Optimal Prediction of Synchronization-Reversal Free Races

ANONYMOUS AUTHOR(S)

Concurrent programs are notoriously hard to write correctly, as scheduling nondeterminism introduces subtle errors that are both hard to detect and reproduce. The most common concurrency errors are *(data) races*, which occur when memory-conflicting actions are executed concurrently. Consequently, considerable effort has been made towards identifying efficient techniques for race detection. The most common approach is *dynamic race prediction*: given an observed, race-free trace σ of a concurrent program, the task is to decide whether events of σ can be correctly reordered to a trace σ^* that witnesses a race hidden in σ.

In this work we introduce the notion of *sync(hronization)-reversal free races.* A sync-reversal free race occurs in σ when there is a witness σ^* in which synchronization operations (e.g., acquisition and release of locks) appear in the same order as in σ. This is a broad definition that *strictly subsumes* the famous notion of happens-before races. Our main results are as follows. First, we develop a *sound and complete* algorithm for predicting sync-reversal free races that runs in $\widetilde{O}(\mathcal{N})$ time and space, for up to a moderate number of other parameters, such as the number of threads. Second, we show that the problem has a $\Omega(\mathcal{N}/\log^2 \mathcal{N})$ space lower bound, and thus our algorithm is essentially *time and space optimal.* Third, we show that predicting races with *even just a single* reversal of two sync operations is NP-complete and even $\mathrm{W}[1]$-hard when parameterized by the number of threads. Thus, sync-reversal freeness characterizes *exactly* the tractability boundary of race prediction, and our algorithm is nearly *optimal* for the tractable side. Our experiments show that our algorithm is fast in practice, while sync-reversal freeness characterizes races often missed by state-of-the-art methods.

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, dynamic analysis, race detection, complexity

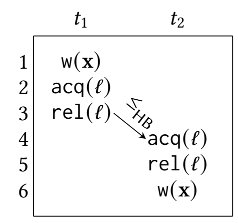
1 INTRODUCTION

The verification of concurrent programs is one of the main challenges in formal methods. Concurrency adds a dimension of non-determinism to program behavior which stems from inter-process communication. Accounting for such non-determinism during program development is a challenging mental task, making concurrent programming significantly error-prone. At the same time, bugs due to concurrency are very hard to reproduce manually, and automated techniques for doing so are crucial in enhancing productivity of software developers.

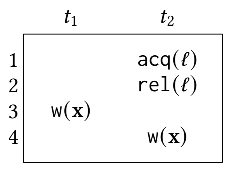
Data races are the most common form of concurrency errors. A data race (sometimes just called race) occurs when a thread of a multi-threaded program accesses a shared memory location while another thread is modifying it without proper synchronization. The presence of a data race is often symptomatic of a serious bug in the program [Lu et al. 2008]; they have lead to data corruption and compilation errors [Boehm 2011; Kasikci et al. 2013; Narayanasamy et al. 2007], and have lead to significant system errors in the past [Boehm 2012; Zhivich and Cunningham 2009]. Therefore, considerable research has focused on detecting and preventing races in multi-threaded programs.

One of the most popular approaches to race prediction is via dynamic analysis [Bond et al. 2010; Flanagan and Freund 2009; Pozniansky and Schuster 2003]. Unlike static analysis, dynamic race prediction is performed at runtime. Such techniques determine if an observed execution provides evidence for the existence of a *possibly alternate* program execution that can concurrently perform

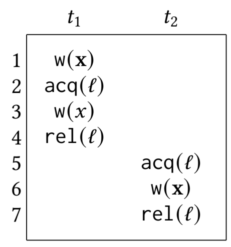
2021. 2475-1421/2021/1-ART1 $15.00
https://doi.org/



(a) An observed trace σ_1.



(b) A witness σ_1^* of a race in σ_1.



(c) An observed trace σ_2.

Fig. 1. (1a) shows a trace σ_1 with sync-reversal free race (e_1, e_6) missed by $\leq_{\text{HB}}$. (1b) shows the witness σ_1^* that exposes the race. (1c) shows a sync-reversal-free race (e_1, e_6) that is non-consecutive, due to the intermediate event e_3.

conflicting data accesses [1]. The underlying principle is that a race is present but "hidden" in a large number of different program executions; hence techniques that uncover such hidden races can accelerate the process of debugging concurrent programs. The popularity of dynamic race prediction techniques further stems (i) from their scalability to large production software, and (ii) from their ability to produce only sound error reports.

The most popular dynamic race prediction techniques are based on Lamport's happens-before partial order [Lamport 1978]. These techniques scan the input trace, determine happens-before orderings on-the-fly, and report a race on a pair of conflicting data accesses if they are *unordered* by happens-before. This approach is sound, in that the presence of unordered conflicting data accesses ensures the existence of an execution with a race. While happens-before based analysis fails to predict races in various cases [Smaragdakis et al. 2012], its wide deployment is based on the fact that the algorithm is fast, single pass, and runs in linear time. The principle that forms the basis of its efficiency is the following. When reasoning about alternate executions, happens-before analysis does not consider any execution in which the order of synchronization primitives is reversed from that in the observed execution. We call such alternate executions *sync(hronization)-reversal free executions* [2] in this paper. Other, more powerful race prediction techniques [Genç et al. 2019; Huang et al. 2014; Huang and Rajagopalan 2016; Pavlogiannis 2019; Roemer et al. 2018; Smaragdakis et al. 2012] sacrifice this principle and consider alternate executions that are not sync-reversal free free. Naturally, this typically results in performance degradation, as the problem is in general NP-hard [Mathur et al. 2020], and considerable efforts are made towards improving the scalability of such techniques [Roemer and Bond 2019; Roemer et al. 2020].

Although happens-before only detects races whose exposure preserves the ordering of synchronization primitives, it can still miss simple races that adhere to this pattern. For example, consider the trace σ_1 shown in Figure 1a. Let us name the events of this trace based on the order in which they appear in the trace; thus, e_i denotes the i^{th} event of the trace. Here, the partial order happens-before orders the first $\mathtt{w}(x)$ (event e_1) and the last $\mathtt{w}(x)$ (event e_6), and therefore, does not detect any race in this execution. However events e_1 and e_6 are in race. This can be exposed by the alternate execution shown in Figure 1b, which is obtained by dropping the critical section of lock ℓ performed by thread t_1. Notice that the order of synchronization events (namely, $\mathtt{acq}(\ell)$ and $\mathtt{rel}(\ell)$ events) *that appear in the trace* of Figure 1b, are in the same order as in the trace of Figure 1a, and hence this is

[1]Conflicting data accesses are data accesses by two threads to a common memory location such that at least one of them is a write.
[2]Precise definition of such executions given in Section 2.1.

a sync-reversal free execution. Thus, the notion of sync-reversal freeness captures races beyond standard happens-before races.

Another important limitation of happens-before and virtually all partial-order methods [Kini et al. 2017; Mathur et al. 2018; Roemer et al. 2018, 2020; Smaragdakis et al. 2012] is highlighted in Figure 1c. The trace σ_2 has a race between e_1 and e_6, both conflicting on variable x. Notice, however, that the intermediate event e_3 also accesses x, but is not in race with either e_1 or e_6. Partial-order methods for race prediction are limited to capturing races *only between successive* conflicting accesses[3]. Hence, distant races that are interjected with intermediate conflicting but non-racy events, are missed by such methods. On the other hand, sync-reversal freeness is not bound to such limitations: (e_1, e_6) is characterized as a race under this criterion, regardless of the intermediate, non-racy e_3, exposed by a witness that omits the critical section on lock ℓ in the thread t_1.

Our Contributions. Motivated by observations like the above, our contributions are as follows.

(1) We introduce the novel notion of *sync(hronization)-reversal free* races. This is a *sound* notion of predictable races, and it *strictly subsumes* the standard notion of happens-before races. Moreover, it characterizes races between events that can be arbitrarily far apart in the input trace, as opposed to happens-before and other partial-order methods that only characterize races between *successive* conflicting accesses. Our notion is applicable to all concurrency settings, and interestingly, it is also *complete* for systems with synchronization-deterministic concurrency [Aguado et al. 2018; Bocchino et al. 2009; Cui et al. 2015; Zhao et al. 2019].
(2) We develop an efficient, single-pass, nearly linear-time algorithm SRFree that, given a trace σ, detects whether σ contains a sync-reversal free free race. In fact, our algorithm soundly reports *all* events e_2 which are in a sync-reversal free free race with an event e_1 that appears earlier in σ. Given $\mathcal{N}$ events in σ, our algorithm spends $\widetilde{O}(N)$ time, where $\widetilde{O}$ hides factors poly-logarithmic in $\mathcal{N}$, when other parameters of the input (e.g., number of threads) are $\widetilde{O}(1)$.
(3) Although our algorithm performs a single pass of the trace, in the worst case, it might use space that is nearly linear in the length of the trace, i.e., $\widetilde{O}(\mathcal{N})$ space. Hence follows a natural question: is there an efficient algorithm for sync-reversal free race prediction that uses considerably less space? We answer this question in negative, by showing that *any* single-pass algorithm for detecting even a single sync-reversal free race must use nearly linear space. Hence, our algorithm SRFree has nearly optimal performance in both time and space.
(4) We next study the complexity of race prediction with respect to the number of synchronization reversals that might occur when constructing a witness that exposes the race. In the case of synchronization via locks, this number corresponds to the number of critical sections whose order is reversed in the witness trace. We prove that the problem of predicting races which can be witnessed by *a single* reversal (of two critical sections) is NP-complete and even W[1]-hard when parameterized by the number of threads. Thus, sync-reversal freeness characterizes *exactly* the tractability boundary of race prediction, and our algorithm is nearly *optimal* for the tractable side. Moreover, our result shows that *any level* of synchronization suffices to make the problem of race prediction as hard as in the general case, improving a recent result of [Mathur et al. 2020].
(5) Finally, we have implemented our race prediction algorithm SRFree and evaluated its performance on standard benchmarks from the literature. Our results show that sync-reversal freeness characterizes many races that are missed by state-of-the-art methods, and SRFree successfully detects them.

[3]When the earlier access is a read instead of a write, this statement is true *per thread.*

2 PRELIMINARIES

In this section, we establish notations useful for the rest of the paper.

Traces and Events. Our objective is to develop a dynamic analysis technique which works over execution traces, or simply *traces* of concurrent programs. We work with the sequential consistency memory model. Under this assumption, traces are sequences of events. We will use $\sigma, \sigma', \ldots, \sigma_1, \sigma_2, \ldots$ to denote traces in this presentation. Every event of σ can be represented as a tuple $e = \langle i, t, \mathsf{op}\rangle$, where i is a unique identifier of e in σ, t is the thread that performs e and op is the operation performed in the event e. We often omit the unique identifier of such a tuple and simply write $e = \langle t, \mathsf{op}\rangle$. We use $\mathsf{thr}(e)$ and $\mathsf{op}(e)$ to denote the thread performing e and the operation performed by e. An operation can be one of read from or write to a shared memory location or *variable* x, denoted $\mathtt{r}(x)$ and $\mathtt{w}(x)$, and acquisition or release of a lock ℓ, denoted $\mathtt{acq}(\ell)$ or $\mathtt{rel}(\ell)$. Forks and joins can be naturally handled, but we avoid introducing them here for notational convenience. We denote by Events_σ the set of events in a trace σ. We use Thr_σ, Vars_σ and Locks_σ to denote respectively the threads, variables and locks that appear in the trace σ. Likewise, we use $\mathsf{Acquires}_\sigma(\ell)$ and $\mathsf{Releases}_\sigma(\ell)$ to denote the set of acquire and release events of σ on lock $\ell \in \mathsf{Locks}_\sigma$.

We assume that traces obey lock semantics. This means, every lock ℓ is released by a thread t only if there is an earlier matching acquire event by the same thread t, and that each such lock is held by at most one thread at a time. Formally, let $\sigma|_\ell$ denote the projection of σ to the set of events $\mathsf{Acquires}_\sigma(\ell) \cup \mathsf{Releases}_\sigma(\ell)$. We require that for every lock ℓ, the sequence $\sigma|_\ell$ is a prefix of some sequence that belongs to the regular language corresponding to the regular expression $\Big(\sum\limits_{t \in \mathsf{Thr}_\sigma} \langle t, \mathtt{acq}(\ell)\rangle \cdot \langle t, \mathtt{rel}(\ell)\rangle\Big)^*$.

For an acquire event e, we use $\mathsf{match}_\sigma(e)$ to denote the matching release event of e if one exists (and $\bot$ otherwise). Similarly, for a release event e, $\mathsf{match}_\sigma(e)$ is the matching acquire of e on the same lock. For an acquire event e, the critical section protected by e, denoted $\mathsf{CS}_\sigma(e)$ is the set of events e' such that $\mathsf{thr}(e') = \mathsf{thr}(e)$ and e' occurs after e (and before the matching release $\mathsf{match}_\sigma(e)$, if it exists) in σ. For a release event e, we have $\mathsf{CS}_\sigma(e) = \mathsf{CS}_\sigma(\mathsf{match}_\sigma(e))$.

Orders on Traces. A partial order $\leq^\sigma_{\mathsf{P}}$ defined over a trace σ is a reflexive, anti-symmetric and transitive binary relation on Events_σ; the symbol P is an optional identifier for the partial order. We will often write $e_1 \leq^\sigma_{\mathsf{P}} e_2$ to denote $(e_1, e_2) \in \leq^\sigma_{\mathsf{P}}$, where $e_1, e_2 \in \mathsf{Events}_\sigma$. For a partial order $\leq^\sigma_{\mathsf{P}}$, we use $<^\sigma_{\mathsf{P}}$ to denote the strict order $\leq^\sigma_{\mathsf{P}} \setminus \{(e, e) \mid e \in \mathsf{Events}_\sigma\}$. We write $e_1 \not\leq^\sigma_{\mathsf{P}} e_2$ to denote that $(e_1, e_2) \notin \leq^\sigma_{\mathsf{P}}$. Events $e_1, e_2 \in \mathsf{Events}_\sigma$ are said to be *unordered* by $\leq^\sigma_{\mathsf{P}}$, denoted $e_1 \parallel^\sigma_P e_2$ if $e_1 \not\leq^\sigma_{\mathsf{P}} e_2$ and $e_2 \not\leq^\sigma_{\mathsf{P}} e_1$; otherwise, we write $e_1 \nparallel^\sigma_P e_2$, denoting that e_1 and e_2 are ordered by $\leq^\sigma_{\mathsf{P}}$ in one or the other way. When σ is clear from context, we will use $\leq_{\mathsf{P}}$, $<_{\mathsf{P}}$, $\not\leq_{\mathsf{P}}$, $\parallel_P$ and $\nparallel_P$ instead of respectively $\leq^\sigma_{\mathsf{P}}$, $<^\sigma_{\mathsf{P}}$, $\not\leq^\sigma_{\mathsf{P}}$, $\parallel^\sigma_P$ and $\nparallel^\sigma_P$. For a partial order $\leq^\sigma_{\mathsf{P}}$, a set $S \subseteq \mathsf{Events}_\sigma$ is said to be *downward-closed with respect to* $\leq^\sigma_{\mathsf{P}}$ if for every $e, e' \in \mathsf{Events}_\sigma$, if $e \leq^\sigma_{\mathsf{P}} e'$ and $e' \in S$, then $e \in S$.

The *trace-order* $\leq^\sigma_{\mathsf{tr}}$ defined by σ is the total order on Events_σ imposed by the sequence σ, i.e., $e_1 \leq^\sigma_{\mathsf{tr}} e_2$ iff the event e_1 occurs before e_2 in σ. The *thread-order* (or *program-order*) $\leq^\sigma_{\mathsf{TO}}$ of σ is the partial order on Events_σ that orders events in the same thread: for two events $e_1, e_2 \in \mathsf{Events}_\sigma$, $e_1 \leq^\sigma_{\mathsf{TO}} e_2$ iff $e_1 \leq^\sigma_{\mathsf{tr}} e_2$ and $\mathsf{thr}(e_1) = \mathsf{thr}(e_2)$.

Conflicting Events and Data Races. Let σ be a trace. Two events $e_1, e_2 \in \mathsf{Events}_\sigma$ are said to be *conflicting*, denoted $e_1 \asymp e_2$, if $\mathsf{thr}(e_1) \neq \mathsf{thr}(e_2)$, and there is a common variable $x \in \mathsf{Vars}_\sigma$ such that $\mathsf{op}(e_1), \mathsf{op}(e_2) \in \{\mathtt{r}(x), \mathtt{w}(x)\}$ and at least one of $\mathsf{op}(e_1)$ and $\mathsf{op}(e_2)$ is $\mathtt{w}(x)$. Let ρ be a trace with $\mathsf{Events}_\rho \subseteq \mathsf{Events}_\sigma$. An event $e \in \mathsf{Events}_\sigma$ is said to be *σ-enabled* in ρ if $e \notin \mathsf{Events}_\rho$ and for all

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

(a) Trace σ_3 with data race

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

(b) Trace σ_4 with predictable race

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

(c) Trace σ_5 with no predictable race

Fig. 2. Traces, data races and predictable data races

events $e' \in \mathsf{Events}_\sigma$ such that $e' <^{\sigma}_{\mathsf{TO}} e$, we have $e' \in \mathsf{Events}_\rho$. A pair of conflicting events (e_1, e_2) in σ is said to be a data race of σ if σ has a prefix σ' such that both e_1 and e_2 are σ-enabled in σ'. Trace σ is said to have a data race if there is a pair of conflicting events (e_1, e_2) in σ that constitutes a data race of σ.

Example 1. Consider the trace σ_3 in Figure 2a. The set of events of σ_3 is $\mathsf{Events}_{\sigma_3} = \{e_1, e_2, \ldots, e_6\}$, $\mathsf{Thr}_\sigma = \{t_1, t_2\}$, $\mathsf{Vars}_\sigma = \{x\}$ and $\mathsf{Locks}_\sigma = \{\ell\}$. For the event $e_1 = \langle t_1, \mathtt{acq}(\ell)\rangle$, we have $\mathsf{thr}(e_1) = t_1$ and $\mathsf{op}(e_1) = \mathtt{acq}(\ell)$. The trace order of this trace is $\leq^{\sigma_3}_{\mathsf{tr}} = \{(e_i, e_j) \mid i \leq j\}$ and the thread-order is $\leq^{\sigma_3}_{\mathsf{TO}} = \{(e_1, e_2), (e_1, e_5), (e_2, e_5), (e_3, e_4), (e_3, e_6), (e_4, e_6)\}$. Events e_5 and e_6 conflict because they access the same variable x and are performed by different threads. For the prefix trace $\sigma'_3 = e_1 \cdot e_2 \cdot e_3 \cdot e_4$, both e_5 and e_6 are σ_3-enabled in σ'_3. Thus, (e_5, e_6) constitutes a data race of σ_3.

Correct Reorderings. Execution traces of concurrent programs are sensitive to thread scheduling and looking for a trace with a specific pattern is like searching for a needle in a haystack. In terms of data race detection, this means that a dynamic analysis technique that looks for executions with conflicting events enabled (data races) is likely going to miss many data races that might have otherwise been captured in alternate executions of the same program that arise due to slightly different thread scheduling. The notion of data race *prediction* attempts to alleviate this problem by capturing a more robust notion of data races. The idea here is to infer data races that might occur in alternate reorderings of an observed trace, thereby detecting data races beyond those in just the execution that was observed. The set of allowable reorderings of an observed trace σ is defined in a manner that ensures that data races can be detected agnostic of the program that generated σ in the first place. Such a notion is captured by a *correct reordering* which we define next.

For a trace σ and a read event e, we use $\mathsf{lw}_\sigma(e)$ to denote the write event observed by e. That is, $e' = \mathsf{lw}_\sigma(e)$ is the last (according to the trace order $\leq^{\sigma}_{\mathsf{tr}}$) write event e' of σ such that e and e' access the same variable and $e' \leq^{\sigma}_{\mathsf{tr}} e$; if no such e' exists, then we write $\mathsf{lw}_\sigma(e) = \bot$.

