SmartTrack: Efficient Predictive Race Detection

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

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

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

Abstract

Widely used data race detectors, including the state-of-the-art *FastTrack* algorithm, incur performance costs that are acceptable for regular in-house testing, but miss races detectable from the analyzed execution. *Predictive analyses* detect more data races in an analyzed execution than FastTrack detects, but at significantly higher performance cost.

This paper presents *SmartTrack*, an algorithm that optimizes predictive race detection analyses, including two analyses from prior work and a new analysis introduced in this paper. SmartTrack incorporates two main optimizations: (1) *epoch and ownership optimizations* from prior work, applied to predictive analysis for the first time, and (2) novel *conflicting critical section optimizations* introduced by this paper. Our evaluation shows that SmartTrack achieves performance competitive with FastTrack—a qualitative improvement in the state of the art for data race detection.

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

Keywords: Data race detection, dynamic predictive analysis

ACM Reference Format:
Jake Roemer, Kaan Genç, and Michael D. Bond. 2020. SmartTrack: Efficient Predictive Race Detection. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3385412.3385993

1 Introduction

Data races are common concurrent programming errors that lead to crashes, hangs, and data corruption [7, 11, 13, 25, 38, 40, 48, 57, 71], incurring significant monetary and human costs [45, 61, 77, 85]. Data races also cause shared-memory programs to have weak or undefined semantics [1, 8, 50].

PLDI '20, June 15–20, 2020, London, UK

ACM ISBN 978-1-4503-7613-6/20/06…$15.00
https://doi.org/10.1145/3385412.3385993

Data races are hard to detect. They occur nondeterministically under specific thread interleavings, program inputs, and execution environments, and can stay hidden even for extensively tested programs [29, 48, 77, 86]. The prevailing approach for detecting data races is to use dynamic analysis—usually *happens-before (HB) analysis* [42]—during in-house testing. *FastTrack* [24] is a state-of-the-art algorithm for HB analysis that is implemented by commercial detectors [37, 73, 74]. However, HB analysis misses races that are detectable in the observed execution (Section 2).

To detect more races than HB analysis detects, researchers have developed dynamic *predictive analyses* [14, 28, 34, 35, 41, 47, 49, 60, 67, 68, 72, 76]. SMT-based predictive analyses are powerful but fail to scale beyond bounded windows of execution (Section 6). In contrast, recently introduced *partial-order-based* predictive analyses scale to full program executions. Notably, *weak-causally-precedes (WCP)* and *doesn't-commute (DC)* analyses detect more races than HB analysis [41, 67], but they are substantially slower than FastTrack-optimized HB analysis: 27–50× vs. 6–8× according to prior work [24, 27, 41, 67] and our evaluation (Section 5).

Why are the WCP and DC predictive analyses significantly slower than FastTrack-optimized HB analysis? Can FastTrack's optimizations be applied to *predictive* analyses to achieve significant speedups? In a nutshell, as we show, FastTrack's optimizations *can* be applied to predictive analyses, but there still remains a significant performance gap between predictive and HB analyses. This gap exists because predictive partial orders such as WCP and DC are inherently more complex than HB. Chiefly, predictive partial orders, in contrast with HB, order critical sections on the same lock only if they contain conflicting accesses,[1] which we call *conflicting critical sections (CCSs)*. In addition, WCP and DC order releases of the same lock if any part of their critical sections are ordered with each other. These sources of predictive analysis complexity—especially detecting CCSs—present nontrivial performance challenges with non-obvious solutions.

Contributions. This paper introduces novel contributions that enable predictive analysis to perform competitively with optimized HB analysis. Table 1 summarizes our contributions, in the same order that Sections 3–4 present them. Our principal technical contribution is *conflicting critical section (CCS) optimizations* (last row of Table 1). These CCS optimizations introduce novel analysis state and techniques to avoid

[1] Conflicting accesses are accesses to the same variable by different threads such that at least one is a write.

Source of poor performance	Contribution		Speedup
Release–release ordering	WDC relation and analysis		1.04–1.25×
Frequent vector clock operations	Epoch and ownership optimizations	SmartTrack	2.15–2.62×
Detecting conflicting critical sections (CCSs)	CCS optimizations	SmartTrack	1.51–1.74×

Table 1. Sources of poor performance for existing partial-order-based predictive analyses (WCP and DC [41, 67]), and corresponding solutions introduced in this paper. Speedups associated with each solution are the geomean across evaluated programs. The second and third optimizations constitute this paper's *SmartTrack* algorithm, with speedups ranging across predictive analyses. *WDC* is this paper's *weak-doesn't-commute*, with speedups ranging across multiple optimization levels.

		Prior work		This work	
Non-predictive analysis: HB		6.3×	(4.9×)	—	
Predictive analysis	WCP	34×	(47×)	8.3×	(7.5×)
	DC	28×	(32×)	8.6×	(7.6×)
	WDC	—		6.9×	(6.2×)

Table 2. Slowdowns and (in parentheses) relative memory usage over native execution, for state-of-the-art analyses without and with this paper's contributions. Lower is better. Each value is the geomean across the evaluated programs.

computing redundant CCS ordering. A novel but smaller contribution is a new predictive analysis, *weak-doesn't-commute (WDC) analysis* (first row), that elides release–release ordering from DC analysis, a strength–complexity tradeoff that proves worthwhile in practice. In addition, this work applies FastTrack's *epoch and ownership optimizations* (middle row) to predictive analysis for the first time.

The CCS optimizations and epoch and ownership optimizations together constitute the new *SmartTrack* algorithm, which applies to the WCP, DC, and WDC predictive analyses.

This paper's contributions, evaluated on large, real Java programs, improve the performance of predictive analyses substantially, as Table 2 summarizes (based on Section 5's results). The *Predictive analysis* rows show that SmartTrack optimizations substantially improve the performance of predictive analyses compared with prior work. The last row shows that the new WDC analysis is cheaper than prior predictive analyses. Furthermore, the table shows that the optimized predictive analyses perform nearly as well as high-performance HB analysis.

Predictive analysis thus not only finds more races than HB analysis for an observed execution, but this paper shows how predictive analysis can close the performance gap with HB analysis. This result suggests the potential for using predictive analysis instead of HB analysis as the prevailing approach for detecting data races.

2 Background and Motivation

This section describes non-predictive and predictive analyses that detect data races and explains their limitations. Some notation and terminology follow prior work's [41, 67].

2.1 Execution Traces and Other Preliminaries

An execution trace tr is a totally ordered list of events, denoted by $<_{tr}$, that represents a linearization of events in a multithreaded execution.[2] Each event consists of a thread identifier (e.g., T1 or T2) and an operation with the form wr(x), rd(x), acq(m), or rel(m), where x is a variable and m is a lock. (Other synchronization events, such as Java volatile and C++ atomic accesses and thread fork/join, are straightforward for our analysis implementations to handle; Section 5.1.) Throughout the paper, we often denote events simply by their operation (e.g., wr(x) or acq(m)).

An execution trace must be *well formed*: a thread only acquires an un-held lock and only releases a lock it holds.

Figure 1(a) shows an example execution trace, in which (as for all example traces in the paper) top-to-bottom ordering denotes observed execution order $<_{tr}$, and column placement denotes which thread executes each event.

For convenience, we define *program-order (PO)*, a strict partial order over events in the same thread:

Definition (Program-order)**.** Given a trace tr, $\prec_{\text{PO}}$ is the smallest relation such that, for two events e and e', $e \prec_{\text{PO}} e'$ if both $e <_{tr} e'$ and e and e' are executed by the same thread.

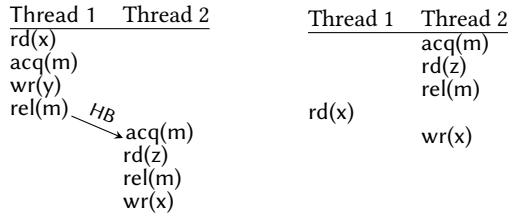
Throughout the paper, ordering notation such as $e \prec_{\text{PO}} e'$ that omits *which trace* the ordering applies to, generally refers to ordering *in the observed execution trace tr* (not some trace tr' predicted from tr—a concept explained next).

2.2 Predicted Traces and Predictable Races

A trace tr' is a *predicted trace* of tr if tr' is a feasible execution derived from the existence of tr. In a predicted trace tr', every event is also present in tr (but not every event in tr is present in tr' in general); event order preserves tr's PO ordering; every read in tr' has the same last writer (or lack of a preceding writer) as in tr; and tr' is well formed (i.e., obeys locking rules).[3]

[2] Data-race-free programs have sequential consistency (SC) semantics under the Java and C++ memory models [8, 50]. An execution of a program with a data race may have non-SC behavior [1, 18], but instrumentation added by dynamic race detection analysis typically ensures SC for every execution.

[3] Prior work provides formal definitions of predicted traces [34, 41, 67].



(a) An execution trace with a predictable race

(b) A predicted trace of (a) exposing the race

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

The execution in Figure 1(b) is a predicted trace of the execution in Figure 1(a): its events are a subset of the observed trace's events, it preserves the original trace's PO and last-writer ordering, and it is well formed.

An execution trace *tr* has a *predictable race* if some predicted trace of *tr*, tr', contains conflicting events that are consecutive (no intervening event). Events e and e' are *conflicting*, denoted $e \asymp e'$, if they are accesses to the same variable by different threads, and at least one is a write.

By definition, Figure 1(a) has a predictable race (involving accesses to x) as demonstrated by Figure 1(b). Intuitively, it is *knowable from the observed execution alone* that the conflicting accesses rd(x) and wr(x) can execute simultaneously in *another* execution.

