Dependence-Aware, Unbounded Sound Predictive Race Detection*

KAAN GENÇ, Ohio State University, USA
JAKE ROEMER, Ohio State University, USA
YUFAN XU, Ohio State University, USA
MICHAEL D. BOND, Ohio State University, USA

Data races are a real problem for parallel software, yet hard to detect. Sound predictive analysis observes a program execution and detects data races that exist in some *other, unobserved* execution. However, existing predictive analyses miss races because they do not scale to full program executions or do not precisely incorporate data and control dependence.

This paper introduces two novel, sound predictive approaches that incorporate data and control dependence and handle full program executions. An evaluation using real, large Java programs shows that these approaches detect more data races than the closest related approaches, thus advancing the state of the art in sound predictive race detection.

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

Additional Key Words and Phrases: data race detection, dynamic predictive analysis

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

1 INTRODUCTION

With the rise in parallel software, *data races* represent a growing hazard. Programs with data races written in shared-memory languages including Java and C++ have weak or undefined semantics, as a result of assuming data race freedom for performance reasons [Adve and Boehm 2010; Boehm and Adve 2008; Manson et al. 2005]. Data races are culprits in real software failures, resulting in substantial financial losses and even harm to humans [Boehm 2011; Burnim et al. 2011; Cao et al. 2016; Flanagan and Freund 2010a; Kasikci et al. 2012, 2015; Leveson and Turner 1993; Lu et al. 2008; Narayanasamy et al. 2007; PCWorld 2012; Sen 2008; U.S.–Canada Power System Outage Task Force 2004; Zhivich and Cunningham 2009].

Writing scalable, data-race-free code is challenging, as is detecting data races, which occur nondeterministically depending on shared-memory interleavings and program inputs and environments. The most common approach for dealing with data races is to detect them during in-house testing

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

Authors' addresses: Kaan Genç, Ohio State University, USA, genc.5@osu.edu; Jake Roemer, Ohio State University, USA, roemer.37@osu.edu; Yufan Xu, Ohio State University, USA, xu.2882@osu.edu; Michael D. Bond, Ohio State University, USA, mikebond@cse.ohio-state.edu.

2475-1421/2019/10-ART179
https://doi.org/10.1145/3360605

using dynamic *happens-before (HB)* analysis [Elmas et al. 2007; Flanagan and Freund 2009; Intel Corporation 2016; Pozniansky and Schuster 2007; Serebryany and Iskhodzhanov 2009; Serebryany et al. 2012], which detects conflicting accesses (two memory accesses, at least one of which is a write, to the same variable by different threads) unordered by the HB partial order [Lamport 1978]. However, HB analysis misses data races when accesses *could* race in some *other* execution but are ordered by critical sections on the same lock in the observed execution.

A promising alternative to HB analysis is *sound predictive analysis*, which detects additional predictable data races from an observed execution [Chen et al. 2008; Huang et al. 2014; Huang and Rajagopalan 2016; Kini et al. 2017; Liu et al. 2016; Pavlogiannis 2019; Roemer et al. 2018; Said et al. 2011; Şerbănuţă et al. 2013; Smaragdakis et al. 2012]; an analysis is *sound* if it detects no false races (Section 2). Some predictive analyses rely on generating and solving SMT constraints, so in practice they cannot scale to full program executions and instead analyze *bounded windows* of execution, missing races between accesses that do not execute close together [Chen et al. 2008; Huang et al. 2014; Huang and Rajagopalan 2016; Liu et al. 2016; Said et al. 2011; Şerbănuţă et al. 2013] (Section 8). In contrast, *unbounded* predictive analyses avoid this limitation by detecting races based on computing a partial order weaker than HB, using analyses with linear running time in the length of the trace [Kini et al. 2017; Roemer et al. 2018]. However, these partial-order-based analyses miss predictable races because they do not incorporate precise notions of *data and control dependence*. More precisely, existing predictive partial orders do *not* encode the precise conditions for reordering memory accesses to expose a race: The reordering can change the last writer of a memory read (data dependence) if the read in turn cannot affect whether the racing accesses execute (control dependence). Encoding data and control dependence precisely in a partial order is fundamentally challenging (Section 2).

Contributions. This paper designs and evaluates new predictive analyses, making the following contributions:

- A partial order called *strong-dependently-precedes (SDP)* that improves over the highest-coverage sound partial order from prior work [Kini et al. 2017] by incorporating data dependence more precisely (Section 4).
- A proof that SDP is sound, i.e., detects no false races (Section 4.2).
- A partial order called *weak-dependently-precedes (WDP)* that improves over the previous highest-coverage partial order from prior work [Roemer et al. 2018] by incorporating data and control dependence precisely (Section 4).
- A proof that WDP is complete (sometimes called *maximal* [Huang et al. 2014; Şerbănuţă et al. 2013]), detecting all races knowable from an observed execution (Section 4.2).
- Dynamic analyses that compute SDP and WDP and detect SDP- and WDP-races (Section 5).
- An algorithm for filtering out WDP-races that are false races, by extending prior work's *vindication* algorithm [Roemer et al. 2018], yielding an overall sound approach (Section 6).
- An implementation and evaluation of SDP and WDP analyses and WDP-race vindication on benchmarked versions of real, large Java programs (Section 7). The evaluation shows that the analyses find predictable races missed by the closest related approaches [Huang et al. 2014; Kini et al. 2017; Roemer et al. 2018].

2 BACKGROUND AND MOTIVATION

Recent partial-order-based predictive analyses can scale to full program executions, enabling detection of predictable races that are millions of executed operations apart [Kini et al. 2017; Roemer et al. 2018]. However, these partial orders are fundamentally limited and miss predictable races, as this section explains. First, we introduce formalisms used throughout the paper.

2.1 Execution Model

An *execution trace tr* is a sequence of events, ordered by the total order $<_{tr}$, that represents a multithreaded execution without loss of generality, corresponding to a linearization of a sequentially consistent (SC) execution.[1] We assume every event in *tr* is a unique object (e.g., has a unique identifier), making it possible to identify the same event *e* across other, predicted traces. Each event has two attributes: (1) an identifier for the thread that executed the operation; and (2) an operation, which is one of wr(x), rd(x), acq(m), rel(m), or br, where x is a program memory location and m is a program lock. (Later we consider how to extend analyses to handle lock-free accesses and Java volatile / C++ atomic accesses.) An execution trace must be *well formed*: a thread may only acquire an unheld lock and may only release a lock it has acquired.

Each br (branch) event *b* represents an executed conditional operation—such as a conditional jump, polymorphic call, or array element access—that may be dependent on some prior read event(s) by the same thread. We assume a helper function $brDepsOn(b, r)$ exists that returns true if the value read by read event *r* may affect *b*'s outcome. An implementation could use static dependence analysis to identify reads on which a branch is data dependent. For simplicity, the paper's examples assume $brDepsOn(b, r)$ always returns true, i.e., every branch is assumed dependent on preceding reads by the same thread. Our implementation and evaluation make the same assumption, as explained later. This assumption limits the capability of predictive analysis to predict different executions; in other words, it limits the number of knowable data races from a single execution.

Three example traces are shown in Figures 1(b), 1(c), and 1(e), in which top-to-bottom order represents trace order, and column placement denotes an event's executing thread. We discuss these examples in detail later.

Two read or write events to the same variable are *conflicting*, notated $e \asymp e'$, if the events are executed by different threads and at least one is a write.

Program-order (PO) is a partial order that orders events in the same thread: $e \prec_{\text{PO}} e'$ if $e <_{tr} e'$ and the events are executed by the same thread.

The function $CS(e)$ returns the set of events in the critical section started or ended by acquire or release event *e*, including the bounding acquire and release events. $R(a)$ returns the release event ending the critical section started by acquire event *a*, and $A(r)$ returns the acquire event starting the critical section ended by release event *r*. The function $lockset(e)$ returns the set of locks held at a read or write event *e* by its executing thread.

2.2 Predictable Traces and Predictable Races

By observing one execution of a program, it is possible to predict data races in both the observed execution and some *other* executions of the program. The information present in the observed execution implies the existence of other, different executions, called *predictable traces*. To define what traces can be predicted from an observed trace, we first define several relevant concepts.

Definition 2.1 (Last writer). Given a trace *tr*, let $lastwr_{tr}(r)$ for a read event *r* be the last write event before *r* in *tr* that accesses the same variable as *r*, or $\varnothing$ if no such event exists.

To ensure that a predictable trace is feasible, each read in a predictable trace must have the same last writer as in the observed trace—with one exception: a read can have a different last writer if the read cannot take the execution down a different control-flow path than the observed execution. An example of such a read is Thread 1's rd(z) event in Figure 1(c). Next, we introduce a concept that helps in identifying reads whose last writer must be preserved in a predictable trace.

[1] Although programs with data races may violate SC [Adve and Boehm 2010; Boehm and Adve 2008; Dolan et al. 2018; Manson et al. 2005], dynamic race detection analyses (including ours) add synchronization instrumentation before accesses, generally ensuring SC.

```
int z = 0, y = 0; Object m = new Object();
new Thread(() -> { synchronized (m) {
                     int t = z;
                     y = 1;
                   } }).start();
new Thread(() -> { synchronized (m) {
                     z = 1;
                     x = 1;
                   } }).start();
new Thread(() -> { synchronized (m) {
                     int t = x;
                     if (t == 0) return;
                   }
                   int t = y;
                 } }).start();
```

(a) Java code that could lead to the executions in (b) and (c).

Thread 1	Thread 2	Thread 3
acq(m)		
rd(z)		
wr(y)		
rel(m)		
	acq(m)	
	wr(z)	
	wr(x)	
	rel(m)	
		acq(m)
		rd(x)
		br
		rel(m)
		rd(y)

(b) Execution with a predictable race

Thread 1	Thread 2	Thread 3
	acq(m)	
	wr(z)	
	wr(x)	
	rel(m)	
acq(m)		
rd(z)		
wr(y)		
		rd(y)

(c) Predictable trace of (b)

```
int z = 0, y = 0; Object m = new Object();
new Thread(() -> { synchronized (m) {
                     int t = z;
                     if (t == 0)
                       y = 1;
                   } }).start();
new Thread(() -> { synchronized (m) {
                     z = 1;
                     x = 1;
                   } }).start();
new Thread(() -> { synchronized (m) {
                     int t = x;
                     if (t == 0) return;
                   }
                   int t = y;
                 } }).start();
```

(d) Java code that could lead to the execution in (e).

Thread 1	Thread 2	Thread 3
acq(m)		
rd(z)		
br		
wr(y)		
rel(m)		
	acq(m)	
	wr(z)	
	wr(x)	
	rel(m)	
		acq(m)
		rd(x)
		br
		rel(m)
		rd(y)

(e) Execution with no predictable race

Fig. 1. Two code examples, with potential executions they could lead to. The execution in (b) has a predictable race, as demonstrated by the predictable trace in (c). The execution in (e) has no predictable race.

Definition 2.2 (Causal events). Given a trace tr, set of events S, and event e, let $causal(tr, S, e)$ be a function that returns true if at least one of the following properties holds, and false otherwise.

- e is a read, and there exists a branch event b such that $b \in S \wedge e \prec_{\text{PO}} b \wedge brDepsOn(b, e)$.
- e is a write, and there exists a read event e' such that $e' \in S \wedge e = lastwr_{tr}(e')$.
- e is a read, and there exists a write event e' such that $e' \in S \wedge e \prec_{\text{PO}} e'$ (e and e' may access different variables).

Intuitively, $causal(tr, S, e)$ tells us whether an event e could have affected some event e' in S directly. For example, $causal(tr, S, e)$ if read e may affect a branch event in S; if e writes a variable later read by an event in S; or if e reads a value that may affect a later write by the same thread in S (even if the read and write are to different variables, to account for intra-thread data flow).

We can now define a predictable trace of an observed trace, which is a trace that is definitely a feasible execution of the program, given the existence of the observed execution.

Definition 2.3 (Predictable trace). An execution trace tr' is a *predictable trace* of trace tr if tr' contains only events in tr (i.e., $\forall e : e \in tr' \implies e \in tr$) and all of the following rules hold:

Program order (PO) rule: For any events e_1 and e_2, if $e_1 \prec_{\text{PO}} e_2$, then $e_1 <_{tr'} e_2 \vee e_2 \notin tr'$.

Last writer (LW) rule: For every read event e such that $\mathit{causal}(tr, tr', e)$, $\mathit{lastwr}_{tr'}(e) = \mathit{lastwr}_{tr}(e)$. (In this context, tr' means the set of events in the trace tr'.)

Lock semantics (LS) rule: For acquire events e_1 and e_2 on the same lock, if $e_1 <_{tr'} e_2$ then $e_1 <_{tr'} R(e_1) <_{tr'} e_2$.

The PO and LW rules ensure key properties from tr also hold in tr', while the LS rule ensures that tr' is well formed. The intuition behind the LW rule is that any read that may (directly or indirectly) affect the control flow of the program must have the same last writer in predictable trace tr' as in observed trace tr.

Note that throughout the paper, partial ordering notation such as $e \prec e'$ refers to the order of e and e' in the *observed* trace tr (not a predictable trace tr').

Predictable traces do not in general contain every event in the observed trace they are based on. For the purposes of race detection, a predictable trace will *conclude with* a pair of conflicting events, which are preceded by events necessary according to the definition of predictable trace. For example, consider Figure 1(c), which is a predictable trace of Figure 1(b) that excludes Thread 1's event after wr(y). The PO rule is satisfied, and the LS rule is satisfied after reordering Thread 2 and 3's critical sections before Thread 1's. The LW rule is satisfied because rd(z) is not a causal event in tr'.