Given the above notation, a trace ρ is said to be a correct reordering of trace σ if

(a) $\mathsf{Events}_\rho \subseteq \mathsf{Events}_\sigma$
(b) Events_ρ is downward closed with respect to $\leq^{\sigma}_{\mathsf{TO}}$, and further $\leq^{\rho}_{\mathsf{TO}} \subseteq \leq^{\sigma}_{\mathsf{TO}}$,
(c) for every read event $e \in \mathsf{Events}_\rho$, $\mathsf{lw}_\rho(e) = \mathsf{lw}_\sigma(e)$.

The above definition ensures that if ρ is a correct reordering of σ, then every program that generates the execution trace σ also generates ρ. This is because ρ preserves both intra-thread ordering, as well as the values read by every read occurring in ρ, thereby preserving any control flow that might have been taken by σ. This style of formalizing alternative executions based on semantics of concurrent objects was popularized by [Herlihy and Wing 1990] and by prior race detection works [Said et al. 2011; Şerbănuţă et al. 2012]. Our definition of correct reordering has been derived from [Smaragdakis et al. 2012], which has subsequently also been used in the literature [Genç et al. 2019; Kini et al. 2017; Mathur et al. 2018, 2020; Pavlogiannis 2019; Roemer et al. 2018].

Data Race Prediction. Armed with the notion of correct reorderings, we can now define a more robust notion of data races. A pair of conflicting events (e_1, e_2) in σ is said to be a *predictable* data race of σ if there is a correct reordering ρ of σ such that e_1, e_2 are σ-enabled in ρ. We remark that a pair of conflicting events (e_1, e_2) in trace σ may not be a data race of σ, but nevertheless may still be a *predictable* data race of σ.

Example 2. Consider the trace σ_4 in Figure 2b. Observe that there is no prefix of σ_4 in which both e_1 and e_6 are enabled. However, (e_1, e_6) is a predictable race of σ_4 that is witnessed by the singleton correct reordering $\sigma_4^{\mathsf{CR}} = e_5$ in which both e_1 and e_6 are enabled; σ_4^{CR} is both downward closed with respect to and respects $\leq_{\mathsf{TO}}^{\sigma_4}$. Further, it has no read events and thus vacuously every read observes the same last write as in σ_4. The other pair of conflicting events in σ_4, namely (e_3, e_6), however, is not a predictable race. These events are protected by a common lock, and there is no correct reordering in which e_3 and e_6 are simultaneously enabled — any attempt at doing so will lead to overlapping critical sections on ℓ, thereby violating lock semantics.

Example 3. Now, consider σ_5 in Figure 2c. Here, the conflicting pair (e_3, e_6) cannot be a predictable race as in the case of σ_4— the lock ℓ protects both e_3 and e_6. Now consider the other conflicting pair (e_1, e_8). Let ρ be a correct reordering of σ_5 in which e_8 is enabled. We must have $e_6 \in \mathsf{Events}_\rho$ (ρ must be $\leq_{\mathsf{TO}}^{\sigma_5}$-downward closed) and further $e_3 \in \mathsf{Events}_\rho$ (as $e_3 = \mathsf{lw}_{\sigma_5}(e_6) = \mathsf{lw}_\rho(e_6)$). Clearly, e_1 cannot be enabled in any such trace ρ, and thus, the trace σ_5 has no predictable data race.

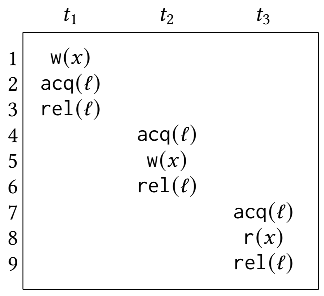
The central theme of race prediction is to solve the problem below.

Problem 1 (Data Race Prediction). Given a trace σ, determine if σ has a predictable data race.

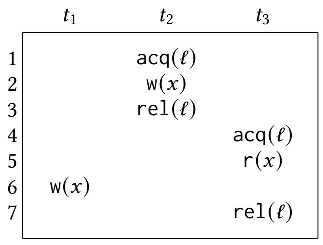
A note on soundness. We say that an algorithm for data race prediction is *sound* if whenever the algorithm reports a YES answer, then the given trace has a predictable data race. Likewise, an algorithm is complete if the algorithm reports YES whenever the input trace has a data race. Our convention for this nomenclature ensures that no false positives are reported by a *sound* algorithm [Sergey 2019] and is consistent with prior work on data race prediction [Genç et al. 2019; Kini et al. 2017; Pavlogiannis 2019; Roemer et al. 2018; Smaragdakis et al. 2012]. Soundness is often a desirable property for dynamic race predictors for widespread adoption [Gorogiannis et al. 2019].

2.1 Sync-Reversal Free Data Races

In general, the problem of data race prediction is intractable [Mathur et al. 2020], and a sound and complete algorithm for data race prediction is unlikely to scale beyond programs of even modest size. A recent trend in predictive analysis for race detection instead, aims to develop techniques that are sound but incomplete, with successively better prediction power (ability to report more data races) than previous techniques [Genç et al. 2019; Kini et al. 2017; Pavlogiannis 2019; Roemer et al. 2018; Smaragdakis et al. 2012]. Most of these techniques are either based on partial orders [Kini et al. 2017; Pozniansky and Schuster 2003; Smaragdakis et al. 2012] or use graph-based algorithms [Pavlogiannis 2019; Roemer et al. 2018]. In this paper, we characterize a class of predictable data races, called *sync-reversal free* races, which we define shortly. We will later (Section 4) present an algorithm that reports a race iff the input trace has a sync-reversal free race. Since sync-reversal free races are predictable races, our algorithm will be sound for race prediction.



(a) Trace σ_6



(b) Sync-reversal free correct reordering σ_6^{CR} of σ_6

Fig. 3. Sync-reversal free correct reordering and sync-reversal free races

Sync-Reversal Free Correct Reordering. A correct reordering of a trace is called *synchronization-reversal free* (*sync-reversal free*, for short) if it does not reverse the order of synchronization constructs; in our formalism, traces uses locks as synchronization primitives to enforce mutual exclusion. Formally, a correct reordering ρ of a given trace σ is *sync-reversal free* with respect to σ if for every lock ℓ and for any two acquire events $e_1, e_2 \in \mathsf{Acquires}_\rho(\ell)$, we have $e_1 \leq^\rho_{\mathsf{tr}} e_2$ iff $e_1 \leq^\sigma_{\mathsf{tr}} e_2$. In other words, the order of two critical sections on the same locks do not change across a trace σ and its sync-reversal free correct reordering ρ. Let us illustrate the notion of sync-reversal free-ness using an example.

Example 4. Consider trace σ_6 in Figure 3a. This trace has 3 critical sections on lock ℓ. Now consider the correct reordering σ_6^{CR} (Figure 3b) of σ_6. Here, the critical section in thread t_1 is not present. But, nevertheless, the order amongst the remaining critical sections on ℓ (in threads t_2 and t_3) is the same as in σ_6, making σ_6^{CR} also a sync-reversal free correct reordering of σ_6. This example also demonstrates that the order of read and write events may be different in a trace and its sync-reversal free correct reordering (as in Figure 3).

A pair of conflicting events (e_1, e_2) of a trace σ is said to be a **sync-reversal free race** of σ if there is a sync-reversal free correct reordering ρ of σ in which e_1 and e_2 are σ-enabled.

Example 5. Let us again consider traces from Figure 3. Events e_1 and e_8 in σ_6 (Figure 3a) correspond respectively to events e_6 and e_7 in σ_6^{CR} (Figure 3b). These two events are σ_6-enabled in the prefix $\rho = e_1{\cdot}e_2{\cdot}e_3{\cdot}e_4{\cdot}e_5$ of σ_6^{CR}. As a result, (e_1, e_8) is a sync-reversal free race of σ_6. Likewise, (e_1, e_4) is also a sync-reversal free race of σ_6 witnessed by the singleton sync-reversal free correct reordering $\rho' = \langle t_2, \mathsf{acq}(\ell)\rangle$, in which both e_1 and e_4 are enabled.

In this article, we will present a linear algorithms for the following decision problem, giving a *sound* algorithm for Problem 1.

Problem 2 (Sync-Reversal Free Race Prediction)**.** Given trace σ, determine if there is a pair of conflicting events (e_1, e_2) in σ such that (e_1, e_2) is a sync-reversal free data race of σ.

Comparison with other approaches. Here we briefly compare sync-reversal free races with other approaches in the literature for sound dynamic race prediction. The famous *happens-before* HB partial order [Pozniansky and Schuster 2003], and its extension to *schedulable-happens-before SHB* [Mathur et al. 2018] are strictly subsumed by this notion. That is, they only compute sync-reversal free races, but can also miss simple cases of sync-reversal freeness, as already illustrated in the examples of Figure 1. The *causally precedes* (CP) partial order [Smaragdakis et al. 2012], and its extension to the *weak causally precedes* (WCP) partial order [Kini et al. 2017] are capable of predicting races that reverse critical sections. However, they are closed under composition with HB, and as such can miss even simple sync-reversal free races, even on two-threaded traces. The *does not commute* DC partial order [Roemer et al. 2018] is an unsound weakening to WCP, that further undergoes a vindication phase to filter out unsound reports. Nevertheless, DC is somewhat similar to WCP, while the two are identical for two-threaded traces. As such, DC also misses sync-reversal free races on such inputs. The recently introduced partial order *strong-dependently-precedes* (SDP) [Genç et al. 2019], while claimed to be sound in that paper, is actually unsound. In Appendix D, we show a counter-example to the soundness theorem of SDP. We further refer to Section C for a few examples that illustrate the above comparison.

3 SUMMARY OF MAIN RESULTS

Here we give an outline of the main results of this paper. In later sections we present the details, i.e., algorithms, proofs and examples. Due to limited space, some technical proofs are relegated to the appendix. Our first result is an algorithm for dynamic prediction of sync-reversal free races. We show the following theorem.

Theorem 3.1. *Sync-reversal free race prediction is solvable in $O(\mathcal{N} \cdot \mathcal{T}^2 + \mathcal{A} \cdot \mathcal{V} \cdot \mathcal{T}^3)$ time and $O(\mathcal{N} + \mathcal{T}^3 \cdot \mathcal{V} \cdot \mathcal{L})$ space, for a trace σ with length $\mathcal{N}$, $\mathcal{T}$ threads, $\mathcal{A}$ lock-acquires, and $\mathcal{V}$ variables.*

In many settings the number of events $\mathcal{N}$ and number of lock-acquires $\mathcal{A}$ are the dominating parameters, whereas the other parameters are much smaller, i.e., $\mathcal{T}, \mathcal{V} = \widetilde{O}(1)$, where $\widetilde{O}$ hides poly-logarithmic factors. Hence, the complexity of our algorithm becomes $\widetilde{O}(\mathcal{N})$ for both time and space. Our next result shows that a linear space complexity is essentially unavoidable when predicting sync-reversal free races with one-pass streaming algorithms, which captures most algorithms in the literature.

Theorem 3.2. *Any one-pass algorithm for sync-reversal free race prediction on traces with ≥ 2 threads, $\mathcal{N}$ events and $\Omega(\log \mathcal{N})$ locks uses $\Omega(\mathcal{N}/\log^2 \mathcal{N})$ space.*

Clearly, any algorithm for the problem must spend linear time, while Theorem 3.2 shows that the algorithm must also use (nearly) linear space. As our algorithm has $\widetilde{O}(\mathcal{N})$ time and space complexity, it is optimal for both resources, modulo poly-logarithmic improvements. Our next theorem shows a combined time-space lower bound for the problem, which highlights that reducing the space usage must lead to an increased running time, given that the algorithm is executed on the Turing Machine model.

Theorem 3.3. *Consider the problem of sync-reversal free race prediction on traces with ≥ 2 threads, $\mathcal{N}$ events and $\Omega(\log \mathcal{N})$ locks. Consider any Turing Machine algorithm for the problem with time and space complexity $T(\mathcal{N})$ and $S(\mathcal{N})$, respectively. Then we have $T(\mathcal{N}) \cdot S(\mathcal{N}) = \Omega(\mathcal{N}^2/\log^2 \mathcal{N})$.*

Finally, we study the complexity of general race prediction as a function of the number of reversals of synchronization operations. Given our positive result in Theorem 3.1, can we relax our restriction of sync-reversal freeness while retaining a tractable definition of predictable races? Our next theorem answers this question in negative.

Theorem 3.4. *Dynamic race prediction on traces with a single lock and two critical sections is* $\mathrm{W}[1]$*-hard parameterized by the number of threads.*

Note that $\mathrm{W}[1]$-hardness implies NP-hardness. The theorem has two important implications.

(1) Any witness of predictable races in the setting of Theorem 3.4 is either a sync-reversal free reordering, or reverses the order of a single pair of lock-acquire events. Thus, Theorem 3.1 and Theorem 3.4 establish a *tight dichotomy* on the tractability of the problem, based on the number of synchronization reversals: the problem is as hard as in the general case for just 1 reversal, while it is efficiently solvable for no reversals.
(2) The general problem of dynamic race prediction was shown to be W[1]-hard in [Mathur et al. 2020]. However, that proof requires traces with $\Omega(\mathcal{N})$ critical sections, and hence it applies to traces that essentially comprise of synchronization events entirely. In contrast to, the class of traces in Theorem 3.4 have the smallest level of synchronization possible, i.e., just a single lock and two critical sections on that lock. Hence, Theorem 3.4 shows that *any* amount of lock-based synchronization suffices to make the problem as hard as in the general case.

Together, Theorem 3.1, Theorem 3.2 and Theorem 3.4 characterize *exactly* the tractability boundary of race prediction, and show that our algorithm is *time and space optimal* for the tractable side.

4 DETECTING SYNC-REVERSAL FREE RACES

In this section, we discuss our algorithm SRFree for detecting sync-reversal free data races. The complete algorithm is presented in Section 4.4. The algorithm may appear complex at first glance and to make the presentation simple, we first present a high-level overview of the algorithm in Section 4.1. In the overview, we highlight important observations and algorithmic insights for solving smaller subproblems of the main problem of sync-reversal free race prediction. Section 4.2 and Section 4.3 present details of the algorithms for the smaller subproblems, and pave the way for the final algorithm in Section 4.4. In Section 4.5, we present a matching space lowerbound result for detecting sync-reversal free races, thereby showing the optimality of our algorithm.

4.1 Insights and Overview of the Algorithm

Our algorithm, SRFree, relies on several important observations that are crucial for detecting sync-reversal free races in linear time. In order to present these observations, it would be helpful to define intermediate subproblems.

Problem 3 (Sync-Reversal Free Race Prediction Given Pair)**.** Given a trace σ and a pair of conflicting events (e_1, e_2) of σ, determine if (e_1, e_2) is a sync-reversal free data race of σ.

Problem 4 (Sync-Reversal Free Race Prediction Given Event and Thread)**.** Given a trace σ, an event e in σ and a thread $t \neq \text{thr}(e)$, check if there is an event $e' \leq^{\sigma}_{\text{tr}} e$ such that $\text{thr}(e') = t$ and (e', e) constitutes a sync-reversal free race of σ.

Observe that a trace with $\mathcal{N}$ events can have $O(\mathcal{N}^2)$ conflicting pair of events. Thus, an algorithm for Problem 3 that runs in time $O(T(\mathcal{N}))$ can be used to obtain an algorithm for Problem 4 (resp. Problem 2) that runs in time $O(\mathcal{N} \cdot T(\mathcal{N}))$ (resp. $O(\mathcal{N}^2 \cdot T(\mathcal{N}))$) by checking if every other event of the given thread t that conflicts with e is also in race with e (resp. every conflicting pair of events is a race). We will, however, present algorithms for all three problems that run in $O(\mathcal{N})$ time.

4.1.1 Solving Problem 3. . We first observe that when checking for the existence of a sync-reversal free reordering (of a trace σ) that witnesses a race on a given pair (e_1, e_2), it, in fact, suffices to only search for those reorderings ρ which impose the same order on *all of its events* (and not just critical section) as in σ. We formalize this in Lemma 4.1.

Lemma 4.1. *If (e_1, e_2) is a sync-reversal free race of σ, then there is a correct reordering ρ of σ such that both e_1, e_2 are σ-enabled in ρ and $\leq^{\rho}_{\text{tr}} \subseteq \leq^{\sigma}_{\text{tr}}$.*

The implication of Lemma 4.1 is the following. If we want to search for a correct reordering ρ of σ that witnesses a race (e_1, e_2), and further, if we already have access to the set of events $S \subseteq \mathsf{Events}_\sigma$ of a candidate reordering ρ, a simple check suffices – does S form a correct reordering of σ when linearized according to $\leq^\sigma_{\mathsf{tr}}$? In other words, we do not need to enumerate and check all the (exponentially many) permutations of events in S. Thus, Problem 3 — 'search for a sync-reversal free correct reordering ρ' — reduces to a simpler problem — 'search for an appropriate set of events'. Of course, not all sets $S \subseteq \mathsf{Events}_\sigma$ of events can be linearized (according to $\leq^\sigma_{\mathsf{tr}}$) to obtain a correct reordering of σ. At the very least, S should satisfy some sanity conditions which we outline next.

Definition 1 (Thread-Order and Last-Write Closure)**.** Let σ be a trace. A set $S \subseteq \mathsf{Events}_\sigma$ is said to be $(\leq^\sigma_{\mathsf{TO}}, \mathsf{lw}_\sigma)$-closed if (a) S is downward-closed with respect to $\leq^\sigma_{\mathsf{TO}}$, and (b) for any read event $r \in \mathsf{Events}_\sigma$, if $r \in S$ and if $\mathsf{lw}_\sigma(r)$ exists, then $\mathsf{lw}_\sigma(r) \in S$.
The $(\leq^\sigma_{\mathsf{TO}}, \mathsf{lw}_\sigma)$-closure of a set $S \subseteq \mathsf{Events}_\sigma$, denoted $\mathsf{TLClosure}_\sigma(S)$ is the smallest set $S' \subseteq \mathsf{Events}_\sigma$ such that $S \subseteq S'$ and S' is $(\leq^\sigma_{\mathsf{TO}}, \mathsf{lw}_\sigma)$-closed.

We remark that any correct reordering ρ of σ that contains events in the set S must also contain all the events in $\mathsf{TLClosure}_\sigma(S)$.

Definition 2 (Sync-Reversal Free Closure)**.** Let σ be a trace. A set $S \subseteq \mathsf{Events}_\sigma$ is said to be sync-reversal free closed if

(a) S is $(\leq^\sigma_{\mathsf{TO}}, \mathsf{lw}_\sigma)$-closed, and
(b) for any two acquire events $a_1, a_2 \in \mathsf{Acquires}_\sigma(\ell)$ acquiring the same lock ℓ with $a_1 \leq^\sigma_{\mathsf{tr}} a_2$, if both $a_1, a_2 \in S$, then $\mathsf{match}_\sigma(a_1) \in S$

The sync-reversal free closure of a set $S \subseteq \mathsf{Events}_\sigma$, denoted $\mathsf{SRFClosure}_\sigma(S)$ is the smallest set $S' \subseteq \mathsf{Events}_\sigma$ such that $S \subseteq S'$ and S' is sync-reversal free closed.

