Fast, Sound, and Effectively Complete Dynamic Race Prediction

ANDREAS PAVLOGIANNIS, Aarhus University, Denmark

Writing concurrent programs is highly error-prone due to the nondeterminism in interprocess communication. The most reliable indicators of errors in concurrency are *data races*, which are accesses to a shared resource that can be executed concurrently. We study the problem of predicting data races in lock-based concurrent programs. The input consists of a concurrent trace t, and the task is to determine all pairs of events of t that constitute a data race. The problem lies at the heart of concurrent verification and has been extensively studied for over three decades. However, existing polynomial-time sound techniques are highly incomplete and can miss simple races.

In this work we develop M2: a new polynomial-time algorithm for this problem, which has no false positives. In addition, our algorithm is *complete* for input traces that consist of two processes, i.e., it provably detects *all* races in the trace. We also develop sufficient criteria for detecting completeness *dynamically* in cases of more than two processes. We make an experimental evaluation of our algorithm on a challenging set of benchmarks taken from recent literature on the topic. Our algorithm soundly reports *hundreds* of real races, many of which are missed by existing methods. In addition, using our dynamic completeness criteria, M2 concludes that it has detected *all* races in the benchmark set, hence the reports are both sound and complete. Finally, its running times are comparable, and often smaller than the theoretically fastest, yet highly incomplete, existing methods. To our knowledge, M2 is the first sound algorithm that achieves such a level of performance on both running time and completeness of the reported races.

CCS Concepts: • **Software and its engineering → Software verification and validation**; • **Theory of computation** → *Theory and algorithms for application domains*; *Program analysis*.

Additional Key Words and Phrases: concurrency, race detection, predictive analyses

ACM Reference Format:

Andreas Pavlogiannis. 2020. Fast, Sound, and Effectively Complete Dynamic Race Prediction. *Proc. ACM Program. Lang.* 4, POPL, Article 17 (January 2020), 29 pages. https://doi.org/10.1145/3371085

1 INTRODUCTION

Verification of concurrent programs. Writing concurrent software is notoriously hard due to the inherent nondeterminism in the way that accesses to shared resources are scheduled. Accounting for all possible nondeterministic choices is hard, even to experienced developers. This makes the development of concurrent software prone to concurrency bugs [Lu et al. 2008; Shi et al. 2010], i.e., bugs that are present only in a few among the (possibly exponentially) many executions of the program. Since developers have no control over the scheduler, concurrency bugs are also

Author's address: Andreas Pavlogiannis, Aarhus University, Aabogade 34, Aarhus, 8200, Denmark, pavlogiannis@cs.au.dk.

2475-1421/2020/1-ART17
https://doi.org/10.1145/3371085

hard to reproduce by testing (often categorized as Heisenbugs [Gray 1985; Musuvathi et al. 2008]). Consequently, testing alone is considered an ineffective approach for detecting bugs in concurrent programs. To circumvent this difficulty, testing techniques are often combined with model checking. First, a testing phase produces a set of concrete program executions. Then, a verification phase makes a formal treatment of these executions and identifies whether there exist other "neighboring" executions that are not present in the test set but (i) constitute valid executions of the program and (ii) manifest a bug. Hence, even though the scheduler might "hide" a bug in the test set, this bug can be effectively caught by formal techniques applied on the test set.

Data races. Two events (e_1, e_2) of a concurrent program are called *conflicting* if they access the same shared resource (e.g., the same global variable x) and at least one of them modifies the resource (e.g., writes to x). A *data race* is typically defined as a conflicting pair (e_1, e_2) that can be executed concurrently [Bond et al. 2010; Flanagan and Freund 2009; Helmbold et al. 1991; O'Callahan and Choi 2003]. Data races are the prime suspects of erroneous behavior, and there have been significant efforts spanning across several decades towards detecting data races efficiently, starting with seminal papers found in [Dinning and Schonberg 1991; Helmbold et al. 1991; Schonberg 1989].

Dynamic race detection. Dynamic algorithms for race detection operate on a single execution (i.e., a *trace*) of the concurrent program, and their task is to identify pairs of events of the trace that constitute a race, even though the race might not be manifested in the input trace. Dynamic race detection is a popular technique that combines testing with formal reasoning. Existing dynamic algorithms typically fall into one of the following three categories.

Lockset-based techniques [Dinning and Schonberg 1991; Elmas et al. 2007; Savage et al. 1997] report races by comparing the sets of locks which guard conflicting data accesses. This approach typically reports spurious races, as data accesses protected by different locks can nevertheless be separated by other control-flow and data dependencies, and thus not constitute a race.

Exhaustive predictive-runtime techniques [Chen and Roşu 2007; Huang et al. 2014; Said et al. 2011; Sen et al. 2005] report races by exploring all possible valid reorderings of the input trace. These techniques typically rely on SAT/SMT solvers and are sound and complete in theory; however, as there are exponentially many valid reorderings, they have *exponential complexity*. In practice, completeness is traded for runtime, by using windowing techniques which slice the input trace into small fragments and analyze each fragment separately.

Partial-order-based techniques are probably the most well-known and widely-used. The underlying principle is to construct a partial order P on the events of the input trace. Afterwards, a race is reported between a pair of events if the two events are unordered by P. These techniques are usually efficient, as constructing the partial order typically requires polynomial time. However, in order for P to admit a linearization to a valid witness trace that exposes the race, P enforces many arbitrary orderings between events. These arbitrary orderings often result in an ordering between the events of an actual race, and thus P misses the race.

Most of the above techniques are based on Lamport's *happens-before* (HB) partial order [Lamport 1978] which is implemented in various tools [Bond et al. 2010; Christiaens and Bosschere 2001; Flanagan and Freund 2009; Pozniansky and Schuster 2003; Schonberg 1989; Yu et al. 2005]. As HB is highly incomplete, there have been several efforts for constructing weaker partial orders that are efficiently computable, such as the *causally-precedes* partial order CP [Smaragdakis et al. 2012]. Partial-order techniques recently led to important advances in predictive race detection, based on the *weakly-causally-precedes* WCP [Kini et al. 2017], *schedulably-happens-before* SHB [Mathur

et al. 2018] and *doesn't-commute* DC [Roemer et al. 2018] partial orders. We next discuss these approaches in more detail and outline the motivation behind our work.

1.1 Motivating Examples

We illustrate the motivation behind our work with a few simple examples (Figure 1) which highlight some completeness issues that the existing approaches based on HB, WCP and DC partial orders suffer from. We focus on single races here, in which case SHB is subsumed by HB. We remark that we focus on polynomial-time, sound methods here, and hence we do not consider unsound techniques (e.g., lockset-based [Savage et al. 1997]) or techniques that rely on SAT/SMT solvers and are thus not polynomial time (e.g., [Huang et al. 2014]). In each example, we use the notation τ_i to refer to the local trace of the i-th process, and e_j to refer to the j-th event in the concurrent trace. We note that the underlying memory model is sequentially consistent, i.e., in every trace, a read event observes the value of the last write event that writes to the location read by the read event.

To develop some context, we briefly outline how each of these techniques works by ordering events of the input trace. We refer to Appendix B for the formal definitions. In all cases, events that belong to the same process are always totally ordered according to their order in the input trace.

(1) The HB and WCP techniques operate in a similar manner. They perform a single pass of t and construct a partial order $\leq_{\mathsf{HB}}$ (resp., $\leq_{\mathsf{WCP}}$). A race (e, e') is reported if e, e' are conflicting and $e \not\leq_{\mathsf{HB}} e'$ (resp., $e \not\leq_{\mathsf{WCP}} e'$), i.e., the two events are unordered by the respective partial order.
(2) DC operates in three phases, which all have to succeed for (e, e') to be reported as a race.
   (a) In Phase 1, a DC partial order is constructed, similarly to HB and WCP. If $e \leq_{\mathsf{DC}} e'$ then (e, e') is reported as a non-race.
   (b) In Phase 2, a constraint graph G is constructed which contains the DC orderings. Then, more ordering constraints are inserted in G. If G becomes cyclic during this process, (e, e') is reported as a non-race. If t^* fails to respect lock semantics, (e, e') is reported as a non-race.

	τ_1	τ_2
1	acq(ℓ)	
2	**w**(**x**)	
3	rel(ℓ)	
4		acq(ℓ)
5		$w(x)$
6		rel(ℓ)
7		**r**(**x**)

(a) Is (e_2, e_7) a race?

	τ_1	τ_2	τ_3
1	acq(ℓ_1)		
2	**w**(**x**)		
3	$w(y)$		
4	rel(ℓ_1)		
5		acq(ℓ_1)	
6		acq(ℓ_2)	
7		$w(z)$	
8		rel(ℓ_2)	
9		$w(y)$	
10		rel(ℓ_1)	
11			acq(ℓ_2)
12			$r(z)$
13			rel(ℓ_2)
14			**w**(**x**)

(b) Is (e_2, e_{14}) a race?

Fig. 1. Examples in which HB, WCP and DC are incomplete. (a) A race (e_2, e_7) missed by HB, WCP and DC. (b) A race (e_2, e_{14}) missed by HB, WCP and DC (in Phase 2).

Incompleteness. Each of HB, WCP and DC methods are incomplete i.e., the input trace t can have arbitrarily many predictable races, however each of these methods falsely reports that there is no race in t. We present a couple of examples where HB, WCP and DC fail to detect simple races.

Figure 1a. There is a predictable race (e_2, e_7). HB defines $e_3 \leq_{\text{HB}} e_4$, and thus $e_2 \leq_{\text{HB}} e_7$, hence missing the race. Similarly, WCP (resp., DC) defines $e_3 \leq_{\text{WCP}} e_5$ (resp., $e_3 \leq_{\text{DC}} e_5$) and thus $e_2 \leq_{\text{WCP}} e_7$ (resp., $e_2 \leq_{\text{DC}} e_7$), hence missing the race. Intuitively, WCP and DC fail to swap the two critical sections because they contain the conflicting events $w(x)$. Note that here DC fails in Phase 1. However, (e_2, e_7) is a true race that is detected by the techniques developed in this work, exposed by the witness trace $t^* = e_4, e_5, e_6, e_1, e_2, e_7$.

Figure 1b. There is a predictable race (e_2, e_{14}). HB defines $e_4 \leq_{\text{HB}} e_5$ and $e_8 \leq_{\text{HB}} e_{11}$, and thus $e_2 \leq_{\text{HB}} e_{14}$, hence missing the race. Similarly, WCP defines $e_4 \leq_{\text{WCP}} e_5$ and $e_8 \leq_{\text{WCP}} e_{12}$ and thus $e_2 \leq_{\text{WCP}} e_{14}$, hence missing the race. Intuitively, WCP fails to swap the critical sections of τ_1 and τ_2 on ℓ_1 because WCP is closed under composition with HB, and in turn HB totally orders critical sections as in the input trace. On the other hand, DC does not compose with HB, and the only enforced orderings are $e_4 \leq_{\text{DC}} e_9$ and $e_8 \leq_{\text{DC}} e_{12}$. Hence DC proceeds with Phase 2, where it constructs a constraint graph G. Since $e_4 \leq_{\text{DC}} e_9$ and e_9 belongs in a critical section on lock ℓ_1 which is released by e_4, in order to not violate lock semantics, G forces the ordering $e_4 \rightsquigarrow e_5$. In addition, G forces the ordering $e_5 \rightsquigarrow e_2$, since e_2 is the racy event and must appear last in the witness trace. Note that this creates a cycle and hence DC fails in Phase 2. However, (e_2, e_{14}) is a true race that is detected by the techniques developed in this work, exposed by the witness trace $t^* = e_5, e_6, e_7, e_8, e_9, e_{10}, e_{11}, e_{12}, e_{13}, e_1, e_2, e_{14}$.

Algorithmic challenge. We have seen that state-of-the-art approaches fail to catch simple races. Intuitively, the algorithmic challenge that underlies race detection is that of constructing a partial order P with the following properties.

(1) P is as weak as possible, so that a race (e_i, e_j) remains unordered in P.
(2) P is efficiently linearizable to a valid trace that exposes the race.

These two features are opposing each other, as the weaker the partial order, the more linearizations it admits, and finding a valid one becomes harder.

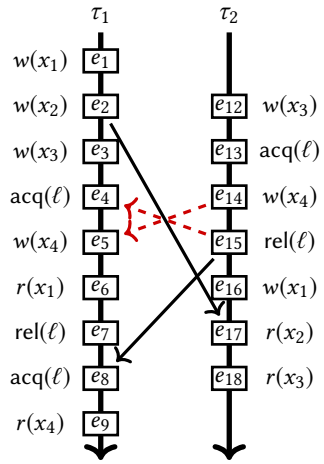
Our approach. In this work we develop a new predictive technique for race detection. At its core, our algorithm constructs partial orders that are much weaker than existing approaches (hence detecting more races), while these partial orders are efficiently (polynomial-time) linearizable to valid traces (hence the reported races are exposed efficiently). To give a complete illustration of our insights, we use the more involved example in Figure 2.

The task is to decide whether (e_{10}, e_{19}) is a predictable race of the input trace t (Figure 2a). To keep the presentation simple, we ignore the other data races that occur, which can be trivially avoided by inserting additional lock events. Note that HB, WCP and DC report no race in t, as they all order $e_{11} \leq e_{14}$. In order to detect this race, we need to make some non-trivial reasoning about reordering certain events in t. Our reasoning can be summarized in the following steps.

(1) If (e_{10}, e_{19}) is a race of t, a witness trace t^* can be constructed in which both e_{10} and e_{19} are the last events. Observe that t^* will not contain the $\text{rel}(\ell)$ event e_{11}.
(2) Since we ignore event e_{11}, that critical section of τ_1 remains open in t^*. Hence the $\text{rel}(\ell)$ event e_{15} must be ordered before the $\text{acq}(\ell)$ event e_8. In addition, the $w(x_2)$ event e_2 is observed by the $r(x_2)$ event e_{17}, hence e_2 must be ordered before e_{17}. These constraints, together with the program order which requires events of each process to occur in the same order as in the input trace, are captured by the partial order shown in solid edges in Figure 2b. Note that several conflicting accesses to x_1, x_3 and x_4 are still unordered. How can we obtain a valid linearization? First, we can infer a few more orderings.