Definition 2.4 (Predictable race). An execution tr has a predictable race if a predictable trace tr' of tr has two *conflicting*, *consecutive* events: $e_1 \asymp e_2 \wedge e_1 <_{tr'} e_2 \wedge (\nexists e : e_1 <_{tr'} e <_{tr'} e_2)$.

Figure 1(b) has a predictable race, as demonstrated by the predictable trace in Figure 1(c). In contrast, Figure 1(e) has no predictable race. The difference between Figures 1(b) and 1(e) is the br event in Thread 1. Since no br exists in Thread 1 in Figure 1(b), rd(z) is not a causal event, which in turn allows the critical sections in Threads 2 and 3 to be reordered above the critical section in Thread 1, allowing wr(y) and rd(y) to be consecutive in the predictable trace. In contrast, a br event executes before wr(y) in Figure 1(e). No predictable trace of this example can exclude br without excluding wr(y); otherwise the PO rule would be violated. rd(z) is a causal event in any predictable trace where wr(y) is included, which makes it impossible to reorder the critical sections. As a result, no predictable trace exists in which wr(y) and rd(y) are consecutive. Figures 1(a) and 1(d) show source code that could lead to the executions in Figures 1(b) and 1(e), respectively. The code in Figure 1(d) has no race; in fact, any deviation of critical section ordering from Figure 1(e)'s causes wr(y) or rd(y) *not* to execute.

2.3 Existing Predictive Partial Orders

Here we overview three relations introduced in prior work, called *happens-before (HB)*, *weak-causally-precedes (WCP)*, and *doesn't-commute (DC)*, that can be computed in time linearly proportional to the length of the execution trace [Kini et al. 2017; Roemer et al. 2018]. Intuitively, each relation orders events that may not be legal to reorder in a predictable trace, so that two unordered conflicting events represent a true or potential data race (depending on whether the relation is sound). An execution trace has an *HB-race*, *WCP-race*, or *DC-race* if it contains two conflicting events that are unordered by HB, WCP, or DC, respectively.

Property	$\prec_{\text{HB}}$	$\prec_{\text{WCP}}$	$\prec_{\text{DC}}$
Same-lock critical section ordering	All	Confl.	Confl.
Orders rel to…	acq	wr/rd	wr/rd
Includes $\prec_{\text{PO}}$?	Yes	No	Yes
Left-and-right composes with $\prec_{\text{HB}}$?	Yes	Yes	No
acq(m) $\prec$ rel(m) implies rel(m) $\prec$ rel(m)?	Yes	Yes	Yes
Transitive?	Yes	Yes	Yes

Table 1. Definitions of three strict partial orders over events in an execution trace. Each order is the minimum relation satisfying the listed properties.

Definitions of relations. Table 1 gives definitions of HB, WCP, and DC by presenting their properties comparatively. The first two rows of the table say how the relations order critical sections on the same lock. HB orders all critical sections on the same lock, and it orders the first critical section's rel(m) to the second critical section's acq(m). WCP and DC order only *conflicting* critical sections (critical sections on the same lock containing conflicting events), and they order from the first critical section's rel(m) to the second critical section's conflicting access event. That is, if r_1 and r_2 are release events on the same lock such that $r_1 \prec_{tr} r_2$, and e_1 and e_2 are conflicting events ($e_1 \asymp e_2$) such that $e_1 \in CS(r_1) \wedge e_2 \in CS(r_2)$, then $r_1 \prec_{\text{WCP}} e_2$ and $r_1 \prec_{\text{DC}} e_2$. The intuition behind these properties of WCP and DC is that non-conflicting critical sections can generally be reordered in a predictable trace; and even in the case of conflicting critical sections, the second critical section can be "reordered" so that it executes only up to its conflicting access and the first critical section does not execute at all in the predictable trace.

The next two table rows show whether the relations include PO or compose with HB. HB and DC include (i.e., are supersets of) PO: if $e_1 \prec_{\text{PO}} e_2$, then $e_1 \prec_{\text{HB}} e_2$ and $e_1 \prec_{\text{DC}} e_2$. In contrast, WCP does not include PO but instead *composes with* the stronger HB: if $e_1 \prec_{\text{WCP}} e_2 \prec_{\text{HB}} e_3$ or $e_1 \prec_{\text{HB}} e_2 \prec_{\text{WCP}} e_3$, then $e_1 \prec_{\text{WCP}} e_2$. (By virtue of being transitive, HB composes with itself.) The intuition behind including or composing with PO (a subset of HB) is that PO-ordered events cannot be reordered in a predictable trace. The intuition behind WCP composing with HB, in essence, is to avoid predicting traces that violate the LS rule of predictable traces. As a result, WCP is sound while DC is unsound, as we will see.

The last two rows show properties shared by all relations. First, if two critical sections on the same lock $a_1 \prec_{\text{PO}} r_1 \prec_{tr} a_2 \prec_{\text{PO}} r_2$ are ordered at all (meaning simply $a_1 \prec_* r_2$ because all relations minimally compose with PO), then their release events are ordered ($r_1 \prec_* r_2$). Second, all of the relations are transitive. As a result of being transitive, antisymmetric, and irreflexive, all of the relations are strict partial orders.

Example. As an example of WCP and DC ordering, consider the execution in Figure 2(b). Both relations order Thread 1's rel(m) to Thread 2's wr(x) because the critical sections on m contain conflicting accesses to x. By WCP's composition with HB (and thus PO) and DC's inclusion of PO, both WCP and DC transitively order rd(x) to wr(x) and wr(y) to rd(y) (wr(y) $\prec_{\text{WCP}}$ rd(y) and wr(y) $\prec_{\text{DC}}$ rd(y)), so the execution has no WCP- or DC-races.

Soundness and completeness. A relation or analysis is *sound* if it detects a race only for an execution trace with a predictable race or deadlock.[2] A relation or analysis is *complete* if it detects a race for every execution trace with a predictable race.

[2]A trace has a *predictable deadlock* if there exists a valid reordering with a deadlock. We define soundness to include predictable deadlocks because prior work's WCP relation [Kini et al. 2017] and our SDP relation are sound in this way.

WCP (and HB) are sound: a WCP-race (HB-race) indicates a predictable race or deadlock. DC is unsound: an execution with a DC-race may have no predictable race or deadlock. However, prior work shows that DC-races are generally true predictable races in practice, and an efficient *vindication* algorithm can verify DC-races as predictable races by computing additional constraints and building a predictable trace exposing the race [Roemer et al. 2018]. Later in the paper, we provide more details about vindication, when introducing a new relation that (like DC) is unsound and makes use of a vindication algorithm.

2.4 Limitations of Existing Predictive Partial Orders

WCP and DC analyses are the state of the art in detecting as many predictable races as possible using online, unbounded analysis [Kini et al. 2017; Roemer et al. 2018]. However, WCP and DC are *incomplete*, failing to detect some predictable races. WCP and DC are overly strict because they order all conflicting accesses, conservatively ruling out some predictable traces that still preserve the last writer of each causal read. This strictness arises from imprecise handling of data and control dependence:

Data dependence: WCP and DC order all conflicting accesses, which is imprecise because the order of a write–write or read–write conflict does not necessarily need to be preserved to satisfy the last-writer (LW) rule of predictable traces.

Consider the executions in Figures 2(a) and 2(b), in which WCP and DC order rd(x) to wr(x). WCP and DC's read–write ordering assumes that no predictable trace exists where a pair of conflicting accesses are reordered. This rationale works for Figure 2(a), in which no predictable race exists. However, conflicting accesses may be reordered as long as the LW rule of predictable traces is satisfied. Figure 2(c) is a predictable trace of Figure 2(b) that reorders the critical sections and exposes a race on y.

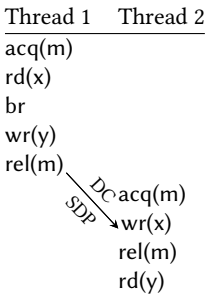
Similarly for write–write conflicts, consider Figure 3(a), in which WCP and DC order the two wr(x) events, leading to no WCP- or DC-race on accesses to y. However, the wr(x) events can be reordered, as the predictable trace in Figure 3(b) shows, exposing a race. (The reader can ignore Figure 3(c) until Section 4.)

It is difficult to model read–write and write–write dependencies more precisely using a partial order. In the case of a read–write dependency, the accesses can be reordered *as long as the read cannot impact a branch's outcome in the predictable trace* (i.e., the read is not a causal event in the predictable trace, or is not part of the predictable trace). For a write–write dependency, the accesses can be reordered *as long as they do not change a causal read's last writer in the predictable trace.* Incorporating either kind of constraint into a partial order is challenging but also desirable because partial orders can be computed efficiently.

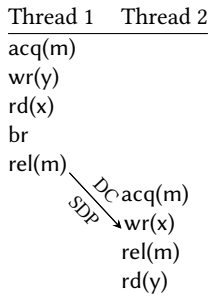
Control dependence: WCP and DC order true (write–read) dependencies even when the read may not affect a branch outcome that affects whether a race happens.

Figure 4(a) shows an execution with a predictable race, as the predictable trace in Figure 4(b) demonstrates. Note that in Figure 4(b), rd(x) has a different last writer than in Figure 4(a), but the lack of a following br event means that rd(y) is still guaranteed to happen (i.e., rd(x) is not a causal event). A variant of this example is in Figure 4(c), which has a branch event dependent on a read outcome, but the branch can be absent from a predictable trace demonstrating a predictable race (Figure 4(d)).

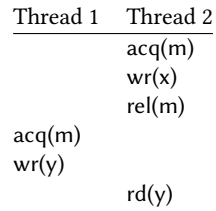
WCP and DC miss the predictable races in Figures 4(a) and 4(c) by conservatively assuming that any event after a rd(x) may be control dependent on the read value. Similarly, WCP and DC miss the predictable race in Figure 5, which our implementation found in the Java program pmd (Section 7). Essentially, WCP and DC conservatively assume that a dependent branch immediately



(a) WDP-race but no WCP-, DC-, or SDP-race. No predictable race exists.

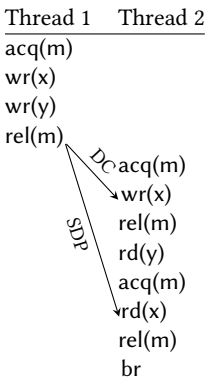


(b) WDP-race but no WCP-, DC-, or SDP-race. A predictable race exists, as (c) shows.

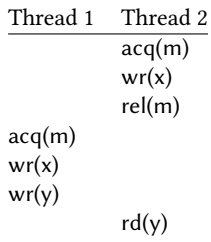


(c) Predictable trace of (b) showing that (b) has a predictable race

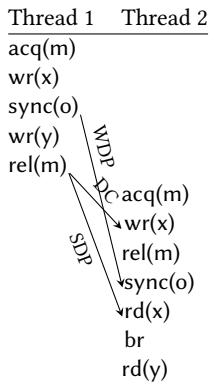
Fig. 2. Executions showing WCP and DC's overly strict handling of read–write dependencies. Edges represent ordering, labeled using the weakest applicable relation(s) (and omitting ordering established by HB alone), implying ordering by strictly stronger relations (see Figure 6(a) for comparison of relations).



(a) Execution with SDP- and WDP-race but no WCP- or DC-race



(b) Valid reordering of (a) showing that (a) has a predictable race



(c) Execution with WDP-race but no SDP-race or predictable race

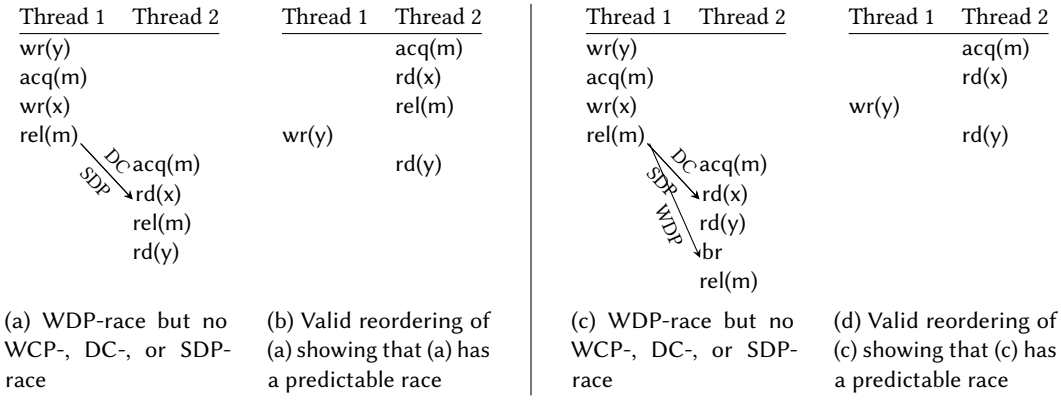
Fig. 3. Executions showing WCP and DC's overly strict handling of write–write dependencies. sync(o) is an abbreviation for the sequence acq(o); rd(oVar); br; wr(oVar); rel(o).

follows each read. This limitation is unsurprising considering the challenge of modeling control dependencies using a partial order. In particular, it is difficult for a partial order to model the fact that *a read must have the same last writer only if the read may affect a branch in the predictable trace.*

This work develops partial orders that are weaker than WCP and DC and thus predict more races. At the same time, these new partial orders retain key properties of the existing relations: WCP's soundness and DC's amenability to a vindication algorithm that ensures soundness, respectively.

3 OVERVIEW

The previous section introduced prior work's weak-causally-precedes (WCP) [Kini et al. 2017] and doesn't-commute (DC) [Roemer et al. 2018], and explained their limitations that lead to missing predictable races. The next two sections introduce new relations and analyses that overcome these



(a) WDP-race but no WCP-, DC-, or SDP-race

(b) Valid reordering of (a) showing that (a) has a predictable race

(c) WDP-race but no WCP-, DC-, or SDP-race

(d) Valid reordering of (c) showing that (c) has a predictable race

Fig. 4. Executions showing WCP and DC's overly strict handling of control dependencies.