Intuitively, the set $S' = \mathsf{SRFClosure}_\sigma(S)$ captures the additional set of events that must be present in any sync-reversal free correct reordering ρ of σ given that ρ contains all events in S. First, any correct reordering of σ containing S will contain $\mathsf{TLClosure}_\sigma(S)$ and thus $\mathsf{TLClosure}_\sigma(S) \subseteq S'$ (Condition a). Second, if a correct reordering ρ is sync-reversal free and contains two acquires $a_1 \leq^\sigma_{\mathsf{tr}} a_2$ on the same lock ℓ, then we must also have $a_1 \leq^\rho_{\mathsf{tr}} a_2$. Then, in order to ensure well-formedness of ρ, $\mathsf{CS}_\sigma(a_1)$ must also finish entirely before a_2 in ρ, and thus ρ must contain $\mathsf{match}_\sigma(a_1)$ (Condition b).
For two events $e_1, e_2 \in \mathsf{Events}_\sigma$, we define

$$\mathsf{SRFIdeal}_\sigma(e_1, e_2) = \mathsf{SRFClosure}_\sigma(\{\mathsf{prev}_\sigma(e_1)\} \cup \{\mathsf{prev}_\sigma(e_2)\}),$$

where, for an event e, $\{\mathsf{prev}_\sigma(e)\} = \emptyset$ if $\mathsf{prev}_\sigma(e)$ does not exist. In essence, $\mathsf{SRFIdeal}_\sigma(e_1, e_2)$ contains the necessary set of events that must be present in any sync-reversal free correct reordering that witnesses the race (e_1, e_2). We next show that, in fact, it is also a sufficient set of events, given that it is disjoint from $\{e_1, e_2\}$.

Lemma 4.2. *(e_1, e_2) is a sync-reversal free race of σ iff $\{e_1, e_2\} \cap \mathsf{SRFIdeal}_\sigma(e_1, e_2) = \emptyset$.*

Lemma 4.2, therefore, gives us a straightforward algorithm for solving Problem 3 — compute $I = \mathsf{SRFIdeal}_\sigma(e_1, e_2)$ and check if neither e_1 nor e_2 is in I. In Section 4.2 we outline how to perform this computation in linear time.

4.1.2 Efficiently Solving Problem 3. As noted before, a linear time algorithm for Problem 3 guarantees a *quadratic* time algorithm for Problem 4. In order to design a more efficient *linear time* algorithm, we will exploit *monotonicity* of $\mathsf{SRFIdeal}_\sigma(\cdot, \cdot)$, which we formalize next.

Lemma 4.3. *Let σ be a trace and let $e_1, e_2, e'_1, e'_2 \in \mathsf{Events}_\sigma$ such that $e_1 <^\sigma_{\mathsf{TO}} e'_1$ and $e_2 <^\sigma_{\mathsf{TO}} e'_2$. Then,* $\mathsf{SRFIdeal}_\sigma(e_1, e_2) \subseteq \mathsf{SRFIdeal}_\sigma(e'_1, e'_2)$.

Our linear time algorithm for Problem 4 exploits Lemma 4.3 as follows. Suppose we are checking if a given event e in the given trace σ is in sync-reversal free race with some earlier conflicting event of thread t. To accomplish this, we can scan σ and enumerate the list L of events that belong to t and conflict with e. When checking for race with the first such event e'_{first}, we compute $I_{\mathsf{first}} = \mathsf{SRFIdeal}_\sigma(e'_{\mathsf{first}}, e)$. If a race is found, we are done. Otherwise, we analyze the next event e'_{next} in L and compute $I_{\mathsf{next}} = \mathsf{SRFIdeal}_\sigma(e'_{\mathsf{next}}, e)$. Here Lemma 4.3 ensures that $I_{\mathsf{first}} \subseteq I_{\mathsf{next}}$. Our algorithm exploits this observation by computing the latter set I_{next} incrementally, spending time that is proportional only to the number of *extra* events (cardinality of $I_{\mathsf{next}} \setminus I_{\mathsf{first}}$). This principle is applied repeatedly to subsequent events of L, giving us an overall linear time algorithm (Section 4.3).

4.1.3 Efficiently solving Problem 2. A final ingredient in our incremental linear time algorithm for Problem 2 is the following observation which builds on Lemma 4.3.

Lemma 4.4. *Let σ be a trace and let $e_1, e_2, e'_2 \in \mathsf{Events}_\sigma$ such that $e_1 \leq^\sigma_{\mathsf{tr}} e_2 \leq^\sigma_{\mathsf{TO}} e'_2$, $e_1 \asymp e_2$ and $e_1 \asymp e'_2$. If (e_1, e_2) is not a sync-reversal free race, then (e_1, e'_2) is also not a sync-reversal free race of σ.*

Intuitively, this observation suggests the following. Suppose that when looking for a sync-reversal free race, the algorithm determines that e_2 is not in race with any earlier conflicting event. Then, for an event e'_2 (that appears later in the thread of e_2), we only need to investigate if e'_2 is in race with conflicting events e'_1 that appear *after* e_2 in the trace (i.e., $e_2 \leq_{\mathsf{tr}} e'_1 \leq_{\mathsf{tr}} e_2$), instead of additionally looking for races (e''_1, e'_2) where $e''_1 \leq_{\mathsf{tr}} e_2$.

Equipped with Lemma 4.3 and Lemma 4.4, we can now describe our incremental algorithm for Problem 2 that works in linear time. For ease of explanation, let us focus on the question —- is there a write-write race on some fixed variable $x \in \mathsf{Vars}$ when accessed in two fixed threads $t_1, t_2 \in \mathsf{Vars}$. The algorithm scans the trace in a streaming forward pass and analyzes every access event $e = \langle t_2, \mathtt{w}(x)\rangle$, checking if there is an earlier conflicting event $e' = \langle t_1, \mathtt{w}(x)\rangle \leq_{\mathsf{tr}} e$ so that (e', e) is a sync-reversal free race. In doing so, it computes $I = \mathsf{SRFIdeal}_\sigma(e', e)$ in linear time and checks if $e' \notin I$. If not, (e', e) is not a race and the algorithm checks if there is a different event $e'_{\mathsf{next}} \leq_{\mathsf{tr}} e$ so that (e'_{next}, e) is a race. This continues until there are no earlier events remaining that conflict with e. Each time, the ideal computation is performed incrementally, by using the previously computed ideals. After this, the algorithm moves to the event $e_{\mathsf{next}}\langle t_2, \mathtt{w}(x)\rangle$ in t_2 and checks if it is in race with some earlier event e'', where this time, e'' appears after the previously discarded event e of t_2. Again, the closed set $\mathsf{SRFIdeal}_\sigma(e'', e_{\mathsf{next}})$ is done incrementally based on previously computed ideals. We show that all the incremental computation can be performed efficiently and present an outline for our algorithm for Problem 2 in Section 4.4.

Next, we present high-level descriptions of the intermediate steps that we outlined above, and discuss important algorithmic insights and data-structures that help achieve efficiency.

4.2 Checking if a given pair of conflicting events is a sync-reversal free race

Algorithm 1 outlines our solution to Problem 3 (check if a given pair of events (e_1, e_2) is a sync-reversal free race of σ). This algorithm computes the closure $\mathsf{SRFClosure}_\sigma(e_1, e_2)$ in an iterative fashion and checks if it contains neither e_1 nor e_2 (see Lemma 4.2). We remark that when $e_1 \leq^\sigma_{\mathsf{tr}} e_2$, Definition 2 ensures that $e_2 \notin \mathsf{SRFIdeal}_\sigma(e_1, e_2)$. Consequently, the check '$e_1 \notin \mathsf{SRFClosure}_\sigma(e_1, e_2)$' (Line 9 in Algorithm 1) is equivalent to the condition '$\{e_1, e_2\} \cap \mathsf{SRFClosure}_\sigma(e_1, e_2) = \varnothing$' (due to Lemma 4.2). The function `ComputeSRFClosure` performs a fixpoint computation, starting from

Algorithm 1: *Checking if a given conflicting pair constitutes a Sync-Reversal Free race*

```
   Input: Trace $\sigma$, Conflicting events $e_1$ and $e_2$ with $e_1 \leq^{\sigma}_{\mathsf{tr}} e_2$
 1 function ComputeSRFClosure($\sigma$, $e_1$, $e_2$, $I_0$)
 2     $I \leftarrow I_0 \cup \mathsf{TLClosure}_\sigma(\{\mathsf{prev}_\sigma(e_1)\}) \cup \mathsf{TLClosure}_\sigma(\{\mathsf{prev}_\sigma(e_2)\})$
 3     repeat
 4         if $\exists \ell \in \mathsf{Locks}_\sigma, \exists a_1, a_2 \in \mathsf{Acquires}_\sigma(\ell)$ such that $a_1 \leq^{\sigma}_{\mathsf{tr}} a_2$ and $a_1, a_2 \in I$ then
 5             $I \leftarrow I \cup \mathsf{TLClosure}_\sigma(\{\mathsf{match}_\sigma(a_1)\})$
 6     until $I$ does not change
 7     return $I$
 8 $I \leftarrow$ ComputeSRFClosure($\sigma$, $e_1$, $e_2$, $\varnothing$)
 9 if $e_1 \notin I$ then
10     declare 'race'
```

the set $I = \bigcup_{i \in \{1,2\}} \mathsf{TLClosure}_\sigma(\mathsf{prev}_\sigma(e_i))$ (when $I_0 = \varnothing$). The correctness of the algorithm follows from the correctness of the function `ComputeSRFClosure`, which we formalize below.

Lemma 4.5. *Let σ be a trace, $e_1, e_2 \in \mathsf{Events}_\sigma$ and $I_0 \subseteq \mathsf{Events}_\sigma$ be a $(\leq^{\sigma}_{\mathsf{TO}}, \mathsf{lw}_\sigma)$-closed set. Let I be the set returned by* `ComputeSRFClosure` *(σ, e_1, e_2, I_0) in Algorithm 1. Then, $I = \mathsf{SRFClosure}_\sigma(I_0 \cup \{prev_\sigma(e_1)\} \cup \{prev_\sigma(e_2)\})$.*

Let us discuss the data-structures we use to ensure that Algorithm 1 runs in linear time and space.

4.2.1 Vector Timestamps. Vector timestamps [Fidge 1991; Mattern 1988] are routinely exploited in distributed computing and also prior work on race prediction [Flanagan and Freund 2009; Kini et al. 2017; Pozniansky and Schuster 2003; Roemer et al. 2018]. We use vector timestamps to represent sets of events that are $(\leq^{\sigma}_{\mathsf{TO}}, \mathsf{lw}_\sigma)$-closed; a formal definition of vector timestamps is deferred to Section 4.4. In Algorithm 1, the sets $\mathsf{TLClosure}_\sigma(\{\mathsf{prev}_\sigma(e_i)\})$ are $(\leq_{\mathsf{TO}}, \mathsf{lw})$-closed (Line 2). Further, if the input I_0 is also assumed to be $(\leq_{\mathsf{TO}}, \mathsf{lw})$-closed, then all subsequent values of I in `ComputeSRFClosure` will also be $(\leq_{\mathsf{TO}}, \mathsf{lw})$-closed. All these sets can then be represented as vector timestamps. The advantage of using vector timestamps is two-folds. First, these timestamps provide a succinct representation of sets — instead of representing a set explicitly as a collection of events, a vector timestamp only uses $\mathcal{T}$ integers (where $\mathcal{T} = |\mathsf{Thr}_\sigma|$). Second, the vector timestamps for $(\leq^{\sigma}_{\mathsf{TO}}, \mathsf{lw}_\sigma)$-closed sets can be computed in a streaming fashion, incrementally, using vector timestamps of smaller subsets.

4.2.2 Projecting trace to threads and locks. Let us consider the check in Line 4. Here, we look for two acquire events $a_1 \leq^{\sigma}_{\mathsf{tr}} a_2$ in the current ideal I that acquire the same lock ℓ. How do we discover two such acquires? A straightforward way is to enumerate all pairs of events in I and check if they are acquire events of the above kind. But this can take $O(\mathcal{N}^3)$, where $\mathcal{N} = |\mathsf{Events}_\sigma|$ because the number of such pairs can be $O(\mathcal{N}^2)$ in the worst case and the number of times the ideal can change is $O(\mathcal{N})$. Instead, we rely on the following observation:

Proposition 4.6. *Let $I \subseteq \mathsf{Events}_\sigma$ be downward closed with respect to $\leq^{\sigma}_{\mathsf{TO}}$. For every $t \in \mathsf{Thr}_\sigma$ and every $\ell \in \mathsf{Locks}_\sigma$, there is at most one acquire event $a = \langle t, \mathsf{acq}(\ell) \rangle$ such that $a \in I$ and $\mathsf{match}_\sigma(a) \notin I$. When such an event a exists, then $\mathsf{match}_\sigma(a') \in I$ for every other acquire $a' <^{\sigma}_{\mathsf{TO}} a$ of the form $a' = \langle t, \mathsf{acq}(\ell) \rangle$.*

The above observation can be exploited as follows. Let $e^I_{t,\ell}$ be the last acquire on lock ℓ performed by thread t such that $e^I_{t,\ell} \in I$. Let $\mathsf{Acq}^I_\ell = \{e^I_{t,\ell}\}_{t \in \mathsf{Thr}_\sigma}$ and let e^I_ℓ be the last event (according to trace order

$\leq^{\sigma}_{\mathrm{tr}}$) in Acq^I_ℓ. Then, for every other acquire $e' \in \mathsf{Acq}^I_\ell \setminus \{e^I_\ell\}$, the matching release $\mathsf{match}_\sigma(e')$ must be included in I . Hence, if we can efficiently determine the events $e^I_{t,\ell}$ each time, then we can also efficiently determine e^I_ℓ and thus efficiently perform the closure each time. So, how do we determine $e^I_{t,\ell}$ efficiently each time? We achieve this by maintaining lists of the form $\{\mathsf{CSHist}_{t,\ell}\}_{t\in\mathsf{Thr}_\sigma,\ell\in\mathsf{Locks}_\sigma}$. Each such list $\mathsf{CSHist}_{t,\ell}$ is a sequence of entries corresponding to the critical sections on ℓ performed by thread t, in the order in which they were performed in the trace σ. Every entry (corresponding to critical section with acquire a) is a pair $(\mathsf{TLClosure}_\sigma(a), \mathsf{TLClosure}_\sigma(\mathsf{match}_\sigma(a)))$, represented as a pair of vector timestamps. We can now traverse each list in $\mathsf{CSHist}_{t,\ell}$ ($t \in \mathsf{Thr}_\sigma$), until the entry corresponding to the last acquire $e^I_{t,\ell}$ that belongs to I (this corresponds to a simple timestamp comparison). All entries in $\mathsf{CSHist}_{t,\ell}$ prior to the identified event $e^I_{t,\ell}$ can then be discarded from $\mathsf{CSHist}_{t,\ell}$, because the ideal now contains them and only grows monotonically through the course of the fixpoint computation. Since every entry in the lists $\{\mathsf{CSHist}_{t,\ell}\}_{t,\ell}$ is traversed only once, the overall fixpoint computation runs in linear time when the number of $\mathcal{T}$ is constant.

4.3 Checking for a Sync-Reversal Free race on a given event with a given thread

Algorithm 2: *Checking if there is a sync-reversal free race on a given event with a given thread*

```
  Input: Trace $\sigma$, Event $e = \langle t, a(x)\rangle$, Thread $u$
1 for each $b \in \{\mathtt{r}, \mathtt{w}\}$ such that $b \asymp a$[4]
2     let $L_b$ be the list of events $e'$ of the form $\langle u, b(x)\rangle$ such that $e' \leq^{\sigma}_{\mathrm{tr}} e$
3 for each $b \in \{\mathtt{r}, \mathtt{w}\}$ such that $b \asymp a$
4     $I_b \leftarrow \emptyset$
5     for each $e' \in L_b$
6         $I_b \leftarrow$ ComputeSRFClosure($\sigma$, $e'$, $e$, $I_b$)
7         if $e' \notin I_b$ then
8             declare 'race' and exit
```

Let us now consider Algorithm 2. This algorithm takes as input a trace σ, an event $e = \langle t, a(x)\rangle$ and a threads $u \neq t$, and checks if there is an event e' of thread u such that $e' \leq^{\sigma}_{\mathrm{tr}} e$ and (e', e) is is a sync-reversal free race. Algorithm 2 works as follows. For the sake of simplicity, assume that e is a read event, i.e., $a = \mathtt{r}$. The algorithm assumes access to the list $L_{\mathtt{w}}$ of write events $e' = \langle u, \mathtt{w}(x)\rangle$ that appear prior to e (Line 2); these lists can be constructed in linear time, in a single pass traversal of the trace. The algorithm simply traverses $L_{\mathtt{w}}$ (according to trace order $\leq^{\sigma}_{\mathrm{tr}}$) and checks for races with each event in $L_{\mathtt{w}}$, by computing the fixpoint closure sets as in Algorithm 1. Instead of computing the ideal from scratch, the algorithm exploits the monotonicity property outlined earlier in Lemma 4.3 by reusing the ideal $I_{\mathtt{w}}$ computed so far. As with Algorithm 1, this algorithm also uses vector timestamps and maintains lists $\{\mathsf{CSHist}_{t,\ell}\}_{t\in\mathsf{Thr},\ell\in\mathsf{Locks}}$ for computing successive ideals efficiently. Overall again, each entry in $\{\mathsf{CSHist}_{t,\ell}\}_{t\in\mathsf{Thr},\ell\in\mathsf{Locks}}$ is visited a constant number of times and thus Algorithm 2 runs uses linear time and space.