	τ_1	τ_2
1	$w(x_1)$	
2	$w(x_2)$	
3	$w(x_3)$	
4	acq(ℓ)	
5	$w(x_4)$	
6	$r(x_1)$	
7	rel(ℓ)	
8	acq(ℓ)	
9	$r(x_4)$	
10	**w(x)**	
11	rel(ℓ)	
12		$w(x_3)$
13		acq(ℓ)
14		$w(x_4)$
15		rel(ℓ)
16		$w(x_1)$
17		$r(x_2)$
18		$r(x_3)$
19		**r(x)**

(a) Is (e_{10}, e_{19}) a race?



(b) Ordering constraints before (solid edges) and after the closure (solid and dashed edges).

	τ_1	τ_2
1	$w(x_1)$	
2	$w(x_2)$	
3	$w(x_3)$	
4		$w(x_3)$
5		acq(ℓ)
6		$w(x_4)$
7		rel(ℓ)
8	acq(ℓ)	
9	$w(x_4)$	
10	$r(x_1)$	
11	rel(ℓ)	
12	acq(ℓ)	
13	$r(x_4)$	
14		$w(x_1)$
15		$r(x_2)$
16		$r(x_3)$
17	**w(x)**	
18		**r(x)**

(c) The witness trace.

Fig. 2. Example of a race that requires non-trivial reasoning about reorderings of the input trace.

(3) The $r(x_4)$ event e_9 must observe the same write event as in t. Due to the previous step, the $w(x_4)$ event e_{14} now is ordered before e_9. To avoid e_9 observing e_{14}, we perform an *observation-closure* step, by ordering e_{14} before the observation e_5 of e_9 (see dashed edge in Figure 2b).
(4) Due to the previous step, the acq(ℓ) event e_{13} is now ordered before the rel(ℓ) event e_7. In order to not violate lock semantics, the critical section of the second process must be ordered before the first critical section of the first process. Hence we perform a *lock-closure* step, by ordering the rel(ℓ) event e_{15} before the lock-acquire event e_4 (see dashed edge in Figure 2b).
(5) At this point, no other closure step is performed, and the partial order is called *trace-closed*. Note that there still exist conflicting accesses to variables x_1 and x_3 which are pairwise unordered and quite distant, hence *not every linearization* produces a valid trace, and a correct linearization is not obvious. We observe that we can obtain a valid trace by starting from the beginning of τ_1 and τ_2, and execute the former *maximally* and the latter *minimally*, according to the partial order. That is, we repeatedly execute τ_1 until we reach an event that is preceded by an event of τ_2, and then execute τ_2 only until an event of τ_1 becomes enabled again. This *max-min linearization* produces a valid witness trace (see Figure 2c).

In this work we make the above insights formal. We define the notion of trace-closed partial orders, which captures observation and lock-closure steps, and develop an efficient (polynomial-time) algorithm for computing the closure. For two processes, we show that max-min linearizations *always* produce valid traces, *as long as* the partial order is trace-closed. Hence, in this case, we have a sound and complete algorithm. The case of three or more processes is more complicated, and our algorithm might eventually order some (but crucially, not all) conflicting events arbitrarily. Although these choices might sacrifice completeness, the resulting partial orders are much weaker than before, so that complex races can still be exposed soundly by a max-min linearization.

1.2 Our Contributions

In summary, the contributions of this work are as follows.

A new algorithm for dynamic race detection. Our main contribution is a *polynomial-time* and *sound* algorithm for detecting predictable races present in the input trace. In addition, our algorithm is *complete* for input traces that consist of events of two processes. First we study the *decision problem*, that is, given an input trace t and a pair of events (e_1, e_2) of t, decide whether the pair constitutes a data race of t. We present a sound algorithm for the problem that operates in $O(n^2 \cdot \log n)$ time, where n is the length of t. Since all data races can be computed by solving the decision problem for each of the $\binom{n}{2}$ event pairs, we obtain a sound algorithm for reporting *all races* that requires $O(n^4 \cdot \log n)$ time. In all cases, if the input trace consists of events of two processes, our race reports are also complete.

Our techniques rely on a new notion of *trace-closed partial orders*, which might be of independent interest. Informally, a closed partial order wrt a trace t is a partial order over a subset of events of t that respects (i) the observation $w(x)$ of each read event $r(x)$ in t, and (ii) the lock semantics. We define *max-min* linearizations of closed partial orders, and prove sufficient conditions under which a max-min linearization produces a valid trace. Finally, we show that given a partial order of small width, its closure can be computed in $O(n^2 \cdot \log n)$ time. To this end, we develop a data structure DS for maintaining the incremental transitive closure of directed acyclic graphs of small width. DS requires $O(n)$ initialization time, after which it supports edge insertions and reachability queries in $O(\log n)$ time. Here, the width of partial orders is bounded by the number of processes, which is a small constant compared to the length of the trace, and hence our data structure is relevant.

A practical algorithm and implementation. We develop an algorithm for the *function problem* of race detection that is more practical than simply solving the decision problem for all possible pairs. The efficiency of the algorithm comes while retaining the soundness and completeness guarantees. We also develop sufficient conditions for detecting dynamically that our algorithm is complete for a given input, even in cases where completeness is not guaranteed theoretically.

We make a prototype implementation of our practical algorithm and evaluate it on a standard set of benchmark traces that contain hundreds of millions of events. We compare the performance of our tool against state-of-the-art, polynomial-time, partial-order-based methods, namely the HB [Lamport 1978], WCP [Kini et al. 2017], DC [Roemer et al. 2018] and SHB [Mathur et al. 2018] methods. Our approach detects significantly more races than each of these methods, while it has comparable running time, and typically being faster. In fact, our algorithm does not simply detect more races; it detects *all* races in the benchmark traces, and soundly reports that no more races (other than the detected ones) exist. To our knowledge, this is the first sound algorithm that achieves such a level of performance on both running time and completeness of the reported races.

2 PRELIMINARIES

In this section we introduce useful notation and define the problem of dynamic race detection for lock-based concurrent programs. The model follows similar recent works (e.g., [Kini et al. 2017]).

Concurrent program. Given a natural number k, let $[k]$ denote the set $\{1, \ldots, k\}$. We consider a shared-memory concurrent program $\mathcal{P}$ that consists of k processes $\{p_i\}_{i \in [k]}$, under sequential consistency semantics. For simplicity of presentation we assume that k is fixed a-priori, and no process is created dynamically. All results presented here can be extended to a setting with dynamic process creation. Communication between processes occurs over a set of global variables $\mathcal{G}$, and synchronization over a set of locks $\mathcal{L}$. We let $\mathcal{V} = \mathcal{G} \cup \mathcal{L}$ be the set of all variables of $\mathcal{P}$. Each process is deterministic, and performs a sequence of operations on execution. We are only interested in the operations that access a global variable or a lock, which are called *events*.

(1) Given a global variable $x \in \mathcal{G}$, a process can *write/read* to x via an event $w(x)/r(x)$.
(2) Given a lock $l \in \mathcal{L}$, a process can *acquire* ℓ via an event $\mathrm{acq}(l)$ and *release* l via an event $\mathrm{rel}(l)$.

Each such event is atomic. Given an event e, we let $\mathrm{loc}(e)$ denote the global variable (or lock) that e accesses. We denote by $\mathcal{W}_p$ (resp. $\mathcal{R}_p$, $\mathcal{L}_p^A$, $\mathcal{L}_p^R$) the set of all write (resp. read, acquire, release) events that can be performed by process p. We let $\mathcal{E}_p = \mathcal{W}_p \cup \mathcal{R}_p \cup \mathcal{L}_p^A \cup \mathcal{L}_p^R$, and assume that $\mathcal{E}_p \cap \mathcal{E}_{p'} = \emptyset$ for every $p \neq p'$. We denote by $\mathcal{E} = \bigcup_p \mathcal{E}_p$, $\mathcal{W} = \bigcup_p \mathcal{W}_p$, $\mathcal{R} = \bigcup_p \mathcal{R}_p$, $\mathcal{L}^A = \bigcup_p \mathcal{L}_p^A$, $\mathcal{L}^R = \bigcup_p \mathcal{L}_p^R$ the events, write, read, acquire and release events of the program $\mathcal{P}$, respectively. Given an event $e \in \mathcal{E}$, we denote by $\mathrm{p}(e)$ the process that e belongs to. Finally, given a set of events $X \subseteq \mathcal{E}$, we denote by $\mathcal{R}(X)$ (resp., $\mathcal{W}(X)$, $\mathcal{L}^A(X)$, $\mathcal{L}^R(X)$) the set of read (resp., write, lock-acquire, lock-release) events of X.

Conflicting events. Given two distinct events $e_1, e_2 \in \mathcal{W} \cup \mathcal{R}$, we say that e_1 and e_2 are *conflicting*, denoted by $e_1 \bowtie e_2$, if (i) $\mathrm{loc}(e_1) = \mathrm{loc}(e_2)$ (i.e., they access the same global variable) and (ii) $\{e_1, e_2\} \cap \mathcal{W} \neq \emptyset$ (i.e., at least one is a write event). We extend the notion of conflict to locks, and say that two events $e_1, e_2 \in \mathcal{L}^A \cup \mathcal{L}^R$ are conflicting if $\mathrm{loc}(e_1) = \mathrm{loc}(e_2)$ (i.e., they use the same lock).

Event sequences. Let t be a sequence of events. We denote by $\mathcal{E}(t)$ the set of events, by $\mathcal{L}(t)$ the set of locks, and by $\mathcal{G}(t)$ the set of global variables in t. We let $\mathcal{W}(t)$ (resp., $\mathcal{R}(t)$, $\mathcal{L}^A(t)$, $\mathcal{L}^R(t)$) denote the set $\mathcal{W}(\mathcal{E}(t))$ (resp., $\mathcal{R}(\mathcal{E}(t))$, $\mathcal{L}^A(\mathcal{E}(t))$, $\mathcal{L}^R(\mathcal{E}(t))$). Given two distinct events $e_1, e_2 \in \mathcal{E}(t)$, we say that e_1 *is earlier than* e_2 in t, denoted by $e_1 <_t e_2$ iff e_1 appears before e_2 in t. We say that e_1 is *program-ordered earlier than* e_2, denoted by $e_1 <_{\mathrm{PO}(t)} e_2$, to mean that $e_1 <_t e_2$ and $\mathrm{p}(e_1) = \mathrm{p}(e_2)$. When t is clear from the context, we simply write PO to denote $\mathrm{PO}(t)$. We let $=^t$ be the identity relation on $\mathcal{E}(t)$, and denote by $\leq_t$, $\leq_{\mathrm{PO}}$ the relations $<_t \cup =_t$ and $<_{\mathrm{PO}} \cup =_t$ respectively. Given a set of events $X \subseteq \mathcal{E}$, we denote by $t|X$ the *projection* of t onto X, i.e., it is the sub-sequence of events of t that belong to X. Given two event sequences t_1, t_2, we denote by $t_1 \circ t_2$ the concatenation of t_1 with t_2. Finally, given a process p_i, we let $t|p_i = t|\mathcal{E}_{p_i}$.

Lock events. Given a sequence of events t and a lock-acquire event $\mathrm{acq} \in \mathcal{L}^A(t)$, we denote by $\mathrm{match}_t(\mathrm{acq})$ the earliest lock-release event in $\mathrm{rel} \in \mathcal{L}^R(t)$ such that $\mathrm{rel} \bowtie \mathrm{acq}$ and $\mathrm{acq} <_t \mathrm{rel}$, and let $\mathrm{match}_t(\mathrm{acq}) = \bot$ if no such lock-release event exists. If $\mathrm{match}_t(\mathrm{acq}) \neq \bot$, we require that $\mathrm{p}(\mathrm{acq}) = \mathrm{p}(\mathrm{match}_t(\mathrm{acq}))$, i.e., the two lock events belong to the same process. Similarly, given a lock-release event $\mathrm{rel} \in \mathcal{L}^R(t)$, we denote by $\mathrm{match}_t(\mathrm{rel})$ the acquire event $\mathrm{acq} \in \mathcal{L}^A(t)$ such that $\mathrm{match}_t(\mathrm{acq}) = \mathrm{rel}$ and require that such a lock-acquire event always exists.

Traces and observation functions. A sequence t is called a *trace* if it satisfies the following.

(1) For every read event $r \in \mathcal{R}(t)$, there exists a write event $w \in \mathcal{W}(t)$ such that $\mathrm{loc}(r) = \mathrm{loc}(w)$ and $w <_t r$.
(2) For any two lock-acquire events $\mathrm{acq}_1, \mathrm{acq}_2 \in \mathcal{L}^A(t)$, if $\mathrm{loc}(\mathrm{acq}_1) = \mathrm{loc}(\mathrm{acq}_2)$ and $\mathrm{acq}_1 <_t \mathrm{acq}_2$, then $\mathrm{rel}_1 = \mathrm{match}_t(\mathrm{acq}_1) \in \mathcal{L}^R(t)$ and $\mathrm{rel}_1 <_t \mathrm{acq}_2$.

Given a trace t, we define its *observation function* $O_t : \mathcal{R}(t) \to \mathcal{W}(t)$ as follows: $O_t(r) = w$ iff

$$\mathrm{loc}(r) = \mathrm{loc}(w) \quad \text{and} \quad w <_t r \quad \text{and} \quad \forall w' \in \mathcal{W}(t) \setminus \{w\} \text{ with } w \bowtie w' : w' <_t r \Rightarrow w' <_t w$$

In words, O_t maps every read event r to the write event w that r observes in t. For simplicity, we assume that t starts with a write event to every location, hence O_t is well-defined.

Enabled events and races. An event $e \in \mathcal{E}$ is said to be *enabled* in a trace t if $t^* = t \circ e$ is a trace of $\mathcal{P}$. A trace t is said to exhibit a *race* if there exist two consecutive conflicting events in t that belong

to different processes. Formally, there exist two events $e_1, e_2 \in \mathcal{R} \cup \mathcal{W}$ such that (i) $\mathrm{p}(e_1) \neq \mathrm{p}(e_2)$, (ii) $e_1 \bowtie e_2$, (iii) $e_1 <_t e_2$, and (iv) for every $e \in \mathcal{E}(t) \setminus \{e_1, e_2\}$, we have that $e <_t e_2 \Rightarrow e <_t e_1$.