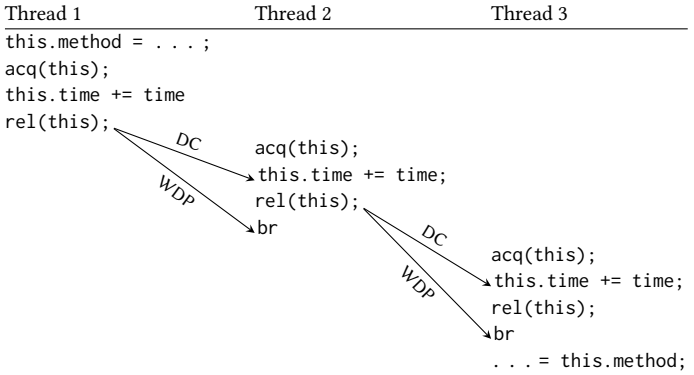


Fig. 5. A predictable race in the Java program pmd that was detected by WDP, but not WCP, DC, or SDP. Note the transitive edges formed by DC, which WDP avoids as the branches are outside the critical sections. The code has been simplified and abbreviated.

limitations. Section 4 introduces the *strong-dependently-precedes (SDP)* and *weak-dependently-precedes (WDP)* relations, which are weaker than WCP and DC, respectively. Section 5 presents online dynamic analyses for computing SDP and WDP and detecting SDP- and WDP-races.

4 NEW DEPENDENCE-AWARE PREDICTIVE RELATIONS

This section introduces new partial orders called *strong-dependently-precedes (SDP)* and *weak-dependently-precedes (WDP)* that overcome the limitations of prior work's predictive relations [Kini et al. 2017; Roemer et al. 2018] (Section 2.4) by incorporating more precise notions of data and control dependence.

4.1 The SDP and WDP Partial Orders

SDP is weaker than WCP[3] by not ordering write–write conflicts, based on the insight that writes can be unordered unless they can affect the outcome of a read, but write–read and read–write

[3]It may seem confusing that SDP is *weaker* than WCP. SDP is so named because it is stronger than WDP, while WCP is so named because it is weaker than prior work's *causally-precedes (CP)* [Smaragdakis et al. 2012].

	Prior work			This paper	
Property	$\prec_{HB}$	$\prec_{WCP}$	$\prec_{DC}$	$\prec_{SDP}$	$\prec_{WDP}$
Same-lock critical section ordering	All	Confl.	Confl.	Confl.	Last wr–rd only
Orders rel to…	acq	wr/rd	wr/rd	wr/rd or to next rd*	Next br
Includes $\prec_{PO}$?	Yes	No	Yes	No	Yes
Left-and-right composes with $\prec_{HB}$?	Yes	Yes	No	Yes	No
acq(m) $\prec$ rel(m) implies rel(m) $\prec$ rel(m)?	Yes	Yes	Yes	Yes	Yes
Transitive?	Yes	Yes	Yes	Yes	Yes

Table 2. Definitions of five strict partial orders over events in an execution trace. Each order is the minimum relation satisfying the listed properties. This table adds columns $\prec_{SDP}$ and $\prec_{WDP}$ to Table 1 (page 6).
* As the text explains, SDP adds release–access ordering for write–read and read–write conflicts, and adds ordering from the release to the next read for write–write conflicts.

ordering already handles that ordering soundly. WDP only orders the last writer of a read to a branch that depends on that read, which is the only reordering constraint that does not lead to missing predictable races.

Table 2 defines SDP and WDP. The table shows that SDP is like WCP and WDP is like DC except in how they order critical sections (first two table rows).

The SDP relation. SDP only orders conflicting critical sections when one critical section contains a read. Like WCP, SDP orders the first critical section's rel(m) to the second critical section's access. The intuition behind this property is that write–write conflicts generally do not impose any limitations on what traces can be predicted. Figure 3(a) shows an example in which two conflicting writes can be safely reordered in a predictable trace.

However, ignoring write–write conflicts altogether would be unsound, as Figure 3(c) shows: the execution has no predictable race. To ensure soundness, SDP handles write–write conflicts by ordering the first critical section to the second thread's next *read* to the same variable.

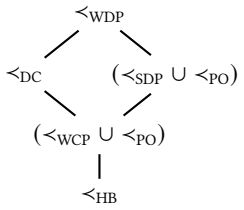
More formally, SDP handles conflicting critical sections as follows. If r_1 and r_2 are release events on the same lock, $r_1 \prec_{tr} r_2$, e_1 and e_2 are write events and e_3 is a read event, $e_1 \asymp e_2$, $e_1 \asymp e_3$, $e_2 \prec_{PO} e_3$, $e_1 \in CS(r_1)$, and $e_2 \in CS(r_2)$, then $r_1 \prec_{SDP} e_3$.

SDP addresses a limitation of WCP via more precise handling of data dependencies. SDP certainly does not address all imprecise data dependencies (e.g., read–write dependencies), and it does not address control dependence. SDP is the weakest known sound partial order.

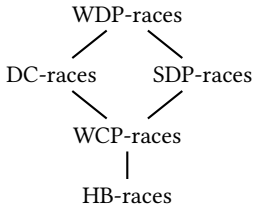
The WDP relation. A separate but worthwhile goal is to develop a partial order that is weaker than DC but produces few false positives so that it is practical to vindicate potential races. WDP achieves this goal and is in fact complete, detecting all predictable races. WDP orders the last writer of each read to the earliest branch that depends on that read (and orders no other conflicting critical sections). The intuition behind this behavior is that the only constraint that is universally true for all predictable traces is that the last writer of a read must not occur after the read if there is a branch that depends on the read.

More formally, if r_1 and r_2 are releases on the same lock, $e_1 \in CS(r_1)$, $e_2 \in CS(r_2)$, $e_1 = lastwr_{tr}(e_2)$, $e_2 \prec_{PO} b$, and $brDepsOn(b, e_2)$, then $r_1 \prec_{WDP} b$.

Unlike DC, WDP integrates control dependence by ordering the write's critical section to the first branch dependent on the read. WDP does not model *local* data dependencies, where a read affects the value written by a write in the same thread. As a result, WDP may find false races, but Section 6 describes a method for ruling out such false races. These properties make WDP complete (as we show). WDP is the strongest known complete partial order.



(a) Weaker above stronger relations



(b) Supersets above subsets

Fig. 6. Lattices showing the relationships among the relations and corresponding kinds of races. Only WCP and SDP do not include PO and thus do not in general order events within the same thread, but this property is irrelevant for comparing relation strength because same-thread accesses cannot race in any case, so the relation lattice uses $(\prec_{WCP} \cup \prec_{PO})$ and $(\prec_{SDP} \cup \prec_{PO})$ to make the relations more directly comparable. (WCP and SDP analyses in fact detect races by comparing access events using $(\prec_{WCP} \cup \prec_{PO})$ and $(\prec_{SDP} \cup \prec_{PO})$, respectively.)

	Sound?	Complete?
HB	Yes [Lamport 1978]	No (Fig. 2(b), 3(a), 4(a), 4(c))
WCP	Yes [Kini et al. 2017]	No (Fig. 2(b), 3(a), 4(a), 4(c))
DC	No [Roemer et al. 2018]	No (Fig. 2(b), 3(a), 4(a), 4(c))
SDP	Yes (Section 4.2)	No (Fig. 2(b), 4(a), 4(c))
WDP	No (Fig. 2(a) and 3(c))	Yes (Section 4.2)

Table 3. Soundness and completeness of each relation.

SDP- and WDP-races. Unlike WCP and DC, SDP and WDP do not inherently order all conflicting accesses that hold a common lock. Thus the following definition of SDP- and WDP-races explicitly excludes conflicting accesses holding a common lock.

A trace has a *SDP-race* (or *WDP-race*) if it has two conflicting events unordered by SDP (WDP) that hold no lock in common. That is, tr has an SDP-race (WDP-race) on events e and e' if $e <_{tr} e'$, $e \asymp e'$, $e \not\prec_{SDP} e'$ ($e \not\prec_{WDP} e'$), and $lockset(e) \cap lockset(e') = \emptyset$.

Examples. To illustrate SDP and WDP, we refer back to examples from pages 8–9.

The executions in Figures 2(a) and 2(b) have no SDP-races: SDP orders the read–write conflicts. In contrast, these executions have WDP-races: there is no cross-thread WDP ordering because the executions have no lock-protected write–read conflicts.

Figure 3(a) has SDP- and WDP-races. SDP and WDP do not order the write–write conflict on x. Nor does WDP order events on the write–read conflict on x, since Thread 1's wr(x) is not the last writer of rd(x).

Figure 3(c), which has a WDP-race but no SDP-race or predictable race, shows the need for SDP's release–read ordering for write–write conflicts.

The executions in Figures 4(a) and 4(c) have no SDP-race since SDP does not take branches into account. On the other hand, both executions have WDP-races: Figure 4(a) has no branch dependent on the read, and Figure 4(c) has a branch, but it occurs after rd(y).

WDP analysis discovers the predictable race in Figure 5. In this case, the fact that there is no branch within the critical section allows WDP to avoid creating an unnecessary transitive edge that otherwise would hide the race.

4.2 Soundness and Completeness

Figure 6 and Table 3 illustrate the relationships among the different relations and corresponding race types. SDP never misses a race that WCP finds, and WDP never misses a race that DC or SDP finds. SDP is sound but incomplete (never reports a false race but may miss predictable races), while WDP is unsound but complete (may report false races but never misses a predictable race).

Here we prove that SDP is sound and WDP is complete. The proofs are manual and have not been verified by a theorem prover.

Theorem (SDP soundness). *If an execution trace has a SDP-race, then it has a predictable race or a predictable deadlock.*

Proof. We define $\prec_{\text{SDP}(i)}$ to be a variant of $\prec_{\text{WCP}}$ and $\prec_{\text{SDP}}$ that orders critical sections like SDP for the first i conflicting writes in tr, and orders critical sections like WCP otherwise (Table 2). Formally, for conflicting events e and e' in critical sections on the same lock, $e \prec_{\text{SDP}(i)} e'$ if either:

- e or e' is a read; or
- there are i many conflicting pairs of write events (i.e., two conflicting write events without an intervening conflicting write event) before the write pair (e, e') (a write pair (w, w') is before a write pair (e, e') if $w' <_{tr} e'$).

Note that $\prec_{\text{SDP}(0)} \equiv \prec_{\text{WCP}}$ and $\prec_{\text{SDP}(\infty)} \equiv \prec_{\text{SDP}}$.

The rest of the proof proceeds by induction to show that SDP(i) is sound for all i, i.e., if an execution trace has an SDP(i)-race, then it has a predictable race or deadlock.

Base case: Since $\prec_{\text{SDP}(0)} \equiv \prec_{\text{WCP}}$ and WCP is sound [Kini et al. 2017], SDP(0) is sound.

Inductive step: Let σ be an execution trace whose first SDP(i)-race is between events e_1 and e_2, where *first* means that e_2 is as early as possible in σ, and among SDP(i)-races whose second event is e_2, e_1 is as late as possible in σ. Proceeding with proof by contradiction, suppose σ has no predictable race or deadlock.

Now let tr be a trace equivalent to σ that moves all events between e_1 and e_2 that are not HB ordered with both events, to outside of e_1 and e_2, and additionally removes all events after e_2. Specifically:

- if $e_1 <_\sigma e \wedge e_1 \not\prec_{\text{HB}} e \prec_{\text{HB}} e_2$, move e before e_1 in tr;
- if $e_1 <_\sigma e <_\sigma e_2 \wedge e \not\prec_{\text{HB}} e_2$, omit e from tr;
- if $e_2 <_\sigma e$, omit e from tr.

Thus the last event in tr is e_2. Like σ, tr's first SDP(i)-race is between events e_1 and e_2, and tr has no predictable race or deadlock. That is, $e_1 <_{tr} e_2$, $e_1 \asymp e_2$, and $e_1 \not\prec_{\text{SDP}} e_2$.

Because tr has no predictable race or deadlock, $e_1 \prec_{\text{SDP}(i-1)} e_2$ by the induction hypothesis. Because of the disparity between $e_1 \prec_{\text{SDP}(i-1)} e_2$ and $e_1 \not\prec_{\text{SDP}(i)} e_2$, it must be that $l \prec_{\text{SDP}(i-1)} w'$ and $l \not\prec_{\text{SDP}(i)} w'$, where w' is the ith conflicting write in tr, w is the latest write before w' such that $w \asymp w'$, and l is the outermost release event of a critical section containing w that releases the same lock as any critical section containing w'.

If there is a read event r that reads the same variable as w and w' such that $w <_{tr} r <_{tr} e_2$, then either

- $w' \asymp r \wedge r <_{tr} w'$, in which case $e_1 \prec_{\text{HB}} r \prec_{\text{SDP}(i)} w' \prec_{\text{HB}} e_2$;
- $w' \asymp r \wedge w' <_{tr} r$, in which case $e_1 \prec_{\text{HB}} w' \prec_{\text{SDP}(i)} r \prec_{\text{HB}} e_2$;
- $w \asymp r \wedge w' \prec_{\text{PO}} r$, in which case $e_1 \prec_{\text{HB}} l \prec_{\text{SDP}(i)} r \prec_{\text{HB}} e_2$; or
- $w \asymp r \wedge r \prec_{\text{PO}} w'$, in which case $e_1 \prec_{\text{HB}} w \prec_{\text{SDP}(i)} r \prec_{\text{HB}} e_2$.

In each of these cases, by SDP(i)'s composition with HB, $e_1 \prec_{\text{SDP}(i)} e_2$, a contradiction.