4.4 Algorithm SRFree for Sync-Reversal Free Race Prediction

The pseudo-code for SRFree is presented in Algorithm 3. This is a one pass streaming algorithm that processes events as they appear in the trace, modifying its state and detecting races on the

[4]$b \asymp a$ whenever not both b and a are read ($\mathtt{r}$) operations

Algorithm 3: *Detailed Streaming Algorithm for Checking Sync-Reversal Free races*

```
function Initialization()
    foreach $t \in$ Thr · do
        $\mathbb{C}_t := \bot$
    foreach $x \in$ Vars · do
        $\mathbb{LW}_x := \bot$
    foreach $\ell \in$ Locks · do
        $\mathbb{g}_\ell := 0$
    foreach $t_1, t_2 \in$ Thr, $a_1, a_2 \in \{$r, w$\}$, $x \in$ Vars do
        $\mathbb{I}^{\langle t_1,t_2,a_1,a_2,x\rangle} := \bot$
        foreach $t \in$ Thr, $\ell \in$ Locks do
            CSHist$^{\langle t_1,t_2,a_1,a_2,x\rangle}_{t,\ell} := \varnothing$
    foreach $u \in$ Thr do
        foreach $t \in$ Thr, $a \in \{$r, w$\}$, $x \in$ Vars do
            AccessHist$^{\langle u\rangle}_{t,a,x} := \varnothing$

function maxLowerBound($U$, $Lst$)
    $(g_{\max}, C_{\max}, C'_{\max}) := (0, \bot, \bot)$
    while not $Lst$ • isEmpty() do
        $(g, C, C') := Lst$ • first()
        if $C \sqsubseteq U$ then
            $(g_{\max}, C_{\max}, C'_{\max}) := (g, C, C')$
        else
            break
        $Lst$ • removeFirst()
    return $(g_{\max}, C_{\max}, C'_{\max})$

function ComputeSRFClosure($I$, $\langle t_1, t_2, a_1, a_2, x\rangle$)
    repeat
        foreach $\ell \in$ Locks do
            foreach $t \in$ Thr do
                $(g_{\ell,t}, C_{\ell,t}, C'_{\ell,t})$ := maxLowerBound($I$,
                    CSHist$^{\langle t_1,t_2,a_1,a_2,x\rangle}_{t,\ell}$)
            $t_{\max} := \mathrm{argmax}_{t \in \mathsf{Thr}} \{g_{\ell,t}\}$
            $I := I \sqcup \bigsqcup_{t \neq t_{\max} \in \mathsf{Thr}} C'_{\ell,t}$
    until $I$ does not change
    return $I$

function checkRace($Lst$, $I$, $\langle t_1, t_2, a_1, a_2, x\rangle$)
    while not $Lst$ • isEmpty() do
        $(C_{\mathrm{prev}}, C) := Lst$ • first()
        $I$ := ComputeSRFClosure($I \sqcup C_{prev}$,
            $\langle t_1, t_2, a_1, a_2, x\rangle$)
        if $C' \not\sqsubseteq I$ then
            declare '$(a_1, a_2)$ race on $x$'
            break
        $Lst$ • removeFirst()
    return $I$

handler read($t$, $x$)
    $C_{\mathrm{prev}} := \mathbb{C}_t$
    $\mathbb{C}_t := \mathbb{C}_t[t \mapsto \mathbb{C}_t(t) + 1] \sqcup \mathbb{LW}_x$
    foreach $u \in$ Thr, $x \in$ Vars do
        AccessHist$^{\langle u\rangle}_{t,\mathsf{r},x}$ • addLast(($C_{\mathrm{prev}}$, $\mathbb{C}_t$))
    foreach $u \neq t \in$ Thr do
        $I := \mathbb{I}^{\langle u,t,\mathsf{w},\mathsf{r},x\rangle} \sqcup C_{\mathrm{prev}}$
        $\mathbb{I}^{\langle u,t,\mathsf{w},\mathsf{r},x\rangle}$ := checkRace(AccessHist$^{\langle u\rangle}_{u,\mathsf{w},x}$, $I$,
            $\langle u, t, \mathsf{w}, \mathsf{r}, x\rangle$)

handler write($t$, $x$)
    $C_{\mathrm{prev}} := \mathbb{C}_t$
    $\mathbb{C}_t := \mathbb{C}_t[t \mapsto \mathbb{C}_t(t) + 1]$;    $\mathbb{LW}_x := \mathbb{C}_t$
    foreach $u \in$ Thr, $x \in$ Vars do
        AccessHist$^{\langle u\rangle}_{t,\mathsf{w},x}$ • addLast(($C_{\mathrm{prev}}$, $\mathbb{C}_t$))
    foreach $u \neq t \in$ Thr, $a \in \{$r, w$\}$ do
        $I := \mathbb{I}^{\langle u,t,a,\mathsf{w},x\rangle} \sqcup C_{\mathrm{prev}}$
        $\mathbb{I}^{\langle u,t,a,\mathsf{w},x\rangle}$ := checkRace(AccessHist$^{\langle u\rangle}_{u,a,x}$, $I$,
            $\langle u, t, a, \mathsf{w}, x\rangle$)

handler acquire($t$, $\ell$)
    $\mathbb{C}_t := \mathbb{C}_t[t \mapsto \mathbb{C}_t(t) + 1]$;    $\mathbb{g}_\ell := \mathbb{g}_\ell + 1$
    foreach $t_1, t_2 \in$ Thr, $a_1, a_2 \in \{$r, w$\}$, $x \in$ Vars do
        CSHist$^{\langle t_1,t_2,a_1,a_2,x\rangle}_{t,\ell}$ • addLast(($\mathbb{g}_\ell$, $\mathbb{C}_t$, $\bot$))

handler release($t$, $\ell$)
    $\mathbb{C}_t := \mathbb{C}_t[t \mapsto \mathbb{C}_t(t) + 1]$
    foreach $t_1, t_2 \in$ Thr, $a_1, a_2 \in \{$r, w$\}$, $x \in$ Vars do
        CSHist$^{\langle t_1,t_2,a_1,a_2,x\rangle}_{t,\ell}$ • last() •
            updateRelease($\mathbb{C}_t$)
```

fly. The algorithm maintains several data structures including vector clocks and FIFO queues in its state. We will first describe these data structures, then discuss how the algorithm initializes and modifies them as it processes the trace, and finally discuss the time and space usage for this algorithm; many of these details have already been spelt out in Sections 4.1-4.3.

Let us first briefly explain the notion of vector timestamps and vector clocks [Fidge 1991; Mattern 1988]. A vector timestamp is a mapping $V : \mathsf{Thr}_\sigma \to \mathbb{N}$ from the threads of a trace to natural numbers, and can be represented as a vector of length $|\mathsf{Thr}_\sigma|$. The *join* of two vector timestamps V_1 and V_2, denoted $V_1 \sqcup V_2$ is the vector timestamp $\lambda t, \max(V_1(t), V_2(t))$. Vector timestamps can be compared in a pointwise fashion: $V_1 \sqsubseteq V_2$ iff $\forall t, V_1(t) \leq V_2(t)$. The minimum timestamp is denoted by $\bot$ — $\bot = \lambda t, 0$. For a scalar $c \in \mathbb{N}$, we use $V[t \mapsto c]$ to denote the timestamp $\lambda u,$ if $u = t$ then c else $V(u)$. Vector clocks are variables that take values from the space of vector timestamps. All operations on vector timestamps also apply to clocks. We will use normal font for timestamps ($C, C', I \ldots$) and boldfaced font for vector clocks ($\mathbb{C}, \mathbb{LW}, \mathbb{I}, \ldots$).

Data structures and Initialization. The algorithm maintains the following data structures.

(1) **Vector Clocks**. The algorithm uses vector clocks primarily for two purposes. First, we assign timestamps to all events in the trace and use the following vector clocks for this purpose — for every thread, a dedicated clock $\mathbb{C}_t$, and for every variable x, a dedicated clock $\mathbb{LW}_x$. The timestamp of an event e is essentially a succinct representation of the set $\mathsf{TLClosure}(e)$. Next, the algorithm computes $\mathsf{SRFClosure}(\cdot,\cdot)$ sets and represents them as timestamps. These are stored in vector clocks $\mathbb{I}^{\langle t_1,t_2,a_1,a_2,x\rangle}$, one for every tuple $(t_1, t_2, a_1, a_2, x) \in \mathsf{Thr}{\times}\mathsf{Thr}{\times}\{\mathtt{r},\mathtt{w}\}{\times}\{\mathtt{r},\mathtt{w}\}{\times}\mathsf{Vars}$. All vector clocks are initialized with the timestamp $\bot = \lambda t, 0$.

(2) **Scalars**. For every lock ℓ, the algorithm maintains a scalar variable $\mathbb{g}_\ell$ to record the index of the last acquire on ℓ seen in the trace seen so far. Each such scalar is initialized with 0.

(3) **FIFO Queues**. The algorithm maintains several FIFO queues, whose entries correspond to different events in the trace. The algorithm ensures that an event appears only once, and additionally ensures that the entries respect the order of appearance of the corresponding events in the trace. The first kind of FIFO queues are used in the fixpoint computation. For this, the algorithm maintains queues $\mathsf{CSHist}_{t,\ell}^{\langle t_1,t_2,a_1,a_2,x\rangle}$, one for every thread t, lock ℓ and tuple (t_1, t_2, a_1, a_2, x). Each entry of $\mathsf{CSHist}_{t,\ell}^{\langle t_1,t_2,a_1,a_2,x\rangle}$ is a triplet (g_e, C_e, C'_e) and corresponds to an acquire event e of the form $\langle t, \ell\rangle$ — here, g_e is the index of e in the trace, C_e is the timestamp of e and C'_e is the timestamp of the matching release $\mathsf{match}(e)$. The second kind of queues are of the form $\mathsf{AccessHist}_{t,x}^{\langle u\rangle}$, one for each $t, u \in \mathsf{Thr}$, $a \in \{\mathtt{r},\mathtt{w}\}$ and $x \in \mathsf{Vars}$. Entry in such a FIFO queue corresponds to access events of the form $e = \langle t, a(x)\rangle$. Each entry is of the form $(C_{\mathsf{prev}(e)}, C_e)$ where $C_{\mathsf{prev}(e)}$ is the timestamp of $\mathsf{prev}(e)$ (and $\bot$ if $\mathsf{prev}(e)$ does not exist) and C_e is the timstamp of e. All FIFO queues are empty ($\varnothing$) in the beginning. We write $L \cdot \mathtt{first()}$ and $L \cdot \mathtt{last()}$ to denote the first and last elements of the FIFO queue L. Further, $L \cdot \mathtt{isEmpty()}$, $L \cdot \mathtt{addLast()}$ and $L \cdot \mathtt{removeFirst()}$ respectively represent functions that check for emptiness of L, add an element at the end of L and remove the earliest (first) element of L.

Let us now describe the working of SRFree. The algorithm works in a streaming fashion and processes each event e as soon as it appears, by calling the appropriate **handler** depending upon the operation performed in e (`read`, `write`, `acquire` or `release`). The argument for each handler is the thread performing the event and the object (variable or lock) accessed in the event. In each handler, the algorithm updates vector clocks to correctly compute timestamps, and maintains the invariants of the FIFO queues. In addition, inside `read` and `write` handler events, the algorithm also checks for races using a fixpoint computation (function `ComputeSRFClosure`). We explain some of these briefly.

Computing Vector Timestamps. The algorithm computes the timestamp of an event e, denoted C_e when processing the event e. The algorithm maintains the following invariant -

$$\forall t \in \mathsf{Thr}, C_e(t) = |\{f \in \mathsf{Events} \mid \mathsf{thr}(f) = t, f \in \mathsf{TLClosure}(e)\}|$$

Observe that, with this invariant, we have $C_e \sqsubseteq C_{e'}$ iff $e \in \mathsf{TLClosure}(e')$. The algorithm uses vector clocks $\{\mathbb{C}_t\}_{t\in\mathsf{Thr}}$ and $\{\mathbb{LW}_x\}_{x\in\mathsf{Vars}}$ and ensures that after processing an event e, (1) $\mathbb{C}_t$ stores the timestamp C_{e_t}, where e_t is the last event by thread t that occurs before e, and (2) $\mathbb{LW}_x$ stores the timestamp C_{e_x}, where e_x is the last event with $\mathsf{op}(e_x) = \mathtt{w}(x)$ that occurs before e. The algorithm correctly maintains these values by appropriate vector clock operations on Lines 45, 53, 60 and 64.

Checking Races. When processing an access event $e = \langle t, a(x)\rangle$, the algorithm checks for a race as follows. For every other thread u and for every other conflicting type $b \asymp a$, the algorithm calls checkRace with the list $Lst = \mathsf{AccessHist}^{\langle t\rangle}_{u,b,x}$ and the timestamp representation of the current ideal as argument. This function, similar to Algorithm 2, scans Lst and reports races by repeatedly performing fixpoint computations and checking membership in some set (using the timestamp comparison in Line 38). The closure computation is performed using the optimizations discussed in Section 4.2 (use of FIFO queues $\mathsf{CSHist}^{\langle\cdots\rangle}_{t,\ell}$).

Space Optimizations. Observe that, for a given thread t and lock ℓ, the algorithm, as presented, maintains $4\mathcal{T}^2{\cdot}\mathcal{V}$ FIFO queues $\{\mathsf{CSHist}^{\langle t_1,t_2,a_1,a_2,x\rangle}_{t,\ell}\}_{t_1,t_2\in\mathsf{Thr},a_1,a_2\in\{\mathtt{r},\mathtt{w}\},x\in\mathsf{Vars}}$. The total number of entries across these queues will then be $O(\mathcal{A}\cdot 4\mathcal{T}^2{\cdot}\mathcal{V})$. To this end, we observe that all the above data structures essentially have the same content, and are suffixes of a common queue corresponding to the critical sections on ℓ in thread t. Indeed, we exploit this redundancy and instead maintain a common underlying data-structure that stores all entries corresponding to acquires and releases on ℓ in t, and maintain a pointer for each $\langle t_1, t_2, a_1, a_2, x\rangle$. Such a pointer keeps track of the starting index of the FIFO queue $\mathsf{CSHist}^{\langle t_1,t_2,a_1,a_2,x\rangle}_{t,\ell}$. With this space optimization, we only store $O(\mathcal{A})$ entries along with additional $4{\cdot}\mathcal{T}^3{\cdot}\mathcal{V}{\cdot}\mathcal{L}$ pointers, one for every tuple $\langle t_1, t_2, a_1, a_2, x\rangle$ and every shared queue indexed by (t, ℓ). The same observations also apply to the FIFO queues $\{\mathsf{AccessHist}^{\langle u\rangle}_{t,a,x}\}_{u\in\mathsf{Thr}}$

Lemma 4.7 (Correctness). *For every access event e in the input trace σ, Algorithm 3 declares a race when processing e iff there is an event e' such that $e' \leq^{\sigma}_{\mathsf{tr}} e$ and (e', e) is a sync-reversal free-race of σ.*

The proof of Lemma 4.7 follows directly from the correctness of the semantics of clocks and other observations outlined in Section 4.1.

The time complexity of the algorithm can be determined as follows. The algorithm visits each entry in the FIFO queues $\mathsf{AccessHist}^{\langle u\rangle}_{t,a,x}$ once, performing constant number of vector clock operations, each running in $O(\mathcal{T})$ time. The total length of all these queues is $O(\mathcal{T}{\cdot}\mathcal{N})$ (more precisely, the number of access events in the trace). Similarly, the algorithm visits each entry in the FIFO queues $\mathsf{CSHist}^{\langle t_1,t_2,a_1,a_2,x\rangle}_{t,\ell}$ once, performing constantly many vector clock operations. The total number of entries in these queues is $O(\mathcal{T}^2{\cdot}\mathcal{V}{\cdot}\mathcal{A})$. This gives us the following complexity for SRFree.

Lemma 4.8 (Complexity). *Let σ be a trace with $\mathcal{T}$ threads, $\mathcal{L}$ locks, $\mathcal{V}$ variables and $\mathcal{N}$ events, of which $\mathcal{A}$ are acquire events. Then, Algorithm 3 runs in time $O(\mathcal{N}\cdot\mathcal{T}^2 + \mathcal{A}\cdot\mathcal{V}\cdot\mathcal{T}^3)$ and uses space $O(\mathcal{N} + \mathcal{T}^3\cdot\mathcal{V}\cdot\mathcal{L})$ on input σ.*

The proof of Theorem 3.1 follows from Lemma 4.7 and Lemma 4.8.

	t_1	t_2
1	acq(b_1)	
2	acq(b_2)	
3	acq(c)	
4	w$_1$(x)	
5	rel(c)	
6	rel(b_2)	
7	rel(b_1)	
8	acq(a_1)	
9	acq(b_2)	
10	r$_2$(x)	
11	rel(b_2)	
12	rel(a_1)	
13	acq(a_2)	
14	acq(b_1)	
15	r$_3$(x)	
16	rel(b_1)	
17	rel(a_2)	
18	acq(a_1)	
19	acq(a_2)	
20	acq(c)	
21	w$_4$(x)	
22	rel(c)	
23	rel(a_2)	
24	rel(a_1)	
25		acq(a_1)
26		acq(a_2)
27		acq(c)
28		w$_1$(x)
29		rel(c)
30		rel(a_2)
31		rel(a_1)
32		acq(b_1)
33		acq(a_2)
34		r$_2$(x)
35		rel(a_2)
36		rel(b_1)
37		acq(b_2)
38		acq(a_1)
39		acq(c)
40		w$_3$(x)
41		rel(c)
42		rel(a_1)
43		rel(b_2)
44		acq(b_1)
45		acq(b_2)
46		acq(c)
47		w$_4$(x)
48		rel(c)
49		rel(b_2)
50		rel(b_1)

Fig. 4. Construction of the trace σ on input $s = u\#v$ where $u = 1001$ and $v = 1011$. Observe that (e_{15}, e_{40}) is a sync-reversal free race, which encodes that $e[3] \neq v[3]$.

4.5 Linear Space Lower Bound

In this section we prove the lower-bounds of Theorem 3.2 and Theorem 3.3, i.e., that any streaming algorithm for sync-reversal free race prediction must essentially use linear space, while the time-space product of any (not necessarily streaming) algorithm for the problem must be quadratic in the length of the input trace.

The language $\mathcal{L}_n$. Given a natural number n, we define the equality language $\mathcal{L}_n = \{u\#v \colon u, v \in \{0,1\}^n \text{ and } u = v\}$, i.e., it is the language of two n-bit strings that are separated by # and are equal.

Lemma 4.9. *Any streaming algorithm that recognizes $\mathcal{L}_n$ uses $\Omega(n)$ space.*

Proof. Assume towards contradiction otherwise, i.e., there is a streaming algorithm that uses $o(n)$ space. Hence the state space of the algorithm is $o(2^n)$. Then, there exist two distinct n-bit strings $u_1 \neq u_2$, such that the algorithm is in the same state after parsing u_1 and u_2. Hence, for any n-bit string v, the algorithm gives the same answer on inputs $u_1\#v$ and $u_2\#v$. Since the algorithm is correct, it reports that $u_1\#u_1$ belongs to $\mathcal{L}_n$. But then the algorithm reports that $u_2\#u_1$ also belongs to $\mathcal{L}_n$, a contradiction. The desired result follows. □

Reduction from $\mathcal{L}_n$ recognition to Sync-Reversal Free race prediction. Consider the language $\mathcal{L}_n$ for some n. We describe a transducer $\mathcal{A}_n$ such that, on input a string $s = u\#v$, the output $\mathcal{A}_n(s)$ is a trace σ with 2 threads, $O(n \cdot \log n)$ events, $2 \cdot \log n + 1$ locks and a single variable such that the following hold.

(1) If $s \notin \mathcal{L}_n$, then σ has no predictable race.
(2) If $s \in \mathcal{L}_n$, then σ has a single predictable race, which is a sync-reversal free race.

Moreover, $\mathcal{A}_n$ uses $O(\log n)$ working space. The transducer $\mathcal{A}_n$ uses a single variable x, two sets of locks $A = \{a_1, \dots, a_{\log n}\}$ and $B = \{b_1, \dots, b_{\log n}\}$, plus one additional lock c. The trace σ consists of two local traces π_1, π_2 of threads t_1 and t_2 which encode the bits of u and v, respectively.

(1) The local trace π_1 is constructed as follows. For every $i \in [n]$, π_1 contains an event e^1_i, which is a write event w(x) if $u[i] = 1$, and a read event r(x) otherwise. The events e^1_i are surrounded by locks from A and B arbitrarily, as long as the following holds. For any $i < j$, we have

$$\mathsf{locksHeld}_\sigma(e^1_j) \cap A \not\subseteq \mathsf{locksHeld}_\sigma(e^1_i) \cap A \text{and } \mathsf{locksHeld}_\sigma(e^1_i) \cap B \not\subseteq \mathsf{locksHeld}_\sigma(e^1_j) \cap B\,.$$

Here, the locks held at an event e has the obvious meaning : $\mathsf{locksHeld}_\sigma(e) = \{\ell \in \mathsf{Locks}_\sigma \mid \exists a \in \mathsf{Acquires}_\sigma(\ell)$ such that $e \in \mathsf{CS}_\sigma(a)\}$.

The above property can be easily met, for example, by making $\mathcal{A}_n$ perform a breadth-first traversal of the subset-lattice of A (resp., B) starting from the top (resp., bottom). Given the current i, the transducer surrounds e^1_i with the locks of the current element in the corresponding lattice. Finally, every write event e^1_i is surrounded by the lock c.

(2) The local trace π_2 is similar to π_1, i.e., we have an event e^2_i for each $i \in [n]$, which is a write event $\mathsf{w}(x)$ if $u[i] = 1$, otherwise it is a read event $\mathsf{r}(x)$. The locks that surround e^2_i are such that

$$\mathsf{locksHeld}_\sigma(e^2_i) \cap (A \cup B) = A \cup B \setminus \mathsf{locksHeld}_\sigma(e^1_j) \ .$$

Finally, similarly to π_1, every write event e^1_2 is surrounded by the lock c.

See Figure 4 for an illustration. Observe that $\mathcal{A}_n$ uses $O(\log n)$ bits of memory, for storing a bit-set of locks for each set A and B that must surround the current event e^1_i and e^2_i.

Any two events e^1_i, e^2_j are surrounded by a common lock from the set $A \cup B$ iff $i \neq j$. Hence, (e^1_i, e^2_j) may be a predictable race of σ only if $i = j$. In turn, if $u[i] = v[j]$, then either both events are read events, or both are write events. In the former case the events are not conflicting, while in the latter case the two events are surrounded by lock c. In both cases no race occurs between e^1_i and e^2_j.