Note that if we replaced rd(z) with rd(y) in Figure 1(a), the execution would *not* have a predictable race. The insight is that executing rd(y) *before* wr(y) might see a different value, which could alter control flow to *not* execute wr(x).

A race detection analysis is *sound* if every reported race is a (true) predictable race.[4] Soundness is an important property because each reported data race, whether true or false, takes hours or days to investigate [2, 9, 24, 29, 48, 51, 57].

2.3 Happens-Before Analysis

Happens-before (HB) [42] is a strict partial order that orders events by PO and synchronization order:

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

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

HB analysis is a dynamic analysis that computes HB over an executing program and detects HB-races.

[4] This definition of soundness follows the predictive race detection literature (e.g., [34, 41, 67, 76]).

An execution trace has an *HB-race* if it has two conflicting events unordered by HB. HB analysis is sound: An HB-race indicates a predictable race [42].

Classical HB analysis uses vector clocks [53] to record variables' last-access times. *FastTrack* and follow-up work perform optimized, state-of-the-art HB analysis, using a lightweight representation of read and write metadata [24, 27, 81]. FastTrack's optimizations result in an average 3 × speedup over vector-clock-based HB analysis (Section 5.4). FastTrack's optimized HB analysis is widely used in data race detectors including Google's ThreadSanitizer [73, 74] and Intel Inspector [37].

Optimzed HB analysis achieves performance acceptable for regular in-house testing—roughly 6–8 × slowdown according to prior work [24, 27] and our evaluation—but it misses predictable races. Consider Figure 1(a): the observed execution has no HB-race, despite having a predictable race.

2.4 Predictive Analyses

A *predictive analysis* is a dynamic analysis that detects predictable races in an observed trace, including races that are not HB-races. (This definition distinguishes HB analysis from predictive analyses.)

Recent work introduces two strict partial orders weaker than HB, *weak-causally-precedes (WCP)* and *doesn't-commute (DC)*, and corresponding analyses [41, 67]. For simplicity of exposition, the paper generally shows details only for DC analysis, which is reasonable because WCP analysis is inefficient for the same reasons as DC analysis, and our optimizations to DC analysis apply directly to WCP analysis.

DC is a strict partial order with the following definition:

Definition 1 (Doesn't-commute)**.** Given a trace *tr*, $\prec_{\text{DC}}$ is the smallest relation that satisfies the following properties:

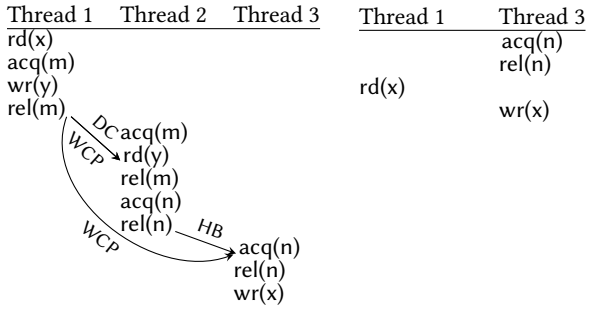
(a) If two critical sections on the same lock contain conflicting events, then the first critical section is ordered by DC to the second event. That is, $r_1 \prec_{\text{DC}} e_2$ if r_1 and r_2 are release events on the same lock, $r_1 <_{tr} r_2$, $e_1 \in CS(r_1)$, $e_2 \in CS(r_2)$, and $e_1 \asymp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

(b) Release events on the same lock are ordered by DC if their critical sections contain DC-ordered events. Because of the next two rules, this rule can be expressed simply as: $r_1 \prec_{\text{DC}} r_2$ if r_1 and r_2 are release events on the same lock and $a_1 \prec_{\text{DC}} r_2$ where a_1 is the acquire event that starts the critical section ended by r_1.

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

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

WCP differs from DC in one way: it composes with HB instead of PO, by replacing DC rules (c) and (d) with a rule



(a) An execution trace with a predictable race

(b) A predicted trace of (a) exposing the race

Figure 2. The execution in (a) has a predictable race and a DC-race ($\text{rd(x)}^{T1} \not\prec_{DC} \text{wr(x)}^{T3}$), but no WCP-race ($\text{rd(x)}^{T1} \prec_{WCP} \text{wr(x)}^{T3}$). Arrows show cross-thread ordering as labeled.

that WCP left- and right-composes with HB [41]. That is, $e \prec_{WCP} e'$ if $\exists e'' \mid e \prec_{HB} e'' \prec_{WCP} e' \vee e \prec_{WCP} e'' \prec_{HB} e'$.

An execution has a *WCP-race* or *DC-race* if it has two conflicting accesses unordered by $\prec_{WCP}$ or $\prec_{DC}$, respectively. The execution from Figure 1(a) has a WCP-race and a DC-race: WCP and DC do not order the critical sections on lock m because the critical sections do not contain conflicting accesses, resulting in $\text{rd(x)} \not\prec_{WCP} \text{wr(x)}$ and $\text{rd(x)} \not\prec_{DC} \text{wr(x)}$. Figure 2(a), on the other hand, has a DC-race but no WCP-race (since WCP composes with HB).

WCP analysis and *DC analysis* compute WCP and DC for an execution and detect WCP- and DC-races, respectively. WCP analysis is sound: every WCP-race indicates a predictable race [41].[5] DC, which is strictly weaker than WCP,[6] is unsound: it may report a race when no predictable race (or deadlock) exists. However, DC analysis reports few if any false races in practice; furthermore, a *vindication* algorithm can rule out false races, providing soundness overall [67].

DC analysis details. Algorithm 1 shows the details of an algorithm for DC analysis based closely on prior work's algorithms for WCP and DC analyses [41, 67]. We refer to this algorithm as *unoptimized DC analysis* to distinguish it from optimized algorithms introduced in this paper.

The algorithm computes DC using vector clocks that represent logical time. A vector clock $C : Tid \mapsto Val$ maps each thread to a nonnegative integer [53]. Operations on vector clocks are pointwise comparison ($\sqsubseteq$) and pointwise join ($\sqcup$):

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

The algorithm maintains the following analysis state:

Algorithm 1 Unoptimized DC analysis

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

- a vector clock C_t for each thread t that represents t's current time;
- vector clocks R_x and W_x for each program variable x that represent times of reads and writes, respectively, to x;
- vector clocks $L^r_{m,x}$ and $L^w_{m,x}$ that represent the times of critical sections on lock m containing reads and writes, respectively, to x;
- sets R_m and W_m of variables read and written, respectively, by each lock m's ongoing critical section (if any); and
- queues $Acq_{m,t}(t')$ and $Rel_{m,t}(t')$, explained below.

Initially, every set and queue is empty, and every vector clock maps all threads to 0, except $C_t(t)$ is 1 for every t.

A significant and challenging source of performance costs is the logic for detecting *conflicting critical sections* to provide DC rule (a)—a cost not present in HB analysis. At each release of a lock m, the algorithm updates $L^r_{m,x}$ and $L^w_{m,x}$ based on the variables accessed in the ending critical section on m (lines 9–10 in Algorithm 1). At a read or write to x by t, the algorithm uses $L^r_{m,x}$ and $L^w_{m,x}$ to join C_t with all prior critical sections on m that performed conflicting accesses to x (lines 15 and 22).

The algorithm checks for DC-races by checking for DC ordering with prior conflicting accesses to x; a failed check indicates a DC-race (lines 17, 18, and 24). The algorithm updates the logical time of the current thread's last write or read to x (lines 19 and 25).

[5] Technically, an execution with a WCP-race has a predictable race or a predictable deadlock [41].

[6] WCP in turn is strictly weaker than prior work's *causally-precedes* (CP) relation [49, 65, 76] and thus predicts more races than CP.

Finally, we explain how unoptimized DC analysis orders events by DC rule (b) (release events are ordered if critical sections are ordered); the details are not important for understanding this paper. The algorithm uses $Acq_{m,t}(t')$ and $Rel_{m,t}(t')$ to detect acquire–release ordering between critical sections and add release–release ordering. Each vector clock in the queue $Acq_{m,t}(t')$ represents the time of an acq(m) by t' that has *not* been determined to be DC ordered to the most recent release of m by t. Vector clocks in $Rel_{m,t}(t')$ represent the corresponding rel(m) times for clocks in $Acq_{m,t}(t')$. At rel(m) by t, the algorithm checks whether the release is ordered to a prior acquire of m by any thread t' (line 5). If so, the algorithm orders the release corresponding to the prior acquire to the current rel(m) (line 7).

Running example. To illustrate how unoptimized DC analysis works and how it compares with optimized algorithms introduced in this paper, Figure 3(a) shows an example execution and the corresponding analysis state updates after each event in the execution—focusing on the subset of analysis state relevant for detecting and ordering conflicting critical sections (DC rule (a)).

At Thread 1's rel(m), the algorithm updates $L^w_{m,x}$ to reflect the fact that x was written in the critical section on m (line 10 in Algorithm 1). Similarly, the algorithm updates $L^r_{m,x}$ or $L^w_{p,x}$ at subsequent lock releases.

At Thread 2's rd(x), unoptimized DC analysis updates C_{T2} to establish ordering with the prior conflicting critical section (line 22). Likewise, at Thread 3's wr(x), the algorithm updates C_{T3} to establish ordering with both prior conflicting critical sections (line 15). (Thread 3 is already transitively ordered with Thread 2's prior conflicting critical section because of the sync(o) events.) As a result, the checks at both threads' accesses to x correctly detect no race (lines 17, 18, and 24).

2.5 Performance Costs of Predictive Analyses

Unoptimized DC (and WCP) analyses [41, 67] incur three costs over FastTrack-optimized HB analysis [24, 27, 81].