Therefore there is no read event r that reads the same variable as w and w' such that $w <_{tr} r$. Any read event that reads the same variable as w and w' must occur *before* w.

Now consider the trace tr' that is equivalent to tr except:

- w' is replaced by a wr(x) event, where x is a brand-new variable not used in tr.
- For every read r in tr that reads the same variable as w', an event r' is appended immediately after r such that r' is a rd(x) event and $r \prec_{\text{PO}} r'$ in tr'.

Note that the SDP($i-1$) ordering for tr' is the same as the SDP(i) ordering for tr: the rd(x)–wr(x) conflicts introduce the same ordering in tr' as the original read–write conflicts between w' and its prior reads in tr, and tr' does not contain the write–write conflict on w and w' found in tr. Thus in tr', $e_1 \not\prec_{\text{SDP}(i-1)} e_2$. By the induction hypothesis, tr' has a predictable race or deadlock. Let tr'' be a predictable trace of tr' that exposes a race or deadlock. However, if we modify tr'' by removing the rd(x) events and replacing the wr(x) event with w', the resulting trace is a predictable trace of tr that exposes a race or deadlock. Thus tr has a predictable race or deadlock, which is a contradiction.

Thus for all i, SDP(i) is sound. Since $\prec_{\text{SDP}(\infty)} \equiv \prec_{\text{SDP}}$, therefore SDP is sound. □

Theorem 4.1 (WDP completeness). *If an execution trace has a predictable race, then it has a WDP-race.*

To prove the theorem, we use the following helper lemma:

Lemma 4.2 (WDP-ordered events cannot be reordered). *Given an execution trace tr, for any events e_1 and e_2 in tr such that $e_1 \prec_{WDP} e_2$, let tr' be a reordering of tr where e_1 and e_2 have been reordered: either $e_2 <_{tr'} e_1$ or $e_2 \in tr' \wedge e_1 \notin tr'$. Then, tr' must not be a valid predictable trace of tr.*

The overall proof strategy is analogous to a corresponding proof for DC [Roemer et al. 2018], so we have relegated the proof of Lemma 4.2 to the extended arXiv version [Genç et al. 2019].

Proof of Theorem 4.1. Let us prove this theorem by contradiction. Let tr be a trace with a predictable race on conflicting events e_1 and e_2 such that $e_1 <_{tr} e_2$, but no WDP-race. Let tr' be a predictable trace of tr in which e_1 and e_2 are consecutive: $e_1 <_{tr'} e_2$ and $\nexists e : e_1 <_{tr'} e <_{tr'} e_2$.

Applying the definition of a WDP-race (Section 4), either $e_1 \prec_{\text{WDP}} e_2$ or $lockset(e_1) \cap lockset(e_2) \neq \emptyset$. If $lockset(e_1) \cap lockset(e_2) \neq \emptyset$, then tr' violates the LS rule of predictable traces.

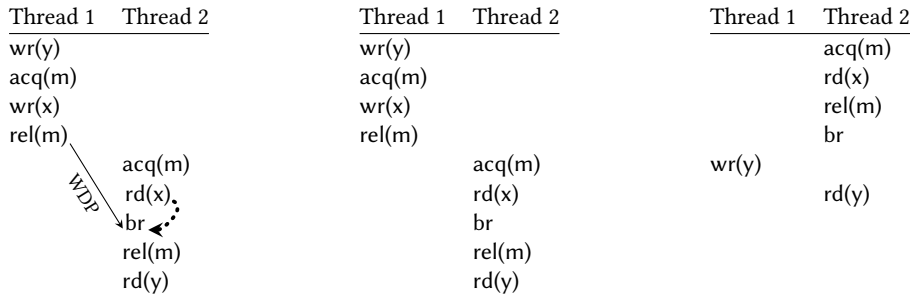
Thus $e_1 \prec_{\text{WDP}} e_2$. By the definition of a predictable race, e_1 and e_2 must be read or write events, and must be on different threads. As a result, the WDP ordering between e_1 and e_2 cannot be established by WDP conflicting critical section ordering or "$\text{acq(m)} \prec_{\text{WDP}} \text{rel(m)} \implies \text{rel(m)} \prec_{\text{WDP}} \text{rel(m)}$," which require e_1 to be a release, and not by PO since the events are on different threads. Therefore, $e_1 \prec_{\text{WDP}} e_2$ by WDP transitivity, so there must exist an event e such that $e_1 \prec_{\text{WDP}} e \prec_{\text{WDP}} e_2$.

Since e_1 and e_2 are consecutive in tr', either $e <_{tr'} e_1$, $e_2 <_{tr'} e$, or $e_2 \in tr' \wedge e \notin tr'$. By Lemma 4.2, any of these possibilities implies tr' is an invalid predictable trace of tr, a contradiction. □

4.3 Using Precise Dependence Information

Up to this point, we have assumed that a branch event depends on every preceding read event in the same thread, meaning that the condition $brDepsOn(b, e_2)$ in WDP's handling of write–read critical sections holds for every read e_2 and branch b. This assumption is needed unless static control dependence information is available from conservative static program analysis (e.g., [Ferrante et al. 1987; Huang and Huang 2017]). We tried out one kind of static analysis to compute static control dependencies but found it provided no benefit, so our experiments do not use it (Section 7). Here we show some examples of how WDP uses static control dependence information if it is available.

Figure 7 shows two executions that differ only in whether precise static control dependency information is available. Figure 7(a) has no control dependency information available, so each



(a) No WDP-race exists if the branch event is dependent on the prior read.

(b) A WDP-race exists if the branch event does not depend on the prior read.

(c) A valid reordering of (b) demonstrating that (b) has a race.

Fig. 7. Example executions that differ only in the static control dependencies between branches and reads. Dotted edges indicate reads that a branch depends on, i.e., $brDepsOn(b, e)$. If precise control dependence information rules out read–branch dependencies, WDP can find additional races, such as the race on y in (b).

branch is conservatively dependent on all prior reads, or the information is available but the branch outcome *may depend* on the prior read. Figure 7(b) has control dependency information that says that the branch outcome does *not* depend on the prior read. As a result, Figure 7(b) has weaker WDP ordering than Figure 7(a), leading to a detected WDP-race in Figure 7(b) only.

5 SDP AND WDP ANALYSES

SDP analysis and *WDP analysis* are new online dynamic program analyses that compute SDP and WDP and detect SDP- and WDP-races, respectively. Algorithms 1 and 2 show SDP and WDP analyses, respectively, for each kind of event. This section's notation and terminology follow the WCP and DC papers' to some extent [Kini et al. 2017; Roemer et al. 2018].

Both algorithms show the differences relative to prior analyses (SDP versus WCP and WDP versus DC) by labeling lines with "+" to show logic added by our analyses and "–" with grayed-out text to show lines removed by our analyses. Algorithm 1 shows that SDP analysis requires few changes to WCP analysis. These changes are for tracking write–write conflicts to add ordering when a future read is detected on the second write's thread. In addition, SDP analysis avoids reporting write–write races for writes in critical sections on the same lock by using $\mathbb{L}^w_{l,x}$ at line 22.

Algorithm 2 shows that WDP analysis makes several significant changes to DC analysis. These changes are primarily to deal with branches, by recording information about write–read dependencies at read events (lines 13–16) and using the recorded information at branch events (lines 26–27). Unlike DC analysis, WDP analysis does not establish ordering at any conflicting accesses, and it never needs to track ordering from a read to another access (since it detects only write–read conflicts). In addition, WDP analysis ensures it does not report races on accesses in critical sections on the same lock, by maintaining and using the locksets $L^w_{x,t'}$ and $L^r_{x,t'}$ when checking for races.

Analysis details. In both SDP and WDP analyses, the procedural parameters t and l are the current thread and lock; L is the set of locks held by the thread performing the current event; R and W are the sets of variables that were read and written in the ending critical section on l; and e represents the current read or branch event (for detecting branch dependencies).

The analysis uses *vector clocks* [Mattern 1988] to represent logical SDP or WDP time. A vector clock $C : Tid \mapsto \mathcal{N}$ maps each thread to a nonnegative integer. Operations on vector clocks

Algorithm 1 SDP analysis at each event type, with differences from WCP analysis

1: **procedure** ACQUIRE(t, l)
2: $\quad \mathbb{H}_t \leftarrow \mathbb{H}_t \sqcup \mathbb{H}_l$
3: $\quad \mathbb{C}_t \leftarrow \mathbb{C}_t \sqcup \mathbb{C}_l$
4: $\quad$ **foreach** $t' \neq t$ **do** $Acq_l(t').Enque(\mathbb{C}_t[t := \mathbb{H}_t(t)])$
5: **procedure** RELEASE(t, l, R, W)
6: $\quad$ **while** $Acq_l(t).Front() \sqsubseteq \mathbb{C}_t[t := \mathbb{H}_t(t)]$ **do**
7: $\quad\quad Acq_l(t).Deque()$
8: $\quad\quad \mathbb{C}_t \leftarrow \mathbb{C}_t \sqcup Rel_l(t).Deque()$
9: $\quad$ **foreach** $x \in R$ **do** $\mathbb{L}^r_{l,x} \leftarrow \mathbb{L}^r_{l,x} \sqcup \mathbb{H}_t$
10: $\quad$ **foreach** $x \in W$ **do** $\mathbb{L}^w_{l,x} \leftarrow \mathbb{L}^w_{l,x} \sqcup \mathbb{H}_t$
11: $\quad \mathbb{H}_l \leftarrow \mathbb{H}_t$
12: $\quad \mathbb{C}_l \leftarrow \mathbb{C}_t$
13: $\quad$ **foreach** $t' \neq t$ **do** $Rel_l(t').Enque(\mathbb{H}_t)$
14: $\quad \mathbb{H}_t(t) \leftarrow \mathbb{H}_t(t) + 1$
15: **procedure** READ(t, x, L)
16: + $\quad \mathbb{C}_t \leftarrow \mathbb{C}_t \sqcup \mathbb{B}_{t,x}$ ▷ Apply prior write–write conflict
17: $\quad \mathbb{C}_t \leftarrow \mathbb{C}_t \sqcup_{l \in L} \mathbb{L}^w_{l,x}$
18: $\quad$ **check** $\mathbb{W}_x \sqsubseteq \mathbb{C}_t[t := \mathbb{H}_t(t)]$ ▷ Write–read race?
19: $\quad \mathbb{R}_x(t) \leftarrow \mathbb{H}_t(t)$
20: **procedure** WRITE(t, x, L)
21: $\quad \mathbb{C}_t \leftarrow \mathbb{C}_t \sqcup_{l \in L} \mathbb{L}^r_{l,x}$
– $\quad \mathbb{C}_t \leftarrow \mathbb{C}_t \sqcup_{l \in L} \mathbb{L}^w_{l,x}$
22: $\quad$ **check** $\mathbb{W}_x \sqsubseteq \mathbb{C}_t[t := \mathbb{H}_t(t)] \sqcup_{l \in L} \mathbb{L}^w_{l,x}$ ▷ Write–write race?
23: + $\quad \mathbb{B}_{t,x} \leftarrow \sqcup_{l \in L} \mathbb{L}^w_{l,x}$ ▷ Records write–write conflict for future read
24: $\quad$ **check** $\mathbb{R}_x \sqsubseteq \mathbb{C}_t[t := \mathbb{H}_t(t)]$ ▷ Read–write race?
25: $\quad \mathbb{W}_x(t) \leftarrow \mathbb{H}_t(t)$
26: **procedure** BRANCH(t, L)
27: $\quad$ **skip** ▷ No analysis at branch events

are pointwise comparison ($C_1 \sqsubseteq C_2 \iff \forall t.C_1(t) \leq C_2(t)$) and pointwise join ($C_1 \sqcup C_2 \equiv \lambda t.\max(C_1(t), C_2(t))$):

Both analyses maintain the following state:

- $\mathbb{C}_t$ is a vector clock that represents the current SDP or WDP time for thread t.
- $\mathbb{R}_x$ and $\mathbb{W}_x$ are vector clocks that represent the SDP or WDP time of the last reads and writes to x.
- $Acq_l(t)$ and $Rel_l(t)$ (SDP) and $Acq_{l,t'}(t)$ and $Rel_{l,t'}(t)$ (WDP) are queues of vector clocks that help compute the "acq(m) $\prec$ rel(m) implies rel(m) $\prec$ rel(m)" property (Table 2).

In addition, SDP analysis maintains the following state:

- $\mathbb{H}_t$ is a vector clock that represents the current HB time for thread t. $\mathbb{C}_t[t := \mathbb{H}_t(t)]$, which evaluates to a vector clock with every element equal to $\mathbb{C}_t$ except that element t is equal to $\mathbb{H}_t(t)$, represents $\prec_{\text{SDP}} \cup \prec_{\text{PO}}$.
- $\mathbb{B}_{t,x}$ is a vector clock that represents the SDP time of a release event e of a critical section on lock m containing a write event w to x such that a later write event w' to x by t conflicts with the write and $m \in lockset(w) \cap lockset(w')$.
- $\mathbb{L}^r_{l,x}$ and $\mathbb{L}^w_{l,x}$ are vector clocks that represent the HB times of critical sections on l that read and wrote x, respectively.