Lemma 4.10. *The following assertions hold.*

(1) If $s \in \mathcal{L}_n$, then σ has no predictable race.

(2) If $s \notin \mathcal{L}_n$, then σ has a single predictable race, which is a sync-reversal free race.

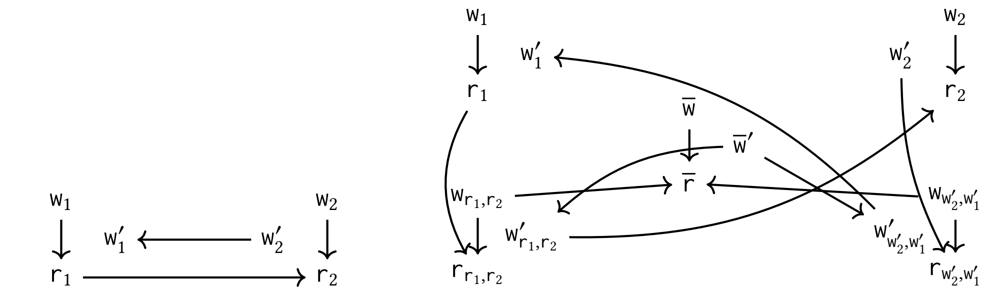
Proof of Theorem 3.2. Consider any algorithm A_1 for sync-reversal free race prediction, executed in the family of traces σ constructed in our above reduction. Let $m = n/\log n$, and assume towards contradiction that A_1 uses $o(m)$ space. Then we can pair A_1 with the transducer $\mathcal{A}_m$, and obtain a new algorithm A_2 for recognizing $\mathcal{L}_m$. Since $\mathcal{A}_m$ uses $O(\log m)$ space, the space complexity of A_2 is $o(m)$. However, this contradicts Lemma 4.9. The desired result follows. □

5 BEYOND SYNCHRONIZATION-REVERSAL FREE RACES

In this section we explore the problem of dynamic race prediction beyond sync-reversal free. We show Theorem 3.4, i.e., that even when just two critical sections are present in the input trace, predicting races with witnesses that might reverse the order of the critical sections becomes intractable. Our reduction is from the realizability problem of Rf-posets, which we present next.

Rf-posets. An rf-poset is a triplet $\mathcal{P} = (X, P, \mathrm{RF})$, where X is a set of read and write events, P defines a partial order $\leq_P$ over X, and $\mathrm{RF}\colon \mathsf{Rds}(X) \to \mathsf{Wts}(X)$ is a reads-from function that maps every read event of X to a write event of X. Given two distinct events $e_1, e_2 \in X$, we write $e_1 \parallel_P e_2$ to denote $e_1 \not\leq_P e_2$ and $e_2 \not\leq_P e_1$. 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 $e_1, e_2 \in Y$, we have $e_1 \leq_{P|Y} e_2$ iff $e_1 \leq_P e_2$. Given a partial order Q over X, we say that Q *refines* P denoted $Q \sqsubseteq P$ if for every two events $e_1, e_2 \in X$, if $e_1 \leq_P e_2$ then $e_1 \leq_Q e_2$. We consider that each event of X belongs to a unique thread, and there is thread order $\leq_{\mathsf{TO}}$ that defines a total order on the events of X that belong to the same thread, and P agrees with $\leq_{\mathsf{TO}}$. The number of threads of $\mathcal{P}$ is the number of threads of the events of X.

The Realizability Problem of Rf-posets. Given an rf-poset $\mathcal{P} = (X, P, \mathrm{RF})$, the *realizability problem* is to decide whether P can be linearized to a total order σ such that $\mathsf{lw}_\sigma = \mathrm{RF}$. It has long been known that the problem is NP-complete [Gibbons and Korach 1997], while it was recently shown that it is even W[1]-hard [Mathur et al. 2020].



(a) An instance of rf-poset realizability. (b) The reduction to reverse rf-poset realizability.

Fig. 5. Reduction of rf-poset realizability (5a) to reverse rf-poset realizability (5b).

Our proof of the lower bound of Theorem 3.4 is by a two-step reduction. First we define a variant of the realizability problem for rf-posets, namely *reverse rf-realizability*, and show that it is W[1]-hard when parameterized by the number of threads. Afterwards, we reduce reverse rf-realizability to the decision problem of dynamic race prediction, which concludes the hardness of the latter.

Rf-triplets. Given an RF-poset $\mathcal{P} = (X, P, \mathrm{RF})$, an *rf-triplet* of $\mathcal{P}$ is a tuple $\lambda = (\mathtt{w}, \mathtt{r}, \mathtt{w}')$ such that (i) $\mathtt{r}$ is a read event, (ii) $\mathrm{RF}(\mathtt{r}) = \mathtt{w}$, and (iii) $\mathtt{w} \asymp \mathtt{w}'$. We refer to $\mathtt{w}$, $\mathtt{r}$ and $\mathtt{w}'$ as the *write*, *read*, and *interfering write* event of λ, respectively. We denote by $\mathsf{Triplets}(\mathcal{P})$ the set of rf-triplets of $\mathcal{P}$.

We next define a variant of rf-poset realizability, and show that, like the original problem, it is W[1]-hard parameterized by the number of threads.

Reverse rf-poset Realizability. The input is a tuple $(\mathcal{P}, \lambda, \sigma)$, where $\mathcal{P} = (X, P, \mathrm{RF})$ is an rf-poset, $\lambda = (\overline{\mathtt{w}}, \overline{\mathtt{r}}, \overline{\mathtt{w}}')$ is a distinguished triplet of $\mathcal{P}$, and σ is a witness to the realizability of $\mathcal{P}$ such that $\overline{\mathtt{w}}' <_{\sigma} \overline{\mathtt{w}}$. The task is to determine whether $\mathcal{P}$ has a linearization σ' with $\overline{\mathtt{w}} <_{\sigma'} \overline{\mathtt{w}}'$. In words, $\mathcal{P}$ is already realizable by a witness that orders $\overline{\mathtt{w}}'$ before $\overline{\mathtt{w}}$, and the task is to decide whether $\mathcal{P}$ also has a witness in which this order is reversed.

Hardness of Reverse Rf-poset Realizability. We show that the problem is W[1]-hard when parameterized by the number of threads of the rf-poset. Our reduction is from rf-realizability. We first present the construction and then argue about its correctness.

Construction. Consider an rf-poset $\mathcal{P} = (X, P, \mathrm{RF})$ with k threads, and we construct an instance of reverse rf-poset realizability $(\mathcal{P}' = (X', P', \mathrm{RF}'), \lambda, \sigma)$ with $k' = O(k^2)$ threads. We refer to Figure 5 for an illustration. For simplicity of presentation, we assume wlog that the following hold.

(1) X contains only the events of the triplets of $\mathcal{P}$.
(2) For every read event $\mathtt{r}$, we have $\mathsf{thr}(\mathtt{r}) = \mathsf{thr}(\mathtt{w})$, i.e., every read observes a local write event.

Let $\{X_i\}_{1 \le i \le k}$ be a partitioning of X such that each $\pi_i = P|X_i$ is a total order containing all events of thread i (i.e., it is the thread order for thread i). We first construct the rf-poset $\mathcal{P}' = (X', P', \mathrm{RF}')$. The threads of $\mathcal{P}'$ are defined implicitly by the sets of events for each thread. In particular, X' is partitioned in the sets X_i (which are the events of $\mathcal{P}$), as well as two sets X_i^j and Y_i^j for each $i, j \in [k]$ with $i \neq j$, where each such X_i^j and Y_i^j contains events of a unique thread of $\mathcal{P}'$. Finally, we have two threads containing the events of the distinguished triplet λ. Hence, $\mathcal{P}'$ has $k' = k + k \cdot (k - 1) + 2 = k^2 + 2$ threads.

We first define the set of triplets $\mathsf{Triplets}(\mathcal{P})$, which defines the event set X' and the observation function RF'. We have $X \subseteq X'$ and $\mathsf{Triplets}(\mathcal{P}) \subseteq \mathsf{Triplets}(\mathcal{P}')$. In addition, we create a distinguished triplet $\lambda = (\overline{\mathtt{w}}, \overline{\mathtt{r}}, \overline{\mathtt{w}}')$, and all its events are in X'. Finally, for every $i, j \in [k]$ with $i \neq j$, we have $X_i^j, Y_i^j \subseteq X'$, where the sets X_i^j and Y_i^j are constructed as follows. We call a pair of events

$(e_1, e_2) \in X_i \times X_j$ with $e_1 <_P e_2$ *dominant* if for any pair $(e_1', e_2') \in X_i \times X_j$ such that $e_1 \leq_P e_1'$, and $e_2' \leq_P e_2$, and $e_1' <_P e_2'$, we have $e_i' = e_i$ for each $i \in [2]$. In words, a dominant pair identifies an ordering in P that cannot be inferred transitively by other orderings. For every dominant pair $(e_1, e_2) \in X_i \times X_j$ we create a triplet $(\mathtt{w}_{e_1,e_2}, \mathtt{r}_{e_1,e_2}, \mathtt{w}'_{e_1,e_2})$, and let $\mathtt{w}_{e_1,e_2}, \mathtt{r}_{e_1,e_2} \in X_i^j$ and $\mathtt{w}'_{e_1,e_2} \in Y_i^j$. We now define the partial order P'. For every triplet $(\mathtt{w}, \mathtt{r}, \mathtt{w}')$ of $\mathcal{P}'$, we have $\mathtt{w} <_{P'} \mathtt{r}$. For every $i, j \in [k]$ with $i \neq j$, for every two events $e_1, e_1' \in X_i$ such that $e_1 <_P e_1'$, for every two events $e_2, e_2' \in X_j$ such that $e_2 <_P e_2'$, if $\mathtt{r}_{e_1,e_2}$ and $\mathtt{w}_{e_1',e_2'}$ are events of X' (i.e., (e_1, e_2) and (e_1', e_2') are dominant pairs), we have (i) $\mathtt{r}_{e_1,e_2} <_{P'} \mathtt{w}_{e_1',e_2'}$ and (ii) $\mathtt{w}'_{e_1,e_2} <_{P'} \mathtt{w}'_{e_1',e_2'}$. Finally, for every triplet of the form $(\mathtt{w}_{e_1,e_2}, \mathtt{r}_{e_1,e_2}, \mathtt{w}'_{e_1,e_2})$, we have

$$e_1 <_{P'} \mathtt{r}_{e_1,e_2} \qquad \text{and} \qquad \mathtt{w}'_{e_1,e_2} <_{P'} e_2 \qquad \text{and} \qquad \mathtt{w}_{e_1,e_2} <_{P'} \overline{\mathtt{r}} \qquad \text{and} \qquad \overline{\mathtt{w}}' <_{P'} \mathtt{w}'_{e_1,e_2} \ .$$

The following lemma establishes that P' is indeed a partial order.

Lemma 5.1. *P' is a partial order.*

We now turn our attention to the solution σ of $\mathcal{P}'$, which is constructed in two steps. First, we construct a partial order $Q \sqsubseteq P'$ over X' which orders in every triplet the interfering write before the write of the triplet. That is, for every triplet $(\mathtt{w}, \mathtt{r}, \mathtt{w}')$ of $\mathcal{P}'$, we have $\mathtt{w}' <_Q \mathtt{w}$. Then, we obtain σ by linearizing Q arbitrarily. The following lemma states that σ witnesses the realizability of $\mathcal{P}'$.

Lemma 5.2. *The trace σ realizes $\mathcal{P}'$.*

Observe that the size of $\mathcal{P}'$ is polynomial in the size of $\mathcal{P}$. The following lemma states the correctness of the reduction. We refer to Appendix B for the detailed proof, while here we sketch the intuition behind the correctness.

Lemma 5.3. *Reverse rf-poset realizability is* W[1]*-hard parameterized by the number of threads.*

Correctness. We now present the key insight behind the correctness of the reduction. Consider any dominant pair of events (e_1, e_2) in the initial rf-poset, i.e., we have $e_1 <_P e_2$. Observe that the two events are unordered in P'. Now consider any trace σ^* that solves the reverse rf-poset realizability problem for $\mathcal{P}'$. By definition, σ^* must reverse the order of the two writes of the conflicting triplet, i.e., we must have $\overline{\mathtt{w}} <_{\sigma^*} \overline{\mathtt{w}}'$.

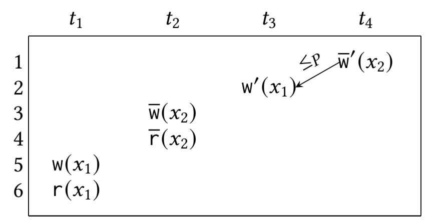
(1) Since, $\overline{\mathtt{w}} <_{\sigma^*} \overline{\mathtt{w}}'$ we must also have $\overline{\mathtt{r}} <_{\sigma^*} \overline{\mathtt{w}}'$, so that the last write of $\overline{\mathtt{r}}$ is not violated in σ^*.
(2) Since $\mathtt{w}_{e_1,e_2} <_{P'} \overline{rd}$, by the previous item we also have transitively $\mathtt{w}_{e_1,e_2} <_{P'} \overline{\mathtt{w}}'$, and since $\overline{\mathtt{w}}' <_{P'} \overline{\mathtt{w}}'_{e_1,e_2}$, we have , transitively $\mathtt{w}_{e_1,e_2} <_{\sigma^*} \mathtt{w}'_{e_1,e_2}$.
(3) Since $\mathtt{w}_{e_1,e_2} <_{\sigma^*} \mathtt{w}'_{e_1,e_2}$, we also have $\mathtt{r}_{e_1,e_2} <_{\sigma^*} \mathtt{w}'_{e_1,e_2}$, so that the last write of $\mathtt{r}_{e_1,e_2}$ is not violated in σ^*.
(4) Finally, since $e_1 < P' \mathtt{r}_{e_1,e_2}$ and $\mathtt{w}'_{e_1,e_2} < e_2$, we also have, transitively, that $e_1 <_{\sigma^*} e_2$.

Hence, the witness σ^* also respects the partial order P, and thus also serves as a witness of the realizability of $\mathcal{P}$ (when projected to the set of events X). Thus, if reverse rf-poset realizability holds for $\mathcal{P}'$, then rf-poset realizability holds for $\mathcal{P}$. The inverse direction is similar.

Hardness of Dynamic Race Prediction. We are now ready to prove our second step of the reduction, i.e., to establish an FPT reduction from reverse rf-poset realizability to the decision problem of dynamic race prediction. We first describe the construction and then prove its correctness.

Consider an instance $(\mathcal{P} = (X, P, \mathrm{RF}), \lambda = (\overline{\mathtt{w}}, \overline{\mathtt{r}}, \overline{\mathtt{w}}'), \sigma)$ of reverse rf-poset realizability, and we construct a trace σ' such a specific event pair of σ' is a predictable race iff $\mathcal{P}$ is realizable by a witness that reverses λ. We assume wlog that X contains only events that appear in triplets of $\mathcal{P}$. We construct σ' by inserting various events in σ, as follows. Figure 6 provides an illustration.

$$\lambda = (\overline{\mathtt{w}}(x_2), \overline{\mathtt{r}}(x_2), \overline{\mathtt{w}}'(x_2))$$



(a) An instance of reverse rf-poset realizability ($\mathcal{P} = (X, P, \mathrm{RF}), \lambda, \sigma$). The figure shows σ, and P is defined as the thread order together with the cross-thread ordering $\mathtt{w}'(x_2) <_P \mathtt{w}'(x_1)$.

	t_1	t_2	t_3	t_4
1				$\mathtt{acq}_1(\ell)$
2				$\overline{\mathtt{w}}'(x_2)$
3				$\mathtt{w}\left(x_{\overline{\mathtt{w}}'(x_2),\mathtt{w}'(x_1)}\right)$
4				$\overline{\mathtt{w}}(\mathbf{y})$
5				$\mathtt{rel}_1(\ell)$
6			$\mathtt{r}\left(x_{\overline{\mathtt{w}}'(x_2),\mathtt{w}'(x_1)}\right)$	
7			$\mathtt{w}'(x_1)$	
8			$\mathtt{w}(x^3)$	
9		$\mathtt{acq}_2(\ell)$		
10		$\overline{\mathtt{w}}(x_2)$		
11		$\mathtt{rel}_2(\ell)$		
12		$\overline{\mathtt{r}}(x_2)$		
13	$\mathtt{w}(x_1)$			
14	$\mathtt{r}(x_1)$			
15	$\mathtt{w}(x^1)$			
16		$\mathtt{r}(x^1)$		
17		$\mathtt{r}(x^3)$		
18		$\overline{\mathtt{r}}(\mathbf{y})$		

(b) The instance of race prediction using our reduction.

Fig. 6. Example of our reduction of an instance of reverse rf-poset realizability (Figure 6a) to an instance of dynamic data-race prediction (Figure 6b) on the event pair (e_4, e_{18}).

(1) For every dominant pair (e_1, e_2) of $\mathcal{P}$, we introduce a new variable x_{e_1,e_2}, and a write event $\mathtt{w}(x_{e_1,e_2})$ and a read event $\mathtt{r}(x_{e_1,e_2})$. We make $\mathrm{thr}(\mathtt{w}(x_{e_1,e_2})) = \mathrm{thr}(e_1)$ and $\mathrm{thr}(\mathtt{r}(x_{e_1,e_2})) = \mathrm{thr}(e_2)$. Finally, we thread-order $\mathtt{w}(x_{e_1,e_2})$ after e_1 and $\mathtt{r}(x_{e_1,e_2})$ before e_2. Notice that any correct reordering σ^* of σ' must order $\mathtt{w}(x_{e_1,e_2}) \leq_{\mathsf{tr}}^{\sigma^*} \mathtt{r}(x_{e_1,e_2})$, and thus, transitively, also order $e_1 \leq_{\mathsf{tr}}^{\sigma^*} e_2$.
(2) For every thread $t_i \neq \mathrm{thr}(\overline{\mathtt{w}})$, $t_i \neq \mathrm{thr}(\overline{\mathtt{w}}')$, we introduce a new variable x^i, and a write event $\mathtt{w}(x^i)$ and a read event $\mathtt{r}(x^i)$. We make $\mathrm{thr}(\mathtt{w}(x^i)) = t_i$ and $\mathrm{thr}(\mathtt{r}(x^i)) = \mathrm{thr}(\overline{\mathtt{r}})$. Finally, we thread-order each $\mathtt{w}(x^i)$ as the last event of t_i, and thread-order all $\mathtt{r}(x^i)$ as final events of $\mathrm{thr}(\overline{\mathtt{r}})$ so far.
(3) We introduce a new variable y, and a write event $\mathtt{w}(y)$ and a read event $\mathtt{r}(y)$. We make $\mathrm{thr}(\mathtt{w}(y)) = \mathrm{thr}(\overline{\mathtt{w}}')$ and $\mathrm{thr}(\mathtt{r}(y)) = \mathrm{thr}(\overline{\mathtt{w}})$. Finally, we thread-order $\mathtt{w}(y)$ and $\mathtt{r}(y)$ at the end of their respective threads. In particular, $\mathtt{r}(y)$ is thread-ordered after the events $\mathtt{r}(x^i)$ introduced in the previous item. Notice that because of this ordering and the previous item, any correct reordering σ^* of σ' must contain all events of X'.
(4) We introduce a lock ℓ and two pairs of lock-acquire and lock-release events $(\mathtt{acq}_i(\ell), \mathtt{rel}_i(\ell))$, for each $i \in [2]$. We make $\mathrm{thr}(\mathtt{acq}_i(\ell)) = \mathrm{thr}(\mathtt{rel}_i(\ell)) = t_j$, where $t_j = \mathrm{thr}(\overline{\mathtt{w}}')$ if $i = 1$ and $t_j = \mathrm{thr}(\overline{\mathtt{w}})$ otherwise. Finally, we surround with the critical section of $\mathtt{acq}_1(\ell), \mathtt{rel}_1(\ell)$ all events of the corresponding thread, and surround with the critical section of $\mathtt{acq}_2(\ell), \mathtt{rel}_2(\ell)$ the event $\overline{\mathtt{w}}$. Notice that any correct reordering σ^* that witnesses a race on $(\mathtt{w}(y), \mathtt{r}(y))$ is missing $\mathtt{rel}_1(\ell)$, and thus must order $\mathtt{rel}_2(\ell) \leq_{\mathsf{tr}}^{\sigma^*} \mathtt{acq}_1(\ell)$. In turn, this leads to a transitive ordering $\overline{\mathtt{w}} \leq_{\mathsf{tr}}^{\sigma^*} \overline{\mathtt{w}}'$, and since the last write of $\overline{\mathtt{r}}$ must be $\mathsf{lw}_{\sigma^*}(\overline{\mathtt{r}}) = \overline{\mathtt{w}}$, we must also have $\overline{\mathtt{r}} \leq_{\mathsf{tr}}^{\sigma^*} \overline{\mathtt{w}}'$.