Conflicting critical section (CCS) ordering. Tracking DC rule (a) requires $O(T \times L)$ time (lines 14–16 and 21–23 in Algorithm 1) for each access inside of critical sections on L locks, where T is the thread count; we find that many of our evaluated real programs have a high proportion of accesses executing inside one or more critical sections (Section 5). Furthermore, $L^r_{m,x}$ and $L^w_{m,x}$ store information for lock–variable pairs, requiring indirect metadata lookups. Note that $L^r_{m,x}$ and $L^w_{m,x}$ cannot be represented using epochs, and applying FastTrack's epoch optimizations to last-access metadata does not optimize detecting CCS ordering.

Vector clocks. Unoptimized DC analysis uses full vector clock operations to update write and read metadata and check for races (lines 17–19 and 24–25).

Release–release ordering. Computing DC rule (b) requires complex queue operations at every synchronization operation (lines 2 and 4–8).[7]

The next two sections describe our optimizations for these challenges, starting with release–release ordering.

3 Weak-Doesn't-Commute

This section introduces a new *weak-doesn't-commute (WDC)* relation, and a *WDC analysis* that detects *WDC-races*.

WDC is a strict partial order that has the same definition as DC except that it *omits DC rule (b)* (Definition 1).[8] Removing lines 2 and 4–8 from unoptimized DC analysis (Algorithm 1) yields unoptimized WDC analysis. This change addresses the "release–release ordering" cost explained in Section 2.5. The DC-races in Figures 1 and 2 are by definition WDC-races.

The motivation for WDC is that it is simpler than DC and thus cheaper to compute. WDC is strictly weaker than DC and thus finds some races that DC does not—but they are generally false races (i.e., not predictable races). Figure 4 shows an execution with a WDC-race but no DC-race or predictable race. The execution has no DC-race because $\text{acq(m)}^{T1} \prec_{DC} \text{rel(m)}^{T3}$ implies $\text{rel(m)}^{T1} \prec_{DC} \text{rel(m)}^{T3}$ by DC rule (b). In contrast, $\text{rel(m)}^{T1} \not\prec_{WDC} \text{rel(m)}^{T3}$. Thus $\text{rd(x)}^{T1} \not\prec_{WDC} \text{wr(x)}^{T3}$.

To ensure soundness, the prior work's *vindication* algorithm for DC analysis [67] can be used without modification to verify WDC-races as predictable races. Section 4.3 discusses vindication and its costs. However, like DC analysis, WDC analysis detects few if any false races in practice. In our evaluation, despite WDC being weaker than DC, WDC analysis does *not* report more races than DC analysis.

The next section's optimizations apply to WCP, DC, and WDC analyses alike.

4 SmartTrack

This section introduces *SmartTrack*, a set of analysis optimizations applicable to predictive analyses:

- *Epoch and ownership optimizations* are from prior work that optimizes HB analysis [24, 27, 81]. We apply them to predictive analysis for the first time (Section 4.1).
- *Conflicting critical section (CCS) optimizations* are novel analysis optimizations that represent the paper's most significant technical contribution (Section 4.2).

4.1 Epoch and Ownership Optimizations

In 2009, Flanagan and Freund introduced *epoch optimizations* to HB analysis, realized in the *FastTrack* algorithm [24]. The core idea is that HB analysis only needs to track the latest

[7] *WCP* analysis provides the same property at lower cost because it can maintain per-lock queues for each thread, instead of each pair of threads, as a consequence of WCP composing with HB [41].

[8] Weakening WCP in the same way would result in an unsound relation, giving up a key property of WCP. In contrast, DC is already unsound.

Execution events			Analysis state after event							
T1	T2	T3	C_{T1}	C_{T2}	C_{T3}	W_x	R_x	$L^w_{m,x}$	$L^r_{m,x}$	$L^w_{p,x}$	$L^r_{p,x}$
			<1, 0, 0>	<0, 1, 0>	<0, 0, 1>	<0, 0, 0>	<0, 0, 0>	<0, 0, 0>	<0, 0, 0>	<0, 0, 0>	<0, 0, 0>
acq(p)											
acq(m)											
wr(x)						<1, 0, 0>					
rel(m)			<2, 0, 0>					<1, 0, 0>			
	acq(m)										
	rd(x)			<1, 1, 0>			<0, 1, 0>				
rel(p)			<3, 0, 0>							<2, 0, 0>	
	rel(m)			<1, 2, 0>					<1, 1, 0>		
	sync(o)			<1, 3, 0>							
		sync(o)			<1, 2, 2>						
		acq(p)									
		wr(x)			<2, 2, 2>	<1, 0, 2>					
		rel(p)			<2, 2, 3>					<2, 2, 2>	

(a) Analysis state for **unoptimized DC analysis** (Algorithm 1).

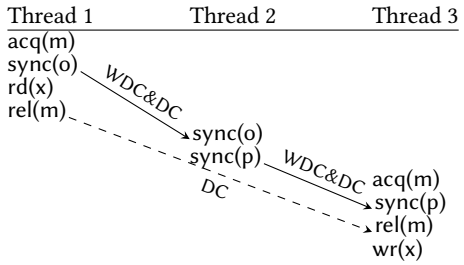
Execution events			Analysis state after event							
T1	T2	T3	C_{T1}	C_{T2}	C_{T3}	W_x	R_x	$L^w_{m,x}$	$L^r_{m,x}$	$L^w_{p,x}$	$L^r_{p,x}$
			<1, 0, 0>	<0, 1, 0>	<0, 0, 1>	⊥	⊥	<0, 0, 0>	<0, 0, 0>	<0, 0, 0>	<0, 0, 0>
acq(p)			<2, 0, 0>								
acq(m)			<3, 0, 0>								
wr(x)						3@T1	3@T1				
rel(m)			<4, 0, 0>					<3, 0, 0>	<3, 0, 0>		
	acq(m)			<0, 2, 0>							
	rd(x)			<3, 2, 0>			2@T2				
rel(p)			<5, 0, 0>							<4, 0, 0>	<4, 0, 0>
	rel(m)			<3, 3, 0>					<3, 2, 0>		
	sync(o)			<3, 5, 0>							
		sync(o)			<3, 4, 3>						
		acq(p)			<3, 4, 4>						
		wr(x)			<4, 4, 4>	4@T3	4@T3				
		rel(p)			<4, 4, 5>					<4, 4, 4>	<4, 4, 4>

(b) Analysis state for **FTO-based DC analysis** (Algorithm 2).

Execution events			Analysis state after event						
T1	T2	T3	C_{T1}	C_{T2}	C_{T3}	W_x	R_x	L^w_x	L^r_x
			<1, 0, 0>	<0, 1, 0>	<0, 0, 1>	⊥	⊥	⟨⟩	⟨⟩
acq(p)			<2, 0, 0>						
acq(m)			<3, 0, 0>						
wr(x)						3@T1	3@T1	⟨⟨<∞, 0, 0>, m⟩, ⟨<∞, 0, 0>, p⟩⟩	⟨⟨<∞, 0, 0>, m⟩, ⟨<∞, 0, 0>, p⟩⟩
rel(m)			<4, 0, 0>					⟨⟨<3, 0, 0>, m⟩, ⟨<∞, 0, 0>, p⟩⟩	⟨⟨<3, 0, 0>, m⟩, ⟨<∞, 0, 0>, p⟩⟩
	acq(m)			<0, 2, 0>					
	rd(x)			<3, 2, 0>			<3, 2, 0>		<⟨⟨<3, 0, 0>, m⟩, ⟨<∞, 0, 0>, p⟩⟩, ⟨⟨<0, ∞, 0>, m⟩⟩, ⟨⟩>
rel(p)			<5, 0, 0>					⟨⟨<3, 0, 0>, m⟩, ⟨<4, 0, 0>, p⟩⟩	<⟨⟨<3, 0, 0>, m⟩, ⟨<4, 0, 0>, p⟩⟩, ⟨⟨<0, ∞, 0>, m⟩⟩, ⟨⟩>
	rel(m)			<3, 3, 0>					<⟨⟨<3, 0, 0>, m⟩, ⟨<4, 0, 0>, p⟩⟩, ⟨⟨<3, 2, 0>, m⟩⟩, ⟨⟩>
	sync(o)			<3, 5, 0>					
		sync(o)			<3, 4, 3>				
		acq(p)			<3, 4, 4>				
		wr(x)			<4, 4, 4>	4@T3	4@T3	⟨⟨<0, 0, ∞>, p⟩⟩	⟨⟨<0, 0, ∞>, p⟩⟩
		rel(p)			<4, 4, 5>			⟨⟨<4, 4, 4>, p⟩⟩	⟨⟨<4, 4, 4>, p⟩⟩

(c) Analysis state for **SmartTrack-based DC analysis** (Algorithm 3).

Figure 3. The operation of three DC analysis algorithms (Algorithms 1–3) on the same example execution. The entries show the analysis state (right side) updated after each executed event. The event sync(o) represents the event sequence acq(o); rd(oVar); wr(oVar); rel(o) and serves to establish DC ordering between two threads. Arrows show cross-thread DC ordering.



Note: sync(o) represents acq(o); rd(oVar); wr(oVar); rel(o)

Figure 4. An execution that has no predictable race and no DC-race ($rd(x)^{T1} \prec_{DC} wr(x)^{T3}$), but does have a WDC-race ($rd(x)^{T1} \not\prec_{WDC} wr(x)^{T3}$). Arrows show cross-thread ordering.

write to a variable x, and in some cases only needs to track the latest read to x, to detect the first race. So FastTrack replaces the use of a vector clock with an *epoch*, $c@t$, to represent the latest write or read, where c is an integer clock value and t is a thread ID. The lightweight epoch representation is sufficient for detecting the first race soundly because whenever an access races with a prior write *not* represented by the last-write epoch, then it must also race with the last write (similarly for reads in some cases). That is, if the current access does not race with the last write, then either (1) the current access does not race with any earlier write or (2) the last write races with an earlier write (which would have been detected earlier). A similar argument applies to reads.

