Detection in Multithreaded C++ Programs. *CCPE*, 19(3):327–340, 2007.

[70] P. Pratikakis, J. S. Foster, and M. Hicks. LOCKSMITH: Context-Sensitive Correlation Analysis for Race Detection. In *PLDI*, pages 320–331, 2006.

[71] D. Rhodes, C. Flanagan, and S. N. Freund. BigFoot: Static Check Placement for Dynamic Race Detection. In *PLDI*, pages 141–156, 2017.

[72] M. C. Rinard and M. S. Lam. The Design, Implementation, and Evaluation of Jade. *TOPLAS*, 20:483–545, 1998.

[73] J. Roemer and M. D. Bond. An Online Dynamic Analysis for Sound Predictive Data Race Detection. Technical Report OSU-CISRC-4/16-TR01, Computer Science & Engineering, Ohio State University, 2016.

[74] M. Said, C. Wang, Z. Yang, and K. Sakallah. Generating Data Race Witnesses by an SMT-based Analysis. In *NFM*, pages 313–327, 2011.

[75] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs. In *SOSP*, pages 27–37, 1997.

[76] C. Segulja and T. S. Abdelrahman. Clean: A Race Detector with Cleaner Semantics. In *ISCA*, pages 401–413, 2015.

[77] K. Sen. Race Directed Random Testing of Concurrent Programs. In *PLDI*, pages 11–21, 2008.

[78] A. Sengupta, S. Biswas, M. Zhang, M. D. Bond, and M. Kulkarni. Hybrid Static–Dynamic Analysis for Statically Bounded Region Serializability. In *ASPLOS*, pages 561–575, 2015.

[79] T. F. Şerbănuţă, F. Chen, and G. Roşu. Maximal Causal Models for Sequentially Consistent Systems. In *RV*, pages 136–150, 2013.

[80] K. Serebryany and T. Iskhodzhanov. ThreadSanitizer – data race detection in practice. In *WBIA*, pages 62–71, 2009.

[81] K. Serebryany, A. Potapenko, T. Iskhodzhanov, and D. Vyukov. Dynamic Race Detection with LLVM Compiler. In *RV*, pages 110–114, 2012.

[82] J. Ševčík and D. Aspinall. On Validity of Program Transformations in the Java Memory Model. In *ECOOP*, pages 27–51, 2008.

[83] T. Sheng, N. Vachharajani, S. Eranian, R. Hundt, W. Chen, and W. Zheng. RACEZ: A Lightweight and Non-Invasive Race Detection Tool for Production Applications. In *ICSE*, pages 401–410, 2011.

[84] A. Singh, D. Marino, S. Narayanasamy, T. Millstein, and M. Musuvathi. Efficient Processor Support for DRFx, a Memory Model with Exceptions. In *ASPLOS*, pages 53–66, 2011.

[85] A. Singh, S. Narayanasamy, D. Marino, T. Millstein, and M. Musuvathi. End-to-End Sequential Consistency. In *ISCA*, pages 524–535, 2012.

[86] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan. Sound Predictive Race Detection in Polynomial Time. In *POPL*, pages 387–400, 2012.

[87] Z. Sura, X. Fang, C.-L. Wong, S. P. Midkiff, J. Lee, and D. Padua. Compiler Techniques for High Performance Sequentially Consistent Java Programs. In *PPoPP*, pages 2–13, 2005.

[88] U.S.–Canada Power System Outage Task Force. Final Report on the August 14th Blackout in the United States and Canada. Technical report, Department of Energy, 2004.

[89] K. Veeraraghavan, P. M. Chen, J. Flinn, and S. Narayanasamy. Detecting and Surviving Data Races using Complementary Schedules. In *SOSP*, pages 369–384, 2011.

[90] C. von Praun and T. R. Gross. Object Race Detection. In *OOPSLA*, pages 70–82, 2001.

[91] C. von Praun and T. R. Gross. Static Conflict Analysis for Multi-Threaded Object-Oriented Programs. In *PLDI*, pages 115–128, 2003.

[92] J. W. Voung, R. Jhala, and S. Lerner. RELAY: Static Race Detection on Millions of Lines of Code. In *ESEC/FSE*, pages 205–214, 2007.

[93] B. P. Wood, L. Ceze, and D. Grossman. Low-Level Detection of Language-Level Data Races with LARD. In *ASPLOS*, pages 671–686, 2014.

[94] T. Zhang, C. Jung, and D. Lee. ProRace: Practical Data Race Detection for Production Use. In *ASPLOS*, pages 149–162, 2017.

[95] P. Zhou, R. Teodorescu, and Y. Zhou. HARD: Hardware-Assisted Lockset-based Race Detection. In *HPCA*, pages 121–132, 2007.