We now outline the correctness of the reduction (see Appendix B for the proof). Consider any correct reordering σ^* that witnesses a predictable race $(\mathtt{w}(y), \mathtt{r}(y))$ on σ'. Item 2 and Item 3 above guarantee that $X \subseteq \mathsf{Events}_{\sigma^*}$, while Item 1 guarantees that σ^* linearizes P, and Item 4 guarantees that σ^* reverses λ, i.e., $\overline{\mathtt{r}} \leq_{\mathsf{tr}}^{\sigma^*} \overline{\mathtt{w}}'$. Finally, note that σ' has size that is polynomial in n, while the number of threads of σ' equals the number of threads of $\mathcal{P}$. This concludes the proof of Theorem 3.4.

6 EXPERIMENTS

In this section we report on an implementation and experimental evaluation of the techniques presented in this work. Our objective is two-fold. The first goal is to quantify the practical relevance of sync-reversal freeness, i.e., whether in practice the definition captures races that are missed by the standard notion of happens-before and WCP [Kini et al. 2017] races. The second goal is to evaluate the performance of our algorithm SRFree for detecting sync-reversal free races.

6.1 Experimental Setup

We have implemented SRFree (Algorithm 3) for predicting all sync-reversal free races in Java, and evaluated its performance on a standard set of benchmarks.

Benchmarks. Our benchmark set consists of standard benchmarks found in the recent literature [Huang et al. 2014; Kini et al. 2017; Mathur et al. 2018; Pavlogiannis 2019; Roemer et al. 2018; Yu et al. 2018]. It consists of 30 concurrent programs taken from standard benchmark suites: (i) the IBM Contest benchmark suite [Farchi et al. 2003], (ii) the Java Grande forum benchmark suite [Smith and Bull 2001], (iii) the DaCapo benchmark suite [Blackburn et al. 2006], (iv) the Software Infrastructure Repository [Do et al. 2005], and (v) some standalone benchmarks. For each benchmark, we used the tool RV-Predict [Rosu 2018] to instrument it and monitor its execution, which created a single trace per program. The same trace was used for evaluating all algorithms.

Compared Methods. We compare our algorithm with state-of-the-art sound race detectors, namely, SHB [Mathur et al. 2018], WCP [Kini et al. 2017] and M2 [Pavlogiannis 2019]. Recall that SHB and WCP are linear-time algorithms that perform a single pass of the input trace σ. SHB computes happens-before races and is sound even beyond the first race. On the other hand, WCP is only sound for the first race report. In order to allow WCP to soundly report more than one race, whenever a race is reported on an event pair (e_1, e_2) (i.e., we have $e_1 \parallel^{\sigma}_{\mathsf{WCP}} e_2$), we force an order $e_1 \leq^{\sigma}_{\mathsf{WCP}} e_2$ before proceeding with the next event of σ. This is a standard practice that has been followed in other works, e.g.,[Pavlogiannis 2019; Roemer et al. 2018]. Finally, M2 is a heavyweight algorithm that makes sound reports for all races by design, though its running time is a large polynomial (of order n^4) [Pavlogiannis 2019].

Optimizations. In general, the benchmark traces can be huge and often scale to sizes of order as large as 10^8. A closer inspection shows that many events, even though they perform accesses to shared memory, are non-racy and even totally ordered by fork-join mechanisms and data flows in the trace. We have implemented a lightweight, linear-time, single-pass optimization of the input trace σ that filters out such events. The optimization simply identifies memory locations x whose conflicting accesses are totally ordered in σ by thread and data-flow orderings, and ignores all such accesses in σ. For a fair comparison, we employ the optimization in all compared methods.

Reported Results. Each of the compared methods is evaluated on the same input trace σ. For every such input, the respective method reports the following race warnings.

(1) *Racy events.* We report the number of events e_2 such that there is an event e_1 with $e_1 <_{\mathsf{tr}} e_2$ for which a race (e_1, e_2) is detected. We remark that this is the standard way of reporting race warnings [Flanagan and Freund 2009; Genç et al. 2019; Kini et al. 2017; Mathur et al. 2018; Roemer et al. 2018], as it allows for one-pass, linear-time algorithms that avoid the overhead of testing for races between all possible $\Theta(n^2)$ pairs of events.
(2) *Racy source-code lines.* We report the number of distinct source-code lines which correspond to events e_2 that are found as racy in Item 1. This is a meaningful measure, as the same source-code line might be reported by many different events e_2.

Table 1. Dynamic race reports in our benchmarks. $\mathcal{N}$ and $\mathcal{T}$ denote the number of events and number of threads in the respective trace. For races, an entry 'r (s)' denotes the number r of events e_2 found to be in race with an earlier event e_1, as well as the number s of unique source-code lines corresponding to such events e_2. Bold-face entries highlight cases where there are sync-reversal free races that are not happens-before races.

Benchmark	$\mathcal{N}$	$\mathcal{T}$	SHB		WCP		M2		SRFree	
			Races	**Time**	**Races**	**Time**	**Races**	**Time**	**Races**	**Time**
array	51	4	0 (0)	0.02s	0 (0)	0.03s	0 (0)	0.09s	0 (0)	0.04s
critical	59	5	3 (3)	0.19s	1 (1)	0.03s	3 (3)	0.11s	3 (3)	0.07s
account	134	5	3 (1)	0	3 (1)	0.06s	3 (1)	0.23s	3 (1)	0.09s
airtickets	140	5	8 (3)	0.02s	5 (2)	0.03s	8 (3)	0.13s	8 (3)	0.05s
pingpong	151	7	8 (3)	1.09s	8 (3)	0.04s	8 (3)	0.17s	8 (3)	0.06s
twostage	193	13	4 (1)	0.02s	4 (1)	0.09s	4 (1)	0.20s	4 (1)	0.10s
wronglock	246	23	12 (2)	0.02s	3 (2)	0.09s	25 (2)	0.43s	**25 (2)**	0.15s
bbuffer	332	3	3 (1)	0.01s	1 (1)	0.05s	3 (1)	0.11s	3 (1)	0.06s
prodcons	658	9	1 (1)	0.03s	1 (1)	0.09s	1 (1)	0.20s	1 (1)	0.10s
clean	1.0K	10	59 (4)	0.04s	33 (4)	0.14s	110 (4)	0.85s	**60 (4)**	0.17s
mergesort	3.0K	6	1 (1)	11m10s	1 (1)	0.12s	5 (2)	0.96s	**3 (1)**	0.13s
bubblesort	4.0K	13	269 (5)	0.03s	100 (5)	0.27s	374 (5)	8.05s	269 (5)	0.50s
lang	6.0K	8	400 (1)	0.10s	400 (1)	0.23s	400 (1)	1.31s	400 (1)	0.31s
readswrites	11K	6	92 (4)	0.12s	92 (4)	0.41s	228 (4)	12.74s	**199 (4)**	0.77s
raytracer	15K	4	8 (4)	0.02s	8 (4)	0.30s	8 (4)	0.40s	8 (4)	0.30s
bufwriter	22K	7	8 (4)	0.10s	8 (4)	0.70s	8 (4)	2.65s	8 (4)	0.84s
ftpserver	49K	12	69 (21)	6.91s	69 (21)	1.34s	85 (21)	4.11s	**85 (21)**	4.69s
moldyn	200K	4	103 (3)	0.05s	103 (3)	1.83s	103 (3)	1m25s	103 (3)	1.86s
linkedlist	1.0M	13	5.0K (4)	7.25s	5.0K (3)	27.07s	TO	TO	**7.0K (4)**	5m19s
derby	1.0M	5	29 (10)	0.01s	28 (10)	16.48s	30 (11)	22.49s	29 (10)	24.07s
jigsaw	3.0M	12	4 (4)	0.41s	4 (4)	19.53s	6 (6)	11.69s	**6 (6)**	17.30s
sunflow	11M	17	84 (6)	39.66s	58 (6)	47.14s	130 (7)	50.24s	**119 (7)**	55.30s
cryptorsa	58M	9	11 (5)	3m4s	11 (5)	6m35s	TO	TO	**35 (7)**	9m42s
xalan	122M	7	31 (10)	0.15s	21 (7)	15m30s	TO	TO	**37 (12)**	10m44s
lufact	134M	5	21K (3)	7m26s	21K (3)	14m57s	TO	TO	21K (3)	10m38s
batik	157M	7	10 (2)	9m49s	10 (2)	22m56s	TO	TO	10 (2)	11m59s
lusearch	217M	8	232 (44)	12.63s	119 (27)	13m40s	232 (44)	27m9s	232 (44)	14m5s
tsp	307M	10	143 (6)	15m2s	140 (6)	29m10s	TO	TO	143 (6)	20m19s
luindex	397M	3	1 (1)	24m40s	2 (2)	31m6s	TO	TO	**15 (15)**	31m46s
sor	606M	5	0 (0)	38m38s	0 (0)	TO	TO	TO	0 (0)	44m36s
Totals	2.0B	-	29520 (157)	1h51m	29133 (134)	$\geq$ 3h15m	1846 (131)	$\geq$ 8h30m	**30862 (178)**	2h40m

(3) *Racy memory locations.* We report the number of different memory locations that are accessed by all the events e_2 that are found as racy in Item 1.

(4) *Running time.* We measure the time of the algorithm required to process each benchmark, while imposing a 1-hour timeout (TO).

6.2 Experimental Results

We now turn our attention to the experimental results. Table 1 shows the races and running times reported by each method on each benchmark.

Coverage of Sync-reversal Freeness. We first discuss the coverage of sync-reversal free races. We find that *every* race reported by SHB or WCP is a sync-reversal free race, also reported by

Table 2. Numbers of different memory locations that are detected as racy.

Benchmark	SHB	WCP	SRFree
ftpserver	49	49	50
jigsaw	4	4	5
xalan	7	6	9
cryptorsa	4	4	5
luindex	1	2	9
sunflow	14	10	17
linkedlist	927	927	932
Total	1006	1002	1027

Table 3. Maximum race distances.

Benchmark	SHB	WCP	SRFree
tsp	11K	11K	224M
batik	1.7M	1.7M	4.8M
cryptorsa	7.9M	7.9M	8.3M
jigsaw	428	428	121K
sunflow	10M	1.0M	10M
xalan	4K	4K	13K
ftpserver	11K	11K	11K
linkedlist	165K	165K	165K
luindex	783	783	6.9K
mergesort	57	57	1.4K
clean	355	47	1.2K
readswrites	13	13	696
wronglock	50	6	113

SRFree. On the other hand, bold-face entries highlight benchmarks which have sync-reversal free races that are not happens-before races. We see that such races are found in 11 out of 30 benchmarks. Interestingly, in the 5 most challenging out of these 11 benchmarks, the same pattern occurs if we focus on source-code lines (i.e., the entries in the parentheses). Hence, for these benchmarks, sync-reversal freeness is *necessary* to capture many racy source-code lines, which happens-before would completely miss. We also remark that the more heavyweight analysis M2 misses several of these races due to frequent timeouts. In total, we have 18 unique source-code lines that are racy but only detected by SRFree. On the other hand, there are only 2 source-code lines that are caught by M2 but not by SRFree.

Running Times. Our experimental times indicate that SRFree is quite efficient in practice. Among all algorithms, SRFree is the second fastest, being about 1.4 times slower that the fastest, lightweight SHB, while at the same time, being able to detect considerably more races the SHB (i.e., 1342 more racy events, and 21 more racy source-code lines). On the other hand, SRFree detects even more races than M2, due to timeouts, and even has almost equal detection capability with M2 on the cases that M2 does not time out. Due to the slow performance of M2 (i.e., over 8.5 hours and with several timeouts), we exclude it from the more refined analysis that follows.

Racy Memory Locations. We next proceed to evaluate the capability of SRFree in detecting racy memory locations. As all races detected by SHB or WCP are sync-reversal free, the same follows for the racy memory locations, i.e., they are all detected as racy by SRFree. On the other hand, Table 2 shows a few cases in which SRFree has discovered racy variables that are missed by SHB and WCP. Hence, sync-reversal freeness is more adequate to capture not only racy program locations, but also racy memory locations.

Race Distances. Finally, we examine the capability of SRFree to detect races that are far apart in the input trace. Table 3 shows maximum race distance of races (e_1, e_2) in various benchmarks, including the ones that contains sync-reversal free races that are missed by happens-before. In each case, the distance is counted as the number of events in the input trace between e_1 and e_2, for every event e_2 reported as racy. We see a sharp contrast between SHB/WCP and SRFree, with the latter being able to detect races that are far more distant in the input. This is in direct alignment with our theoretical observations already illustrated earlier in Section 1 (see Figure 1c). Indeed, as partial orders, SHB/WCP can only detect races between conflicting accesses that are successive in the input trace. On the other hand, sync-reversal free races may be interleaved with arbitrarily many conflicting, non-racy accesses, and our complete algorithm SRFree is guaranteed to detect them. Overall, all our experimental observations suggest that sync-reversal freeness is an elegant notion:

it finely characterizes almost all races that are efficiently detectable, while it captures several races that are beyond the standard happens-before relation.

7 RELATED WORK

Happens-before has been the standard approach to sound dynamic race detection. The respective partial order is computable in linear time [Mattern 1988] and has formed the basis of many race detectors in the literature [Bond et al. 2010; Christiaens and Bosschere 2001; Flanagan and Freund 2009; Pozniansky and Schuster 2003; Schonberg 1989]. As we have seen in this work, happens-before only characterizes a small subset of predictable races, and several works have tried to increase coverage at only a small increase of computational resources [Genç et al. 2019; Kini et al. 2017; Pavlogiannis 2019; Roemer et al. 2018, 2020; Smaragdakis et al. 2012].

Another direction to dynamic race prediction is symbolic techniques that typically rely on SAT/SMT encodings of the condition of a correct reordering, and dispatch such encodings to the respective solver [Huang et al. 2014; Liu et al. 2016; Said et al. 2011; Wang et al. 2009]. The encodings are typically sound and complete in theory, but the solution takes exponential time. In practice, windowing techniques are used to fragment the trace into chunks, and analyze each chunk independently. This introduces incompleteness, as races between events of different chunks are naturally missed. Dynamic techniques have also been used for predicting other types of errors, such as deadlocks, atomicity violations and synchronization errors [Chen et al. 2008; Farzan and Madhusudan 2009; Farzan et al. 2009; Flanagan et al. 2008; Kalhauge and Palsberg 2018; Mathur and Viswanathan 2020; Sen et al. 2005; Sorrentino et al. 2010].

Another common approach to race prediction is via lockset-based methods. At a high level, a lockset is a set of locks that guards all accesses to a given memory location. Such techniques report races when they discover a write access to a memory location which has not been consistently protected by a common lock (i.e., whose lockset is empty). They were introduced in [Dinning and Schonberg 1991] and equipped the Eraser [Savage et al. 1997]. The lockset criterion is a complete one, as all races are required to meet it. However, it is unsound, and various works attempt to reduce false positives by enhancements such as random testing [Sen 2008] and static analysis [Choi et al. 2002; von Praun and Gross 2001].

8 CONCLUSION

In this work, we have introduced the new notion of synchronization-reversal free races. Conceptually, this is a completion of the principle behind happens-before races, namely that such races can be witnessed without reversing the order in which synchronization operations are observed. We have shown that sync-reversal freeness strictly subsumes happens-before, and can detect races that are far apart in the input trace. We have developed a new algorithm SRFree that is sound and complete for sync-reversal free races, and has nearly linear time and space complexity. Moreover, we have shown that linear space complexity is necessary for detecting sync-reversal free races, and thus our algorithm is optimal. In addition, we have shown that relaxing our definition even slightly, i.e., by allowing a single synchronization reversal suffices to make the problem W[1]-hard, i.e., as hard as in the general case. Finally, we have performed an extensive experimental evaluation of SRFree on a standard set of benchmarks. Our experiments show that sync-reversal freeness is an elegant notion that characterizes almost all races that are efficiently detectable, while it captures several races that are beyond the standard happens-before. Given the demonstrated relevance of this new notion, we identify as important future work the development of more efficient race detectors for sync-reversal free races, in a similar manner that happens-before race detectors have been refined over the years.

REFERENCES

Joaquín Aguado, Michael Mendler, Marc Pouzet, Partha Roop, and Reinhard von Hanxleden. 2018. Deterministic Concurrency: A Clock-Synchronised Shared Memory Approach. In *Programming Languages and Systems*, Amal Ahmed (Ed.). Springer International Publishing, Cham, 86–113.

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 *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA) *(OOPSLA '06)*. ACM, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

Robert L. Bocchino, Vikram S. Adve, Sarita V. Adve, and Marc Snir. 2009. Parallel Programming Must Be Deterministic by Default. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism* (Berkeley, California) *(HotPar'09)*. USENIX Association, USA, 4.

Hans-J. Boehm. 2011. How to Miscompile Programs with "Benign" Data Races. In *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism* (Berkeley, CA) *(HotPar'11)*. USENIX Association, USA, 3.

Hans-J. Boehm. 2012. Position Paper: Nondeterminism is Unavoidable, but Data Races Are Pure Evil. In *Proceedings of the 2012 ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability* (Tucson, Arizona, USA) *(RACES '12)*. Association for Computing Machinery, New York, NY, USA, 9–14. https://doi.org/10.1145/2414729.2414732

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* (Toronto, Ontario, Canada) *(PLDI '10)*. ACM, New York, NY, USA, 255–268. https://doi.org/10.1145/1806596.1806626

Feng Chen, Traian Florin Şerbănuţă, and Grigore Roşu. 2008. jPredictor: a predictive runtime analysis tool for Java. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering* (Leipzig, Germany). ACM, New York, NY, USA, 221–230.

Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. 2002. Efficient and Precise Datarace Detection for Multithreaded Object-oriented Programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany) *(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.

Heming Cui, Rui Gu, Cheng Liu, Tianyu Chen, and Junfeng Yang. 2015. Paxos Made Transparent. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 105–120. https://doi.org/10.1145/2815400.2815427

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* (Santa Cruz, California, USA) *(PADD '91)*. ACM, New York, NY, USA, 85–96. https://doi.org/10.1145/122759.122767

Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. 2005. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Software Engineering: An International Journal* 10, 4 (2005), 405–435.

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* (San Diego, California, USA) *(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–.