It is straightforward to adapt FastTrack's epoch optimizations to *predictive* analysis's last-access metadata updates: changes to R_x and W_x's representations will not affect the logic for detecting CCSs. We apply epoch optimizations together with *ownership optimizations* from Wood et al.'s *FastTrack-Ownership (FTO)* algorithm [81]. FTO's invariants enable a more elegant formulation for SmartTrack. We explain FTO shortly, in the context of applying it to DC analysis.

Algorithm 2 shows *FTO-DC*, which applies FTO's optimizations to unoptimized DC analysis (Algorithm 1). Differences between Algorithms 1 and 2 are highlighted in gray. Optimizing WCP and WDC analyses similarly is straightforward.

As mentioned above briefly, an epoch is a scalar $c@t$, where c is a nonnegative integer, and the leading bits represent t, a thread ID. For simplicity of exposition, for the rest of the paper, we redefine vector clocks to map to epochs instead of integers, $C : Tid \mapsto Epoch$, and redefine $C_1 \sqsubseteq C_2$ and $C_1 \sqcup C_2$ in terms of epochs. The notation $e \preceq C$ checks whether an epoch $e = c@t$ is ordered before a vector clock C, and evaluates to $c \leq c'$ where $c'@t = C(t)$. An "uninitialized" epoch representing no prior access is denoted as $\bot$, and $\bot \preceq C$ for every vector clock C.

FTO-DC modifies the metadata used by unoptimized DC analysis (Algorithm 1) in the following ways:

- W_x is an epoch representing the latest write to x.
- R_x is either an epoch or a vector clock representing the latest reads *and write* to x.

Initially, every R_x and W_x is $\bot$.

Additionally, although FTO-DC does not change the representations of $L^r_{m,x}$ and R_m from unoptimized DC analysis, in FTO-DC they represent reads *and writes*, not just reads, within a critical section on m.

Compared with unoptimized DC analysis, FTO-DC significantly changes the maintenance and checking of R_x and W_x, by using a set of increasingly complex cases:

Algorithm 2 FTO-DC (FTO-based DC analysis)

Differences with unoptimized DC analysis (Algorithm 1) are highlighted gray.

```
1:  procedure Acquire(t, m)
2:      foreach t' ≠ t do Acq_{m,t'}(t).Enque(C_t)
3:      C_t(t) ← C_t(t) + 1                          ▷ Supports same-epoch checks
4:  procedure Release(t, m)
5:      foreach t' ≠ t do
6:          while Acq_{m,t}(t').Front() ⊑ C_t do
7:              Acq_{m,t}(t').Deque()
8:              C_t ← C_t ⊔ Rel_{m,t}(t').Deque()
9:      foreach t' ≠ t do Rel_{m,t'}(t).Enque(C_t)
10:     foreach x ∈ R_m do L^r_{m,x} ← L^r_{m,x} ⊔ C_t
11:     foreach x ∈ W_m do L^w_{m,x} ← L^w_{m,x} ⊔ C_t
12:     R_m ← W_m ← ∅
13:     C_t(t) ← C_t(t) + 1
14: procedure Write(t, x)
15:     if W_x = C_t(t) then return                  [Write Same Epoch]
16:     foreach m ∈ HeldLocks(t) do
17:         C_t ← C_t ⊔ (L^r_{m,x} ⊔ L^w_{m,x})
18:         W_m ← W_m ∪ {x}
19:         R_m ← R_m ∪ {x}
20:     if R_x = c@t then skip                       [Write Owned]
21:     else if R_x = c@u then                       [Write Exclusive]
22:         check R_x ⪯ C_t
23:     else                                         [Write Shared]
24:         check R_x ⊑ C_t
25:     W_x ← R_x ← C_t(t)
26: procedure Read(t, x)
27:     if R_x = C_t(t) then return                  [Read Same Epoch]
28:     if R_x(t) = C_t(t) then return               [Shared Same Epoch]
29:     foreach m ∈ HeldLocks(t) do
30:         C_t ← C_t ⊔ L^w_{m,x}
31:         R_m ← R_m ∪ {x}
32:     if R_x = c@t then                            [Read Owned]
33:         R_x ← C_t(t)
34:     else if R_x = c@u then
35:         if R_x ⪯ C_t then                        [Read Exclusive]
36:             R_x ← C_t(t)
37:         else                                     [Read Share]
38:             check W_x ⪯ C_t
39:             R_x ← {c@u, C_t(t)}
40:     else if R_x(t) = c@t then                    [Read Shared Owned]
41:         R_x(t) ← C_t(t)
42:     else                                         [Read Shared]
43:         check W_x ⪯ C_t
44:         R_x(t) ← C_t(t)
```

Same-epoch cases. At a write (or read) to x by t, if t has already written (or read or written) x since the last synchronization event, then the access is effectively redundant (it cannot introduce a race or change last-access metadata). FTO-DC checks these cases by comparing the current thread's epoch with R_x or W_x, shown in the [Read Same Epoch], [Shared Same Epoch], and [Write Same Epoch] cases in Algorithm 2.

FTO-DC's same-epoch check works because a thread increments its logical clock $C_t(t)$ at not only release events but also acquire events (line 3 in Algorithm 2). The same-epoch check thus succeeds only for accesses redundant since the last synchronization operation.

If a same-epoch case does not apply, then FTO-DC adds ordering from prior conflicting critical sections (lines 16–19 and 29–31), just as in unoptimized DC analysis, before checking other FTO-DC cases. Because R_x, R_m, and $L^r_{m,x}$ represent last reads *and writes*, at writes FTO-DC updates R_x as well as W_x (line 25) and R_m as well as W_m (line 19).

Owned cases. At a read or write to x by t, if R_x represents a prior access by t (i.e., $R_x = c@t$ or $R_x(t) \neq \bot$), then the current access cannot race with any prior accesses. The [Read Owned], [Read Shared Owned], and [Write Owned] cases thus skip race check(s) and proceed to update R_x and/or W_x.

Exclusive cases. If an owned case does not apply and R_x is an epoch, FTO-DC compares the current time with R_x. If the current access is a write, this comparison acts as a race check [Write Exclusive]. If the current access is a read, then the comparison determines whether R_x can remain an epoch or must become a vector clock. If R_x is DC ordered before the current access, then R_x remains an epoch [Read Exclusive]. Otherwise, the algorithm checks for a write–read race by comparing the current access with W_x, and upgrades R_x to a vector clock representing both the current read and prior read or write [Read Share].

Shared cases. Finally, if an owned case does not apply and R_x is a vector clock, a shared case handles the access. Since R_x may *not* be DC ordered before the current access, [Read Shared] checks for a race by comparing with W_x, while [Write Shared] checks for a race by comparing with R_x (comparing with W_x is unnecessary since $W_x \preceq R_x$).

Running example. Figure 3(b) shows how FTO-DC works, using the same execution that we used to show how unoptimized DC works (Figure 3(a), described in Section 2.4). Here we focus on the differences between the two algorithms.

First, unlike unoptimized DC analysis, FTO-DC increments thread vector clocks at acquire events, leading to larger vector clock times. Second, FTO-DC uses epochs instead of vector clocks to represent last-access times when possible, as illustrated by the W_x and R_x columns in Figure 3(b). Third, FTO-DC essentially treats each write to x as both a write and read to x. As a result, at the execution's wr(x) events, the algorithm updates R_x as well as W_x; and at all release events for a critical section containing a wr(x), the algorithm updates $L^r_{m,x}$ or $L^r_{p,x}$ in addition to updating $L^w_{m,x}$ or $L^w_{p,x}$.

4.2 Conflicting Critical Section Optimizations

While epoch and ownership optimizations improve the performance of predictive analyses, they cannot optimize detecting *conflicting critical sections (CCSs)* to compute DC (or WCP or WDC) rule (a).

Instead, our insight for efficiently detecting CCSs is that, in common cases, an algorithm can unify how it maintains CCS metadata and last-access metadata for each variable x. Our *CCS optimizations* use new analysis state L^w_x and L^r_x, which have a correspondence with W_x and R_x at all times. L^w_x represents critical sections containing the write represented by W_x. L^r_x represents critical sections containing the read or write represented by R_x if R_x is an epoch, or a vector of critical sections containing the reads and/or writes represented by R_x if R_x is a vecor clock. Representing CCSs in this manner leads to cheaper logic than prior algorithms for predictive analysis in the common case.

The idea is that if an access within a critical section conflicts with a prior access in a critical section on the same lock *not* represented by L^w_x and L^r_x, then it must conflict with the last access within a critical section, represented by L^w_x and L^r_x, or else it races with the last access. Furthermore, CCS optimizations exploit the synergy between CCS and last-access metadata, often avoiding a race check after detecting CCSs.

SmartTrack is our new algorithm that combines CCS optimizations with epoch and ownership optimizations. Algorithm 3 shows *SmartTrack-DC*, which applies the SmartTrack algorithm to DC analysis. SmartTrack-DC modifies FTO-DC (Algorithm 2) by integrating CCS optimizations; differences between the algorithms are highlighted in gray. (Applying SmartTrack to WCP or WDC analysis is analogous.) In particular, removing lines 2 and 8–12 from Algorithm 3 yields SmartTrack-WDC.