Algorithm 2 WDP analysis at each event type, with differences from DC analysis

```
1:  procedure ACQUIRE(t, l)
2:      foreach t' ≠ t do Acq_{l,t'}(t).Enque(ℂ_t)
3:  procedure RELEASE(t, l, R, W)
4:      foreach t' ≠ t do
5:          while Acq_{l,t}(t').Front() ⊑ ℂ_t do
6:              Acq_{l,t}(t').Deque()
7:              ℂ_t ← ℂ_t ⊔ Rel_{l,t}(t').Deque()
8:      foreach x ∈ W do 𝕃_{l,x} ← ℂ_t                ▷ Record release time for writes in critical section
  –     foreach x ∈ R do 𝕃^r_{l,x} ← ℂ_t
9:      foreach t' ≠ t do Rel_{l,t'}(t).Enque(ℂ_t)
10:     ℂ_t(t) ← ℂ_t(t) + 1
11: procedure READ(t, x, e, L)
  –     ℂ_t ← ℂ_t ⊔_{l∈(L∩L^w_{x,t'})} 𝕃_{l,x}
  –     check 𝕎_x ⊑ ℂ_t
12: +   foreach thread t' ≠ t check 𝕎_x(t') ≤ ℂ_t(t') ∨ L^w_{x,t'} ∩ L ≠ ∅      ▷ Check write–read race
13: +   let t' ← T_x                                                          ▷ Get last writer thread of x
14: +   if t' ∉ {∅, t} ∧ L ∩ L^w_{x,t'} ≠ ∅ then                              ▷ Write–read conflict
15: +       𝔹_{t,x} ← ⊔_{l∈(L∩L^w_{x,t'})} 𝕃_{l,x}                            ▷ Record time of writer thread's release for later use
16: +       if 𝔹_{t,x} ⋢ ℂ_t then D_t ← D_t ∪ {⟨x, e⟩}                        ▷ Record read
17:     ℝ_x(t) ← ℂ_t(t)
18: +   L^r_{x,t} ← L
19: procedure WRITE(t, x, L)
  –     ℂ_t ← ℂ_t ⊔ ⊔_{l∈(L∩L^w_{x,t'})} 𝕃_{l,x} ⊔ ⊔_{l∈(L∩L^r_{x,t'})} 𝕃^r_{l,x}
  –     check 𝕎_x ⊑ ℂ_t
20: +   foreach thread t' ≠ t check 𝕎_x(t') ≤ ℂ_t(t') ∨ L^w_{x,t'} ∩ L ≠ ∅      ▷ Check write–write race
  –     check ℝ_x ⊑ ℂ_t
21: +   foreach thread t' ≠ t check ℝ_x(t') ≤ ℂ_t(t') ∨ L^r_{x,t'} ∩ L ≠ ∅      ▷ Check read–write race
22:     𝕎_x(t) ← ℂ_t(t)
23: +   L^w_{x,t} ← L
24: +   T_x ← t                                                               ▷ Set last writer thread of x
25: procedure BRANCH(t, e, L)
  –     skip
26: +   foreach⟨x, r⟩ ∈ D_t : brDepsOn(e, r) do ℂ_t ← ℂ_t ⊔ 𝔹_{t,x}           ▷ Add release–branch ordering
27: +   D_t ← D_t \ {⟨x, r⟩ ∈ D_t : brDepsOn(e, r)}                           ▷ Remove dependencies that were applied
```

WDP analysis maintains the following additional state:

- $\mathbb{L}_{l,x}$ is a vector clock that represents the WDP time of critical sections on l that wrote x.
- $\mathbb{B}_{t,x}$ is a vector clock that represents the WDP time of the last write event e to x such that a later read event e' to x by t conflicts with e and $lockset(e) \cap lockset(e') \neq \emptyset$.
- T_x is the last thread to write to x, or $\varnothing$ if no thread has yet written x.
- D_t is a set of pairs $\langle x, e\rangle$ such that event e is a read to x by a thread that has not (yet) executed a branch b such that $brDepsOn(b, e)$.
- $L^r_{x,t}$ and $L^w_{x,t}$ are sets of locks that were held by thread t when it last read and wrote variable x, respectively.

Initially, every vector clock maps every thread to 0, except $\forall t.\mathbb{H}_t(t) = 1$ for SDP analysis, and $\forall t.\mathbb{C}_t(t) = 1$ for WDP analysis. Every queue and set is initially empty.

The analyses provide SDP and WDP's handling of conflicting critical sections by detecting some kinds of conflicts on accesses holding a common lock. SDP analysis orders the earlier release of a common lock to the current event for write–read and read–write conflicts using $\mathbb{L}^r_{l,x}$ and $\mathbb{L}^w_{l,x}$ (lines 17 and 21 in Algorithm 1). For write–write conflicts, SDP analysis stores the time of the earlier release of a common lock in $\mathbb{B}_{t,x}$ (line 23) to order the earlier write to a later read of x by the current thread (line 16).

WDP analysis orders the release on the writer's executing thread to a later branch dependent on the read. The analysis does so by recording the time of the last writer's release in $\mathbb{L}_{l,x}$ (line 8 in Algorithm 2). Later, when a conflicting read occurs on thread t holding l, the analysis uses $\mathbb{L}_{l,x}$ to get the time for the last conflicting writer T_x's release, and stores this time in $\mathbb{B}_{t,x}$ (line 15). When t executes a branch dependent on the prior conflicting read, WDP adds ordering from the release to the current branch (line 26). The analysis detects the dependent branch using D_t, which contains a set of $\langle x, e\rangle$ pairs for which a branch dependent on read event e has not yet executed (line 24). The exact representation of e and behavior of $brDepsOn(b, e)$ are implementation dependent.

The analyses compute the "acq(m) $\prec$ rel(m) implies rel(m) $\prec$ rel(m)" property (Table 2) in the same way as WCP and DC analyses, respectively [Kini et al. 2017; Roemer et al. 2018]. Briefly, $Acq_l(t)$ and $Rel_l(t)$ contain times of acq(l) and rel(l) operations (by any thread other than t) such that the acq(l) operation is not yet SDP ordered to a following rel(l) by thread t. $Acq_{l,t'}(t)$ and $Rel_{l,t'}(t)$ contain times of acq(l) and rel(l) operations by thread t such that the acq(l) operation is not yet WDP ordered to a following rel(l) by thread t'.

SDP analysis provides composition with HB using $\mathbb{C}_t$, $\mathbb{H}_t$, and $\mathbb{C}_t[t := \mathbb{H}_t(t)]$.[4] WDP analysis includes PO with the increment of $\mathbb{C}_t(t)$ at line 10.

The analyses check the conditions for a SDP- or WDP-race by using $\mathbb{R}_x$ and $\mathbb{W}_x$. Since the analyses do not order all pairs of conflicting accesses, unordered conflicting accesses are not sufficient to report a race. SDP analysis uses the vector clock $\mathbb{L}^w_{l,x}$ and WDP analysis uses the locksets $L^r_{x,t}$ and $L^w_{x,t}$ to check if the current and prior conflicting accesses' held locks overlap (lines 18, 22, and 24 in Algorithm 1; lines 12, 20, and 21 in Algorithm 2).

Atomic accesses and operations. We extend SDP and WDP analyses to handle accesses that have ordering or atomicity semantics: *atomic accesses* that introduce ordering such as Java volatile and C++ atomic accesses, and *atomic read-modify-write operations* such as atomic test-and-set.

The following pseudocode shows how we extend WDP analysis (Algorithm 2) to handle atomic reads and writes and atomic operations. (The extensions to SDP analysis are similar but also add conflicting read–write and write–write–read ordering.)

```
1:  procedure atomicRead(t, x, e)
2:      let t' ← T_x                                  ▷ Get last writer thread of x
3:      if t' ∉ {∅, t} ∧ W_x ⋢ C_t then               ▷ Write–read conflict
4:          B_{t,x} ← W_x                             ▷ Record the write
5:          if B_{t,x} ⋢ C_t then D_t ← D_t ∪ {⟨x, e⟩}
6:  procedure atomicWrite(t, x)
7:      W_x ← C_t
8:      T_x ← t                                       ▷ Set last writer thread of x
9:  procedure atomicReadModifyWrite(t, x, e)
10:     atomicRead(t, x, e)
11:     atomicWrite(t, x)
```

[4] In Algorithm 1, $\mathbb{C}_t$ and $\mathbb{C}_t[t := \mathbb{H}_t(t)]$ are analogous to the WCP paper's $\mathbb{P}_t$ and $\mathbb{C}_t$, respectively [Kini et al. 2017].

In essence, the analysis handles atomic accesses like regular accesses contained in single-access critical sections on a unique lock to the accessed variable. The analysis treats an atomic operation as an atomic read followed by an atomic write.

Handling races. The behavior of programs with data races is unreliable [Adve and Boehm 2010; Dolan et al. 2018], but our analyses' instrumentation performs synchronization operations before accesses, which generally ensures sequential consistency (SC) for all program executions. A different problem is that if an analysis continues detecting races after the first race, then additional detected races are not necessarily real races because they may depend on an earlier race (i.e., if the earlier race were ordered, the later race would not exist). Our implementation (Section 7.1) addresses this issue by treating racing accesses as if they were contained in single-access critical sections on the same lock. Specifically, SDP analysis orders one racing event to the other for write–read and read–write races, and WDP analysis orders write–read races to a branch that depends on the read if the write is the last writer of the read. For example, after detecting a race in line 12, WDP analysis performs the following: **if** $t' = T_x$ **then** $\mathbb{B}_{t,x} \leftarrow \mathbb{W}_x(t')$; $D_t \leftarrow D_t \cup \{\langle x, e\rangle\}$.

Time and space complexity. SDP analysis and WDP analysis's run times are each linear in the number of events. Like WCP and DC analyses [Kini et al. 2017; Roemer et al. 2018], for N events, L locks, and T threads, time complexity for an entire execution trace is $O(N \times (L \times T + T^2))$. Briefly, SDP or WDP analysis at a read or write takes $O(L \times T + T^2)$ time even considering the additional computation it performs compared with WCP or DC analysis; and WDP analysis's run time at branch events can be amortized over read events.

6 VERIFYING WDP-RACES

WDP analysis is unsound, so a WDP-race may not indicate a predictable race (i.e., there may be no race exposed in any predictable trace). To avoid reporting false races, our approach post-processes each detected WDP-race with an algorithm called VindicateWDPRace. Here we overview VindicateWDPRace; the extended arXiv version [Genç et al. 2019] presents VindicateWDPRace in detail with an algorithm and examples.

To support performing VindicateWDPRace on WDP-races, WDP analysis builds a *constraint graph* in which execution events are nodes, and initially edges represent WDP ordering. VindicateWDPRace discovers and adds additional constraints to the graph that enforce lock semantics (LS) and last-writer (LW) rules. VindicateWDPRace uses the constraint graph to attempt to construct a predictable trace that exposes the WDP-race as a predictable race.

VindicateWDPRace extends prior work's VindicateDCRace algorithm for checking DC-races [Roemer et al. 2018]. VindicateWDPRace differs from VindicateDCRace primarily in the following way. VindicateWDPRace computes and adds LW constraints to the constraint graph for all reads that must be causal events in the predictable trace. Importantly, VindicateWDPRace computes causal reads and adds LW constraints at each iteration of adding constraints and at each attempt at building a predictable trace.

Algorithm 3 shows VindicateWDPRace at a high level; the extended arXiv version [Genç et al. 2019] presents VindicateWDPRace in detail. VindicateWDPRace takes the initial constraint graph (G) and a WDP-race (e_1, e_2) as input (line 1). It first calls AddConstraints (line 2), which adds necessary constraints to G and returns an updated constraint graph. AddConstraints first adds *consecutive-event* constraints (i.e., edges) to G to enforce that any predictable trace must execute e_1 and e_2 consecutively (line 9). AddConstraints then computes the set of causal events for any predictable trace constrained by G, which it uses to add LW constraints to G, ensuring that every causal read in a predictable trace can have the same last writer as in the original trace (line 11). Next, AddConstraints adds LS constraints to G, by identifying critical sections on the

Algorithm 3 Check if WDP-race is a true predictable race (high-level version of algorithm)

```
1:  procedure VindicateWDPRace(G, e1, e2)          ▷ Inputs: constraint graph and WDP-race events
2:      G ← AddConstraints(G, e1, e2)
3:      if G has a cycle reaching e1 or e2 then return No predictable race
4:      else
5:          tr′ ← ConstructReorderedTrace(G, e1, e2)      ▷ Non-empty iff predictable trace constructed
6:          if tr′ ≠ ⟨⟩ then return Predictable race witnessed by tr′      ▷ Check for non-empty trace
7:          else return Don't know

8:  procedure AddConstraints(G, e1, e2)
9:      Add consecutive-event constraints to G
10:     do
11:         Compute causal reads and add last-writer (LW) constraints to G
12:         Add lock-semantics (LS) constraints to G
13:     while G has changed
14:     return G
```

same lock that are partially ordered and thus must be fully ordered to obey LS rules (line 12). Since added LW and LS constraints may lead to new LS and LW constraints being detected, respectively, AddConstraints iterates until it finds no new constraints to add (lines 10–13).

The constraints added by AddConstraints are necessary but insufficient constraints on any trace exposing a predictable race on e_1 and e_2. Thus if G has a cycle that must be part of any predictable trace, then the original trace has no predictable race on e_1 and e_2 (line 3). Otherwise, AddConstraints calls ConstructReorderedTrace (line 5), which attempts to construct a legal predictable trace tr'. ConstructReorderedTrace is a greedy algorithm that starts from e_1 and e_2 and works backward, adding events in reverse order that satisfy G's constraints and also conform to LS and LW rules (G's constraints are necessary but insufficient). If ConstructReorderedTrace returns a (non-empty) trace tr', it is a legal predictable trace exposing a race on e_1 and e_2 (line 6). Otherwise, ConstructReorderedTrace returns an empty trace, which means that it could not find a predictable race, although one may exist (line 7).

7 EVALUATION

This section evaluates the predictive race detection effectiveness and run-time performance of this paper's approaches.