Predictable races. A trace t' is a (prefix) *correct reordering* of another trace t if (i) for every process p_i, we have that $t'|p_i$ is a prefix of $t|p_i$ and (ii) $O_{t'} \subseteq O_t$, i.e., the observation functions of t' and t agree on their common read events. We say that t has a *predictable race* on a pair of events $e_1, e_2 \in \mathcal{E}(t)$ if there exists a correct reordering t' of t such that $t^* = t' \circ e_1 \circ e_2$ is a trace that exhibits the race (e_1, e_2).

Computational problems. The aim of this work is to present sound and fast algorithms for race detection, that also have certain completeness guarantees. As usual in algorithmic parlance, we are concerned with two versions of the problem, namely the following. Given an input trace t,

(1) the *decision problem* is stated on two events $e_1, e_2 \in \mathcal{E}(t)$, and asks whether (e_1, e_2) is a predictable race of t, and
(2) the *function problem* asks to compute the set of all pairs $\{(e_1^i, e_2^i)\}_i$ such that each (e_1^i, e_2^i) is a predictable race of t.

Soundness, completeness and complexity. A predictive race-detection algorithm is called *sound* if on every input trace t, every reported race is a predictable race of t. The algorithm is called *complete* if it reports all predictable races of t. We note that these notions are often used in reverse in program verification. However, here we align with the terminology used in predictive techniques, hence soundness (resp., completeness) means the absence of false positives (resp., false negatives). We measure complexity in terms of the length n of t. Other important parameters are the number of processes k and the number of global variables $\mathcal{G}$. Typically k is much smaller than n, and is treated as a constant. For simplicity, we also ignore $\mathcal{G}$ in our complexity statements. In all cases, our algorithms have a dependency of factor $k^2 \cdot |\mathcal{G}|$ (and hence polynomial) on these parameters.

Dynamic process creation and other synchronization primitives. To keep the presentation simple, in the theoretical part of this work we neglect dynamic process creation (i.e., fork/join events). We note that such events can be handled naturally in our framework. In our experiments (Section 6) we explain how we handle dynamic process creation, which is present in our benchmark set. Similarly, our focus on locks is for simplicity of presentation and not restrictive to our model. For dynamic race detection, other synchronization primitives, such as compare-and-swap, intrinsic locks and synchronized methods can be simulated with locks and extra orderings in the partial orders. Indeed, this modeling approach has been taken in many other works, as e.g. in [Kini et al. 2017; Mathur et al. 2018; Roemer et al. 2018; Smaragdakis et al. 2012].

Due to limited space, all proofs are relegated to a technical report [Pavlogiannis 2019].

3 TRACE-CLOSED PARTIAL ORDERS

In this section we present relevant notation on partial orders, and introduce the concept of closed partial orders. We also present max-min linearizations which linearize closed partial orders to valid traces. Since this our most technical section, we provide here an overview to assist the reader.

(1) In Section 3.1 we define general notation on partial orders. Since these are partial orders over sets of events X of an input trace t, we introduce a *feasibility* criterion for these sets, which requires that certain events are present in the partial order. For example, for every two conflicting lock-acquire events in X, at least one corresponding lock-release event must also be in X.
(2) In Section 3.2 we define trace-closed partial orders. Intuitively, this notion requires certain orderings between conflicting events to be present in the partial order. Note that *not* every

linearization of a partial order leads to a valid trace (e.g., some linearizations might not respect the lock semantics). Nevertheless, we show that for a specific class of trace-closed partial orders, a specific type of *max-min* linearization is guaranteed to *always* produce a valid trace.

(3) In Section 3.3 we develop an algorithm that computes the trace-closure of a partial order efficiently. To this end, we develop a data structure DS for the efficient representation of partial orders. For ease of presentation, we relegate the technical description of DS in Appendix A.

3.1 Partial Orders

Feasible sets. Given a set of events $X \subseteq \mathcal{E}(t)$, we say that X is *prefix-closed* for t if for every pair of events $e_1, e_2 \in \mathcal{E}(t)$ if $e_1 \leq_{\text{PO}} e_2$ and $e_2 \in X$, then $e_1 \in X$ (i.e., X is an ideal of $\leq_{\text{PO}}$). We define the *open acquires* of X under t as $\text{OpenAcqs}_t(X) = \{\text{acq} \in \mathcal{L}^A(X) : \ \text{match}_t(\text{acq}) \notin X\}$.

We call X *observation-feasible* for t if for every read event $r \in \mathcal{R}(X)$, we have $O_t(r) \in X$. We call X *lock-feasible* for t if (i) for every lock-release event $\text{rel} \in \mathcal{L}^R(X)$, we have $\text{match}_t(\text{rel}) \in X$, and (ii) for every distinct pair of lock-acquire events $\text{acq}_1, \text{acq}_2 \in \text{OpenAcqs}_t(X)$, we have $\text{loc}(\text{acq}_1) \neq \text{loc}(\text{acq}_2)$. In words, X is lock-feasible if every release event of X has its matching acquire event also in X, and every open lock of X remains open by exactly one acquire event of X. Finally, we call X *feasible* for t if X is prefix-closed, observation-feasible, and lock-feasible for t.

Partial orders. Given a trace t and a set $X \subseteq \mathcal{E}(t)$, a *partial order* $P(X)$ over X is a reflexive, antisymmetric and transitive relation over X (i.e., $\leq_{P(X)} \subseteq X \times X$). When X is clear from the context, we will simply write P instead of $P(X)$. Given two events e_1, e_2 we write $e_1 <_P e_2$ to denote that $e_1 \leq_P e_2$ and $e_1 \neq e_2$. Given two distinct events $e_1, e_2 \in X$, we say that e_1 and e_2 are *unordered* by P, denoted by $e_1 \parallel_P e_2$, if neither $e_1 <_P e_2$ nor $e_2 <_P e_1$. Given a set $Y \subseteq X$, we denote by $P|Y$ the *projection* of P on Y, i.e., we have $\leq_{P|Y} \subseteq Y \times Y$, and for all $e_1, e_2 \in Y$, $e_1 \leq_{P|Y} e_2$ iff $e_1 \leq_P e_2$. Given two partial orders P and Q over a common set X, we say that Q *refines* P, denoted by $Q \sqsubseteq P$, if for every pair of events $e_1, e_2 \in X$, if $e_1 \leq_P e_2$ then $e_1 \leq_Q e_2$. If Q refines P, we say that P is *weaker* than Q. A *linearization* of P is a total order that refines P. We make the following remark.

Remark 1. *Not every linearization of a partial order P is a valid trace, and generally, P is not guaranteed to have such a linearization. Our algorithm for dynamic race detection relies on developing sufficient conditions under which P indeed has a linearization to a valid trace.*

Width and Mazurkiewicz traces. Let P be a partial order over a set $X \subseteq \mathcal{E}(t)$. The *width* $\text{width}(P)$ of P is the length of its longest antichain. i.e., it is the largest size of a set $Y \subseteq X$ such that for every pair of distinct elements $e_1, e_2 \in Y$ we have $e_1 \parallel_P e_2$. The partial order P is called a *Mazurkiewicz trace* (or *M-trace* for short) if for every two conflicting events $e_1, e_2 \in X$, we have $e_1 \not\parallel_P e_2$ [Mazurkiewicz 1987]. Note that if $\text{width}(P) = 1$ then P is trivially an M-trace.

3.2 Trace-closed Partial Orders

In this section we define the notion of trace-closed partial orders. This is a central concept in this work, as our race-detection algorithm is based on computing trace-closed partial orders efficiently.

Trace-respecting partial orders. Let t be a trace, and P a partial order over a feasible set $X \subseteq \mathcal{E}(t)$. We say that P *respects* t if the following conditions hold.

(1) $P \sqsubseteq \text{PO}|X$, i.e., P refines the program order when restricted to the set X.
(2) For every read event $r \in \mathcal{R}(X)$ we have $O_t(r) <_P r$.
(3) For every lock-acquire event $\text{acq} \in \mathcal{L}^A(X)$, if $\text{match}_t(\text{acq}) \notin X$, then for every lock-release event $\text{rel} \in \mathcal{L}^R(X)$ such that $\text{rel} \bowtie \text{acq}$, we have $\text{rel} <_P \text{acq}$.

We denote by $R_t(X)$ the weakest partial order over X that respects t.

Trace-closed partial orders. Let t be a trace, and P a partial order over a feasible set $X \subseteq \mathcal{E}(t)$ such that P respects t. We call P *observation-closed* if the following condition holds. For every read event $r \in \mathcal{R}(X)$, let $w = O_t(r)$. For every write event $w' \in \mathcal{W}(X) \setminus \{w\}$ such that $w' \bowtie r$, we have

$$\text{if } w' <_P r \text{ then } w' <_P w \quad \text{and} \quad \text{if } w <_P w' \text{ then } r <_P w'$$

For a pair of lock-release events $\mathsf{rel}_1, \mathsf{rel}_2 \in \mathcal{L}^R(X)$, let $\mathsf{acq}_i = \mathsf{match}_t(\mathsf{rel}_i)$. We call P *lock-closed* if for every $\mathsf{acq}_1, \mathsf{acq}_2 \in \mathcal{L}^A$ and $\mathsf{rel}_1, \mathsf{rel}_2 \in \mathcal{L}^R$, if $\mathsf{rel}_2 \bowtie \mathsf{acq}_1$ and $\mathsf{acq}_1 \leq_P \mathsf{rel}_2$, then $\mathsf{rel}_1 \leq_P \mathsf{acq}_2$. Finally, we call P *trace-closed* (or simply *closed*) if it is both observation-closed and lock-closed. See Figure 3 for an illustration. Note that a closed partial order can still contain conflicting events that are unordered. In addition, it does not necessarily admit a linearization to a valid trace. In the next paragraph we develop sufficient conditions for when such a linearization exists.

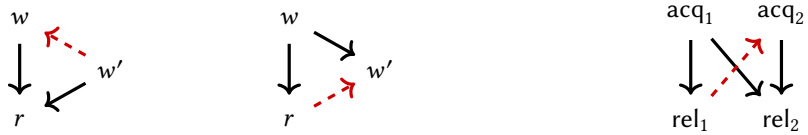


Fig. 3. The conditions of observation closure (a,b) and lock closure (c). Solid edges and dashed edges represent existing and inferred orderings, respectively.

Max-min linearizations. The key technical challenge in race prediction is, given a trace t, to construct a partial order P over $\mathcal{E}(t)$ such that P is efficiently linearizable to a correct reordering of t that manifests the race. Here we use trace-closed partial orders to provide a sufficient condition for efficient linearization, which we call the *max-min linearization*. In later sections, our race-detection algorithm constructs trace-closed partial orders. The max-min linearization of such partial orders will guarantee that the races exposed by these partial orders are indeed valid races, which are exhibited by a trace constructed using the max-min linearization.

Let t be a trace, and consider a partial order P over a feasible set $X \subseteq \mathcal{E}(t)$ such that P is trace-closed for t and X can be partitioned into two sets $X_1, X_2 \subseteq X$ such that (i) $\mathsf{width}(P|X_1) = 1$ and (ii) $P|X_2$ is an M-trace. The *max-min* linearization t^* is a linearization of P given by Algorithm 1. In words, first every event of X_1 is ordered before every event of X_2, as long as this is allowed by P, and then the resulting partial order is linearized arbitrarily. Intuitively, we obtain the sequence t^* by linearizing X_1 maximally, and X_2 minimally. See Figure 4 for an illustration.

Intuition. First, observe that P can contain pairs of conflicting events that are unordered, i,.e., between the sets X_1 and X_2. Conceptually, MaxMin shows that as we attempt to linearize P, we do not have to make an exhaustive search over all the possible (exponentially many) orderings of such pairs. Instead, the specific orderings made by MaxMin are guaranteed to produce a correct linearization. The intuition behind the correctness of MaxMin can be summarized as follows.

(1) Since $\mathsf{width}(P|X_1) = 1$, ordering every two events $e_1 \in X_1$, $e_2 \in X_2$ as $e_1 \to e_2$ (provided that $e_2 \not<_P e_1$) creates a partial order (i.e., no cycle is formed).
(2) Since P is closed and $P|X_2$ is an M-trace, this ordering respects the observation w of every read event r. Indeed, if the ordering was forcing some other conflicting write event w' between w and r, then w' must be ordered with at least one of w and r, and then the corresponding closure rule (Figure 3) would have resolved this conflict entirely.

Theorem 3.1. *Let t be a trace and P a partial order over a feasible set $X \subseteq \mathcal{E}(t)$ such that P is trace-closed for t and X can be partitioned into two sets X_1, X_2 so that (i) $\mathsf{width}(P|X_1) = 1$ and (ii) $P|X_2$ is a Mazurkiewicz trace. The max-min linearization of P produces a correct reordering of t.*

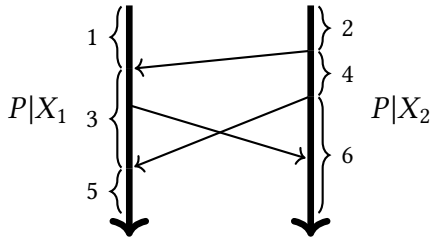


Fig. 4. Illustration of the max-min linearization. Here $\mathsf{width}(P|X_1) = \mathsf{width}(P|X_2) = 1$. The numbers show the order in which various segments of the sets X_1 and X_2 are executed, i.e., the linearization t^* consists of the segments $t^* = \langle 1\rangle \circ \langle 2\rangle \circ \langle 3\rangle \circ \langle 4\rangle \circ \langle 5\rangle \circ \langle 6\rangle$. Theorem 3.1 guarantees that t^* is a correct reordering.

Algorithm 1: MaxMin

Input: A trace t, a closed partial order P over a feasible set $X \subseteq \mathcal{E}(t)$, a partitioning of X to X_1, X_2 s.t. $\mathsf{width}(P|X_1) = 1$ and $P|X_2$ is an M-trace.
Output: A linearization of P that is a correct reordering of t.

```
Let Q ← P
foreach e1 ∈ X1, e2 ∈ X2 such that e1 ||_P e2 do
    Insert (e1 → e2) in Q
end
return any linearization t* of Q
```

3.3 Computing the Closure of a Partial Order

In this section we define the trace-closure of partial orders, and develop an efficient algorithm that, given a partial order P, either computes the closure of P or concludes that the closure does not exist. In the next section we will solve the decision problem of race detection by constructing specific partial orders and computing their closure.