Analysis state. SmartTrack introduces a new data type: the *critical section (CS) list*, which represents the logical times for releases of active critical sections by thread t at some point in the execution. A CS list has the following form:

$$\langle \langle C_1, m_1 \rangle, \ \ldots, \ \langle C_n, m_n \rangle \rangle$$

where $m_1 \ldots m_n$ are locks held by t, in innermost to outermost order; and $C_1 \ldots C_n$ are *references* to (equivalently, shallow copies of) vector clocks representing the release time of each critical section, in innermost to outermost order. CS lists store *references* to vector clocks in order to allow the update of C_i to be deferred until the release of m_i executes.

SmartTrack-DC maintains analysis state similar to Algorithm 2 with the following additions and changes:

- H_t for each thread t, which is a current CS list for t;
- L^w_x for each variable x (replaces FTO-DC's $L^w_{m,x}$), which is a CS list for the last write access to x;

Algorithm 3 SmartTrack-DC (SmartTrack-based DC analysis)

Differences with FTO-based DC analysis (Algorithm 2) are highlighted gray.

```
1:  procedure Acquire(t, m)
2:      foreach t' ≠ t do Acq_{m,t'}(t).Enque(C_t(t))
3:      let C = reference to new vector clock
4:      C(t) ← ∞
5:      H_t ← prepend(⟨C, m⟩, H_t)      ▷ Add ⟨C, m⟩ as head of list
6:      C_t(t) ← C_t(t) + 1
7:  procedure Release(t, m)
8:      foreach t' ≠ t do
9:          while Acq_{m,t}(t').Front() ≤ C_t do
10:             Acq_{m,t}(t').Deque()
11:             C_t ← C_t ⊔ Rel_{m,t}(t').Deque()
12:     foreach t' ≠ t do Rel_{m,t'}(t).Enque(C_t)
13:     let ⟨C, _⟩ = head(H_t)          ▷ head() returns first element
14:     C ← C_t                         ▷ Update vector clock referenced by C
15:     H_t ← rest(H_t)                 ▷ rest() returns list without first element
16:     C_t(t) ← C_t(t) + 1
17: procedure Write(t, x)
18:     if W_x = C_t(t) then return                    [Write Same Epoch]
19:     if A^r_x ≠ ∅ then
20:         foreach m ∈ HeldLocks(t) do
21:             C_t ← C_t ⊔ (⊔_{u≠t} A^r_x(u)(m))
22:             foreach u ≠ t do A^r_x(u)(m) ← A^w_x(u)(m) ← ∅
23:         A^r_x(t) ← A^w_x(t) ← ∅
24:     if R_x = c@t then skip                         [Write Owned]
25:     else if R_x = c@u then                         [Write Exclusive]
26:         let A = MultiCheck(L^r_x, u, R_x)
27:         if A ≠ ∅ then
28:             A^r_x(u) ← A
29:             A^w_x(u) ← MultiCheck(L^w_x, u, ⊥)
30:     else                                           [Write Shared]
31:         foreach u ≠ t do
32:             let A = MultiCheck(L^r_x(u), u, R_x(u))
33:             if A ≠ ∅ then
34:                 A^r_x(u) ← A
35:                 A^w_x(u) ← MultiCheck(L^w_x(u), u, ⊥)
36:     L^w_x ← L^r_x ← H_t
37:     W_x ← R_x ← C_t(t)
38: procedure Read(t, x)
39:     if R_x = C_t(t) then return                    [Read Same Epoch]
40:     if R_x(t) = C_t(t) then return                 [Shared Same Epoch]
41:     if A^w_x ≠ ∅ then
42:         foreach m ∈ HeldLocks(t) do
43:             C_t ← C_t ⊔ (⊔_{u≠t} A^w_x(u)(m))
44:     if R_x = c@t then                              [Read Owned]
45:         L^r_x ← H_t
46:         R_x ← C_t(t)
47:     else if R_x = c@u then
48:         let c'@u = { C'(u) s.t. ⟨C', _⟩ = tail(L^r_x)  if L^r_x ≠ ⟨⟩
                       { R_x                               otherwise
49:         if c'@u ≤ C_t then                         [Read Exclusive]
50:             L^r_x ← H_t
51:             R_x ← C_t(t)
52:         else                                       [Read Share]
53:             MultiCheck(L^w_x, tid(W_x), W_x)
54:             L^r_x ← {L^r_x, H_t}
55:             R_x ← {c@u, C_t(t)}
56:     else if R_x(t) = c@t then                      [Read Shared Owned]
57:         L^r_x(t) ← H_t
58:         R_x(t) ← C_t(t)
59:     else                                           [Read Shared]
60:         MultiCheck(L^w_x, tid(W_x), W_x)
61:         L^r_x(t) ← H_t
62:         R_x(t) ← C_t(t)
63: procedure MultiCheck(L, u, a)
64:     let A = ∅                                      ▷ Empty map
65:     foreach ⟨C, m⟩ in L in tail-to-head order do
66:         if C(u) ≤ C_t then return A
67:         if m ∈ heldby(t) then
68:             C_t ← C_t ⊔ C
69:             return A
70:         A(m) ← C
71:     check a ≤ C_t
72:     return A
```

- L^r_x (replaces FTO-DC's $L^r_{m,x}$) has a form dependent on R_x:
  - if R_x is an epoch, L^r_x is a CS list for the last access to x;
  - if R_x is a vector clock, L^r_x is a thread-indexed vector of CS lists ($Tid \mapsto CS\ list$), with $L^r_x(t)$ representing the CS list for the last access to x by t;
- A^w_x and A^r_x ("ancillary" metadata) for each variable x, which are vectors of maps from locks to *references to* vector clocks ($Tid \mapsto Lock \mapsto VC$). A^w_x and A^r_x represent critical sections containing accesses to x that are *not necessarily* captured by L^w_x and L^r_x, respectively.

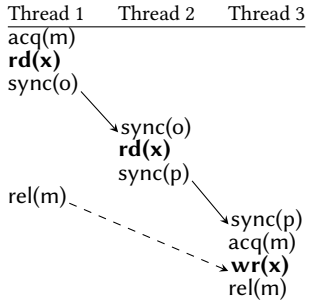
In addition to the above changes to integrate CCS optimizations, SmartTrack-DC makes the following change to FTO-DC as a small optimization:

- $Acq_{m,t}(t')$ is now a queue of *epochs*.

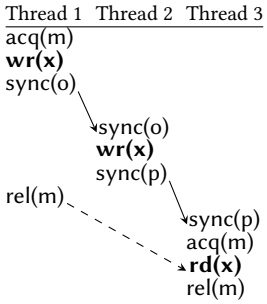
Initially all CS lists are empty; A^w_x and A^r_x are empty maps.

Maintaining CS lists. SmartTrack-DC uses the same analysis cases as FTO-DC. At each read or write to x, SmartTrack-DC maintains CCS metadata in L^w_x and L^r_x that corresponds to last-access metadata in W_x and R_x. At an access, the algorithm updates L^r_x and/or L^w_x to represent the current thread's active critical sections.

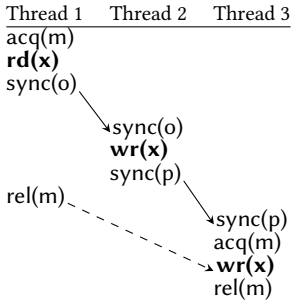
SmartTrack-DC obtains the CS list representing the current thread's active critical sections from H_t, which the algorithm maintains at each acquire and release event. At an acquire, the algorithm prepends a new entry $\langle C, m\rangle$ to H_t representing the new innermost critical section (lines 3–5). C is a reference to (i.e., shallow copy of) a newly allocated vector clock that represents the critical section's release time, which is not yet known and will be updated at the release. In the meantime, another thread u may query whether t's release of m is DC ordered before u's current time (line 66; explained later). To ensure that this query returns false before t's release of m, the algorithm initializes $C(t)$ to ∞ (line 4).



(a) An execution motivating the need for [Read Share] when FTO-DC takes [Read Exclusive]



(b) An execution motivating the need for "ancillary" metadata A_x^w and A_x^r



(c) Another execution motivating the need for A_x^w and A_x^r

Figure 5. Example executions used by the text to illustrate how SmartTrack-DC computes DC accurately. All arrows show DC ordering. Dashed arrows represent ordering that would be missed without specific SmartTrack features. sync(o) represents the sequence acq(o); rd(oVar); wr(oVar); rel(o).

When the release of m happens, the algorithm removes the first element $\langle C, m\rangle$ of H_t, representing the critical section on m, and updates the vector clock referenced by C with the release time (lines 13–15).

Checking for CCSs and races. At a read or write that may conflict with prior access(es), SmartTrack-DC combines the CCS check with the race check. To perform this combined check, the algorithm calls the helper function MultiCheck. MultiCheck traverses a CS list in reverse (outermost-to-innermost) order, looking for a prior critical section that is ordered to the current access or that conflicts with one of the current access's held locks (lines 65–70). If a critical section matches, it subsumes checking for inner critical sections or a DC-race, so MultiCheck returns. If no critical section matches, MultiCheck performs the race check (line 71).

Running example. Figure 3(c) shows how SmartTrack-DC works, focusing on differences with FTO-DC.

Unique to SmartTrack-DC are L_x^w and L_x^r. At each access to x by a thread t, the algorithm updates L_x^r and/or L_x^w using the current value of H_t, the CS list representing t's ongoing critical sections (line 68 in Algorithm 3). Note that H_t and thus L_x^r and/or L_x^w contain *references to* (i.e., shallow copies of) vector clocks. At each release of a lock, the algorithm updates vector clocks *referenced by* L_x^r and/or L_x^w.