Azadeh Farzan and P. Madhusudan. 2009. The Complexity of Predicting Atomicity Violations. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,* (York, UK) *(TACAS '09)*. Springer-Verlag, Berlin, Heidelberg, 155–169. https://doi.org/10.1007/978-3-642-00768-2_14

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* (Grenoble, France) *(CAV '09)*. Springer-Verlag, Berlin, Heidelberg, 248–262. https://doi.org/10.1007/978-3-642-02658-4_21

Colin Fidge. 1991. Logical Time in Distributed Computing Systems. *Computer* 24, 8 (Aug. 1991), 28–33. https://doi.org/10.1109/2.84874

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* (Dublin, Ireland) *(PLDI '09)*. ACM, New York, NY, USA, 121–133. https://doi.org/10.1145/1542476.1542490

Cormac Flanagan, Stephen N. Freund, and Jaeheon Yi. 2008. Velodrome: A Sound and Complete Dynamic Atomicity Checker for Multithreaded Programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. ACM, New York, NY, USA, 293–303. https://doi.org/10.1145/1375581.1375618

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

Phillip B. Gibbons and Ephraim Korach. 1997. Testing Shared Memories. *SIAM J. Comput.* 26, 4 (Aug. 1997), 1208–1244. https://doi.org/10.1137/S0097539794279614

Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 2019. A True Positives Theorem for a Static Race Detector. *Proc. ACM Program. Lang.* 3, POPL, Article 57 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290370

Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463–492. https://doi.org/10.1145/78969.78972

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* (Edinburgh, United Kingdom) *(PLDI '14)*. ACM, New York, NY, USA, 337–348. https://doi.org/10.1145/2594291.2594315

Jeff Huang and Arun K. Rajagopalan. 2016. Precise and Maximal Race Detection from Incomplete Traces. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. ACM, New York, NY, USA, 462–476. https://doi.org/10.1145/2983990.2984024

Christian Gram Kalhauge and Jens Palsberg. 2018. Sound Deadlock Prediction. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 146 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276516

Baris Kasikci, Cristian Zamfir, and George Candea. 2013. RaceMob: Crowdsourced Data Race Detection. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. ACM, New York, NY, USA, 406–422.

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* (Barcelona, Spain) *(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.

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* (Saarbr&#252;cken, Germany) *(ISSTA 2016)*. ACM, New York, NY, USA, 59–69.

Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (Seattle, WA, USA) *(ASPLOS XIII)*. ACM, New York, NY, USA, 329–339. https://doi.org/10.1145/1346281.1346323

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

Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. 2020. The Complexity of Dynamic Data Race Prediction. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science* (Saarbrücken, Germany) *(LICS '20)*. Association for Computing Machinery, New York, NY, USA, 713–727. https://doi.org/10.1145/3373718.3394783

Umang Mathur and Mahesh Viswanathan. 2020. Atomicity Checking in Linear Time Using Vector Clocks. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 183–199. https://doi.org/10.1145/3373376.3378475

Friedemann Mattern. 1988. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms*. North-Holland, 215–226.

Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. 2007. Automatically Classifying Benign and Harmful Data Races Using Replay Analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 22–31. https://doi.org/10.1145/1250734.1250738

Robert O'Callahan and Jong-Deok Choi. 2003. Hybrid Dynamic Data Race Detection. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, California, USA) *(PPoPP '03)*. ACM, New York, NY, USA, 167–178. https://doi.org/10.1145/781498.781528

Andreas Pavlogiannis. 2019. Fast, Sound, and Effectively Complete Dynamic Race Prediction. *Proc. ACM Program. Lang.* 4, POPL, Article 17 (Dec. 2019), 29 pages. https://doi.org/10.1145/3371085

Eli Pozniansky and Assaf Schuster. 2003. Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, California, USA) *(PPoPP '03)*. ACM, New York, NY, USA, 179–190. https://doi.org/10.1145/781498.781529

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

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

Jake Roemer, Kaan Genç, and Michael D. Bond. 2020. SmartTrack: Efficient Predictive Race Detection. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 747–762. https://doi.org/10.1145/3385412.3385993

Grigore Rosu. 2018. *RV-Predict, Runtime Verification.* Accessed: 2018-04-01.

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* (Pasadena, CA) *(NFM'11)*. Springer-Verlag, Berlin, Heidelberg, 313–327.

Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A Dynamic Data Race Detector for Multi-threaded Programs. *SIGOPS Oper. Syst. Rev.* 31, 5 (Oct. 1997), 27–37.

D. Schonberg. 1989. On-the-fly Detection of Access Anomalies. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation* (Portland, Oregon, USA) *(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* (Tucson, AZ, USA) *(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 *Proceedings of the 7th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems* (Athens, Greece) *(FMOODS'05)*. Springer-Verlag, Berlin, Heidelberg, 211–226.

Traian Florin Şerbănuţă, Feng Chen, and Grigore Roşu. 2012. Maximal causal models for sequentially consistent systems. In *International Conference on Runtime Verification*. Springer, 136–150.

Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: Data Race Detection in Practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications* (New York, New York, USA) *(WBIA '09)*. ACM, New York, NY, USA, 62–71.

Ilya Sergey. 2019. *What Does It Mean for a Program Analysis to Be Sound?* Accessed: 2019-08-07.

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* (Philadelphia, PA, USA) *(POPL '12)*. ACM, New York, NY, USA, 387–400. https://doi.org/10.1145/2103656.2103702

Lorna A Smith and J Mark Bull. 2001. A multithreaded java grande benchmark suite. In *Proceedings of the third workshop on Java for high performance computing*.

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* (Santa Fe, New Mexico, USA) *(FSE '10)*. ACM, New York, NY, USA, 37–46. https://doi.org/10.1145/1882291.1882300

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* (Tampa Bay, FL, USA) *(OOPSLA '01)*. ACM, New York, NY, USA, 70–82. https://doi.org/10.1145/504282.504288

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* (Eindhoven, The Netherlands) *(FM '09)*. Springer-Verlag, Berlin, Heidelberg, 256–272.

Andrew Chi-Chih Yao. 1979. Some Complexity Questions Related to Distributive Computing(Preliminary Report). In *Proceedings of the Eleventh Annual ACM Symposium on Theory of Computing* (Atlanta, Georgia, USA) *(STOC '79)*. Association for Computing Machinery, New York, NY, USA, 209–213. https://doi.org/10.1145/800135.804414

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.

Qi Zhao, Zhengyi Qiu, and Guoliang Jin. 2019. Semantics-Aware Scheduling Policies for Synchronization Determinism. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (Washington, District of Columbia) *(PPoPP '19)*. Association for Computing Machinery, New York, NY, USA, 242–256. https://doi.org/10.1145/3293883.3295731

M. Zhivich and R. K. Cunningham. 2009. The Real Cost of Software Errors. *IEEE Security and Privacy* 7, 2 (March 2009), 87–90. https://doi.org/10.1109/MSP.2009.56

A PROOFS FROM SECTION 4

Lemma 4.1. *If (e_1, e_2) is a sync-reversal free race of σ, then there is a correct reordering ρ of σ such that both e_1, e_2 are σ-enabled in ρ and $\leq^{\rho}_{\mathsf{tr}} \subseteq \leq^{\sigma}_{\mathsf{tr}}$.*

Proof. Let π be a sync-reversal free correct reordering of σ. Let $E = \mathsf{Events}_\pi$. Observe that E is $(\leq^{\sigma}_{\mathsf{TO}}, \mathsf{lw}_\sigma)$-closed because π is a correct reordering of σ. We note a few observations about E. First, $E \subseteq \mathsf{Events}_\sigma$. Second, E is downward closed with respect to $\leq^{\sigma}_{\mathsf{TO}}$. Third, for every read event $e \in E$, we have $\mathsf{lw}_\sigma(e) \in E$. Fourth, for every lock ℓ, there is at most one acquire a of ℓ such that $a \in E$ but $\mathsf{match}_\sigma(a) \notin E$.

Now, consider the sequence ρ obtained by linearizing E according to the total order of $\leq^{\sigma}_{\mathsf{tr}}$ (i.e., ρ is the projection of σ onto the set E). We first argue that ρ is a well-formed trace. This follows because for every lock ℓ, there is at most one unmatched acquire event e in E of lock ℓ, and all the other acquires are earlier than e in σ (and hence in ρ). Next, ρ respects $\leq^{\sigma}_{\mathsf{TO}}$ because E is downward closed with respect to $\leq^{\sigma}_{\mathsf{TO}}$ and respects $\leq^{\sigma}_{\mathsf{tr}}$. For the same reason, for every read event $e \in E$, we have $\mathsf{lw}_\rho(e) = \mathsf{lw}_\sigma(e)$. Thus, ρ is a correct reordering of σ. Further, ρ is trivially a sync-reversal free correct reordering of σ. □

Lemma 4.2. *(e_1, e_2) is a sync-reversal free race of σ iff $\{e_1, e_2\} \cap \mathsf{SRFIdeal}_\sigma(e_1, e_2) = \emptyset$.*

Proof. ($\Rightarrow$) Assume $\mathsf{SRFIdeal}_\sigma(e_1, e_2) \cap \{e_1, e_2\} \neq \emptyset$. Then we must have $e_1 \in \mathsf{SRFIdeal}_\sigma(e_1, e_2)$ and in particular $e_1 \in \mathsf{SRFClosure}_\sigma(\mathsf{prev}_\sigma(e_2))$ (and thus $e = \mathsf{prev}_\sigma(e_2) \neq \bot$). Then, either $e_1 \leq^{\sigma}_{\mathsf{TO}} e$ or there is a release event e' such that $e_1 \leq^{\sigma}_{\mathsf{TO}} e'$ and $e' \in \mathsf{SRFClosure}_\sigma(e)$. In either case, e_1 cannot be enabled in a sync-reversal free correct reordering containing e.

($\Leftarrow$) Let $E = \mathsf{SRFIdeal}_\sigma(e_1, e_2)$ and let ρ be the sequence obtained by linearizing E as per $\leq^{\sigma}_{\mathsf{tr}}$. Observe that ρ is well formed, respects $\leq^{\sigma}_{\mathsf{TO}}$ and lw_σ and thus is a correct reordering of σ. Further, the order of all critical sections is the same. Also, e_1 and e_2 are both enabled in ρ. □

Lemma 4.3. *Let σ be a trace and let $e_1, e_2, e'_1, e'_2 \in \mathsf{Events}_\sigma$ such that $e_1 <^{\sigma}_{\mathsf{TO}} e'_1$ and $e_2 <^{\sigma}_{\mathsf{TO}} e'_2$. Then, $\mathsf{SRFIdeal}_\sigma(e_1, e_2) \subseteq \mathsf{SRFIdeal}_\sigma(e'_1, e'_2)$.*

Proof. Proof follows from the following observations. $e_1 \leq^{\sigma}_{\mathsf{TO}} \mathsf{prev}_\sigma(e'_1)$, $e_2 \leq^{\sigma}_{\mathsf{TO}} \mathsf{prev}_\sigma(e'_2)$ and $\mathsf{SRFClosure}_\sigma(S)$ is downward closed with respect to $(\leq^{\sigma}_{\mathsf{TO}})$ and sync-reversal free-closed. □

Lemma 4.10. *The following assertions hold.*

(1) If $s \in \mathcal{L}_n$, then σ has no predictable race.
(2) If $s \notin \mathcal{L}_n$, then σ has a single predictable race, which is a sync-reversal free race.

Proof. We prove each item separately.

1. $s \in \mathcal{L}'_n$. First, notice that by construction, for every $i \in [n]$, the pair (e^1_i, e^2_i) is not a predictable race of σ, as we have $u[i] = v[i]$ and thus either both events are read events or both are write events and thus each is protected by the lock c. It remains to argue that $e(^1_i, e^2_j)$ is not a predictable race for any $i, j \in [n]$ with $i \neq j$. It suffices to show that $\mathsf{locksHeld}_\sigma(e^1_i)) \cap \mathsf{locksHeld}_\sigma(e^2_j) \neq \emptyset$. We split cases based on the relation between i and j.

$i < j$. By construction, we have $\mathsf{locksHeld}_\sigma(e^1_j) \cap B \not\subseteq \mathsf{locksHeld}_\sigma(e^1_i) \cap B$. Moreover, we have $(e^2_j) \cap B = B \setminus \mathsf{locksHeld}_\sigma(e^1_j)$. Thus, we have $\mathsf{locksHeld}_\sigma(e^1_i)) \cap \mathsf{locksHeld}_\sigma(e^2_j) \cap B \neq \emptyset$, as desired.

$j < i$. By construction, we have $\mathsf{locksHeld}_\sigma(e_i^1) \cap A \not\subseteq \mathsf{locksHeld}_\sigma(e_j^1) \cap A$. Moreover, we have $(e_i^2) \cap A = A \setminus \mathsf{locksHeld}_\sigma(e_i^1)$. Thus, we have $\mathsf{locksHeld}_\sigma(e_i^1)) \cap \mathsf{locksHeld}_\sigma(e_j^2) \cap A \neq \emptyset$, as desired.

2. $s \notin \mathcal{L}'_n$. First, notice that by construction, one of e_i^1 and e_i^2 is a read event and the other is a write event. Hence, at least one of them is not surrounded by lock c. Finally, by construction we have $(e_i^2) \cap (A \cup B) = A \cup B \setminus \mathsf{locksHeld}_\sigma(e_j^1)$, and thus $\mathsf{locksHeld}_\sigma(e_i^1)) \cap \mathsf{locksHeld}_\sigma(e_j^2) = \emptyset$.

The desired result follows. □

The set-equality problem. Finally, we turn our attention to Theorem 3.3. Our proof uses the set-equality problem, i.e., given two bit-sets $u, v \in \{0,1\}^n$, the task is to decide whether $u = v$. The problem has a $\Omega(n)$ lower-bound for communication complexity [Yao 1979], i.e., if u and v is given separately to Alice and Bob, respectively, the two parties need to exchange $\Omega(n)$ bits of information in order to decide whether $u = v$.

Proof of Theorem 3.3. Consider the language $\mathcal{L}'_n = \{u\#^n v : u, v \in \{0,1\}^n \text{ and } u = v\}$, and any Turing Machine M that decides $\mathcal{L}'_n$ using $T_M(n)$ time and $S_M(n)$ space. Let $K_M(n)$ be an upper-bound on the number of "passes" that M makes over the sequence $\#^n$ as it decides membership in $\mathcal{L}'_n$. In each pass, M "communicates" at most $S_M(n)$ bits of information. Since the set-equality problem has communication complexity $\Omega(n)$ [Yao 1979], we have $K_M(n) \cdot S_M(n) = \Omega(n)$, i.e., M makes $\Omega(n/S_M(n))$ passes. Since each pass has to traverse n symbols $\#$, each pass costs time $\Omega(n)$. Hence, the total time is $T_M(n) = \Omega(n \cdot K_M(n))$, and thus $T_M(n) \cdot S_M(n) \geq n^2$.

We now turn our attention to sync-reversal free race prediction. Let $m = n/\log n$ Using essentially the same reduction as above, we reduce the membership problem for $\mathcal{L}'_m$ to the sync-reversal free race prediction problem on a trace σ with 2 threads, n events and $O(\log n)$ locks. Lemma 4.10 guarantees that σ has a predictable race iff $u \neq v$, and if so, then it is a sync-reversal free race. Hence, any algorithm that solves sync-reversal free race prediction on σ in time $T(n)$ and space $S(n)$ must satisfy that $T(n) \cdot S(n) = \Omega(m^2) = \Omega(n^2/\log^2 n)$. The desired result follows. □

B PROOFS FROM SECTION 5

In this section we present the proofs of Section 5, i.e., Lemma 5.1, Lemma 5.2 and Lemma B.1.

Lemma 5.1. *P' is a partial order.*

Proof. Assume towards contradiction otherwise, hence P' has a cycle $e_1 <_{P'} e_2 <_{P'} \cdots <_{P'} e_1$. Let $C = \{e_i : e_i \in X' \setminus X\}$, and note that $C \neq \emptyset$ as $P \sqsubseteq P'|X$. Observe that C cannot contain any event of the distinguished triplet λ, as every event of λ is either minimal or maximal in P'. On the other hand, C cannot contain any event of any set Y_i^j, as every such event only has predecessors that are either in Y_i^j or in λ, and clearly $P'|Y_i^j$ is acyclic. Finally, C cannot contain any event of any set X_i^j, as every such event only has successors that are either in X_i^j or in λ, and clearly $P'|X_i^j$ is acyclic. Hence $C = \emptyset$, a contradiction.

The desired result follows. □

Lemma 5.2. *The trace σ realizes $\mathcal{P}'$.*

Proof. We argue that Q is indeed a partial order, from which follows that σ is a witness of the realizability of $\mathcal{P}'$. Observe that for every interfering write event $\mathtt{w}'$ of each triplet, every predecessor $\mathtt{w}''$ of $\mathtt{w}'$ is also an interfering write event of some triplet. On the other hand, every new successor of $\mathtt{w}'$ in Q is not an interfering write event of any triplet. Hence, since P' is acyclic, Q is also acyclic.

The desired result follows. □

Lemma B.1. *Reverse rf-poset realizability is* W[1]*-hard parameterized by the number of threads.*

Proof. Here we argue that $\mathcal{P}$ has a witness σ_1 iff $\mathcal{P}'$ has a witness σ_2.
($\Rightarrow$). Consider the witness σ_1 for $\mathcal{P}$, and we show how to obtain the witness σ_2 for $\mathcal{P}'$. We construct a partial order Q over X' such that (i) $Q \sqsubseteq P'$, (ii) for every triplet $(\mathtt{w}, \mathtt{r}, \mathtt{w}') \in \mathsf{Triplets}(\mathcal{S}) \setminus \mathsf{Triplets}(\mathcal{P})$, we have $\mathtt{r} <_Q \mathtt{w}'$, and (iii) $Q|X = \sigma_1$ (i.e., Q totally orders the events of X according to σ_1). Afterwards, we construct σ_2 by linearizing Q arbitrarily. Note that (ii) makes $Q|X = P$, hence the linearization in (iii) is well defined.
($\Leftarrow$). Consider the witness σ_2 for $\mathcal{P}'$, and we show how to obtain the witness σ_1 for $\mathcal{P}$. We construct σ_1 simply as $\sigma_1 = \sigma_2|X$. To see that σ_1 is a linearization of (X, P), consider any two events $e, e' \in X$ such that $e <_P e'$. Consider the triplet $(\mathtt{w}_{e,e'}, \mathtt{r}_{e,e'}, \mathtt{w}'_{e,e'})$ of $\mathcal{P}'$, and we have $\mathtt{w}_{e,e'} <_{\sigma_2} \mathtt{w}'_{e,e'}$, thus $\mathtt{r}_{e,e'} <_{\sigma_2} \mathtt{w}'_{e,e'}$. This leads to $e <_{\sigma_2} e'$, and thus $e <_{\sigma_1} e'$, as desired.
The desired result follows. □

Theorem 3.4. *Dynamic race prediction on traces with a single lock and two critical sections is* W[1]*-hard parameterized by the number of threads.*

Proof. We argue that $\mathcal{P}$ has a witness σ_1 iff $(\mathtt{w}(y), \mathtt{r}(y))$ is a predictable data race of σ', witnessed by a trace σ_2.
($\Rightarrow$). Consider the witness σ_1, hence $\overline{\mathtt{r}} <_{\sigma_1} \overline{\mathtt{w}}'$ for the distinguished triplet $\lambda = (\overline{\mathtt{w}}, \overline{\mathtt{r}}, \overline{\mathtt{w}}')$. The witness trace σ_2 is constructed as follows.

(1) We insert in σ_1 all events that where inserted in σ to produce σ', in the same order.
(2) We remove from σ_1 the events $\{\mathtt{rel}_1, \mathtt{w}(y), \mathtt{r}(y)\}$.