Feasible partial orders. Let t be a trace and P a partial order over a feasible set $X \subseteq \mathcal{E}(t)$ such that P respects t. If there exists a partial order Q over X such that (i) $Q \sqsubseteq P$ and (ii) Q is closed, we define the *closure* of P as the smallest such partial order Q. If no such partial order Q exists, then the closure of P is undefined (i.e., P does not have a closure). We call P *feasible* iff it has a closure. The following lemma states that P has a unique closure.

Lemma 3.2. *There exists at most one smallest partial order Q such that (i) $Q \sqsubseteq P$ and (ii) Q is closed.*

Computing the closure of a partial order. It is straightforward to verify that, given a partial order P, the closure of P (or deducing that P is not feasible) can be computed in polynomial time. This is simply achieved by iteratively detecting whether one of the cases shown in Figure 3 is violated, and strengthening P with the appropriate orderings. However, since our goal is to handle large traces with hundreds of millions of events, polynomial-time guarantees are not enough, and

the goal is to develop an algorithm with low polynomial complexity. Here we develop such an algorithm called, Closure, that computes the closure of a partial order in $O(n^2 \cdot \log n)$ time.

The data structure DS. To make the closure computation efficient, we develop a data structure DS for manipulating partial-orders efficiently. Given a partial order P over n events such that P has width $k = O(1)$, DS represents P in $O(n)$ space and supports the following operations: (i) initialization in $O(n)$ time, (ii) querying whether $e_1 \leq_P e_2$, for any two events e_1, e_2 in $O(\log n)$ time, and (iii) inserting an ordering $e_1 \leq_P e_2$, for any two events e_1, e_2 in $O(\log n)$ time. For ease of presentation, we relegate the formal description of DS to Appendix A.

The event maps After, Before *and* $\mathcal{F}$. Consider a trace t. For every lock $l \in \mathcal{L}(t)$, we define the maps $\text{After}_l^{\mathcal{L}^A}, \text{After}_l^{\mathcal{L}^R}, \text{Before}_l^{\mathcal{L}^A}, \text{Before}_l^{\mathcal{L}^R} : \mathcal{E}(t) \to \mathcal{E}(t) \cup \{\bot\}$, as follows. Given an event $e \in \mathcal{E}(t)$, the maps $\text{After}_l^{\mathcal{L}^A}(e)$ and $\text{Before}_l^{\mathcal{L}^A}(e)$ point to the first lock-acquire event acq after e in t, and last lock-acquire event acq before e in t, respectively, such that $\text{p}(e) = \text{p}(\text{acq})$ and $\text{loc}(\text{acq}) = l$. The maps $\text{After}_l^{\mathcal{L}^R}(e)$ and $\text{Before}_l^{\mathcal{L}^R}(e)$ are defined analogously, pointing to lock-release instead of lock-acquire events. Similarly, for every global variable $x \in \mathcal{G}(t)$, we define the maps $\text{After}_x^{\mathcal{W}}, \text{After}_x^{\mathcal{R}}, \text{Before}_x^{\mathcal{W}}, \text{Before}_x^{\mathcal{R}} : \mathcal{E}(t) \to \mathcal{E}(t) \cup \{\bot\}$, as follows. Given an event $e \in \mathcal{E}(t)$, the map $\text{After}_x^{\mathcal{W}}(e)$ (resp. $\text{Before}_x^{\mathcal{W}}(e)$) points to the first write event w after (resp., before) e in t such that $\text{p}(e) = \text{p}(w)$ and $\text{loc}(w) = x$. The maps $\text{After}_l^{\mathcal{R}}(e)$ and $\text{Before}_l^{\mathcal{R}}(e)$ are defined analogously, pointing to read instead of write events. Finally, the *flow map* $\mathcal{F}_p : \mathcal{W}(t) \to \mathcal{R}(t) \cap \mathcal{R}_p$ of t is a partial function that maps each write event w to the last read event of p that observes w. In all the above cases, if no corresponding event exists, the respective map points to $\bot$. Observe that each of these maps has size $O(|G| \cdot n)$, where $|G|$ is the number of memory locations of t. The maps can be constructed in $O(|G| \cdot n)$ time, simply by traversing t and maintaining on-the-fly each map.

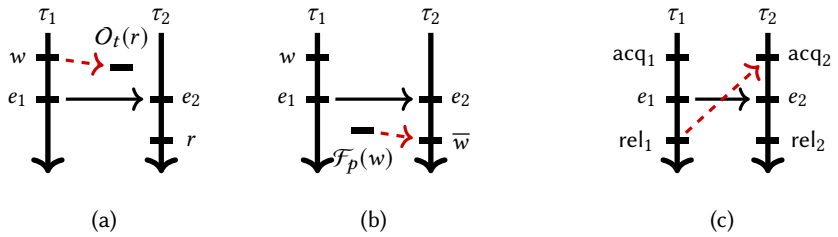


Fig. 5. Illustration of ObsClosure(e_1, e_2) (a, b) and LockClosure(e_1, e_2) (c) for an edge (e_1, e_2) added in P. In each case, the dashed edge corresponds to the new ordering inserted in P. Recall that $O_t(r)$ denotes the observation of r in t, and $\mathcal{F}_p(w)$ denotes the last read event of process p that observes w in t.

Algorithm 3: ObsClosure(e_1, e_2)

```
1 foreach x ∈ G(t) do
2 |  Let r ← After_x^R(e_2)
3 |  Let w ← Before_x^W(e_1)
4 |  if O_t(r) ≠ w then Q.push(w, O_t(r))
5 |  Let w̄ ← After_x^W(e_2)
6 |  foreach p ∈ {p_i}_i do Q.push(F_p(w), w̄)
7 end
```

Algorithm 4: LockClosure(e_1, e_2)

```
1 foreach ℓ ∈ L(t) do
2 |  Let acq_1 ← Before_ℓ^{L^A}(e_1)
3 |  Let rel_1 ← match_t(acq_1)
4 |  Let rel_2 ← After_l^{L^R}(e_2)
5 |  Let acq_2 ← match_t(rel_2)
6 |  Q.push(rel_1, acq_2)
7 end
```

The algorithm Closure. We now present Closure for computing the closure of a partial order P over a set X, or concluding that P is not feasible. The algorithm maintains a partial order as a DAG

Algorithm 2: Closure

Input: A trace t, a partial order P over a set X s.t. P respects t and is represented as a DAG $G = (V, E)$.
Output: The closure of P, if it exists, otherwise $\bot$.

```
// Initialization - P is represented as k total orders {τ_i}_i with extra orderings between τ_i
Initialize the data structure DS for G
Q ← an empty worklist
foreach e_1 ∈ V do                                              // Push partial-order edges
    foreach i ∈ [k] do
        Let e_2 ← DS.successor(e_1, i)          // The first successor of e_1 in the total order τ_i
        DS.insert(e_1, e_2)                                      // Insert the edge in DS
        ObsClosure(e_1, e_2)                                     // Resolve observations
        LockClosure(e_1, e_2)                                    // Resolve locks
    end
end
// Main computation
while Q is not empty do
    (ē_1, ē_2) ← Q.pop()
    if DS.query(ē_2, ē_1) = True then return ⊥                   // Cycle formed, abort
    if DS.query(ē_1, ē_2) = False then                           // Edge not present
        DS.insert(ē_1, ē_2)              // Besides ē_1 ≤_Q ē_2, inserts O(k^2) transitive orderings
        foreach (e_1, e_2) inserted do
            ObsClosure(e_1, e_2)                                 // Resolve observations
            LockClosure(e_1, e_2)                                // Resolve locks
        end
end
return DS                                  // At this point DS represents the closure of P
```

represented by the data structure DS. Conceptually, DS consists of k total orders, $\tau_1, \ldots, \tau_k$, where $k = \text{width}(P)$, with some extra orderings that go across the τ_i. Each total order τ_i contains the events of process p_i in X. Initially DS represents P. The main computation iterates over a worklist Q which holds edges to be inserted in DS. Upon extracting such an edge $(\overline{e}_1, \overline{e}_2)$ from Q, the algorithm inserts the edge in DS using the operation DS.insert. This operation results in various edges (e_1, e_2) inserted in the graph, transitively through $(\overline{e}_1, \overline{e}_2)$. For every (e_1, e_2), the algorithm calls methods ObsClosure and LockClosure to resolve any violation of observation and lock constraints created by the insertion of (e_1, e_2). Figure 5 illustrates ObsClosure and LockClosure. Algorithm 2, Algorithm 3 and Algorithm 4 give the description of Closure, ObsClosure and LockClosure, respectively.

Correctness and complexity. It is rather straightforward that if P has a closure Q, then for each Q.push(e_1, e_2) operation performed by Closure, we have $e_1 <_Q e_2$. It follows that if Closure returns $\bot$, then P is unfeasible. On the other hand, if Closure does not return $\bot$, then the partial order Q stored in the data structure DS is the closure of P. Indeed, each of the closure rules can only be violated by an ordering $e_1 <_Q e_2$. The algorithm guarantees that every such edge is processed by the methods ObsClosure and LockClosure, and new edges will be inserted in DS according to the rules of Figure 5. After such edges have been inserted, the ordering $e_1 <_Q e_2$ can no longer violate any of the conditions of closure. Regarding the time complexity, the algorithm inserts at most n^2 edges in the partial order represented by DS. Using the algorithms for DS (see Lemma A.1 in

Appendix A), for every edge inserted by the algorithm, identifying which other edges are imposed by the closure rules requires only $O(\log n)$ time. We have the following theorem.

Theorem 3.3. *Let t be a trace and P a partial order over a feasible set $X \subseteq \mathcal{E}(t)$ such that P respects t.* Closure *correctly computes the closure of P and requires $O(n^2 \cdot \log n)$ time.*

Incremental closure. In our race detection algorithm, we also make use of the following operation on partial orders. Let t be a trace and P a partial order over a feasible set $X \subseteq \mathcal{E}(t)$ and such that P is closed wrt t and is represented as a DAG using the data structure DS. Given a pair of events $e_1, e_2 \in X$, we define the operation InsertAndClose$(P, e_1 \to e_2)$ as follows. We execute the algorithm Closure starting from Line 15, performing a DS.insert(e_1, e_2). Hence we perform the ObsClosure and LockClosure only for the new orderings added due to (e_1, e_2).

Lemma 3.4. *Let Σ be a sequence of* InsertAndClose *operations. Performing all operations of Σ requires $O(n^2 \cdot \log n + |\Sigma| \cdot \log n)$ time in total, and produces a closed partial order.*

4 THE DECISION PROBLEM OF RACE DETECTION

Here we present a polynomial-time algorithm for the decision problem of dynamic race detection, i.e., given an input trace t and two events $e_1, e_2 \in \mathcal{E}(t)$, decide whether (e_1, e_2) is a predictable race of t. Our algorithm is sound but incomplete in general, and it becomes complete if the input trace contains events of only two processes. To assist the reader, we provide an outline of this section.

(1) First, we introduce the notion of relative causal cone of an event e, which is a subset of events of t. Intuitively, it can be thought of as a trace slice of t up to e. When deciding whether (e_1, e_2) is a race of t, our algorithm tries to find a witness trace for the race, such that the witness consists of events of the causal cones of e_1 and e_2.
(2) Second, we present our main algorithm RaceDecision for deciding whether (e_1, e_2) is a predictable race of t. In high level, the algorithm computes the causal cones of e_1 and e_2, and constructs a partial order P over events of the causal cones. Afterwards, it computes the closure Q of P using algorithm Closure from the previous section, and resolves certain orderings of conflicting events in Q according to the trace order in t. Finally, it uses algorithm MaxMin to linearize Q to a witness trace that exhibits the race. We also prove that this process is *sound*, i.e., if (e_1, e_2) is reported as a race, then it is a true predictable race of t.
(3) Third, we prove that the above process is also *complete* for traces that consist of events of two processes, i.e., it detects all predictable races.
(4) Finally, we illustrate RaceDecision on a few examples.

Relative causal cones. Given a trace t, an event $e \in \mathcal{E}(t)$ and a process p, the *causal past cone* $\mathsf{RCone}_t(e, p)$ of e *relative to* p in t is the smallest set that contains the following events:

(1) For every event $e' \in \mathcal{E}(t)$ with $e' <_{\mathsf{PO}} e$, we have that $e' \in \mathsf{RCone}_t(e, p)$.
(2) For every pair of events $e_1 \in \mathsf{RCone}_t(e, p)$ and $e_2 \in \mathcal{E}(t)$, if $e_2 \leq_{\mathsf{PO}} e_1$ then $e_2 \in \mathsf{RCone}_t(e, p)$.
(3) For every read event $r \in \mathcal{R}(\mathsf{RCone}_t(e, p))$, we have that $\mathsf{O}_t(r) \in \mathsf{RCone}_t(e, p)$.
(4) For every lock-acquire event $\mathsf{acq} \in \mathsf{RCone}_t(e, p)$, if $\mathsf{p}(\mathsf{acq}) \neq \mathsf{p}(e)$ and $\mathsf{p}(\mathsf{acq}) \neq p$, then $\mathsf{match}_t(\mathsf{acq}) \in \mathsf{RCone}_t(e, p)$.

It is easy to verify that $\mathsf{RCone}_t(e, p)$ is always observation-feasible but not necessarily lock-feasible.

Intuition and example on relative causal cones. In order to decide whether an event pair (e_1, e_2) is a predictable race of an input trace t, we first need to decide the events that will constitute

a witness trace t^* that exposes the race. In our race-detection algorithm, we take this set to be $\mathrm{RCone}_t(e_1, \mathrm{p}(e_2)) \cup \mathrm{RCone}_t(e_2, \mathrm{p}(e_1))$, i.e., it is the causal past cone of each focal event relative to the process of the other focal event. Conditions 1-3 ensure that the cones are closed wrt the program order, and the observation of every read event is present, which is required for t^* to be a correct reordering of t. The intuition behind condition 4 is a bit more subtle. To avoid having two critical sections on the same lock open, we include the matching release event of every lock-acquire event. However, this rule does not apply for the processes of the focal events, since for these processes the events we have to include in t^* are precisely the predecessors of the corresponding focal event.