SmartTrack-DC uses L_x^w and L_x^r to detect and order conflicting critical sections and to detect races. At Thread 2's rd(x), the algorithm takes the [Read Share] case after detecting that Thread 1's critical section on p is *not* fully DC ordered before the current time (lines 48–49). (Below we explain why SmartTrack-DC must take the [Read Share] in this situation.) The [Read Share] case inflates both R_x and L_x^r to vectors; L_x^r represents Thread 1 and 2's prior accesses to x within critical sections.

At Thread 3's wr(x), SmartTrack-DC takes the [Write Shared] case, which first checks ordering with Thread 1's wr(x); it detects the conflicting critical sections on p, so it adds ordering from rel(p) to the current access (line 68). The algorithm then checks ordering with Thread 2's rd(x); the check succeeds immediately (line 66) because the events are already DC ordered due to the sync(o) events.

SmartTrack's [Read Share] behavior. SmartTrack's CCS optimizations unify the representations of critical section and last-access metadata. To handle this unification correctly, SmartTrack-DC takes the [Read Share] case in some situations—such as Thread 2's rd(x) in Figure 3—when FTO-DC would take [Read Exclusive].

Figure 5(a) shows an execution that motivates the need for this behavior. If SmartTrack-DC were to take the [Read Exclusive] case at Thread 2's rd(x), then the algorithm would lose information about Thread 1's rd(x) being inside of the critical section on m. As a result, SmartTrack-DC would miss adding ordering from Thread 1's rel(m) to Thread 3's wr(x) (dashed arrow), leading to unsound tracking of DC and potentially reporting a false race later. SmartTrack-DC thus takes [Read Share] in situations like Thread 2's rd(x) when the prior access's critical sections (represented by the CS list L_x^r) are not all ordered before the current access.

Using "ancillary" metadata. Partly as a result of its [Read Share] behavior, SmartTrack-DC loses *no* needed CCS information at *reads*. However, as described so far, SmartTrack-DC can lose needed CCS information at *writes* to x, by overwriting information about critical sections in L_x^r and L_x^w that are not ordered before the current write. Figures 5(b) and 5(c) show two executions in which this situation occurs. In each execution, at Thread 2's wr(x), SmartTrack-DC updates L_x^r and L_x^w to $\langle\rangle$ (representing the access's lack of active critical sections)—which loses information about Thread 1's critical section on m containing an access to x. As a result, in each execution, when Thread 3 accesses x, SmartTrack-DC cannot

use L^r_x or L^w_x to detect the ordering from Thread 1's rel(m) to the current access.

To ensure sound tracking of DC, SmartTrack-DC uses the ancillary metadata A^r_x and A^w_x to track CCS information lost from L^r_x and L^w_x at writes to x. $A^r_x(t)(m)$ and $A^w_x(t)(m)$ each represent the release time of a critical section on m by t containing a read or write (A^r_x) or write (A^w_x) to x. MultiCheck computes a "residual" map A of critical sections that are not ordered to the current access (line 70), which SmartTrack-DC assigns to A^r_x or A^w_x. At a write or read not handled by a same-epoch case, if A^r_x or A^w_x, respectively, is non-empty, the analysis adds ordering for CCSs represented in A^r_x (lines 19–23) or A^w_x (lines 41–43), respectively.

In essence, SmartTrack-DC uses per-variable CCS metadata (L^r_x and L^w_x) that mimics last-access metadata (R_x and W_x) when feasible, and otherwise falls back to CCS metadata (A^r_x and A^w_x) analogous to non-SmartTrack metadata (i.e., $L^r_{m,x}$ and $L^w_{m,x}$ in Algorithms 1 and 2). SmartTrack's performance improvement over FTO relies on A^r_x and A^w_x being empty in most cases.

Optimizing $Acq_{m,t}(t')$. A final optimization that we include as part of SmartTrack-DC is to change $Acq_{m,t}(t')$ from a vector clock (used in FTO-DC) to an epoch. This optimization is correct because an epoch is sufficient for checking if ordering has been established from an acq(m) on t' to a rel(m) on t, since SmartTrack-DC increments $C_t(t)$ after every acquire operation.

4.3 Vindication: Performance Cost of Soundness

A final significant cost of DC analysis is supporting a *vindication* algorithm that checks whether a DC-race is a predictable race (similarly for WDC analysis and WDC-races). Vindication operates on a constraint graph G, constructed during DC analysis, which adds significant time and space overhead.

To avoid the cost of constructing a constraint graph, an implementation of DC analysis can either (1) report all DC-races, which are almost never false races in practice, or (2) *replay* any execution that detects a new (i.e., previously unknown) DC-race—and construct a constraint graph and perform vindication during the replayed execution only. Recent multithreaded record & replay approaches add very low (3%) run-time overhead to record an execution [46, 52].[9] Replay failures caused by undetected HB-races [44] are a non-issue since DC analysis detects all HB-races.

Our optimized DC and WDC analyses do *not* construct a constraint graph and thus do *not* perform vindication.

[9]We have not implemented or tested an approach using record & replay, which is beyond the scope of this paper. The recent practical multithreaded record & replay tools iReplayer [46] and Castor [52] both target C/C++ programs, while our implementation targets Java programs.

5 Evaluation

This section evaluates the effectiveness of this paper's predictive analysis optimizations.

5.1 Implementation

Table 3 presents the analyses that we have implemented and evaluated, categorized by analysis type (row headings) and optimization level (column headings). Each cell in the table (e.g., *FTO-WDC*) is an analysis that represents the application of an algorithm (*FTO*) to an analysis type (*WDC analysis*).

We have made all of these analysis implementations open source and publicly available.[10]

We implemented the optimized analyses (*+ Ownership* and *+ CS optimizations* columns in Table 3) based on the default *FastTrack2* analysis [27] in *RoadRunner*, a dynamic analysis framework for concurrent Java programs [26].[11] Our optimized analysis implementations minimally extend the existing FastTrack analysis that is part of the publicly available RoadRunner implementation.

For the unoptimized analyses (*Unopt* column), we used our RoadRunner-based *Vindicator* implementation[12] which implements vector-clock-based HB, WCP, and DC analyses and the vindication algorithm for checking DC-races [67]. We extended Unopt-DC to implement Unopt-WDC.

All analyses are online and detect races synchronously; none of them build a constraint graph or perform vindication.

Handling events. In addition to handling read, write, acquire, and release events as described so far, every analysis supports additional synchronization primitives. Each analysis establishes order on thread fork and join; between conflicting volatile variable accesses; and from "class initialized" to "class accessed" events. Each analysis treats wait() as a release followed by an acquire.

Every analysis maintains last-access metadata at the granularity of Java memory accesses, i.e., each object field, static field, and array element has its own last-access metadata.

Same-epoch cases. The Unopt-$*$ analysis implementations perform a [Shared Same Epoch]-like check at reads and writes (not shown in Algorithm 1). Thus, the unoptimized predictive analysis implementations (Unopt-{WCP, DC, WDC}) increment $C_t(t)$ at acquires as well as releases, just as for the optimized predictive analyses.

Handling races. In theory, the analyses handle executions up to the first race. In practice, similar to industry-standard race detectors [37, 73, 74], our analysis implementations continue analyzing executions after the first race in order to report more races to users and collect performance results for full executions. At a race, an analysis reports the race with the static program location that detected the race. If

[10]https://github.com/PLaSSticity/SmartTrack-pldi20

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

[12]https://github.com/PLaSSticity/Vindicator

	Unopt	Epochs	+ Ownership	+ CS optimizations
HB	Unopt-HB	FastTrack2 [27]	FTO-HB [81]	N/A
WCP	Unopt-WCP [41]	—	FTO-WCP	SmartTrack-WCP
DC	Unopt-DC (Algorithm 1)	—	FTO-DC (Algorithm 2)	SmartTrack-DC (Algorithm 3)
WDC	Unopt-WDC	—	FTO-WDC	SmartTrack-WDC

Table 3. Implemented and evaluated analyses. Optimizations increase from left to right, and relations weaken from top to bottom.

an analysis detects multiple races at an access (e.g., a write races with multiple last readers), we still count it as a single race. After the analysis detects a race, it continues normally.

Analysis metadata. Each analysis processes events correctly in parallel by using fine-grained synchronization on analysis metadata. An analysis can forgo synchronization for an access if a same-epoch check succeeds. To synchronize this lock-free check correctly (i.e., fence semantics), the read and write epochs in all analyses are volatile variables.

5.2 Methodology

Our evaluation uses the DaCapo benchmarks, version 9.12-bach, which are real, widely used concurrent programs that have been harnessed for evaluating performance [5]. While the DaCapo suite is not expressly intended for evaluating data race detection, the programs do contain data races.

RoadRunner bundles a version of the DaCapo benchmarks, modified to work with RoadRunner, that executes workloads similar to the default workloads. RoadRunner does not currently support eclipse, tradebeans, or tradesoap, and fop is single threaded, so our evaluation excludes those programs.

The experiments run on a quiet system with an Intel Xeon E5-2683 14-core processor with hyperthreading disabled and 256 GB of main memory running Linux 3.10.0. We run the implementations with the HotSpot 1.8.0 JVM and let it choose and adjust the heap size on the fly.

Each reported performance result, race count, or frequency statistic for an evaluated program is the arithmetic mean of 10 trials. We measure execution time as wall-clock time within the benchmarked harness of the evaluated program, and memory usage as the maximum resident set size during execution according to the GNU time program. We measure time, memory, and races within the same runs, and frequency statistics in separate statistics-collecting runs.

