Optimistic Prediction of Synchronization-Reversal Data Races

Zheng Shi
National University of Singapore
Singapore, Singapore
shizheng@u.nus.edu

Umang Mathur
National University of Singapore
Singapore, Singapore
umathur@comp.nus.edu.sg

Andreas Pavlogiannis
Aarhus University
Aarhus, Denmark
pavlogiannis@cs.au.dk

ABSTRACT

Dynamic data race detection has emerged as a key technique for ensuring reliability of concurrent software in practice. However, dynamic approaches can often miss data races owing to nondeterminism in the thread scheduler. Predictive race detection techniques cater to this shortcoming by inferring alternate executions that may expose data races without re-executing the underlying program. More formally, the dynamic data race prediction problem asks, given a trace σ of an execution of a concurrent program, can σ be correctly reordered to expose a data race? Existing state-of-the art techniques for data race prediction either do not scale to executions arising from real world concurrent software, or only expose a limited class of data races, such as those that can be exposed without reversing the order of synchronization operations.

In general, exposing data races by reasoning about synchronization reversals is an intractable problem. In this work, we identify a class of data races, called Optimistic Sync(hronization)-Reversal races that can be detected in a tractable manner and often include non-trivial data races that cannot be exposed by prior tractable techniques. We also propose a sound algorithm OSR for detecting all optimistic sync-reversal data races in overall quadratic time, and show that the algorithm is optimal by establishing a matching lower bound. Our experiments demonstrate the effectiveness of OSR— on our extensive suite of benchmarks, OSR reports the largest number of data races, and scales well to large execution traces.

ACM Reference Format:
Zheng Shi, Umang Mathur, and Andreas Pavlogiannis. 2024. Optimistic Prediction of Synchronization-Reversal Data Races. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 28 pages. https://doi.org/10.1145/3597503.3639099

1 INTRODUCTION

Concurrency bugs such as data races and deadlocks often escape in-house testing and manifest only in production [21, 61], making the development of reliable concurrent software a challenging task. Automated data race detection has emerged as a first line of defense against undesired behaviors caused by data races, has been actively studied over multiple decades, and is also the subject of this paper. In particular, our focus is on dynamic analyses, which, unlike static techniques, are the preferred class of techniques for detecting data races for industrial scale software applications [61].

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04.
https://doi.org/10.1145/3597503.3639099

	t_1	t_2	t_3	t_4
1	$\mathsf{w}(x)$			
2	$\mathsf{w}(y)$			
3		$\mathsf{acq}(\ell)$		
4		$\mathsf{w}(z_1)$		
5		$\mathsf{r}(y)$		
6		$\mathsf{rel}(\ell)$		
7			$\mathsf{acq}(\ell)$	
8			$\mathsf{w}(z_2)$	
9			$\mathsf{rel}(\ell)$	
10				$\mathsf{r}(z_1)$
11				$\mathsf{r}(z_2)$
12				$\mathsf{w}(x)$

(a) Trace σ_1

	t_1	t_2	t_3	t_4
1			$\mathsf{acq}(\ell)$	
2			$\mathsf{w}(z_2)$	
3			$\mathsf{rel}(\ell)$	
4		$\mathsf{acq}(\ell)$		
5		$\mathsf{w}(z_1)$		
6				$\mathsf{r}(z_1)$
7				$\mathsf{r}(z_2)$
8	$\mathsf{w}(x)$			
9				$\mathsf{w}(x)$
10	$\mathsf{w}(y)$			
11		$\mathsf{r}(y)$		
12		$\mathsf{rel}(\ell)$		

(b) Reordering ρ_1

Figure 1: The two conflicting events $e_1 = \langle t_1, \mathsf{w}(x)\rangle$ and $e_{12} = \langle t_4, \mathsf{w}(x)\rangle$ is a *predictable* data race of σ_1 which is also an optimistic sync-reversal race, witnessed by the *correct reordering* ρ_1 that reverses critical sections.