Consider the input trace in Figure 6, where our task is to detect the race (e_2, e_{10}). We outline here the computation of the relative causal cones $\mathrm{RCone}_t(e_1, \mathrm{p}(e_2))$ and $\mathrm{RCone}_t(e_2, \mathrm{p}(e_1))$. Item 1 of relative causal cones leads to $\mathrm{RCone}_t(e_2, p_3) = \{e_1\}$. For $\mathrm{RCone}_t(e_{10}, p_1)$, Item 1 makes $e_9 \in \mathrm{RCone}_t(e_{10}, p_1)$. Since e_9 is a read event, Item 3 makes $O_t(e_9) = e_5 \in \mathrm{RCone}_t(e_{10}, p_1)$, and then Item 2 makes $e_4 \in \mathrm{RCone}_t(e_{10}, p_1)$. Since e_4 is a lock-acquire event and $\mathrm{p}(e_4) \neq p_1, p_3$ (i.e., the process of e_4 is neither the process of e_{10}, nor the process relative to which we are computing the causal cone of e_{10}), Item 4 makes $\mathrm{match}_t(e_4) = e_6 \in \mathrm{RCone}_t(e_{10}, p_1)$. Hence, in the end, $\mathrm{RCone}_t(e_{10}, p_1) = \{e_9, e_6, e_5, e_4\}$.

	τ_1	τ_2	τ_3
1	acq(ℓ)		
2	w(x)		
3	rel(ℓ)		
4		acq(ℓ)	
5		$w(y)$	
6		rel(ℓ)	
7	$w(z)$		
8		$r(z)$	
9			$r(y)$
10			w(x)

(a) An input trace t.

$$\mathrm{RCone}_t(e_2, p_3) = \{e_1\}$$
$$\mathrm{RCone}_t(e_{10}, p_1) = \{e_9, e_6, e_5, e_4\}$$

(b) The relative causal cones $\mathrm{RCone}_t(e_2, p_3)$ and $\mathrm{RCone}_t(e_{10}, p_1)$

Fig. 6. The relative causal cones when testing for a race (e_2, e_{10}).

The algorithm RaceDecision**.** We now describe our algorithm for reporting whether t has a predictable race on a given pair (e_1, e_2). In words, the algorithm constructs a set X that is the union of the causal cones of each e_i relative to the process of e_{3-i}. Afterwards, the algorithm constructs a partial order P that respects X, and computes the closure Q of P. Finally, the algorithm non-deterministically chooses some $i \in [2]$, and examines all events that belong to processes other than p_i. For every two such events $\overline{e}_1, \overline{e}_2$, if they conflict and are unordered by Q, the algorithm orders them according to their order in t. If a cycle is created in Q during this process, the algorithm returns False. Otherwise, at the end of this process, the set X can be naturally partitioned into two sets X_1, X_2 such that $\mathrm{width}(Q|X_1) = 1$ and $Q|X_2$ is an M-trace. The first set is $X_1|\mathrm{p}(e_i)$, i.e., it contains the events of the process in which e_i belongs to, and thus is a totally ordered set under Q. The second set is $X_2 = X \setminus X_1$, and note that all pairs of conflicting events of X_2 are now ordered under Q. Hence, according to Theorem 3.1, the partial order Q is linearizable to a valid trace, and the algorithm returns True. See Algorithm 5 for a formal description.

The complexity of the algorithm is $O(n^2 \cdot \log n)$, which is the time required for computing the closure of the partial order Q in Line 4 and Line 8 (due to Theorem 3.3 and Lemma 3.4, respectively).

Note that the algorithm is sound, i.e., if it returns True then (e_1, e_2) is a true predictable race of t. On the other hand, the algorithm is incomplete in general, i.e., it might return False even though (e_1, e_2) is a true predictable race of t. For example, in Line 9, the algorithm orders some pairs of conflicting events $(\overline{e}_1, \overline{e}_2)$ in the same order as in the input trace. Although these orderings are expected to work most of the time, this choice might not always be correct. In such cases, a cycle will be created in Q, and the algorithm will return False (see [Pavlogiannis 2019] for an example). As we discuss next, the algorithm becomes complete if the input trace consists of only two processes.

Algorithm 5: RaceDecision

Input: A trace t and two events $e_1, e_2 \in \mathcal{E}(t)$ with $e_1 \bowtie e_2$.
Output: True if (e_1, e_2) is detected as a predictable race of t.

1 Let $X \leftarrow \mathsf{RCone}_t(e_1, \mathsf{p}(e_2)) \cup \mathsf{RCone}_t(e_2, \mathsf{p}(e_1))$
2 **if** $\{e_1, e_2\} \cap X \neq \emptyset$ *or* X *is not feasible* **then return** False // No race
3 Let $P \leftarrow R_t(X)$ // The weakest po that respects t
4 Let $Q \leftarrow \mathsf{Closure}(t, P, X)$ // Trace-close P
5 **if** $Q = \bot$ **then return** False // Closure created a cycle, no race
6 Non-deterministically chose $i \in [2]$ // In practice, try both $i = 1$ and $i = 2$
7 **while** $\exists \overline{e}_1, \overline{e}_2 \in X \setminus \mathcal{E}_{\mathsf{p}(e_i)}$ *s.t.* $\overline{e}_1 \bowtie \overline{e}_2$ *and* $\overline{e}_1 \parallel_Q \overline{e}_2$ *and* $\overline{e}_1 <_t \overline{e}_2$ **do**
8 $\quad Q \leftarrow \mathsf{InsertAndClose}(Q, \overline{e}_1 \rightarrow \overline{e}_2)$ // Order $\overline{e}_1 \rightarrow \overline{e}_2$ in Q and close Q
9 $\quad$ **if** $Q = \bot$ **then return** False // Closure created a cycle, no race
10 **end**
11 **return** True

Completeness for two processes. We now discuss the completeness properties of RaceDecision for reporting races on input traces of two processes. Assume that (e_1, e_2) is a race of the input trace. The key insight is that Line 7 of RaceDecision is not executed, as every pair of events $\overline{e}_1, \overline{e}_2$ in that line belong to the same process, and thus are already ordered. Up until that point, all orderings used in constructing the partial order $R_t(X)$ and computing the closure of $R_t(X)$ are necessarily present in every trace that witnesses the race (e_1, e_2). Hence the closure computation cannot return $\bot$, and RaceDecision returns True. The following theorem concludes the results of this section.

Theorem 4.1. *Let t be a trace of $k \geq 2$ processes, and $n = |\mathcal{E}(t)|$. Let $e_1, e_2 \in \mathcal{E}(t)$ be two conflicting events of t. The algorithm* RaceDecision *requires $O(n^2 \cdot \log n)$ time and soundly reports whether (e_1, e_2) is a predictable race of t. If $k = 2$,* RaceDecision *is also complete. If* RaceDecision *reports a race, a witness trace can be constructed in $O(n \cdot \log n)$ time.*

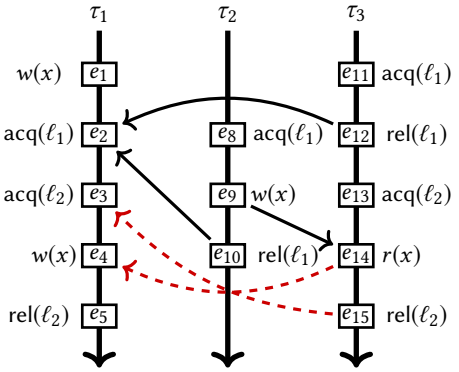
As there are $O(n^2)$ pairs of events in t, Theorem 4.1 yields the following corollary.

Corollary 4.2. *Let t be a trace of $k \geq 2$ processes. There exists a sound algorithm that requires $O(n^4 \cdot \log n)$ time and soundly reports predictable races of t. If $k = 2$, the algorithm is also complete (i.e., it reports* all *predictable races).*

Remark 2. *We note that the dependency of Corollary 4.2 on the number of variables $|\mathcal{G}|$ and number of threads k is $O(|\mathcal{G}| \cdot k^2 \cdot n^4 \cdot \log n)$. To keep the presentation simple, we have neglected the dependency on $|\mathcal{G}|$ and k in the analysis of the algorithm.*

	τ_1	τ_2	τ_3
1	$w(x)$		
2	acq(ℓ_1)		
3	acq(ℓ_2)		
4	$w(x)$		
5	rel(ℓ_2)		
6	w(y)		
7	rel(ℓ_1)		
8		acq(ℓ_1)	
9		$w(x)$	
10		rel(ℓ_1)	
11			acq(ℓ_1)
12			rel(ℓ_1)
13			acq(ℓ_2)
14			$r(x)$
15			rel(ℓ_2)
16			r(y)

(a) Is (e_6, e_{16}) a race?



(b) The partial order P and its closure Q.

	τ_1	τ_2	τ_3
1	$w(x)$		
2		acq(ℓ_1)	
3		$w(x)$	
4		rel(ℓ_1)	
5			acq(ℓ_1)
6			rel(ℓ_1)
7			acq(ℓ_2)
8			$r(x)$
9			rel(ℓ_2)
10	acq(ℓ_1)		
11	acq(ℓ_2)		
12	$w(x)$		
13	rel(ℓ_2)		
14	w(y)		
15			r(y)

(c) The witness trace.

Fig. 7. (a) The input trace. (b) The partial order P (solid edges) and its closure Q (solid and dashed edges). (c) The witness trace obtained by extending MaxMin(Q) with the racy pair (e_6, e_{16}).

4.1 Examples

We now illustrate the algorithm RaceDecision on a few examples.

Example of a race (Figure 7). Consider the trace t shown in Figure 7a, and the task is to decide whether (e_6, e_{16}) is a predictable race of t. The algorithm constructs the causal cones

$$\mathsf{RCone}_t(e_6, p_3) = \{e_i\}_{i=1}^{5} \quad \text{and} \quad \mathsf{RCone}_t(e_{16}, p_1) = \{e_i\}_{i=8}^{15}$$

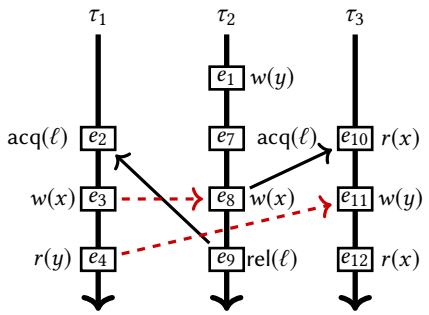
and the partial order P that respects t, shown in Figure 7b in solid edges. Afterwards, the algorithm computes the closure Q of P by inserting the dashed edges in Figure 7b. In particular, since for the write event e_4 we have $e_9 <_P e_4$ and e_9 is observed by the read event e_{14}, we have $e_{14} <_Q e_4$ (i.e., this is an observation-closure edge). After this ordering is inserted, for the lock-acquire event e_{13} we have $e_{13} <_Q e_5$ and e_5 is a lock-release event on the same lock, we also have $e_{15} <_Q e_3$, where $e_{15} = \mathsf{match}_t(e_{13})$ and $e_3 = \mathsf{match}_t(e_5)$ (i.e., this is a lock-closure edge). Now consider the nondeterministic choice made in Line 6 of RaceDecision such that $i = 1$. The algorithm also orders $e_{10} <_Q e_{11}$ by performing InsertAndClose($Q, e_{10} \to e_{11}$). Notice that, after this operation, Q is a closed partial order. In addition, the $w(x)$ event e_1 is unordered with the conflicting events e_9 and e_{10}, i.e., Q is not an M-trace. However, by taking $X_1 = X|_{p_1}$ and $X_2 = X \setminus X_2$, we have that $\mathsf{width}(Q|X_1) = 1$ and $Q|X_2$ is an M-trace. Hence, by Theorem 3.1 Q is linearizable to a correct reordering, constructed as the max-min linearization $t^* = \mathsf{MaxMin}(Q)$. This illustrates the advantage of our technique over existing methods, as here a correct reordering is exposed even though the initial partial order P contains several pairs of conflicting events that are unordered. Indeed, this race is missed by all HB, WCP, DC and SHB. Finally, the witness trace in is constructed by extending t^* with the racy events e_6, e_{16}, shown in Figure 7c.

Example of a non-race (Figure 8). Consider the trace t shown in Figure 8a, and the task is to decide whether (e_5, e_{13}) is a predictable race of t. The algorithm constructs the causal cones

$$\mathsf{RCone}_t(e_5, p_3) = \{e_i\}_{i=1}^{4} \quad \text{and} \quad \mathsf{RCone}_t(e_{13}, p_1) = \{e_1\} \cup \{e_i\}_{i=7}^{12}$$

	τ_1	τ_2	τ_3
1		$w(y)$	
2	acq(ℓ)		
3	$w(x)$		
4	$r(y)$		
5	**w(z)**		
6	rel(ℓ)		
7		acq(ℓ)	
8		$w(x)$	
9		rel(ℓ)	
10			$r(x)$
11			$w(y)$
12			$r(x)$
13			**r(z)**

(a) Is (e_5, e_{13}) a race?



(b) The partial order P is not feasible.

Fig. 8. (a) The input trace. (b) The partial order P (solid edges) and its closure Q (solid and dashed edges).

and the partial order P that respects t, shown in Figure 8b in solid edges. Afterwards, the algorithm computes the closure Q of P by inserting the dashed edges in Figure 8b. Observe that Q contains a cycle and thus P is not feasible, hence the algorithm reports that (e_4, e_{13}) is not a race[1].

Examples from Section 1. Finally, we outline RaceDecision on the two races from Figure 1.

Example from Figure 1a. The algorithm constructs the causal cones

$$\mathsf{RCone}_t(e_2, p_2) = \{e_1\} \quad \text{and} \quad \mathsf{RCone}_t(e_7, p_1) = \{e_i\}_{i=4}^{6}$$

and the partial order P that respects t by forcing the ordering $e_6 <_P e_1$. Note that P is already closed, hence $Q = P$ by the algorithm Closure. Finally, the witness trace is constructed by obtaining the max-min linearization $t^* = \mathsf{MaxMin}(Q)$ and extending t^* with the racy events e_2, e_7, thereby witnessing the race by the trace $t^* = e_4, e_5, e_6, e_1, e_2, e_7$.