Our extended arXiv paper provides detailed performance results, predictable race coverage results, vindication results, 95% confidence intervals for all results, and frequency statistics for SmartTrack algorithm cases [66].

5.3 Run-Time Characteristics

Table 4 shows run-time characteristics relevant to the analyses. The *#Thr* column shows the total number of threads created. *Events* are the total executed program events (*All*) and non-same-epoch accesses (*NSEAs*).

The *Locks held at NSEAs* columns report percentages of non-same-epoch accesses holding at least one, two, or three locks, respectively. These counts are important because non-SmartTrack predictive analyses perform substantial work per held lock at non-same-epoch accesses. While all programs generally benefit from epoch and ownership optimizations, only programs that perform many accesses holding one or more locks benefit substantially from CCS optimizations. Notably, h2, luindex, and xalan have the highest average locks held per access. Unsurprisingly, these programs have the highest FTO-based predictive analysis overhead and benefit the most from SmartTrack's optimizations (Section 5.5).

The *"Ancillary" metadata* columns report percentages of non-same-epoch accesses that detect non-null ancillary metadata at a *Check* (lines 19 and 41 in Algorithm 3) and that *Use* ancillary metadata to add critical section ordering (lines 21 and 43 in Algorithm 3). Ancillary metadata is rarely if ever *used*, but some programs perform a significant number of *checks*, which can degrade performance.

5.4 Comparing Baselines

The rightmost columns of Table 4 show results that help determine whether we are using valid baselines compared with prior work. *Run time* reports slowdowns relative to uninstrumented (unanalyzed) execution, and *Memory usage* reports memory used relative to uninstrumented execution.

FastTrack comparison. The *Run time* and *Memory usage* columns report the performance of two variants of the FastTrack algorithm. *FT2* is our implementation of the FastTrack2 algorithm [27], based closely on RoadRunner's implementation of FastTrack2, which is the default FastTrack tool in RoadRunner. The main difference between FT2 and RoadRunner's FastTrack2 lies in how they handle detected races. RoadRunner's FastTrack2 does not update last-access metadata at read (but not write) events that detect a race (for unknown reasons); it does not perform analysis on future accesses to a variable after it detects a race on the variable; and it limits the number of races it counts by class field and array type. In contrast, our FT2 updates last-access metadata after every event even if it detects a race; it does not stop performing analysis on any events; and it counts every race.

FTO is our implementation of FTO-HB analysis, implemented in the same RoadRunner tool as *FT2*. Overall FTO-HB performs quite similarly to FT2. The rest of the paper's

Program	#Thr	Size (LoC)	Events All	Events NSEAs	Locks held at NSEAs ≥ 1	Locks held at NSEAs ≥ 2	Locks held at NSEAs ≥ 3	"Ancillary" metadata Check	"Ancillary" metadata Use	Run time FT2	Run time FTO-HB	Memory usage FT2	Memory usage FTO-HB
avrora	7	69 K	1,400M	140M	5.9%	<0.1%	0	6.8%	0	5.3×	5.4×	13×	13×
batik	7	188 K	160M	5.8M	46.1%	<0.1%	<0.1%	0	0	4.2×	4.2×	4.9×	4.9×
h2	16	116 K	3,800M	300M	82.8%	80.1%	0.17%	0.46%	<0.001%	9.5×	9.3×	3.0×	3.0×
jython	2	212 K	730M	170M	3.8%	0.23%	<0.1%	0	0	8.3×	8.3×	7.0×	7.0×
luindex	3	126 K	400M	41M	25.8%	25.4%	25.3%	0	0	7.9×	8.0×	4.3×	4.3×
lusearch	16	126 K	1,400M	140M	3.8%	0.39%	<0.1%	<0.001%	0	11×	12×	9.7×	10×
pmd	15	61 K	210M	8.0M	1.1%	0	0	0	0	6.5×	6.6×	2.9×	2.7×
sunflow	29	22 K	9,700M	3.5M	0.78%	<0.1%	0	0	0	17×	17×	8.4×	8.4×
tomcat	55	159 K	44M	9.7M	13.1%	8.0%	3.9%	0.13%	<0.001%	20×	19×	55×	61×
xalan	15	176 K	630M	240M	99.9%	99.7%	1.1%	6.5%	0	4.1×	4.4×	6.3×	6.3×

Table 4. Run-time characteristics of the evaluated programs. NSEAs are *non-same-epoch accesses*. The last two major columns report run time and memory usage for FastTrack-based HB analyses, relative to uninstrumented execution.

	Run time Unopt-	Run time FTO-	Run time SmartTrack-	Memory usage Unopt-	Memory usage FTO-	Memory usage SmartTrack-
HB	19×	6.3×	N/A	25×	4.9×	N/A
WCP	34×	13×	8.3×	47×	13×	7.5×
DC	28×	13×	8.6×	32×	12×	7.6×
WDC	27×	12×	6.9×	31×	11×	6.2×

Table 5. Geometric mean of run time and memory usage across the evaluated programs.

results compare against FTO-HB as the representative from the FastTrack family of optimized HB analyses.

5.5 Run-Time and Memory Performance

This section evaluates the performance of our optimized analyses, compared with competing approaches from prior work. Table 5 presents the paper's main results: run time and memory usage of the 11 analyses from Table 3.

The table reports relative run time and memory usage across all programs. For example, a cell in column *SmartTrack-* and row *DC* shows slowdown or memory usage of SmartTrack-DC analysis relative to uninstrumented execution.

The main takeaway is that SmartTrack's optimizations are effective at improving the performance of all three predictive analyses substantially, achieving performance (notably runtime overhead) close to state-of-the-art HB analysis (FTO-HB). On average across the programs, the FTO optimizations applied to predictive analyses result in a 2.2–2.6 × speedup and 2.7–3.6 × memory usage reduction over unoptimized analyses (Unopt-∗), although the FTO-based predictive analyses are still about twice as slow as FTO-HB on average. SmartTrack's CCS optimizations provide a 1.5–1.7 × average speedup and 1.6–1.8 × memory usage reduction over FTO-∗ analyses, showing that CCS optimizations eliminate most of the remaining costs FTO-based predictive analyses incur compared with FTO-HB. Overall, SmartTrack optimizations yield 3.3–4.1 × average speedups and 4.2–6.3 × memory usage reductions over unoptimized analyses, closing the performance gap compared with FTO-HB. Both FTO and CCS optimizations contribute proportionate improvements to achieve predictive analysis with performance close to that of state-of-the-art HB analysis.

HB analysis generally outperforms predictive analyses at each optimization level because it is the most straightforward analysis, eschewing the cost of computing CCSs. Unopt-WCP performs worse than Unopt-DC due to the additional cost of computing HB (needed to compute WCP). FTO-WCP and SmartTrack-WCP reduce this analysis cost significantly. At the same time, DC rule (b) is somewhat more complex to compute than WCP rule (b) (Section 2.4). These two effects cancel out on average, leading to little or no average performance difference between FTO-WCP and FTO-DC and between SmartTrack-WCP and SmartTrack-DC. WDC analysis eliminates computing rule (b), achieving better performance than DC analysis at all optimization levels.

SmartTrack thus enables three kinds of predictive analysis, each offering a different coverage–soundness tradeoff, with performance approaching that of HB analysis.

5.6 Predictable Race Coverage

Although our evaluation focuses on the performance of our optimizations, and prior work has established that WCP and DC analyses detect more races than HB analysis [41, 67], we have also evaluated how many races each analysis detects.

In general, the results confirm that weaker relations find more races than stronger relations (except WDC analysis does not report more races than DC analysis). In addition, for each relation, the different optimizations (Unopt-, FTO-, and SmartTrack-) generally report comparable race counts. The differences that exist across optimizations are attributable to run-to-run variation (as reported confidence intervals show) and differences in how the optimized analyses detect races after the first race (Section 5.1). Thus the race count differences do not serve to compare race detection effectiveness across optimizations, but rather to verify that the proposed optimizations and our implementations of them lead to reasonable race detection results.

In experiments with configurations of Unopt-DC and Unopt-WDC that build constraint graphs and perform vindication, every detected DC- and WDC-race was successfully vindicated (results not shown). We cross-referenced the static races detected by unoptimized and SmartTrack-based analyses in order to confirm that every race reported by the SmartTrack-based analyses was a true race.

5.7 Results Summary

As the results show, prior work's WCP and DC analyses are costly, especially when accesses in critical sections are frequent. The SmartTrack-optimized WCP and DC analyses improve run time and memory usage by several times on average, achieving performance comparable to HB analysis.

SmartTrack's optimizations are effective across predictive analyses. Sound WCP analysis detects fewer races than other predictive analyses and, in its unoptimized form, has the highest overhead. SmartTrack-WCP provides performance on par with HB analysis and other predictive analyses. At the other end of the coverage–soundness tradeoff, WDC has the most potential for false positives—although in practice it detects only true races—and it has the lowest overhead among predictive analyses. SmartTrack-WDC provides the best performance of any predictive analysis, nearly matching the performance of optimized HB analysis (FTO-HB). The coverage–soundness tradeoff provides flexibility to choose different analyses depending on a programmer's tolerance for the possibility of false races (although deploying with record & replay allows vindicating reported DC- or WDC-races) and the empirically observed differences among the analyses for the programmer's application.

Overall, the results show that predictive analyses can be practical data race detectors that are competitive with standard highly optimized HB data race detectors.

6 Related Work

This section considers prior work *other than* happens-before (HB) and partial-order-based predictive analyses discussed in Section 2 [20, 24, 27, 37, 41, 49, 60, 63, 65, 67, 73, 74, 76, 81].