7.1 Implementation

We implemented SDP and WDP analyses and VindicateWDPRace by extending the publicly available *Vindicator* implementation, which includes HB, WCP, and DC analyses and VindicateDCRace [Roemer et al. 2018].[5] Vindicator is built on top of *RoadRunner*, a dynamic analysis framework for concurrent Java programs [Flanagan and Freund 2010b].[6] We extended RoadRunner to instrument branches to enable WDP analysis at program branches. RoadRunner operates on the Java bytecode of analyzed programs, so analysis properties such as SDP soundness and WDP completeness hold with respect to the execution of the bytecode, even if the JVM compiler optimizes away control or data dependencies.

Our implementation of SDP and WDP analyses and VindicateWDPRace is publicly available.[7]

[5] https://github.com/PLaSSticity/Vindicator
[6] https://github.com/stephenfreund/RoadRunner/releases/tag/v0.5
[7] https://github.com/PLaSSticity/SDP-WDP-implementation

We evaluated *Joana* to perform static analysis for detecting whether a branch depends on prior reads or not [Giffhorn and Hammer 2008], following the system dependency graphs used in MCR-S [Huang and Huang 2017]. We found no practical advantages to using Joana. In most programs, for the vast majority of the write–read branch dependencies executed, the next branch after the read is dependent on the read according to Joana. In pmd and sunflow, static analysis reported many write–read dependencies where the following branch did not depend on the read, but this did not lead to any additional WDP-races being detected. It is unclear whether these results are mainly due to properties of the evaluated programs (i.e., if almost all branches do depend on prior reads) or imprecision of Joana's static analysis. Our implementation and evaluation do not use static analysis, and instead assume that branches always depend on prior reads.

SDP and WDP analyses. We implemented a single analysis tool within RoadRunner that can perform HB, WCP, DC, SDP, and WDP analyses on a single observed execution. The implementation of HB, WCP, and DC analyses are taken from the Vindicator implementation, and implementation of SDP and WDP analyses follows Algorithms 1 and 2. For thread fork and join (including implicitly forked threads [Roemer et al. 2018]) and static class initializer edges [Lindholm and Yellin 1999], each analysis adds appropriate ordering between the two synchronizing events. The analyses treat calls to m.wait() as a release of m followed by an acquire of m. The analyses instrument volatile variable accesses as *atomic accesses* as described in Section 5. The analyses can in theory handle lock-free data structures, such as data structures in java.util.concurrent, by handling atomic operations as in Section 5. However, RoadRunner instruments only application code, not Java library code, and it does not intercept underlying atomic operations (e.g., by instrumenting calls to atomic sun.misc.Unsafe methods). The analyses may thus miss some synchronization in the evaluated programs.

The analyses can determine that some observed events are "redundant" and cannot affect the analysis results. For a read or write event, if the same thread has performed a read or write, respectively, to the same variable without an intervening synchronization operation, then the access is redundant. For a branch event, if the same thread has not performed a read event since the last branch event, then the branch is redundant (since our implementation assumes that a branch is dependent on all prior reads). The implementation "fast path" detects and filters redundant events, and does not perform analysis for them.

The implementation is naturally parallel because application threads running in parallel perform analysis. The implementation uses fine-grained synchronization on metadata to ensure atomicity of the analysis for an event. For WDP analysis, to obtain an approximation of $<_{tr}$ (needed by vindication; see the extended arXiv version [Genç et al. 2019]), the implementation assigns each event node in the constraint graph a Lamport timestamp [Lamport 1978] that respects HB order: $e \prec_{\text{HB}} e' \implies ts(e) < ts(e')$.

Handling races. To keep finding real races after the first detected race, whenever an analysis detects a race, it updates vector clocks (and WDP's constraint graph) so that the execution so far is race free. SDP and WDP analyses treat racing accesses as though minimal critical sections on the same lock protected them, as described in Section 5. HB, WCP, and DC analyses handle detected races by adding ordering between all accesses.

If an analysis detects multiple races involving the current access, it reports only one of the races but adds ordering to eliminate all of the races.

Vindication. WDP analysis constructs a constraint graph representing the observed execution's WDP ordering. When the execution completes, the implementation calls VindicateWDPRace on a configurable subset of the WDP-races, e.g., each WDP-race that is not also a SDP-race.

	#Thr		Total events	Analyzed events All	(acq/rel	wr	rd	br)
avrora	7	(7)	2,400 M	260 M	(1.2%	17.7%	42.2%	38.4%)
batik	7	(7)	490 M	17 M	(0.6%	26.3%	38.4%	34.0%)
h2	34	(33)	9,368 M	768 M	(0.5%	17.1%	43.1%	39.1%)
luindex	3	(3)	910 M	72 M	(0.6%	20.1%	42.4%	36.9%)
lusearch	34	(34)	2,746 M	301 M	(0.9%	19.5%	43.6%	35.6%)
pmd	33	(33)	403 M	41 M	(< 0.1%	28.5%	37.3%	34.2%)
sunflow	65	(33)	14,452 M	887 M	(< 0.1%	44.7%	41.4%	13.8%)
tomcat	106	(67)	113 M	29 M	(2.8%	18.7%	42.1%	36.1%)
xalan	33	(33)	1,306 M	436 M	(2.1%	12.0%	48.8%	37.1%)

Table 4. Dynamic characteristics of the analyzed programs. Event counts (shown in millions) and percentages are collected from WDP analysis; other analyses do not analyze branch events.

7.2 Methodology

The experiments execute large, real Java programs harnessed as the DaCapo benchmarks [Blackburn et al. 2006], version 9.12-bach. We use a version of the DaCapo programs that the RoadRunner authors have modified to work with RoadRunner;[8] the resulting workloads are approximately equal to DaCapo's default workload. The experiments exclude DaCapo programs eclipse, tradebeans, and tradesoap, which the RoadRunner authors have not modified to run with RoadRunner; jython, which failed to run with RoadRunner in our environment; and the single-threaded program fop.

The experiments execute on a quiet Intel Xeon E5-4620 with four 8-core processors with hyper-threading disabled and 256 GB of main memory, running Linux 3.10.0. We execute RoadRunner with the HotSpot 1.8.0 JVM and set the maximum heap size to 245 GB.

We run various combinations of the analyses to collect race results and statistics and measure performance. To account for run-to-run variation, each reported result is the mean of five trials.

Each WDP-race in an execution is a *dynamic* WDP-race (similarly for SDP-, DC-, WCP-, and HB-races). Among dynamic WDP-races, some may be detected at the same static accesses. If two dynamic WDP-races have the same two static source location regardless of order, then they are the same *static* WDP-race (similarly for SDP-, DC-, WCP-, and HB-races).

7.3 Dynamic Characteristics

Table 4 shows properties of the analyzed programs. The *#Thr* column reports total threads created by an execution and, in parentheses, threads active at termination. The rest of the columns count events from WDP analysis; other analyses are similar but exclude branch events. *Total events* are all executed events instrumented by the analysis.

Analyzed events are the events *not* filtered by the fast path that detects redundant events. The rest of the columns show the breakdown of analyzed events by event type. The percentages do not add up to 100% because they do not include other events (e.g., fork, join, wait, volatile access, and static class initializer events), which are always less than 1% of analyzed events. Unsurprisingly, most analyzed events are memory accesses or branches.

7.4 Race Detection Effectiveness

Table 5 reports detected races for two different experiments that each run a combination of analyses on the same executions. Table 5(a)'s results are from an experiment that runs HB, WCP, and SDP analyses together on the same executions, to compare these analyses' race detection

[8] https://github.com/stephenfreund/RoadRunner/releases/tag/v0.5

Program	HB-races		WCP-races		SDP-races	
avrora	5	(205 K)	5	(206 K)	5	(206 K)
batik	0	(0)	0	(0)	0	(0)
h2	9	(52 K)	9	(52 K)	9	(53 K)
luindex	1	(1)	1	(1)	1	(1)
lusearch	0	(0)	0	(0)	0	(0)
pmd	6	(351)	6	(354)	8	(562)
sunflow	2	(19)	2	(25)	2	(25)
tomcat	85	(34 K)	86	(34 K)	91	(38 K)
xalan	6	(203)	21	(520 K)	52	(2.2 M)

(a) HB, WCP, and SDP analyses on the same executions.

Program	DC-races		WDP-races	
avrora	5	(203 K)	5	(406 K)
batik	0	(0)	0	(0)
h2	11	(54 K)	12	(63 K)
luindex	1	(1)	1	(1)
lusearch	0	(0)	1	(30)
pmd	9	(2 K)	10	(3 K)
sunflow	2	(49)	2	(100)
tomcat	94	(36 K)	284	(125 K)
xalan	17	(649 K)	170	(15 M)

(b) DC and WDP analyses on the same executions.

Table 5. Static and dynamic (in parentheses) race counts from two different experiments.

capabilities directly. Likewise, a separate experiment runs DC and WDP analyses together on the same executions to make them directly comparable, resulting in Table 5(b)'s results.

For each race count, the first value is static races, followed by dynamic races in parentheses. For example, on average over the five trials, the analysis detects about 406,000 WDP-races for avrora, which each correspond to one of 5 different unordered pairs of static program locations.

Table 5(a) shows that SDP analysis finds significantly more races than not only HB analysis but also WCP analysis—the state of the art in unbounded sound predictive race detection [Flanagan and Freund 2017; Kini et al. 2017]. These additional races are due to SDP incorporating data dependence more precisely than WCP by not ordering write–write conflicting critical sections, essentially permitting predictable traces that swap writes without changing a causal read's last writer.

Likewise, Table 5(b) shows that WDP analysis finds more races than DC analysis, the state of the art in high-coverage unbounded predictive race detection [Roemer et al. 2018]. These additional races result from WDP being more precise with respect to both data and control dependence than DC, and in fact being complete.

The counts of HB-, WCP-, and DC-races we report here are significantly different from those reported by the Vindicator paper [Roemer et al. 2018]. (While the counts are not directly comparable, both papers show similar trends between relations.) The most significant cause of this effect is that RoadRunner stops tracking a field after the field has 100 races, a behavior that Vindicator used but that we disabled for these results to avoid artificially underreporting race counts. Furthermore, our analyses do not use a Vindicator optimization that merges events, reducing the number of races reported when there are multiple races between synchronization-free regions. We disabled this optimization because WDP analysis must track variable access information for each event, negating the advantages of this optimization. Another difference is that the Vindicator experiments spawned fewer threads for some benchmarks by setting the number of available cores to 8.

Table 6 reports results from an experiment that runs SDP and WDP analyses together and then performs VindicateWDPRace on *WDP-only races*, which are WDP-races that are not also SDP-races. The *SDP-races* and *WDP-races* columns report static and dynamic races, as in Table 5. The *WDP-only* column is *static* WDP-only races, which are static WDP-races that have no dynamic instances that are SDP-races. The last column, *WDP-only* → *Verified*, reports how many static WDP-only races are detected and how many are successfully vindicated as true races by VindicateWDPRace. In this experiment, the implementation tries to vindicate up to 10 dynamic instances of each static WDP-only race. The implementation first attempts to vindicate the five earliest dynamic instances

Program	SDP-races		WDP-races		WDP-only → Verified
avrora	5	(202K)	5	(407K)	0
batik	0	(0)	0	(0)	0
h2	12	(53K)	13	(63K)	1 → 0
luindex	1	(1)	1	(1)	0
lusearch	0	(0)	1	(30)	1 → 0
pmd	9	(456)	10	(3K)	1 → 1
sunflow	2	(32)	2	(100)	0
tomcat	98	(37K)	334	(128K)	236 → 60
xalan	31	(1.7M)	170	(15M)	139 → 137

Table 6. Static and dynamic (in parentheses) race counts from an experiment running SDP and WDP analyses together and vindicating dynamic instances of static WDP-only races. The *WDP-only → Verified* column reports static WDP-only races, followed by how many static WDP-only races were verified as predictable races by VindicateWDPRace.

	Mean ±	Stdev	Max
pmd	24,200 ±	14,100	40,294
tomcat	4,830,000 ±	5,540,000	28,734,020
xalan	52,100 ±	90,500	751,701

Table 7. Characteristics of the distribution of event distances of WDP-only races that are verified predictable races. The table rounds the mean and standard deviation to three significant digits.

of a static WDP-only race, then five random dynamic instances, stopping as soon as it verifies any dynamic instance of the static race.

Around half of the static WDP-only races are verified predictable races: out of 378 static WDP-only races on average, 198 are verified predictable races. As Section 7.5 shows, it can take a few minutes for VindicateWDPRace to check a WDP-race. Given the difficulty and importance of detecting unknown, hard-to-expose data races in mature software—and the amount of time developers currently spend on testing and debugging—the time for VindicateWDPRace is reasonable.

We confirmed that in all of the experiments, SDP analysis detected every race detected by WCP analysis, and WDP analysis detected every race detected by DC or SDP analysis.

Race characteristics. SMT-solver-based predictive race detectors can be as precise as SDP and WDP analyses, but cannot scale to unbounded program executions [Chen et al. 2008; Huang et al. 2014; Huang and Rajagopalan 2016; Liu et al. 2016; Said et al. 2011; Şerbănuţă et al. 2013] (Section 8). These approaches typically analyze bounded windows of an execution trace, missing races involving "far apart" events. We can estimate whether SMT-based approaches would miss a predictable race by computing the race's *event distance*, which is the number of events in the execution trace between the race's two events. Since our implementation does not compute a total order of events, it approximates event distance using Lamport timestamps: event distance is the number of events e such that $ts(e_1) < ts(e) < ts(e_2)$.

Table 7 reports the distribution of event distances between accesses in each successfully vindicated WDP-only race (i.e., the last column of Table 6). The average distance and standard deviation are across all trials. The last column reports the greatest distance found among all trials.