Example from Figure 1b. The algorithm constructs the causal cones

$$\mathsf{RCone}_t(e_2, p_3) = \{e_1\} \quad \text{and} \quad \mathsf{RCone}_t(e_{14}, p_1) = \{e_i\}_{i=5}^{13}$$

and the partial order P that respects t by forcing the orderings $e_{10} <_P e_1$ and $e_7 <_P e_{12}$. Afterwards, the algorithm computes the closure of Q by inserting $e_8 <_Q e_{11}$. Finally, the witness trace is constructed by obtaining the max-min linearization $t^* = \mathsf{MaxMin}(Q)$ and extending t^* with the racy events e_2, e_{14}, thereby witnessing the race by the trace $t^* = e_5, e_6, e_7, e_8, e_9, e_{10}, e_1, e_{11}, e_{12}, e_{13}, e_2, e_{14}$.

5 THE FUNCTION PROBLEM IN PRACTICE

Corollary 4.2 solves the function problem by solving the decision problem on every pair of events of the input trace. Here we present an explicit algorithm for the function problem, called M2, which is the main contribution of this work. Although M2 does not improve the worst-case complexity, it is faster in practice. The algorithm relies on the following simple lemma.

Lemma 5.1. *Consider two conflicting events e_1, e_2 and let $X = \mathsf{RCone}_t(e_1, \mathsf{p}(e_2)) \cup \mathsf{RCone}_t(e_2, \mathsf{p}(e_1))$. If $X \cap \{e_1, e_2\} = \emptyset$ and $\mathsf{OpenAcqs}_t(X) = \emptyset$ then (e_1, e_2) is a predictable race of t.*

[1]While computing the closure different cycles might appear, depending on the order in which the closure rules are applied.

Intuitively, if the conditions of Lemma 5.1 are met, we can postpone the execution of e_1 in t until e_2, and the trace witnessing the race is simply $t|X \circ e_1, e_2$.

The algorithm $\mathsf{M2}^p_{e_1}$. We are now ready to describe an algorithm for partially solving the function problem on an input trace t. In particular, we present the algorithm $\mathsf{M2}^p_{e_1}$ given an event $e_1 \in \mathcal{E}(t)$ and a process $p \neq \mathrm{p}(e_1)$. The algorithm returns the set $\mathcal{Z} \subseteq \{e_1\} \times \mathcal{E}(t)|p$ of races detected between e_1 and events of process p. The algorithm simply iterates over all events e_2 of p in increasing order and computes the causal cone $\mathrm{RCone}_t(e_2, \mathrm{p}(e_1))$. Let $X = \mathrm{RCone}_t(e_1, \mathrm{p}(e_2)) \cup \mathrm{RCone}_t(e_2, \mathrm{p}(e_1))$. If there are open lock-acquire events in X, the algorithm invokes RaceDecision for solving the decision problem on (e_1, e_2). Otherwise, a race (e_1, e_2) is directly inferred, due to Lemma 5.1. Algorithm 6 gives the formal description. The efficiency of $\mathsf{M2}^p_{e_1}$ lies on two observations:

Algorithm 6: $\mathsf{M2}^p_{e_1}$

Input: A trace t, an event $e_1 \in \mathcal{E}(t)$, a process $p \neq \mathrm{p}(e_1)$.
Output: A set $\mathcal{Z} \subseteq \{e_1\} \times \mathcal{E}(t)$ of predictable races of t.

```
1  Let Z ← ∅
2  Let X ← RCone_t(e_1, p)
3  foreach e_2 ∈ E(p) in increasing order of <_PO(t) s.t. e_2 ∉ X and e_1 ⋈ e_2 do
4  |  Insert RCone_t(e, p(e_1)) \ X in X        // At this point X = RCone_t(e_1, p) ∪ RCone_t(e, p(e_1))
5  |  if e_1 ∈ X then return Z                  // e_1 ∈ X for all remaining e_2, return early
6  |  if OpenAcqs_t(X) = ∅ then                 // No open locks, race found
7  |  |  Insert (e_1, e_2) in Z
8  |  else
   |  |  // Open locks, use the decision algorithm
9  |  |  if RaceDecision(e_1, e_2) then Insert (e_1, e_2) in Z
10 |  end
11 end
```

(1) Since critical sections tend to be small, we expect the condition in Line 6 to be False only a few times, thus Lemma 5.1 allows to soundly report a race without constructing a partial order.
(2) The causal cones are closed wrt the program order. That is, given two events e_2 and e_2' with $e_2 <_{\mathrm{PO}} e_2'$, we have that $\mathrm{RCone}_t(e_2, \mathrm{p}(e_1)) \subset \mathrm{RCone}_t(e_2', \mathrm{p}(e_1))$, and thus we only need to consider the difference $\mathrm{RCone}_t(e_2', \mathrm{p}(e_1)) \setminus \mathrm{RCone}_t(e_2, \mathrm{p}(e_1))$ in Line 4. This decreases the total time for constructing all causal cones from quadratic to linear.

The algorithm M2. Finally, we outline the algorithm M2 for solving the function problem. Given an input trace t, the algorithm simply invokes $\mathsf{M2}^p_{e_1}$ for every event $e_1 \in \mathcal{E}(t)$ and process $p \neq \mathrm{p}(e_1)$ and obtains the returned race set $\mathcal{Z}^p_{e_1}$. Since there are $O(n)$ such events, the algorithm makes $O(n)$ invocations. The reported set of predictable races of t is then $\mathcal{Z} = \bigcup_{e_1,p} \mathcal{Z}^p_{e_1}$.

Detecting completeness dynamically. Assume that we execute RaceDecision on input events e_1, e_2 and the algorithm returns False. It can be easily shown that if the following conditions hold, then (e_1, e_2) is not a predictable race of t (and hence correctly rejected by RaceDecision).

(1) When computing the relative causal past cones in Line 1 of RaceDecision, no event is added to the cones due to Item 4 of the definition of relative causal past cones.
(2) RaceDecision returns False before executing Line 8.

Let C be the set of races that are rejected by M2 on input trace t such that at least one of the conditions above does not hold. It follows that C over-approximates the set of false negatives of M2. The algorithm is *dynamically complete* for t if $C_t = \emptyset$.

6 EXPERIMENTS

In this section we report on an implementation and experimental evaluation of our techniques.

6.1 Implementation

We have implemented our algorithm M2 in Java and evaluated its performance on a standard set of benchmarks. We first discuss some details of the implementation.

Handling dynamic processes creation. In the theoretical part of this paper we have neglected dynamic process creation events. In practice such events are common, and all our benchmark traces contain fork(i) and join(j) events (for forking process p_i and joining with process p_j, respectively). To handle such events, we include in the program order PO the following order relationships: If e_1 is a fork(i) event and e_2 is a join(j) event, then we include the order relationship $(e_1 <_{\text{PO}} e_1')$ and $(e_2' <_{\text{PO}} e_2)$, where e_1' is the first event of p_i and e_2' is the last event of p_j such that $e_2' <_t e_2$.

Optimizations. We make two straightforward optimizations, namely, ignoring non-racy locations and over-approximating the racy events. Recall that $R_t(X)$ is the weakest partial order over the events of t that respects t. $R_t(X)$ can be constructed efficiently by a single pass of t. For the first optimization, we simply remove from t all events to a location x if every pair of conflicting events on x is ordered by $R_t(X)$. For the second optimization, we construct the set

$$A = \{(e_1, e_2) : e_1 \bowtie e_2 \quad \text{and} \quad e_1 \parallel_{R_t(X)} e_2 \quad \text{and} \quad e_1, e_2 \text{ are not protected by the same lock}\}.$$

which over-approximates the races of t, and, we only consider the pairs $(e_1, e_2) \in A$ for races.

6.2 Experimental Setup

Benchmarks. Our benchmark set is a standard one found in recent works on race detection [Huang et al. 2014; Kini et al. 2017; Mathur et al. 2018; Yu et al. 2018], and parts of it also exist in other works [Bond et al. 2010; Flanagan and Freund 2009; Roemer et al. 2018; Zhai et al. 2012]. It contains concurrent traces of various sizes, which are concrete executions of concurrent programs taken from standard benchmark suits: (i) the IBM Contest benchmark suite [Farchi et al. 2003], (ii) the Java Grande forum benchmark suite [Smith et al. 2001], (iii) the DaCapo benchmark suite [Blackburn et al. 2006], and (iv) some standalone, real-world software. We have also included the benchmark `cryptorsa` from the SPEC JVM08 benchmark suite [SPEC 2008] which we have found to be racy. We refer to Table 1 for various interesting statistics on each benchmark trace. The columns k and n denote the number of processes and number of events in each input trace. In each case, k is also used as the bound of the width of the partial orders constructed by our algorithm.

Comparison with HB, WCP, DC and SHB. We compare M2 against the standard HB, as well as WCP [Kini et al. 2017], DC [Roemer et al. 2018], and SHB [Mathur et al. 2018] which, to our knowledge, are the most recent advances in race prediction. These are partial-order methods based on vector clocks. All implementations are in Java: we rely on the tool Rapid [Mathur et al. 2018] for running HB, WCP and SHB, and on our own implementation of DC. To obtain all race reports for an input trace t, we use the following process. We construct the corresponding partial order incrementally, by inserting new events in the order they appear in t. In addition, we use an extra vector clock R_x, W_x, for every location x, which records the vector clock of the last read and write event, respectively, that accessed the respective location. These vector clocks are used to determine

Table 1. Statistics on our benchmark set.

Benchmark	n	k	# variables	# locks	Benchmark	n	k	# variables	# locks
array	44	2	30	2	moldyn	164K	2	1.0K	2
critical	49	3	30	0	derby	1.0M	3	185K	1.0K
airtickets	116	2	46	0	jigsaw	3.0M	13	103K	280
account	125	3	41	3	bufwriter	11M	5	56	1
pingpong	126	4	54	0	hsqldb	18M	43	946K	412
bbuffer	322	2	73	2	cryptorsa	57M	7	1.0M	8.0K
mergesort	3.0K	4	621	3	eclipse	86M	14	10M	8.0K
bubblesort	4.0K	10	196	3	xalan	122M	6	4.0M	2.0K
raytracer	16K	2	3.0K	8	lusearch	216M	7	5.0M	118
ftpserver	48K	10	5.0K	304	-	-	-	-	-

whether the current event is racy. After inserting an event e_1 in the partial order, we iterate over each conflicting event e_2 that precedes e_1 in t, and determine whether $e_2 < e_1$. If not, we report a race (e_1, e_2), and join the vector clock of e_1 with the vector clock of the corresponding location.

As HB and WCP are only sound on the first race, and DC is unsound, the above process creates, in general, false positives. In order to have a basis for comparison on sound reports, (e_1, e_2) is regarded as a reported race by each of these methods only if M2 reports it either as a race, or a possibly false negative (i.e., either $(e_1, e_2) \in \mathcal{Z}$ or $(e_1, e_2) \in C$, the sets $\mathcal{Z}$ and C as defined in Section 5) [2].

Race reports. Recall that a race is defined as a pair of events (e_1, e_2) of the input trace. However, the interest of the programmer is on the actual code lines (l_1, l_2) that these events correspond to. Since long traces typically come from code that executes repeatedly in a loop, we expect to have many racy pairs of events $\{(e_1^i, e_2^i)\}_i$ that correspond to the same racy pair of code lines (l_1, l_2), and hence all such pairs (e_1^i, e_2^i) require a single fix. Hence, although the input trace might contain many different event pairs that correspond to the same line pair, these will result in a single race report.

6.3 Experimental Results

Our evaluation is summarized in Table 2. The columns Races and Time show the number of reported races, and the time taken, respectively by each method. The column FN reports the size of the set C, which gives an upper-bound on the number of false negatives of M2 (see Section 5).

Race detection capability. We see that M2 is very effective: overall, it discovers hundreds of real races on all benchmarks, regardless of their size and number of processes. In addition, M2 is found complete on *all* benchmarks (i.e., our over-approximation of the false negatives in column FN always reports at most 0 false negatives). Hence, M2 manages to detect all races in our benchmark set. To our knowledge, this is the first sound technique that reaches such a level of completeness.

On the other hand, the capability of HB, WCP, DC and SHB is more limited, as they all miss several races on several benchmarks. We observe that WCP catches more races than HB, and DC more races than WCP. This is predicted by theory, as HB races are WCP races [Kini et al. 2017], and WCP races are DC races [Roemer et al. 2018]. On the other hand, although SHB captures provably more races than HB, SHB is incomparable with DC. In either case, M2 captures more races than DC on 10 benchmarks, and than SHB on 6 benchmarks. In addition, on 5 benchmarks (shown in

[2] SHB is sound on all race reports, and hence this filtering is not performed.

Table 2. Experimental comparison between HB, WCP, DC, SHB and our algorithm M2. The column FN shows an upper bound on the number of races missed by M2.