Our recent work introduces two partial-order-based analyses, *strong-dependently-precedes (SDP)* and *weak-dependently-precedes (WDP)* analyses, that have more precise notions of dependence than WCP and DC analyses, respectively [28]. SDP and WDP do not generally order write–write conflicting critical sections, making it challenging to apply epoch and CCS optimizations to these analyses.

An alternative to partial-order-based predictive analysis is *SMT-based approaches*, which encode reordering constraints as SMT constraints [14, 34, 35, 47, 68, 72]. However, the number of constraints and the solving time scale superlinearly with trace length, so prior work analyzes bounded windows of execution, typically missing races that are more than a few thousand events apart. Prior work shows that a predictable race's accesses may be millions of events apart [28, 67].

An alternative to HB analysis is *lockset analysis*, which detects races that violate a locking discipline, but inherently reports false races [15, 17, 58, 59, 69, 79]. Hybrid lockset–HB lockset analyses typically incur the disadvantages of at least one kind of analysis [59, 63, 83].

A sound, non-predictive alternative to HB analysis is analyses that detect or infer simultaneous conflicting regions or accesses [3, 4, 19, 22, 71, 78].

Dynamic race detection analyses can target *production runs* by trading race coverage for performance [3, 9, 22, 39, 51, 75, 84] or using custom hardware [16, 62, 70, 82, 86].

Static analysis can detect all data races in all possible executions of a program [21, 55, 56, 64, 80], but for real programs, it reports thousands of false races [3, 43].

RacerD and *RacerDX* are recent static race detectors that find few false races in practice [6, 30]. RacerDX provably reports no false races under a set of well-defined assumptions [30]. However, these assumptions are not realistic; for example, RacerDX reports false races for well-synchronized programs that violate a locking discipline [31]. The RacerDX evaluation uses a few of the same programs as our evaluation, but the results are incomparable because the papers use different methodology for counting distinct races.

Schedule exploration approaches execute programs multiple times using either systematic exploration (often called *model checking*) or using heuristics [10, 12, 23, 32, 33, 36, 54, 71]. Schedule exploration is complementary with predictive analysis, which seeks to find more races in a given schedule.

7 Conclusion

This paper's contributions—notably SmartTrack's novel conflicting critical section (CCS) optimizations—enable predictive race detectors to perform nearly as well as state-of-the-art non-predictive race detectors. SmartTrack's optimizations are applicable to existing predictive analyses and to this paper's new WDC analysis, offering compelling new options in the performance–detection space. This work substantially improves the performance of predictive race detection analyses, making a case for predictive analysis to be the prevailing approach for detecting data races.

Acknowledgments

Thanks to Yufan Xu for early help with this project; Steve Freund for help with RoadRunner; and Ilya Sergey and Peter O'Hearn for discussions about RacerDX. Thanks to the anonymous reviewers and the paper's shepherd, Grigore Roşu, for many suggestions that helped improve the paper.

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

References

[1] 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.

[2] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *CACM* 53, 2 (2010), 66–75.

[3] 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.

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

[5] 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.

[6] Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 2018. RacerD: Compositional Static Race Detection. *PACMPL* 2, OOPSLA, Article 144 (2018).

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

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

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

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

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

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

[13] 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.

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

[15] 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.

[16] 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.

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

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

[19] 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.

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

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

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

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

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

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

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

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

[28] Kaan Genç, Jake Roemer, Yufan Xu, and Michael D. Bond. 2019. Dependence-Aware, Unbounded Sound Predictive Race Detection. *PACMPL* 3, OOPSLA, Article 179 (2019).

[29] Patrice Godefroid and Nachi Nagappan. 2008. Concurrency at Microsoft – An Exploratory Survey. In *(EC)²*.

[30] Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 2019. A True Positives Theorem for a Static Race Detector. *PACMPL* 3, POPL, Article 57 (2019).

[31] Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 2020. Personal communication.

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

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

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

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

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

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

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

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

[40] 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 (2015), 8:1–8:44 pages.

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

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

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

[44] Dongyoon Lee, Benjamin Wester, Kaushik Veeraraghavan, Satish Narayanasamy, Peter M. Chen, and Jason Flinn. 2010. Respec: Efficient Online Multiprocessor Replay via Speculation and External Determinism. In *ASPLOS*. 77–90.

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

[46] Hongyu Liu, Sam Silvestro, Wei Wang, Chen Tian, and Tongping Liu. 2018. iReplayer: In-situ and Identical Record-and-Replay for Multithreaded Applications. In *PLDI*. 344–358.

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

[48] 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.

[49] 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).

[50] Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java Memory Model. In *POPL*. 378–391.
[51] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. 2009. LiteRace: Effective Sampling for Lightweight Data-Race Detection. In *PLDI*. 134–143.
[52] Ali José Mashtizadeh, Tal Garfinkel, David Terei, David Mazieres, and Mendel Rosenblum. 2017. Towards Practical Default-On Multi-Core Record/Replay. In *ASPLOS*. 693–708.
[53] Friedemann Mattern. 1988. Virtual Time and Global States of Distributed Systems. In *Workshop on Parallel and Distributed Algorithms*. 215–226.
[54] Madanlal Musuvathi and Shaz Qadeer. 2007. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. In *PLDI*. 446–455.
[55] Mayur Naik and Alex Aiken. 2007. Conditional Must Not Aliasing for Static Race Detection. In *POPL*. 327–338.
[56] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective Static Race Detection for Java. In *PLDI*. 308–319.
[57] 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.
[58] Hiroyasu Nishiyama. 2004. Detecting Data Races using Dynamic Escape Analysis based on Read Barrier. In *VMRT*. 127–138.
[59] Robert O'Callahan and Jong-Deok Choi. 2003. Hybrid Dynamic Data Race Detection. In *PPoPP*. 167–178.
[60] Andreas Pavlogiannis. 2019. Fast, Sound, and Effectively Complete Dynamic Race Prediction. *PACMPL* 4, POPL, Article 17 (Dec. 2019).
[61] PCWorld. 2012. Nasdaq's Facebook Glitch Came From Race Conditions. http://www.pcworld.com/article/255911/nasdaqs_facebook_glitch_came_from_race_conditions.html.
[62] Yuanfeng Peng, Benjamin P. Wood, and Joseph Devietti. 2017. PARSNIP: Performant Architecture for Race Safety with No Impact on Precision. In *MICRO*. 490–502.
[63] Eli Pozniansky and Assaf Schuster. 2007. MultiRace: Efficient On-the-Fly Data Race Detection in Multithreaded C++ Programs. *CCPE* 19, 3 (2007), 327–340.
[64] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. 2006. LOCKSMITH: Context-Sensitive Correlation Analysis for Race Detection. In *PLDI*. 320–331.
[65] 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
[66] Jake Roemer, Kaan Genç, and Michael D. Bond. 2020. SmartTrack: Efficient Predictive Race Detection. *CoRR* abs/1905.00494 (2020). arXiv:1905.00494 http://arxiv.org/abs/1905.00494 Extended version of PLDI 2020 paper.
[67] Jake Roemer, Kaan Genç, and Michael D. Bond. 2018. High-Coverage, Unbounded Sound Predictive Race Detection. In *PLDI*. 374–389.
[68] Mahmoud Said, Chao Wang, Zijiang Yang, and Karem Sakallah. 2011. Generating Data Race Witnesses by an SMT-based Analysis. In *NFM*. 313–327.
[69] 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.
[70] Cedomir Segulja and Tarek S. Abdelrahman. 2015. Clean: A Race Detector with Cleaner Semantics. In *ISCA*. 401–413.
[71] Koushik Sen. 2008. Race Directed Random Testing of Concurrent Programs. In *PLDI*. 11–21.
[72] Traian Florin Şerbănuţă, Feng Chen, and Grigore Roşu. 2013. Maximal Causal Models for Sequentially Consistent Systems. In *RV*. 136–150.
[73] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer – data race detection in practice. In *WBIA*. 62–71.
[74] Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitriy Vyukov. 2012. Dynamic Race Detection with LLVM Compiler. In *RV*. 110–114.
[75] 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.
[76] Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. 2012. Sound Predictive Race Detection in Polynomial Time. In *POPL*. 387–400.
[77] 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.
[78] Kaushik Veeraraghavan, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2011. Detecting and Surviving Data Races using Complementary Schedules. In *SOSP*. 369–384.
[79] Christoph von Praun and Thomas R. Gross. 2001. Object Race Detection. In *OOPSLA*. 70–82.
[80] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. 2007. RELAY: Static Race Detection on Millions of Lines of Code. In *ESEC/FSE*. 205–214.
[81] Benjamin P. Wood, Man Cao, Michael D. Bond, and Dan Grossman. 2017. Instrumentation Bias for Dynamic Data Race Detection. *PACMPL* 1, OOPSLA, Article 69 (2017).
[82] Benjamin P. Wood, Luis Ceze, and Dan Grossman. 2014. Low-Level Detection of Language-Level Data Races with LARD. In *ASPLOS*. 671–686.
[83] Yuan Yu, Tom Rodeheffer, and Wei Chen. 2005. RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. In *SOSP*. 221–234.
[84] Tong Zhang, Changhee Jung, and Dongyoon Lee. 2017. ProRace: Practical Data Race Detection for Production Use. In *ASPLOS*. 149–162.
[85] M. Zhivich and R. K. Cunningham. 2009. The Real Cost of Software Errors. *IEEE Security & Privacy* 7 (03 2009), 87–90.
[86] Pin Zhou, Radu Teodorescu, and Yuanyuan Zhou. 2007. HARD: Hardware-Assisted Lockset-based Race Detection. In *HPCA*. 121–132.