		Instr. only		Analyses w/o constraint graph				SDP+WDP+graph		
	Base	w/o br	w/ br	WCP	SDP	SDP+DC	SDP+WDP	Slowdown	Failed	Verified
avrora	6.0 s	2.7×	3.4×	21×	23×	32×	38×	50×	-	-
batik	4.2 s	3.2×	4.4×	14×	14×	16×	22×	25×	-	-
h2	9.0 s	6.7×	9.4×	125×	135×	213×	208×	241×	386 s	-
luindex	1.6 s	5.0×	9.5×	66×	69×	82×	100×	120×	-	-
lusearch	4.1 s	3.8×	4.5×	17×	17×	21×	23×	32×	< 0.1 s	-
pmd	3.0 s	6.4×	8.7×	21×	22×	23×	27×	29×	-	0.9 s
sunflow	2.8 s	8.6×	12×	104×	106×	116×	124×	189×	-	-
tomcat	1.9 s	5.1×	5.4×	23×	22×	37×	40×	43×	1.8 s	49 s
xalan	5.5 s	2.4×	3.0×	34×	35×	51×	66×	113×	29 s	0.2 s

Table 8. Slowdowns of program instrumentation and various analyses over uninstrumented execution, and the average time taken to vindicate WDP-only races.

7.5 Performance

Table 8 reports the run-time performance of various combinations of analyses. *Base* is native execution time without any instrumentation. Other columns (excluding *Failed* and *Verified*) are slowdowns relative to *Base*.

The *Instr. only* columns are RoadRunner configurations that instrument events (excluding or including branches) but perform no analysis in the instrumentation.

The *Analyses w/o constraint graph* show configurations that do not construct a constraint graph. Only configurations including WDP analysis instrument branch events. The *WCP* and *SDP* columns show the slowdowns from running the WCP and SDP analyses independently; the performance difference between them is modest, suggesting that there is no significant performance penalty from using SDP analysis over WCP analysis. (SDP's performance improvement over WCP for h2 is not statistically significant, according to confidence intervals in the extended arXiv version [Genç et al. 2019].)

SDP+DC represents performing SDP and DC analyses together. We run DC analysis with SDP analysis to minimize DC-races that need vindication. (Vindicator combined DC analysis with WCP analysis for this purpose [Roemer et al. 2018], but SDP analysis is more powerful.)

SDP+WDP+graph represents the canonical use case for WDP analysis. This configuration performs SDP and WDP analyses and constructs the constraint graph to enable vindication. It uses SDP to reduce how many WDP-races need vindication. For comparison purposes, *SDP+WDP* forgoes constructing the constraint graph, showing the cost of constructing the graph, which we have not optimized. *SDP+WDP* is slower than *SDP+DC* because WDP analysis is generally more complex than DC analysis.

Finally, *Failed* and *Verified* are the average times taken for each dynamic race that VindicateWDPRace fails to verify or successfully verifies, respectively. Vindication times vary significantly across programs; vindication is particularly slow for tomcat because most of its racing accesses are separated by millions of events (Table 7). Vindication is slow for h2, even though its races are not far apart, because VindicateWDPRace discovers new critical section constraints that require analyzing over 500 million events.

7.6 Summary and Discussion

Our SDP- and WDP-based approaches are slower than other predictive approaches, but they find more races, some of which are millions of events apart. SMT-based approaches would not be able to find these far-apart races because they cannot scale past analyzing bounded windows of

executions [Chen et al. 2008; Huang et al. 2014; Huang and Rajagopalan 2016; Liu et al. 2016; Said et al. 2011; Şerbănuţă et al. 2013] (Section 8). Notably, *RVPredict*, which (like WDP) incorporates precise control and data dependence, uses an analysis window of 10,000 events [Huang et al. 2014], meaning it would miss many of the predictable races detected and verified by our approach.

Our evaluation demonstrates the power of SDP and WDP analyses to find more races than prior approaches *in a single execution*. A potential limitation of the evaluation is that it does not compare our analyses with approaches that perform HB analysis on multiple executions (e.g., using one of the many schedule exploration approaches; Section 8). Our work's aim is to push the limit on what can be found in a single execution, which is essentially complementary to approaches that explore multiple schedules. No other known sound technique could have predicted all of these races from the observed executions.

8 RELATED WORK

This section describes and compares with prior work, starting with the most closely related work.

Unbounded predictive analysis. Prior work introduces unbounded predictive analyses, weak-causally-precedes (WCP) and doesn't-commute (DC) analyses [Kini et al. 2017; Roemer et al. 2018], which Sections 2 and 7 covered and evaluated in detail. SDP and WDP analyses predict more races in real programs than WCP and DC analyses, respectively (Section 7).

The WCP relation is weaker (i.e., detects more races) than Smaragdakis et al.'s earlier *causally-precedes (CP)* relation [Smaragdakis et al. 2012]. Smaragdakis et al.'s implementation detects races within bounded windows of 500 events because of the difficulty of developing an efficient unbounded analysis for CP [Roemer and Bond 2019; Smaragdakis et al. 2012].

Recent work introduces the *afterward-confirm (AC)* relation and an approach called *DigHR* for computing AC [Luo et al. 2018]. AC is the same as *CP* except that it removes write–write conflicting critical section ordering. Despite this similarity with our work, the contributions differ significantly, and the DigHR work has major correctness issues. Foremost, the DigHR paper claims incorrectly that AC is sound. AC is, to the best of our understanding, unsound: its removal of write–write ordering leads to detecting false races, including for the execution in Figure 3(c) (with the br event omitted; DigHR's event model does not include branches). The DigHR paper provides a soundness proof, which we believe is incorrect as a result of informality leading to not covering cases such as Figure 3(c). In contrast with DigHR, our work introduces a sound relaxation of WCP (SDP). Additionally, our work introduces a complete relation (WDP), handles control dependencies (br events), and presents linear-time analyses for SDP and WDP (DigHR is superlinear, like existing CP analyses [Roemer and Bond 2019; Smaragdakis et al. 2012]).

Concurrently with our work, Pavlogiannis introduces a predictive race detection approach called *M2* that is related to vindication [Pavlogiannis 2019]. Pavlogiannis uses lockset analysis as an imprecise filter for potential races checked by M2, while our work introduces linear-time WDP analysis as a less-imprecise filter for potential races checked by VindicateWDPRace. Although Pavlogiannis reports performance that is sometimes competitive with the performance of HB, WCP, and DC analyses, Pavlogiannis's implementations of these analyses perform extra passes over execution traces in addition to the efficient single-pass vector-clock-based analyses from prior work [Kini et al. 2017; Roemer et al. 2018]. It is unclear to us how M2 and VindicateWDPRace would compare in terms of detection capability (aside from the fact that only VindicateWDPRace takes branches and control dependence into account). In addition to these differences, our work incorporates branches and control dependence sensitivity, while Pavlogiannis's work does not and thus would miss races such as Figures 1(b) and 5; and our work introduces a sound partial order and linear-time analysis (SDP and SDP analysis).

Our concurrent work introduces the *SmartTrack* algorithm, which optimizes the performance of WCP and DC analyses [Roemer et al. 2019]. SmartTrack's optimizations apply to analyses that compute predictive relations that order all pairs of conflicting accesses—a property that SDP and WDP do not conform to. In any case, optimizing WDP analysis would have limited benefit because the analysis still must construct a constraint graph in order to perform vindication (a necessity considering that so many WDP-races fail vindication in practice). The SmartTrack paper also introduces a new relation *weak-doesn't-commute (WDC)* that is a weak variant of DC [Roemer et al. 2019]. Unlike SDP and WDP, WDC does *not* help find more races than DC, but rather serves to improve analysis performance.

Bounded predictive approaches. Other approaches predict data races by generating and solving satisfiability modulo theories (SMT) constraints [Chen et al. 2008; Huang et al. 2014; Huang and Rajagopalan 2016; Liu et al. 2016; Said et al. 2011; Şerbănuţă et al. 2013]. These SMT-based approaches cannot analyze long execution traces in reasonable time, because constraints are quadratic or cubic in trace length, and constraint-solving time often grows superlinearly with constraints. These approaches thus break traces into bounded "windows" of traces (e.g., 500–10,000 events each), missing predictable races involving accesses that occur further apart in the trace.

One advantage of SMT-based approaches is that they can be both sound and complete (within a bounded window) by encoding precise constraints. Notably, *RVPredict* includes branches in its execution model [Huang et al. 2014]. (RVPredict also incorporates *values* into its event model, so a read in a predictable trace can have a different last writer as long as it writes the same value [Huang et al. 2014].) RVPredict is thus complete by Section 2.3's definition, except that it is practically limited to bounded windows of execution. WDP analysis, on the other hand, is complete without the windowing limitation, but VindicateWDPRace is not guaranteed to vindicate a WDP-race even when a predictable race exists.

Schedule exploration. In contrast to predictive analysis, *schedule exploration* approaches execute the program multiple times to explore more program behaviors [Burckhardt et al. 2010; Cai and Cao 2015; Eslamimehr and Palsberg 2014; Henzinger et al. 2004; Huang 2015; Huang and Huang 2017; Musuvathi and Qadeer 2007; Sen 2008]. These approaches may be systematic (often called *model checking*) or be based on randomness or heuristics. Schedule exploration is complementary to predictive analysis, which aims to glean as much as possible from a given execution. *Maximal causality reduction (MCR)* combines schedule exploration with predictive analysis [Huang 2015; Huang and Huang 2017]. *MCR-S* incorporates static control flow information to reduce the complexity of MCR's generated SMT constraints [Huang and Huang 2017].

Other analyses. Widely used data race detectors typically use dynamic *happens-before (HB)* analysis [Elmas et al. 2007; Flanagan and Freund 2009; Intel Corporation 2016; Lamport 1978; Pozniansky and Schuster 2007; Serebryany and Iskhodzhanov 2009; Serebryany et al. 2012]. HB analysis cannot predict races involving reordered critical sections on the same lock. The detected races thus depend heavily on the scheduling of the analyzed program. Other analyses find a subset of HB-races by detecting simultaneously executing conflicting accesses or regions [Biswas et al. 2017, 2015; Effinger-Dean et al. 2012; Erickson et al. 2010; Sen 2008; Veeraraghavan et al. 2011].

Lockset analysis checks a locking discipline, ensuring that all pairs of conflicting accesses hold some common lock [Choi et al. 2002; Nishiyama 2004; Savage et al. 1997; von Praun and Gross 2001]. Lockset analysis is predictive but unsound, reporting false races for synchronization patterns other than the locking discipline. *Hybrid* lockset–HB analyses generally incur disadvantages of one or both kinds of analysis [Dinning and Schonberg 1991; O'Callahan and Choi 2003; Pozniansky and Schuster 2007; Yu et al. 2005].

Sampling-based analysis trades coverage for performance (opposite of predictive analysis) in order to detect data races in production [Biswas et al. 2017; Bond et al. 2010; Erickson et al. 2010; Kasikci et al. 2013; Marino et al. 2009; Sheng et al. 2011; Zhang et al. 2017]. Custom hardware support can detect data races with low performance cost but has not been implemented [Devietti et al. 2012; Lucia et al. 2010; Marino et al. 2010; Peng et al. 2017; Segulja and Abdelrahman 2015; Singh et al. 2011; Wood et al. 2014; Zhou et al. 2007].

Dynamic analysis can estimate the likely harm of a data race [Boehm 2011; Burnim et al. 2011; Cao et al. 2016; Flanagan and Freund 2010a; Kasikci et al. 2015; Narayanasamy et al. 2007], which is orthogonal to detection. All data races are erroneous under language memory models that ascribe them undefined semantics [Adve and Boehm 2010; Boehm 2012; Boehm and Adve 2008, 2012; Boehm and Demsky 2014; Manson et al. 2005; Ševčík and Aspinall 2008]. Java's memory model defines weak semantics for data races [Manson et al. 2005], but inadvertently prohibits common JVM optimizations [Boehm and Demsky 2014; Ševčík and Aspinall 2008].

Static program analysis can detect all races across all feasible executions of a program [Engler and Ashcraft 2003; Naik and Aiken 2007; Naik et al. 2006; Pratikakis et al. 2006; von Praun and Gross 2003; Voung et al. 2007], but it reports thousands of false races for real programs [Biswas et al. 2017; Lee et al. 2012].

Avoiding or tolerating data races. New languages and type systems can ensure data race freedom, but require significant programmer effort [Abadi et al. 2006; Bocchino et al. 2009; Boyapati et al. 2002; Flanagan and Freund 2007; Matsakis and Klock 2014; Rinard and Lam 1998]. Compilers and hardware can provide well-defined semantics for data races, but incur high run-time costs or hardware complexity [Ahn et al. 2009; Lucia et al. 2010; Marino et al. 2010, 2011; Ouyang et al. 2013; Segulja and Abdelrahman 2015; Sengupta et al. 2015; Singh et al. 2011, 2012; Sura et al. 2005].

9 CONCLUSION

SDP and WDP analyses improve over existing predictive analyses by incorporating precise notions of data and control dependence, finding more races both in theory and in practice while retaining linear (in trace length) run time and thus unbounded analysis. SDP analysis maintains WCP analysis's soundness while increasing race coverage. WDP analysis finds all data races that can be predicted from an observed execution; not all WDP-races are predictable races, but VindicateWDPRace can efficiently filter false races. Experiments show that our new approaches find many predictable races in real programs that prior approaches are unable to find. These properties and results suggest that our contributions advance the state of the art in predictive race detection analysis.

ACKNOWLEDGMENTS

We thank Rob LaTour for early help with this project. Thanks to Steve Freund for making RoadRunner publicly available and providing help with using and modifying it. Thanks to the anonymous reviewers for their thorough and insightful feedback.

REFERENCES

Martín Abadi, Cormac Flanagan, and Stephen N. Freund. 2006. Types for Safe Locking: Static Race Detection for Java. *TOPLAS* 28, 2 (2006), 207–255.