It follows easily that σ_2 is a correct reordering of σ', in which $\mathtt{w}(y)$ and $\mathtt{r}(y)$ are enabled, hence $(\mathtt{w}(y), \mathtt{r}(y))$ is a predictable data race of σ'.
($\Leftarrow$). Consider the witness σ_2, and it is straightforward that $\mathsf{Events}_{\sigma_2} = \mathsf{Events}_{\sigma'} \setminus \{\mathtt{rel}_2, \overline{\mathtt{w}}, \overline{\mathtt{r}}\}$. Hence $\mathtt{rel}_1 <_{\sigma_2} \mathtt{acq}_2$ and thus $\overline{\mathtt{r}} < \overline{\mathtt{w}}'$ for the distinguished triplet $\lambda = (\overline{\mathtt{w}}, \overline{\mathtt{r}}, \overline{\mathtt{w}}')$. The witness σ_1 is constructed as $\sigma_1 = \sigma_2|X$, i.e., by removing from σ_2 all the events that we inserted when we constructed σ' from σ. It follows easily that σ_1 is a linearization of the rf-poset (X, P, RF) and $\mathtt{r} <_{\sigma_1} \mathtt{w}'$.
The desired result follows. □

C COMPARISON WITH OTHER RACE PREDICTION ALGORITHMS

Here, we discuss recent advances in data race prediction and characterize their prediction power with respect to sync-reversal free races. We focus our attention to *sound* race prediction techniques. Specifically, we compare sync-reversal free race detection with race prediction based on the *happens-before* (HB) partial order, the *schedulable-happens-before* (SHB) partial order [Mathur et al. 2018], the *causally precedes partial order* (CP) [Smaragdakis et al. 2012], the *weak causally precedes partial order* (WCP) [Kini et al. 2017], and the *does not commute partial order* (DC) [Roemer et al. 2018]. The recently introduced partial order *strong-dependently-precedes* (SDP) [Genç et al. 2019], while claimed to be sound in that paper, is actually unsound. In Appendix D, we show a counter-example to the soundness theorem of SDP.

C.1 Comparison with HB and SHB

The happens-before ($\leq_{\mathsf{HB}}$) order is a classic partial order employed in popular race detectors like ThreadSanitizer [Serebryany and Iskhodzhanov 2009] and FastTrack [Flanagan and Freund 2009]. The main idea behind race detectors based on $\leq_{\mathsf{HB}}$ is to determine the existence of conflicting pairs of events that are unordered by $\leq_{\mathsf{HB}}$, defined as follows.

Definition 3 (Happens-Before)**.** The happens-before order defined given a trace σ is the smallest partial order $\leq^{\sigma}_{\mathsf{HB}}$ on Events_{σ} such that $\leq^{\sigma}_{\mathsf{TO}} \subseteq \leq^{\sigma}_{\mathsf{HB}}$ and for every lock $\ell \in \mathsf{Locks}_{\sigma}$ and for every two events $e_1 \leq^{\sigma}_{\mathsf{tr}} e_2$ such that $e_1 \in \mathsf{Releases}_{\sigma}(\ell)$ and $e_2 \in \mathsf{Acquires}_{\sigma}(\ell)$, we have $e_1 \leq^{\sigma}_{\mathsf{HB}} e_2$.

A pair (e_1, e_2) of conflicting events of σ is said to be an $\leq_{\mathsf{HB}}$-race if $e_1 \parallel^{\sigma}_{\mathsf{HB}} e_2$. The soundness guarantee of $\leq_{\mathsf{HB}}$ states that if σ has an $\leq_{\mathsf{HB}}$-race, then σ also has a predictable data race. The partial order $\leq_{\mathsf{HB}}$ is known to miss the existence of predictable data races, or, in other words, it is incomplete. Further, as noted in as noted in [Mathur et al. 2018], while $\leq_{\mathsf{HB}}$ is sound for checking the existence of a data race, the soundness guarantee only applies to the first such race identified, beyond which, conflicting pairs of events unordered by $\leq_{\mathsf{HB}}$ may not be predictable races. The partial order $\leq_{\mathsf{SHB}}$ [Mathur et al. 2018], defined below, overcomes this problem.

Definition 4 (Schedulable-Happens-Before [Mathur et al. 2018])**.** The schedulable-happens-before order defined given a trace σ is the smallest partial order $\leq^{\sigma}_{\mathsf{SHB}}$ on Events_{σ} such that $\leq^{\sigma}_{\mathsf{HB}} \subseteq \leq^{\sigma}_{\mathsf{SHB}}$ and for every variable $x \in \mathsf{Locks}_{\sigma}$ and for every read event $e_1 \leq^{\sigma}_{\mathsf{tr}} e_2$ such that $e_1 \in \mathsf{Releases}_{\sigma}(\ell)$ and $e_2 \in \mathsf{Acquires}_{\sigma}(\ell)$, we have $e_1 \leq^{\sigma}_{\mathsf{HB}} e_2$.

A pair of conflicting events (e_1, e_2) in σ is an $\leq_{\mathsf{SHB}}$-race if either $\mathsf{prev}_{\sigma}(e_2) = \bot$ or $e_1 \parallel^{\sigma}_{\mathsf{SHB}} \mathsf{prev}_{\sigma}(e_2)$. The soundness theorem for SHB states that every $\leq_{\mathsf{SHB}}$-race of a trace σ is also a predictable data race of σ [Mathur et al. 2018], and further the first race identified by $\leq_{\mathsf{HB}}$ (for which soundness of $\leq_{\mathsf{HB}}$ holds) is also reported by $\leq_{\mathsf{SHB}}$.
We next make the following observation. The proof follows directly from the soundness proof of SHB [Mathur et al. 2018].

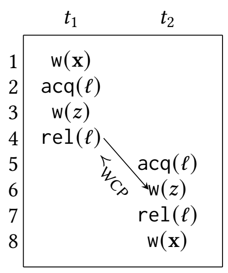
Lemma C.1. *For a trace σ and a conflicting pair of events (e_1, e_2) of σ, if (e_1, e_2) is an SHB-race, then (e_1, e_2) is a sync-reversal free predictable race of σ.*

Proof. Let σ be a trace and let (e_1, e_2) be an SHB-race of σ such that $e_1 \leq^{\sigma}_{\mathsf{tr}} e_2$. Let $S_i = \{e \in \mathsf{Events}_{\sigma} \mid e <^{\sigma}_{\mathsf{SHB}} e_i\}$ and let $S = S_1 \cup S_2$. We observe that $\{e_1, e_2\} \cap S = \emptyset$. Next, consider the sequence ρ' obtained by linearizing the events in S as per $\leq^{\sigma}_{\mathsf{tr}}$ and let $\rho = \rho' \cdot e_1 \cdot e_2$. We remark that S is downward closed with respect to $\leq^{\sigma}_{\mathsf{SHB}}$ and thus ρ is a correct reordering of σ. Further, since $\leq^{\rho}_{\mathsf{tr}} \subseteq \leq^{\sigma}_{\mathsf{tr}}$, ρ is also a sync-reversal free correct reordering of σ. □

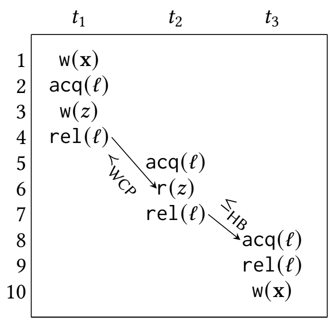
Example 6. Consider the trace σ_1 in Figure 1a. Both $\leq^{\sigma_1}_{\mathsf{HB}}$ and $\leq^{\sigma_1}_{\mathsf{SHB}}$ order all events in σ_1, and thus there is no HB or SHB race in σ. Nevertheless, the correct reordering σ_1^{CR} is a sync-reversal free correct reordering that witnesses the race (e_1, e_6). Notice that the earlier critical section in σ_1 on lock ℓ (performed in thread t_1) is not present in the correct reordering.

Based on Lemma C.1 and Example 6, we have the following.

Observation 1. The prediction power of sync-reversal free race prediction is strictly better than SHB race prediction.



(a) σ_7 with no $\leq_{\mathsf{WCP}}$-race.



(b) Trace σ_8 with no $\leq_{\mathsf{WCP}}$-race.

Fig. 7. Traces σ_7 and σ_8 have sync-reversal free-races. But $\leq_{\mathsf{WCP}}$ does not report any races.

C.2 Comparison with CP and WCP

The partial orders *causally precedes* (denoted $\leq_{\mathsf{CP}}$) [Smaragdakis et al. 2012] and *weak causally precedes* (denoted $\leq_{\mathsf{WCP}}$) [Kini et al. 2017] are two recent partial orders, proposed for data race prediction. Both partial orders are sound and can report races missed by $\leq_{\mathsf{HB}}$ (and $\leq_{\mathsf{SHB}}$). More precisely, $\leq_{\mathsf{CP}}$ can predict races on strictly more traces than $\leq_{\mathsf{HB}}$, and $\leq_{\mathsf{WCP}}$ can, in turn, predict strictly more races than $\leq_{\mathsf{CP}}$. Further, $\leq_{\mathsf{WCP}}$ has a more efficient race detection algorithm than $\leq_{\mathsf{CP}}$. We define $\leq_{\mathsf{WCP}}$ below; the precise definition of $\leq_{\mathsf{CP}}$ is similar to that of $\leq_{\mathsf{WCP}}$ and not important for our discussion here. Here, we say that two acquire events a_1 and a_2 are conflicting if there is a common lock ℓ such that $\mathsf{op}(a_1) = \mathsf{op}(a_2) = \mathtt{acq}(\ell)$.

Definition 5 (Weak Causal Precedence). For a trace σ, $\leq^{\sigma}_{\mathsf{WCP}} = \prec^{\sigma}_{\mathsf{WCP}} \cup \leq^{\sigma}_{\mathsf{TO}}$, where $\prec^{\sigma}_{\mathsf{WCP}}$ is the smallest transitive order such that the following hold:

(a) For two conflicting acquire events a_1, a_2, if there are events $e_1 \in \mathsf{CS}_\sigma(a_1)$ and $e_2 \in \mathsf{CS}_\sigma(a_2)$ such that $e_1 \asymp e_2$, then $\mathsf{match}_\sigma(a_1) \prec^{\sigma}_{\mathsf{WCP}} e_2$.
(b) For two conflicting acquire a_1, a_2 events, if $a_1 \prec^{\sigma}_{\mathsf{WCP}} a_2$, then $\mathsf{match}_\sigma(a_1) \prec^{\sigma}_{\mathsf{WCP}} \mathsf{match}_\sigma(a_2)$.
(c) For any three events e_1, e_2, e_3, if either $e_1 \prec^{\sigma}_{\mathsf{WCP}} e_2 \leq^{\sigma}_{\mathsf{HB}} e_3$, or $e_1 \leq^{\sigma}_{\mathsf{HB}} e_2 \prec^{\sigma}_{\mathsf{WCP}} e_3$, then $e_1 \prec^{\sigma}_{\mathsf{WCP}} e_3$.

The partial order $\leq_{\mathsf{CP}}$ is defined in a similar manner, except that it orders the second conflicting acquire event a_2 in rules a and b in Definition 5. For a trace σ, a $\leq_{\mathsf{WCP}}$-race (resp. $\leq_{\mathsf{CP}}$-race) is a pair of conflicting events (e_1, e_2) in σ unordered by $\leq^{\sigma}_{\mathsf{WCP}}$ (resp. $\leq^{\sigma}_{\mathsf{CP}}$).

The soundness guarantee of $\leq_{\mathsf{WCP}}$ (and $\leq_{\mathsf{CP}}$) is that of *weak soundness*—if a trace σ has a $\leq_{\mathsf{WCP}}$-race (or $\leq_{\mathsf{CP}}$-race), then σ has a predictable data race or a predictable deadlock[5]. We remark that, as with $\leq_{\mathsf{HB}}$, not every $\leq_{\mathsf{WCP}}$-race is a predictable race, and the weak soundness guarantee applies only to the first $\leq_{\mathsf{WCP}}$-race identified.

Let us now consider how $\leq_{\mathsf{WCP}}$-race prediction compares with sync-reversal free race prediction.

Example 7. Consider trace σ_7 in Figure 7a. Here, we have $e_4 \prec^{\sigma_7}_{\mathsf{WCP}} e_6$ due to rule a. Together with composition with $\leq^{\sigma_7}_{\mathsf{HB}}$ (rule c), we have that $e_1 \leq^{\sigma_7}_{\mathsf{WCP}} e_8$ and thus there is no $\leq_{\mathsf{WCP}}$-race in σ_7. However, the trace $\sigma_7^{\mathsf{CR}} = e_5{\cdot}e_6{\cdot}e_7{\cdot}e_1{\cdot}e_8$ is a sync-reversal free correct reordering of σ_7 that exposes the predictable race (e_1, e_8).

We remark that $\leq_{\mathsf{WCP}}$ misses the race in Example 7 because of the ordering $e_4 \prec^{\sigma_7}_{\mathsf{WCP}} e_6$, which is a spurious ordering and correct reorderings may not necessarily respect it. We next highlight

[5]A trace σ has a predictable deadlock, if there is a correct reordering of σ that witnesses a deadlock.

	t_1	t_2
1	acq(ℓ)	
2	w($\mathbf{x}$)	
3	rel(ℓ)	
4		acq(ℓ)
5		rel(ℓ)
6		w($\mathbf{x}$)

(a) Trace σ_9 with predictable race reported by $\leq_{\mathsf{WCP}}$

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

(b) Trace σ_{10} with predictable race missed by $\leq_{\mathsf{WCP}}$

Fig. 8. Traces with no sync-reversal free races. $\leq_{\mathsf{WCP}}$ predicts race in σ_9 but misses in σ_{10}

another source of imprecision in $\leq_{\mathsf{WCP}}$ arising due to the $\leq_{\mathsf{HB}}$-composition rule of $\leq_{\mathsf{WCP}}$ (rule c in Definition 5).

Example 8. Consider trace σ_8 in Figure 7b. Here, due to rule a, we have $e_4 <^{\sigma_8}_{\mathsf{WCP}} e_6$. Further, we have $e_1 \leq^{\sigma_8}_{\mathsf{HB}} e_4$ and $e_6 \leq^{\sigma_8}_{\mathsf{HB}} e_{10}$, giving us $e_1 <^{\sigma_8}_{\mathsf{WCP}} e_{10}$ due to rule c As a result, there is no $\leq_{\mathsf{WCP}}$-race in σ_8. However, the pair (e_1, e_{10}) is, in fact, a sync-reversal free race witnessed by the correct reordering $\sigma_8^{\mathsf{CR}} = e_8 \cdot e_9 \cdot e_1 \cdot e_{10}$ that completely drops the critical sections in t_1 and t_3.

Of course, there are predictable races that are neither $\leq_{\mathsf{WCP}}$-races, nor sync-reversal free races.

Example 9. The trace σ_{10} in Figure 8b has a predictable race (e_2, e_7) which is witnessed by the (only) correct reordering $\sigma_{10}^{\mathsf{CR}} = e_4 \cdot e_5 \cdot e_6 \cdot e_1$. Notice that this is not a sync-reversal free correct ordering. Further, $\leq_{\mathsf{WCP}}$ misses this race as well — $e_3 <^{\sigma_{10}}_{\mathsf{WCP}} e_5$ (rule a), giving $e_1 \leq^{\sigma_{10}}_{\mathsf{WCP}} e_7$.

In the next example, we illustrate that $\leq_{\mathsf{WCP}}$ can predict races that are not sync-reversal free races.

Example 10. Consider trace σ_9 in Figure 8a. Here, (e_2, e_6) is a predictable data race witnessed by the (only) correct reordering $\sigma_9^{\mathsf{CR}} = e_4 \cdot e_5 \cdot e_1 \cdot e_2 \cdot e_6$. Observe that σ_9^{CR} is not a sync-reversal free correct reordering of σ_9 and thus σ_9 does not have any sync-reversal free race. At the same time, $e_2 \parallel^{\sigma}_{\mathsf{WCP}} e_6$ and thus this predictable race is identified by $\leq_{\mathsf{WCP}}$.

We summarize our comparison with $\leq_{\mathsf{WCP}}$ as follows.

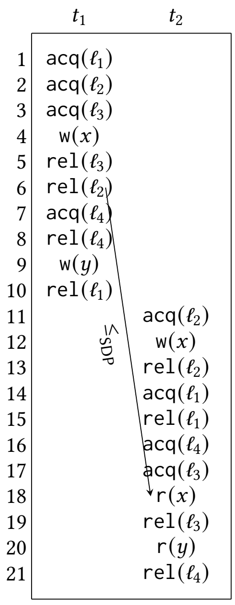
Observation 2. The prediction power of $\leq_{\mathsf{WCP}}$-race prediction and sync-reversal free race prediction are incomparable. Further, $\leq_{\mathsf{WCP}}$ offers a weak soundness guarantee, i.e., a $\leq_{\mathsf{WCP}}$-race may sometimes imply no predictable race but only a predictable deadlock, whereas sync-reversal free race prediction is strongly sound. Finally, $\leq_{\mathsf{WCP}}$ is sound only until the first race, whereas all sync-reversal free-races reported are true races.

C.3 Comparison with DC

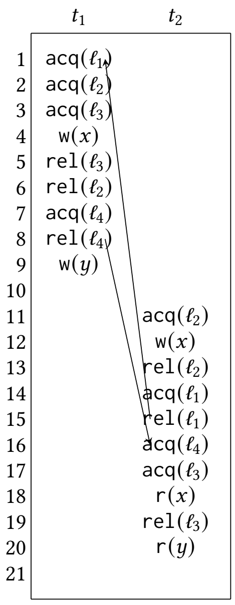
Finally, we briefly outline how sync-reversal free races compare with the methods DC. The DC partial order was introduced in [Roemer et al. 2018] as an unsound weakening to WCP. The difference between the two is that DC does not compose with HB, i.e., it lacks c in the definition of WCP above, and instead only composes with the thread order. Due to its similarity with WCP, DC also misses sync-reversal free races. For example, in the trace of Figure 7a, DC forces the same ordering as WCP, and thus misses the sync-reversal free race (e_1, e_8).

D A NOTE ON THE SOUNDNESS OF SDP

The SDP partial order was recently introduce in [Genç et al. 2019] for dynamic race prediction. [Genç et al. 2019, Theorem p12] states that SDP is sound, i.e., if a trace σ has an SDP-race and σ has a predictable race. In this section we construct a counterexample to soundness.



(a) A trace σ with an SDP-race but no predictable race.



(b) Attempt for a correct reordering of σ with a race on (e_9, e_{20}).

Fig. 9. Counterexample to the soundness of SDP.

Counterexample to SDP soundness. Our counterexample is shown in Figure 9. First, we argue that the trace σ has an SDP-race. Second, we argue that σ has no predictable race.

(1) Observe that e_9 and e_{20} are conflicting and are not protected by the same lock. Hence, it suffices to argue that $e_9 \not\leq^{\sigma}_{\mathsf{SDP}} e_{20}$. Since the two critical sections on ℓ_2 contain the w(x) conflicting events e_4 and e_{12}, SDP will order $e_6 \leq^{\sigma}_{\mathsf{SDP}} e_{18}$, as e_{18} is a r(x) event that is thread-ordered after e_{12}. At this point, SDP will insert no orderings, hence $e_9 \not\leq^{\sigma}_{\mathsf{SDP}} e_{20}$, and (e_9, e_{20}) is an SDP-race.
(2) There are three conflicting event pairs that may constitute a predictable data race, namely, (i) (e_4, e_{12}), (ii) (e_4, e_{17}), and (iii) (e_9, e_{20}). Observe that (e_4, e_{12}) and (e_4, e_{17}) cannot yield a predictable race, as the event pairs are protected by the same locks ℓ_2 and ℓ_3, respectively. For (e_9, e_{20}), consider an attempt for constructing a correct reordering σ^* that witnesses the race, as shown in Figure 9b. Observe that σ^* is missing the rel(ℓ_1) event e_{10} and rel(ℓ_4) event e_{21}. Since σ^* must respect lock semantics, it must satisfy the two orderings shown in Figure 9b. Note, however, that these two orderings necessarily violate the observation of the r(x) event e_{18}. Thus, $\mathsf{RF}_{\sigma} \neq \mathsf{RF}_{\sigma^*}$, and σ^* cannot be a correct reordering of σ.