Benchmark	HB		WCP		DC		SHB		M2		
	Races	Time	Races	Time	Races	Time	Races	Time	Races	FN	Time
array	0	0.30s	0	0.28s	0	2.12s	0	0.29s	0	0	0.12s
critical	3	0.29s	3	0.30s	3	2.11s	8	0.28s	8	0	0.10s
airtickets	3	0.32s	3	0.31s	3	2.10s	4	0.31s	4	0	0.12s
account	1	0.31s	1	0.32s	1	2.12s	1	0.30s	1	0	0.12s
pingpong	2	0.30s	2	0.31s	2	2.04s	2	0.31s	2	0	0.09s
bbuffer	2	0.30s	2	0.31s	2	2.12s	2	0.31s	2	0	0.09s
mergesort	1	0.36s	1	0.41s	1	2.16s	1	0.37s	**2**	0	0.18s
bubblesort	4	0.46s	4	0.54s	5	2.28s	6	0.62s	6	0	0.71s
raytracer	3	0.51s	3	0.56s	3	2.57s	3	0.51s	3	0	0.23s
ftpserver	23	0.79s	23	1.28s	24	2.75s	23	0.73s	**26**	0	0.88s
moldyn	2	1.50s	2	1.81s	2	3.88s	2	1.52s	2	0	1.08s
derby	12	8.53s	12	14.54s	12	15.29s	12	8.32s	12	0	7.84s
jigsaw	8	17.51s	10	21.80s	10	40.89s	9	17.93s	**11**	0	14.65s
bufwriter	2	48.64s	2	2m0s	2	2m59s	2	47.71s	2	0	57.37s
hsqldb	4	3m53s	4	3m5s	5	4m23s	9	3m53s	9	0	7m1s
cryptorsa	5	3m42s	5	3m0s	7	6m58s	5	3m29s	7	0	6m6s
eclipse	33	8m1s	34	7m0s	39	14m44s	54	7m11s	**67**	0	45m23s
xalan	7	8m58s	7	8m25s	9	20m12s	11	9m8s	**15**	0	7m15s
lusearch	30	16m4s	30	9m59s	30	2h49m6s	52	15m28s	52	0	8m9s
Total	**145**	**42m0s**	**148**	**34m14s**	**160**	**3h39m**	**206**	**40m31s**	**231**	**0**	**1h15m**

Table 3. Mean and maximum distances (in number of intervening events) on races detected by M2.

Benchmark	Mean Distance	Max Distance
ftpserver	939	12K
jigsaw	1K	4K
hsqldb	92K	1M
cryptorsa	1M	8M
eclipse	11M	53M
xalan	2.0K	43K
lusearch	44M	125M

Table 4. Racy locations missed by each method. For HB, WCP, DC, SHB the numbers are lower-bounds. For M2, the numbers are upper-bounds.

Benchmark	HB	WCP	DC	SHB	M2
ftpserver	3	3	3	3	0
jigsaw	2	0	0	2	0
cryptorsa	1	1	0	1	0
eclipse	16	11	8	10	0
xalan	3	2	2	2	0
lusearch	11	11	11	0	0
Total	**36**	**28**	**24**	**18**	**0**

bold), M2 captures more races than any other algorithm. In total, M2 detects 71 more races than DC and 25 more races than SHB.

We also remark that our algorithm provides more information than the baseline methods even on benchmarks where the number of reported races is the same. This is because M2 manages to detect that the reports in such benchmarks are complete (i.e., no races are missed).

In terms of race distances, we have found that M2 is able to detect races that are very far apart in the input trace. We refer to Table 3 for a few interesting examples, where the distance of a race (e_1, e_2) is counted as the number of intervening events between e_1 and e_2 in the input trace. For instance,

in lusearch, the maximum race distance detected by M2 is 125M events. Note that, in general, the same memory location can be reported as racy by many data-race pairs (e_1, e_2). To assess the significance of the new races detected by M2, we have also computed the number of racy memory locations that are missed by each method. We see that M2 misses 0 memory locations (i.e., it detects all racy memory locations). For each of HB, WCP, DC and SHB, this number has been computed by counting how many locations have been detected by M2 and missed by the corresponding method. We refer to Table 4 for the cases where at least one method missed some racy memory location. In total, each of the baseline methods misses tens of racy memory locations. Finally, we have also computed location-specific race distances, as follows. For each location x, we computed the minimum distance d_x between all races on location x. Hence, d_x holds the smallest distance of a race which reveals that the location x is racy. The mean and maximum location-specific race distances in eclipse are 3M and 38M events, respectively, while in lusearch, they are 33M and 125M events, respectively. These numbers indicate that windowing techniques, which are typically restricted to windows of a few hundreds/thousands of events, are likely to produce highly incomplete results, and even fail to detect that certain memory locations are racy. Similar observations have also been made in recent works [Kini et al. 2017; Roemer et al. 2018].

Scalability. We see that M2 has comparable running time to the baseline methods, and is sometimes faster. One clear exception is on eclipse, where M2 requires about 45m, whereas the other methods spend between 7m and 14m. However, this is the benchmark on which all other methods miss both the most races (at least 13, see Table 2) and the most racy memory locations (at least 8, see Table 4). Hence, the completeness of our race reports comes at a relatively small increase in running time.

To better understand the efficiency of M2, recall that its worst-case complexity is a product of two factors, $O(\alpha \cdot \beta)$, where α is the number of calls to RaceDecision for verifying race pairs, and β is the time taken by Closure to compute the closure of the underlying partial order P. In the worst case, both α and β are $\Theta(n^2)$ (ignoring log-factors). In practice, we have observed that M2 resorts on calling RaceDecision only a small number of times, hence α is small. This illustrates the practical advantage of M2 over the naive approach that just uses RaceDecision on all $\binom{n}{2}$ event pairs. In addition, we can express β as roughly $\beta = n + m \cdot \gamma$, where m is the number of edges inserted in P during closure, and γ is the time spent for each such edge. Using our data structure DS for representing P, we have $\gamma = O(\log n)$, and, although $m = \Theta(n^2)$ in the worst-case, we have observed that m behaves as a constant in practice.

Finally, we note that the baseline methods may admit further engineering optimizations that reduce their running time. Such optimizations exist for HB, but we are unaware of any attempts to optimize WCP, DC or SHB further. In any case, although our tool is not faster, the take-home message is well-supported: M2 makes sound and effectively complete race predictions, at running times comparable to the theoretically fastest, yet highly incomplete, state-of-the-art methods.

7 RELATED WORK

In this section we briefly review related work on dynamic race detection.

Predictive analyses. Predictive techniques aim at inferring program behavior simply by looking at given traces. In the context of race detection, the CP partial order [Smaragdakis et al. 2012] and WCP partial order [Kini et al. 2017] are sound but incomplete predictive techniques based on partial orders. A somewhat different approach was proposed recently in [Roemer et al. 2018], based on the DC partial order. DC imposes fewer orderings than WCP, but is generally unsound. To create sound warnings, a DC-race is followed by a vindication phase, which is sound but incomplete.

Other works in this domain include [Huang et al. 2014; Liu et al. 2016; Said et al. 2011; Wang et al. 2009], which typically approach the problem based on SAT/SMT encodings. These works are sound and complete in theory, but require exponential time. In practice, techniques such as windowing make these methods operate fast, at the cost of sacrificing completeness. Predictive techniques have also been used for atomicity violations and synchronization errors [Chen et al. 2008; Huang and Rauchwerger 2015; Sen et al. 2005; Sorrentino et al. 2010], as well as in lock-based communication [Farzan et al. 2009; Kahlon et al. 2005; Sorrentino et al. 2010].

Happens-before techniques. A large pool of race detectors are based on Lamport's happens-before relation [Lamport 1978], which yields the HB partial order. HB can be computed in linear time [Mattern 1989] and has been the technical basis behind many approaches [Bond et al. 2010; Christiaens and Bosschere 2001; Flanagan and Freund 2009; Pozniansky and Schuster 2003; Schonberg 1989]. The tradeoff between runtime and space usage in race-detection using the happens-before relation was studied in [Banerjee et al. 2006]. Recently, the SHB partial order was proposed as an extension to HB in order to effectively detect multiple races per trace [Mathur et al. 2018].

Lockset-based techniques. A lockset of a variable is the set of locks that guard critical regions in which the variable is accessed. Lockset-based techniques report races by comparing the locksets of the variables accessed by the corresponding events. They were introduced in [Dinning and Schonberg 1991] and equipped by the tool of [Savage et al. 1997]. Lockset-based techniques tend to produce many false positives and this problem has been targeted by various enhancements such as random testing [Sen 2008] and static analysis [Choi et al. 2002; von Praun and Gross 2001].

Other approaches. To reduce unsound reports, lockset-based techniques have been combined with happens-before techniques [Elmas et al. 2007; O'Callahan and Choi 2003; Yu et al. 2005]. Other approaches include statistical techniques [Bond et al. 2010; Marino et al. 2009] and static race-detectors [Naik et al. 2006; Pratikakis et al. 2011; Voung et al. 2007]. Recently, [Genç et al. 2019] applied a combination of static and dynamic techniques to allow for correct reorderings in which the observation of some read events is allowed to differ between the input and witness trace, as long as the read does not affect the control-flow of the respective thread. Such static information can be directly incorporated in the techniques we have developed in this paper, and is left for interesting follow-up work. Dynamic race detection has also been studied under structured parallelism [Raman et al. 2012] and relaxed memory models [Kim et al. 2009; Lidbury and Donaldson 2017].

8 CONCLUSION

We have presented M2: a new polynomial-time algorithm for the problem that has no false positives. In addition, our algorithm is *complete* for input traces that consist of two processes, i.e., it provably detects *all* races in the trace. We have also developed criteria for detecting completeness dynamically, even in the case of more than two processes. Our experimental validation found that M2 is very effective in practice, as it soundly reported all races in the input benchmark set. Although M2 is not theoretically complete in the general case, we believe that its completeness guarantee on two processes provides some explanation of why it performs so well in practice.

A INCREMENTAL DAG REACHABILITY

In this section we target the problem of solving incremental reachability on Directed Acyclic Graphs (DAGs). Informally, we are given a DAG and an online sequence of (i) edge-insertion and (ii) reachability query operations. The task is to answer each reachability query correctly, accounting for all preceding edge-insertion operations. Here we develop a data structure DS for solving the problem efficiently on DAGs of small width. In the main paper we use DS to compute

the closure of partial orders efficiently. We expect that DS might be of relevance also to other race-detection techniques that are graph-based.

Directed acyclic graphs of small width. Let $G = (V, E)$ be a DAG and E^* be the transitive closure of E. Note that E^* is a partial order, and we let $\text{width}(G) = \text{width}(E^*)$. Our focus is on DAGs of small width, i.e., we take $\text{width}(G) = k = O(1)$. For $u, v \in V$, we write $u \rightsquigarrow v$ if v is reachable from u. We represent G as k (totally ordered) chains with extra edges between them. We let $V \subseteq [k] \times [n]$, so that a node of G is represented as a pair (i, j), meaning that it is the j-th node in the i-th chain. For two nodes $\langle i, j_1\rangle, \langle i, j_2\rangle \in V$ with $j_2 = j_1 + 1$, we have $((i, j_1), (i, j_2)) \in E$. Given two nodes $\langle i, j_1\rangle, \langle i, j_2\rangle$, we say that $\langle i, j_1\rangle$ is *higher than* $\langle i, j_2\rangle$ if $j_1 \leq j_2$. In such a case, we say that $\langle i, j_2\rangle$ is *lower than* $\langle i, j_1\rangle$. The edge set is represented as a set of arrays $\text{Out}_{i_1}^{i_2} \to [n] \cup \{\infty\}$, where $i_1, i_2 \in [k]$ and $i_1 \neq i_2$. We have that $\text{Out}_{i_1}^{i_2}[j_1] = j_2 \in [k]$ iff $(\langle i_1, j_1\rangle, \langle i_2, j_2\rangle) \in E$. Note that since $k = O(1)$, such a representation requires $O(n)$ space even if G is a dense graph.

Incremental reachability on DAGs of small width. The *incremental reachability* problem on a DAG $G = (V, E)$ is defined on an online sequence of operations of the following types.

(1) An $\text{insert}(u, v)$ operation, such that $v \not\rightsquigarrow u$, inserts the edge u, v in G.
(2) A $\text{query}(u, v)$ operation returns True iff $u \rightsquigarrow v$.
(3) A $\text{successor}(u, i)$ operation returns the highest successor of u in the i-th chain.
(4) A $\text{predecessor}(u, i)$ operation returns the lowest predecessor of u in the i-th chain.

The task is to answer query operations correctly, taking into consideration all preceding insert operations. Note that the width of G does not increase after any operation. We will present a data structure that handles each such query in $O(\log n)$ time. Our data structure is based on the dynamic suffix minima problem, presented below.

Dynamic suffix minima and Fenwick-trees. The *dynamic suffix minima* problem is defined given an integer array A of length n, and an online sequence of operations of the following types.

(1) An $\text{update}(i, x)$ operation, for $1 \leq i \leq n$ and $x \in \mathbb{Z}$, sets $A[i] = x$.
(2) A $\min(i)$ operation, for $1 \leq i \leq n$, returns $\min_{i \leq j \leq n} A[j]$.
(3) An $\arg\text{leq}(i)$ operation, for $1 \leq i \leq n$, returns $\max_{j:\, A[j] \leq i} j$.

The task is to answer min and arg leq operations correctly, taking into consideration all preceding update operations. The Fenwick-tree data structure solves the dynamic suffix minima problem in $O(\log n)$ time per operation, after $O(n)$ preprocessing time [Fenwick 1994].

The data structure DS **for solving the incremental reachability problem.** We are now ready to describe our data structure DS for solving the incremental reachability problem given a DAG $G = (V, E)$ of width k, and an online sequence Σ of insert and query operations. We consider that G is given in a sparse representation form, where outgoing edges are represented using the arrays $\text{Out}_{i_1}^{i_2}$, for each $i_1, i_2 \in [k]$ and $i_1 \neq i_2$.

In the initialization phase, DS performs the following steps. For each $i_1, i_2 \in [k]$ such that $i_1 \neq i_2$, we initialize a Fenwick-tree data structure $\text{FenwickTree}_{i_1}^{i_2}$ with array $\text{Out}_{i_1}^{i_2}$. This data structure stores forward reachability information from nodes of the i_1-th chain to nodes in the i_2 chain, by maintaining the invariant that $\text{FenwickTree}_{i_1}^{i_2}.\min(j_1) = j_2$ iff $\langle i_2, j_2\rangle$ is the highest node of the i_2-th chain reachable from $\langle i_1, j_2\rangle$. A $\text{successor}(\langle i_1, j_1\rangle, i)$ (resp., $\text{predecessor}(\langle i_1, j_1\rangle, i)$) operation is handled by DS by returning $\text{FenwickTree}_{i_1}^{i}.\min(j_1)$ (resp., $\text{FenwickTree}_{i_1}^{i}.\arg\text{leq}(j_1)$). Finally, the

operations DS.insert and DS.query are handled by Algorithm 7 and Algorithm 8, respectively. The following lemma establishes the correctness and complexity of the data structure DS.