Sarita V. Adve and Hans-J. Boehm. 2010. Memory Models: A Case for Rethinking Parallel Languages and Hardware. *CACM* 53 (2010), 90–101. Issue 8.

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

Swarnendu Biswas, Man Cao, Minjia Zhang, Michael D. Bond, and Benjamin P. Wood. 2017. Lightweight Data Race Detection for Production Runs. In *CC*. 11–21.

Swarnendu Biswas, Minjia Zhang, Michael D. Bond, and Brandon Lucia. 2015. Valor: Efficient, Software-Only Region Conflict Exceptions. In *OOPSLA*. 241–259.

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. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA*. 169–190.

Robert L. Bocchino, Jr., Vikram S. Adve, Sarita V. Adve, and Marc Snir. 2009. Parallel Programming Must Be Deterministic by Default. In *HotPar*. 4–9.

Hans-J. Boehm. 2011. How to miscompile programs with "benign" data races. In *HotPar*. 6.

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

Hans-J. Boehm and Sarita V. Adve. 2008. Foundations of the C++ Concurrency Memory Model. In *PLDI*. 68–78.

Hans-J. Boehm and Sarita V. Adve. 2012. You Don't Know Jack about Shared Variables or Memory Models. *CACM* 55, 2 (2012), 48–54.

Hans-J. Boehm and Brian Demsky. 2014. Outlawing Ghosts: Avoiding Out-of-Thin-Air Results. In *MSPC*. Article 7, 6 pages.

Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. 2010. Pacer: Proportional Detection of Data Races. In *PLDI*. 255–268.

Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. 2002. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *OOPSLA*. 211–230.

Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. 2010. A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs. In *ASPLOS*. 167–178.

Jacob Burnim, Koushik Sen, and Christos Stergiou. 2011. Testing Concurrent Programs on Relaxed Memory Models. In *ISSTA*. 122–132.

Yan Cai and Lingwei Cao. 2015. Effective and Precise Dynamic Detection of Hidden Races for Java Programs. In *ESEC/FSE*. 450–461.

Man Cao, Jake Roemer, Aritra Sengupta, and Michael D. Bond. 2016. Prescient Memory: Exposing Weak Memory Model Behavior by Looking into the Future. In *ISMM*. 99–110.

Feng Chen, Traian Florin Şerbănuţă, and Grigore Roşu. 2008. jPredictor: A Predictive Runtime Analysis Tool for Java. In *ICSE*. 221–230.

Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. 2002. Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs. In *PLDI*. 258–269.

Joseph Devietti, Benjamin P. Wood, Karin Strauss, Luis Ceze, Dan Grossman, and Shaz Qadeer. 2012. RADISH: Always-On Sound and Complete Race Detection in Software and Hardware. In *ISCA*. 201–212.

Anne Dinning and Edith Schonberg. 1991. Detecting Access Anomalies in Programs with Critical Sections. In *PADD*. 85–96.

Stephen Dolan, KC Sivaramakrishnan, and Anil Madhavapeddy. 2018. Bounding Data Races in Space and Time. In *PLDI*. 242–255.

Laura Effinger-Dean, Brandon Lucia, Luis Ceze, Dan Grossman, and Hans-J. Boehm. 2012. IFRit: Interference-Free Regions for Dynamic Data-Race Detection. In *OOPSLA*. 467–484.

Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. 2007. Goldilocks: A Race and Transaction-Aware Java Runtime. In *PLDI*. 245–255.

Dawson Engler and Ken Ashcraft. 2003. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *SOSP*. 237–252.

John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. 2010. Effective Data-Race Detection for the Kernel. In *OSDI*. 1–16.

Mahdi Eslamimehr and Jens Palsberg. 2014. Race Directed Scheduling of Concurrent Programs. In *PPoPP*. 301–314.

Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *TOPLAS* 9, 3 (1987), 319–349.

Cormac Flanagan and Stephen N. Freund. 2007. Type Inference Against Races. *SCP* 64, 1 (2007), 140–165.

Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and Precise Dynamic Race Detection. In *PLDI*. 121–133.

Cormac Flanagan and Stephen N. Freund. 2010a. Adversarial Memory for Detecting Destructive Races. In *PLDI*. 244–254.

Cormac Flanagan and Stephen N. Freund. 2010b. The RoadRunner Dynamic Analysis Framework for Concurrent Programs. In *PASTE*. 1–8.

Cormac Flanagan and Stephen N. Freund. 2017. *The FastTrack2 Race Detector*. Technical Report. Williams College.

Kaan Genç, Jake Roemer, Yufan Xu, and Michael D. Bond. 2019. Dependence-Aware, Unbounded Sound Predictive Race Detection. *CoRR* abs/1904.13088 (2019). arXiv:1904.13088 http://arxiv.org/abs/1904.13088

Dennis Giffhorn and Christian Hammer. 2008. Precise Analysis of Java Programs Using JOANA. In *SCAM*. 267–268.

Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. 2004. Race Checking by Context Inference. In *PLDI*. 1–13.

Jeff Huang. 2015. Stateless Model Checking Concurrent Programs with Maximal Causality Reduction. In *PLDI*. 165–174.

Jeff Huang, Patrick O'Neil Meredith, and Grigore Roşu. 2014. Maximal Sound Predictive Race Detection with Control Flow Abstraction. In *PLDI*. 337–348.

Jeff Huang and Arun K. Rajagopalan. 2016. Precise and Maximal Race Detection from Incomplete Traces. In *OOPSLA*. 462–476.

Shiyou Huang and Jeff Huang. 2017. Speeding Up Maximal Causality Reduction with Static Dependency Analysis. In *ECOOP*. 16:1–16:22.

Intel Corporation. 2016. Intel Inspector. https://software.intel.com/en-us/intel-inspector-xe.

Baris Kasikci, Cristian Zamfir, and George Candea. 2012. Data Races vs. Data Race Bugs: Telling the Difference with Portend. In *ASPLOS*. 185–198.

Baris Kasikci, Cristian Zamfir, and George Candea. 2013. RaceMob: Crowdsourced Data Race Detection. In *SOSP*. 406–422.

Baris Kasikci, Cristian Zamfir, and George Candea. 2015. Automated Classification of Data Races Under Both Strong and Weak Memory Models. *TOPLAS* 37, 3, Article 8 (May 2015), 44 pages.

Dileep Kini, Umang Mathur, and Mahesh Viswanathan. 2017. Dynamic Race Prediction in Linear Time. In *PLDI*. 157–170.

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

Dongyoon Lee, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2012. Chimera: Hybrid Program Analysis for Determinism. In *PLDI*. 463–474.

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

Tim Lindholm and Frank Yellin. 1999. *The Java Virtual Machine Specification* (2nd ed.). Prentice Hall PTR.

Peng Liu, Omer Tripp, and Xiangyu Zhang. 2016. IPA: Improving Predictive Analysis with Pointer Analysis. In *ISSTA*. 59–69.

Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *ASPLOS*. 329–339.

Brandon Lucia, Luis Ceze, Karin Strauss, Shaz Qadeer, and Hans-J. Boehm. 2010. Conflict Exceptions: Simplifying Concurrent Language Semantics with Precise Hardware Exceptions for Data-Races. In *ISCA*. 210–221.

Peng Luo, Deqing Zou, Hai Jin, Yajuan Du, Long Zheng, and Jinan Shen. 2018. DigHR: precise dynamic detection of hidden races with weak causal relation analysis. *J. Supercomputing* (2018).

Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java Memory Model. In *POPL*. 378–391.

Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. 2009. LiteRace: Effective Sampling for Lightweight Data-Race Detection. In *PLDI*. 134–143.

Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. 2010. DRFx: A Simple and Efficient Memory Model for Concurrent Programming Languages. In *PLDI*. 351–362.

Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. 2011. A Case for an SC-Preserving Compiler. In *PLDI*. 199–210.

Nicholas D. Matsakis and Felix S. Klock, II. 2014. The Rust Language. In *HILT*. 103–104.

Friedemann Mattern. 1988. Virtual Time and Global States of Distributed Systems. In *Workshop on Parallel and Distributed Algorithms*. 215–226.

Madanlal Musuvathi and Shaz Qadeer. 2007. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. In *PLDI*. 446–455.

Mayur Naik and Alex Aiken. 2007. Conditional Must Not Aliasing for Static Race Detection. In *POPL*. 327–338.

Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective Static Race Detection for Java. In *PLDI*. 308–319.

Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. 2007. Automatically Classifying Benign and Harmful Data Races Using Replay Analysis. In *PLDI*. 22–31.

Hiroyasu Nishiyama. 2004. Detecting Data Races using Dynamic Escape Analysis based on Read Barrier. In *VMRT*. 127–138.

Robert O'Callahan and Jong-Deok Choi. 2003. Hybrid Dynamic Data Race Detection. In *PPoPP*. 167–178.

Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2013. ...and region serializability for all. In *HotPar*.

Andreas Pavlogiannis. 2019. Fast, Sound and Effectively Complete Dynamic Race Detection. arXiv:1901.08857 http://arxiv.org/abs/1901.08857

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

Yuanfeng Peng, Benjamin P. Wood, and Joseph Devietti. 2017. PARSNIP: Performant Architecture for Race Safety with No Impact on Precision. In *MICRO*. 490–502.

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

Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. 2006. LOCKSMITH: Context-Sensitive Correlation Analysis for Race Detection. In *PLDI*. 320–331.

Martin C. Rinard and Monica S. Lam. 1998. The Design, Implementation, and Evaluation of Jade. *TOPLAS* 20 (1998), 483–545. Issue 3.

Jake Roemer and Michael D. Bond. 2019. Online Set-Based Dynamic Analysis for Sound Predictive Race Detection. *CoRR* abs/1907.08337 (2019). arXiv:1907.08337 http://arxiv.org/abs/1907.08337

Jake Roemer, Kaan Genç, and Michael D. Bond. 2019. Practical Predictive Race Detection. *CoRR* abs/1905.00494 (2019). arXiv:1905.00494 http://arxiv.org/abs/1905.00494

Jake Roemer, Kaan Genç, and Michael D. Bond. 2018. High-Coverage, Unbounded Sound Predictive Race Detection. In *PLDI*. 374–389.

Mahmoud Said, Chao Wang, Zijiang Yang, and Karem Sakallah. 2011. Generating Data Race Witnesses by an SMT-based Analysis. In *NFM*. 313–327.

Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs. In *SOSP*. 27–37.

Cedomir Segulja and Tarek S. Abdelrahman. 2015. Clean: A Race Detector with Cleaner Semantics. In *ISCA*. 401–413.

Koushik Sen. 2008. Race Directed Random Testing of Concurrent Programs. In *PLDI*. 11–21.

Aritra Sengupta, Swarnendu Biswas, Minjia Zhang, Michael D. Bond, and Milind Kulkarni. 2015. Hybrid Static–Dynamic Analysis for Statically Bounded Region Serializability. In *ASPLOS*. 561–575.

Traian Florin Şerbănuţă, Feng Chen, and Grigore Roşu. 2013. Maximal Causal Models for Sequentially Consistent Systems. In *RV*. 136–150.

Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer – data race detection in practice. In *WBIA*. 62–71.

Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitriy Vyukov. 2012. Dynamic Race Detection with LLVM Compiler. In *RV*. 110–114.

Jaroslav Ševčík and David Aspinall. 2008. On Validity of Program Transformations in the Java Memory Model. In *ECOOP*. 27–51.

Tianwei Sheng, Neil Vachharajani, Stephane Eranian, Robert Hundt, Wenguang Chen, and Weimin Zheng. 2011. RACEZ: A Lightweight and Non-Invasive Race Detection Tool for Production Applications. In *ICSE*. 401–410.

Abhayendra Singh, Daniel Marino, Satish Narayanasamy, Todd Millstein, and Madan Musuvathi. 2011. Efficient Processor Support for DRFx, a Memory Model with Exceptions. In *ASPLOS*. 53–66.

Abhayendra Singh, Satish Narayanasamy, Daniel Marino, Todd Millstein, and Madanlal Musuvathi. 2012. End-to-End Sequential Consistency. In *ISCA*. 524–535.

Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. 2012. Sound Predictive Race Detection in Polynomial Time. In *POPL*. 387–400.

Zehra Sura, Xing Fang, Chi-Leung Wong, Samuel P. Midkiff, Jaejin Lee, and David Padua. 2005. Compiler Techniques for High Performance Sequentially Consistent Java Programs. In *PPoPP*. 2–13.

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

Kaushik Veeraraghavan, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2011. Detecting and Surviving Data Races using Complementary Schedules. In *SOSP*. 369–384.

Christoph von Praun and Thomas R. Gross. 2001. Object Race Detection. In *OOPSLA*. 70–82.

Christoph von Praun and Thomas R. Gross. 2003. Static Conflict Analysis for Multi-Threaded Object-Oriented Programs. In *PLDI*. 115–128.

Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. 2007. RELAY: Static Race Detection on Millions of Lines of Code. In *ESEC/FSE*. 205–214.

Benjamin P. Wood, Luis Ceze, and Dan Grossman. 2014. Low-Level Detection of Language-Level Data Races with LARD. In *ASPLOS*. 671–686.

Yuan Yu, Tom Rodeheffer, and Wei Chen. 2005. RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. In *SOSP*. 221–234.

Tong Zhang, Changhee Jung, and Dongyoon Lee. 2017. ProRace: Practical Data Race Detection for Production Use. In *ASPLOS*. 149–162.

M. Zhivich and R. K. Cunningham. 2009. The Real Cost of Software Errors. *IEEE Security & Privacy* 7 (03 2009), 87–90.

Pin Zhou, Radu Teodorescu, and Yuanyuan Zhou. 2007. HARD: Hardware-Assisted Lockset-based Race Detection. In *HPCA*. 121–132.