Lemma A.1. *Let Σ be an online sequence of incremental reachability operations. The data structure* DS *correctly handles Σ and spends (i) $O(n)$ preprocessing time and (ii) $O(\log n)$ time per operation.*

Algorithm 7: DS.insert($\langle i_1, j_1\rangle, \langle i_2, j_2\rangle$)

```
1 foreach i'_1 ∈ [k] do
2 |  foreach i'_2 ∈ [k] do
3 |  |  Let j'_1 ← predecessor(⟨i_1, j_1⟩, i'_1)
4 |  |  Let j'_2 ← successor(⟨i_2, j_2⟩, i'_2)
5 |  |  FenwickTree_{i'_1}^{i'_2}.update(j'_1, j'_2)
6 |  end
7 end
```

Algorithm 8: DS.query($\langle i_1, j_1\rangle, \langle i_2, j_2\rangle$)

```
1 Let j'_2 ← DS.successor(⟨i_1, j_1⟩, i_2)
2 if j'_2 ≤ j_2 then
3 |  return True
4 else
5 |  return False
6 end
```

B DEFINITION OF HB, SHB, WCP AND DC

For the sake of completeness, here we give the definitions of the partial orders HB, SHB, WCP and DC, based on [Kini et al. 2017; Mathur et al. 2018; Roemer et al. 2018]. In each case, we consider given a trace t, and the respective partial order is over the set of events $\mathcal{E}(t)$ of t.

The HB **partial order** is the smallest partial order that satisfies the following conditions .

(1) $\mathsf{HB} \sqsubseteq \mathsf{PO}$.
(2) For every lock-acquire and lock-release events acq and rel such that $\mathsf{rel} <_t \mathsf{acq}$, if $\mathsf{acq} \bowtie \mathsf{rel}$ then $\mathsf{rel} <_{\mathsf{HB}} \mathsf{acq}$.

The SHB **partial order** is the smallest partial order that satisfies the following conditions.

(1) $\mathsf{SHB} \sqsubseteq \mathsf{HB}$.
(2) For every read event r we have $O_t(r) <_{\mathsf{SHB}} r$. Recall that $O_t(r)$ is the observation of r in t.

The WCP **partial order** is the smallest partial order that satisfies the following conditions.

(1) $\mathsf{WCP} \sqsubseteq \mathsf{PO}$.
(2) For every lock-release event rel and write/read event e such that $\mathsf{rel} <_t e$, if (i) e is protected by a lock $\ell = \mathsf{loc}(\mathsf{rel})$ and (ii) e conflicts with an event in the critical section of rel, then $\mathsf{rel} <_{\mathsf{WCP}} e$.
(3) For every two lock-release events $\mathsf{rel}_1, \mathsf{rel}_2$ such that $\mathsf{rel}_1 <_t \mathsf{rel}_2$, if the critical sections of rel_1 and rel_2 contain WCP-ordered events, then $\mathsf{rel}_1 <_{\mathsf{WCP}} \mathsf{rel}_2$.
(4) WCP is closed under left and right composition with HB.

The DC **partial order** is the smallest partial order that satisfies conditions 1, 2 and 3 of WCP.

ACKNOWLEDGMENTS

I am grateful to Umang Mathur for his valuable assistance in the experimental part of the paper, to Viktor Kunčak for his insightful comments in earlier drafts, and to anonymous reviewers for their constructive feedback. This work is partly supported by the Austrian Science Fund (FWF) Schrödinger grant J-4220.

REFERENCES

Utpal Banerjee, Brian Bliss, Zhiqiang Ma, and Paul Petersen. 2006. A Theory of Data Race Detection. In *Proceedings of the 2006 Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD '06)*. ACM, New York, NY, USA, 69–78. https://doi.org/10.1145/1147403.1147416

Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA*.

Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. 2010. PACER: Proportional Detection of Data Races. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. ACM, New York, NY, USA, 255–268. https://doi.org/10.1145/1806596.1806626

Feng Chen and Grigore Roşu. 2007. Parametric and Sliced Causality. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07)*. Springer-Verlag, Berlin, Heidelberg, 240–253. http://dl.acm.org/citation.cfm?id=1770351.1770387

Feng Chen, Traian Florin Serbanuta, and Grigore Rosu. 2008. jPredictor: A Predictive Runtime Analysis Tool for Java. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. ACM, New York, NY, USA, 221–230. https://doi.org/10.1145/1368088.1368119

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 *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI '02)*. ACM, New York, NY, USA, 258–269. https://doi.org/10.1145/512529.512560

Mark Christiaens and Koenraad De Bosschere. 2001. TRaDe: Data Race Detection for Java. In *Proceedings of the International Conference on Computational Science-Part II (ICCS '01)*. Springer-Verlag, London, UK, UK, 761–770. http://dl.acm.org/citation.cfm?id=645456.654536

Anne Dinning and Edith Schonberg. 1991. Detecting Access Anomalies in Programs with Critical Sections. In *Proceedings of the 1991 ACM/ONR Workshop on Parallel and Distributed Debugging (PADD '91)*. ACM, New York, NY, USA, 85–96. https://doi.org/10.1145/122759.122767

Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. 2007. Goldilocks: A Race and Transaction-aware Java Runtime. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 245–255. https://doi.org/10.1145/1250734.1250762

Eitan Farchi, Yarden Nir, and Shmuel Ur. 2003. Concurrent Bug Patterns and How to Test Them. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing (IPDPS '03)*. IEEE Computer Society, Washington, DC, USA, 286.2–. http://dl.acm.org/citation.cfm?id=838237.838485

Azadeh Farzan, P. Madhusudan, and Francesco Sorrentino. 2009. Meta-analysis for Atomicity Violations Under Nested Locking. In *Proceedings of the 21st International Conference on Computer Aided Verification (CAV '09)*. Springer-Verlag, Berlin, Heidelberg, 248–262. https://doi.org/10.1007/978-3-642-02658-4_21

Peter M. Fenwick. 1994. A new data structure for cumulative frequency tables. *Software: Practice and Experience* 24, 3 (1994), 327–336. https://doi.org/10.1002/spe.4380240306

Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 121–133. https://doi.org/10.1145/1542476.1542490

Kaan Genç, Jake Roemer, Yufan Xu, and Michael D. Bond. 2019. Dependence-Aware, Unbounded Sound Predictive Race Detection *(OOPSLA 2019)*. To appear.

Jim Gray. 1985. Why Do Computers Stop And What Can Be Done About It? *Büroautomation* (1985), 128–145.

D. P. Helmbold, C. E. McDowell, and Jian-Zhong Wang. 1991. Detecting data races from sequential traces. In *Proceedings of the Twenty-Fourth Annual Hawaii International Conference on System Sciences*, Vol. ii. 408–417 vol.2. https://doi.org/10.1109/HICSS.1991.184003

Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. 2014. Maximal Sound Predictive Race Detection with Control Flow Abstraction. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 337–348. https://doi.org/10.1145/2594291.2594315

Jeff Huang and Lawrence Rauchwerger. 2015. Finding Schedule-sensitive Branches. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 439–449. https://doi.org/10.1145/2786805.2786840

Vineet Kahlon, Franjo Ivančić, and Aarti Gupta. 2005. Reasoning About Threads Communicating via Locks. In *Proceedings of the 17th International Conference on Computer Aided Verification (CAV'05)*. Springer-Verlag, Berlin, Heidelberg, 505–518. https://doi.org/10.1007/11513988_49

KyungHee Kim, Tuba Yavuz-Kahveci, and Beverly A. Sanders. 2009. Precise Data Race Detection in a Relaxed Memory Model Using Heuristic-Based Model Checking. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE '09)*. IEEE Computer Society, Washington, DC, USA, 495–499. https://doi.org/10.1109/ASE.2009.82

Dileep Kini, Umang Mathur, and Mahesh Viswanathan. 2017. Dynamic Race Prediction in Linear Time. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 157–170. https://doi.org/10.1145/3062341.3062374

Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978), 558–565. https://doi.org/10.1145/359545.359563

Christopher Lidbury and Alastair F. Donaldson. 2017. Dynamic Race Detection for C++11. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 443–457. https://doi.org/10.1145/3009837.3009857

Peng Liu, Omer Tripp, and Xiangyu Zhang. 2016. IPA: Improving Predictive Analysis with Pointer Analysis. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. ACM, New York, NY, USA, 59–69. https://doi.org/10.1145/2931037.2931046

Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. *SIGOPS Oper. Syst. Rev.* 42, 2 (March 2008), 329–339. https://doi.org/10.1145/1353535.1346323

Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. 2009. LiteRace: Effective Sampling for Lightweight Data-race Detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 134–143. https://doi.org/10.1145/1542476.1542491

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

Friedemann Mattern. 1989. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel & Distributed Algorithms*, M. Cosnard et. al. (Ed.). Elsevier Science Publishers B. V., 215–226.

A Mazurkiewicz. 1987. Trace Theory. In *Advances in Petri Nets 1986, Part II on Petri Nets: Applications and Relationships to Other Models of Concurrency*. Springer-Verlag New York, Inc., 279–324.

Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, USA, 267–280. http://dl.acm.org/citation.cfm?id=1855741.1855760

Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective Static Race Detection for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. ACM, New York, NY, USA, 308–319. https://doi.org/10.1145/1133981.1134018

Robert O'Callahan and Jong-Deok Choi. 2003. Hybrid Dynamic Data Race Detection. *SIGPLAN Not.* 38, 10 (June 2003), 167–178. https://doi.org/10.1145/966049.781528

Andreas Pavlogiannis. 2019. Fast, Sound and Effectively Complete Dynamic Race Detection. (2019). arXiv:arXiv:1901.08857

Eli Pozniansky and Assaf Schuster. 2003. Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs. *SIGPLAN Not.* 38, 10 (June 2003), 179–190. https://doi.org/10.1145/966049.781529

Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. 2011. LOCKSMITH: Practical Static Race Detection for C. *ACM Trans. Program. Lang. Syst.* 33, 1, Article 3 (Jan. 2011), 55 pages. https://doi.org/10.1145/1889997.1890000

Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. 2012. Scalable and Precise Dynamic Datarace Detection for Structured Parallelism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 531–542. https://doi.org/10.1145/2254064.2254127

Jake Roemer, Kaan Genç, and Michael D. Bond. 2018. High-coverage, Unbounded Sound Predictive Race Detection. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 374–389. https://doi.org/10.1145/3192366.3192385

Mahmoud Said, Chao Wang, Zijiang Yang, and Karem Sakallah. 2011. Generating Data Race Witnesses by an SMT-based Analysis. In *Proceedings of the Third International Conference on NASA Formal Methods (NFM'11)*. Springer-Verlag, Berlin, Heidelberg, 313–327. http://dl.acm.org/citation.cfm?id=1986308.1986334

Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Trans. Comput. Syst.* 15, 4 (Nov. 1997), 391–411. https://doi.org/10.1145/265924.265927

D. Schonberg. 1989. On-the-fly Detection of Access Anomalies. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation (PLDI '89)*. ACM, New York, NY, USA, 285–297. https://doi.org/10.

1145/73141.74844

Koushik Sen. 2008. Race Directed Random Testing of Concurrent Programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, New York, NY, USA, 11–21. https://doi.org/10.1145/1375581.1375584

Koushik Sen, Grigore Roşu, and Gul Agha. 2005. Detecting Errors in Multithreaded Programs by Generalized Predictive Analysis of Executions. In *Formal Methods for Open Object-Based Distributed Systems*, Martin Steffen and Gianluigi Zavattaro (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 211–226.

Yao Shi, Soyeon Park, Zuoning Yin, Shan Lu, Yuanyuan Zhou, Wenguang Chen, and Weimin Zheng. 2010. Do I Use the Wrong Definition?: DeFuse: Definition-use Invariants for Detecting Concurrency and Sequential Bugs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*. ACM, New York, NY, USA, 160–174. https://doi.org/10.1145/1869459.1869474

Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. 2012. Sound Predictive Race Detection in Polynomial Time. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*. ACM, New York, NY, USA, 387–400. https://doi.org/10.1145/2103656.2103702

L. A. Smith, J. M. Bull, and J. Obdrzálek. 2001. A Parallel Java Grande Benchmark Suite. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing (SC '01)*. ACM, New York, NY, USA, 8–8. https://doi.org/10.1145/582034.582042

Francesco Sorrentino, Azadeh Farzan, and P. Madhusudan. 2010. PENELOPE: Weaving Threads to Expose Atomicity Violations. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '10)*. ACM, New York, NY, USA, 37–46. https://doi.org/10.1145/1882291.1882300

SPEC. 2008. SPEC releases free SPECjvm2008 benchmark. (2008). https://www.spec.org/jvm2008/press/release.html

Christoph von Praun and Thomas R. Gross. 2001. Object Race Detection. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '01)*. ACM, New York, NY, USA, 70–82. https://doi.org/10.1145/504282.504288

Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. 2007. RELAY: Static Race Detection on Millions of Lines of Code. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC-FSE '07)*. ACM, New York, NY, USA, 205–214. https://doi.org/10.1145/1287624.1287654

Chao Wang, Sudipta Kundu, Malay Ganai, and Aarti Gupta. 2009. Symbolic Predictive Analysis for Concurrent Programs. In *Proceedings of the 2Nd World Congress on Formal Methods (FM '09)*. Springer-Verlag, Berlin, Heidelberg, 256–272. https://doi.org/10.1007/978-3-642-05089-3_17

Misun Yu, Joon-Sang Lee, and Doo-Hwan Bae. 2018. AdaptiveLock: Efficient Hybrid Data Race Detection Based on Real-World Locking Patterns. *International Journal of Parallel Programming* (04 Jun 2018). https://doi.org/10.1007/s10766-018-0579-5

Yuan Yu, Tom Rodeheffer, and Wei Chen. 2005. RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. *SIGOPS Oper. Syst. Rev.* 39, 5 (Oct. 2005), 221–234. https://doi.org/10.1145/1095809.1095832

Ke Zhai, Boni Xu, W. K. Chan, and T. H. Tse. 2012. CARISMA: A Context-sensitive Approach to Race-condition Sample-instance Selection for Multithreaded Applications. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA 2012)*. ACM, New York, NY, USA, 221–231. https://doi.org/10.1145/2338965.2336780