A dynamic data race detector observes an execution of a concurrent program P and infers the presence of a data race by analysing the trace of the observed execution. A key challenge in the design of such a technique is sensitivity to non-deterministic thread schedules — even for a fixed program input, a data race may be observed under a very specific thread schedule, but not under other thread schedules. This means that a simplistic race detector that, say, only checks for two conflicting events appearing simultaneously in the execution trace, is likely going to miss many bugs. This is where *predictive analysis* techniques shine — instead of looking for bugs only in the execution that was observed, they additionally also detect bugs in executions that, while not explicitly observed during testing, can nevertheless be inferred from the observed execution, without rerunning the underlying program P [33, 36, 47, 59, 62, 67]. Predictive techniques identify the space of executions or *reorderings* that can provably be inferred from a given observed execution σ, and then look for a reordering ρ in this space, that can serve as a witness to a bug such as a data race. Consider the execution σ_1 in Figure 1a consisting of events $e_1, e_2, \ldots, e_{12}$ where e_i denotes the i^{th} event from the top. The two write events on variable x, e_1 and e_{12}, are far apart and not witnessed as a data race in σ_1. However, the correct reordering ρ_1 of σ_1, in which the two write events appear consecutively, shows that it is nevertheless, a predictable data race of σ_1. Indeed any program P that generates σ_1 will also generate ρ_1 albeit with a different thread interleaving.

In general, sound (no false positives) and complete (no false negatives) data race prediction is known to be an intractable problem [46]. Soundness is a key desired property, since false positives need to be otherwise vetted manually, a task which is particularly challenging in the case of concurrent programs. Consequently, many recent works counter the intractability by proposing incomplete (but nevertheless sound) predictive race detection algorithms that work in polynomial time and have high precision in practice.

The main contribution of this paper is a new race prediction algorithm OSR that is sound, has higher prediction power than prior algorithms and achieves high scalability in practice.

The design of our algorithm OSR stems from the observation that often, data races can be exposed only by inverting the relative order of (some pairs of) critical sections, or *synchronizations*. The data race (e_1, e_{12}) in Figure 1a, for instance, can in fact only be observed in correct reorderings that invert the order of the two critical sections on lock ℓ. However, reversing synchronization (lock/unlock) operations in the reordering can further force a reversal in the order in which memory access events must appear in the reordering, and can be intractable to reason about [46, 47]. This strong tradeoff between precision (obtained by virtue of reversing the order of many synchronization operations) and performance has materialized on both the extremes. Algorithms such as those based on the happens-before partial order [44, 57] or the recently proposed SyncP [47] run in linear time but fail to expose races that mandate reasoning about synchronization reversals. On the other extreme, methods that exhaustively search for reversals, either resort to expensive constraint solving [33, 62] or saturation style reasoning [20, 56], and do not scale to long execution traces observed in real world concurrent applications. Our proposed algorithm OSR aims to strike a balance — it is designed to *optimistically* reason about *synchronization reversals*, and identifies those reversals that do not lead to the reversal of memory operations. The pair (e_1, e_{12}) in Figure 1 is an example of a race that OSR reports.

OSR reports all *optimistic synchronization-reversal* races in overall time $\widetilde{O}(\mathcal{N}^2)$, spending $\widetilde{O}(\mathcal{N})$ time for processing each event in the given execution trace σ. Here, $\mathcal{N}$ is the number of events in σ and $\widetilde{O}$ hides polynomial multiplicative factors due to number of locks and threads which are typically considered constants. In order to check for the absence of memory reversals, OSR constructs a graph (*optimistic reordering graph*) of events and checks if it is acyclic. Naively, such an acyclicity check would take $\widetilde{O}(\mathcal{N})$ time for every pair of conflicting events, resulting in a total cubic running time. A key technical contribution of our work is to perform this check in amortized constant time by constructing a succinct representation of this graph, called *abstract* optimistic reordering graph, of constant size. We show that this abstract graph preserves acyclicity, and can be constructed in an incremental manner in amortized constant time, allowing us to perform race prediction for the entire input execution in overall quadratic (instead of cubic) time. Finally, we show that the problem of checking the existence of an optimistic sync-reversal race also admits a matching quadratic time lower bound, thereby implying that our algorithm is optimal.

We implemented OSR and evaluate its performance thoroughly. Our evaluation demonstrates the effectiveness of our algorithm on a comprehensive suite of 153 Java and C/C++ benchmarks derived from real-world programs. Our results show OSR has comparable scalability as linear time algorithms SyncP and WCP, while it reports significantly more races than the second most predictive one on many benchmarks, confirming our hypothesis that going beyond the principle of synchronisation preservation allows us to discover significantly more races and with better performance. OSR, thus, advances the state-of-the-art in sound predictive race detection.

The rest of the paper is organized as follows. In Section 2, we discuss relevant background. In Section 3, we formally define the notion of optimistic sync-reversal races, and present our algorithm OSR for detecting all optimistic sync-reversal races in Section 4. Our evaluation of OSR and its comparison with other race prediction algorithms is presented in Section 5. In Section 6 we discuss related work and conclude in Section 7. The proof details can be found in our technical report [9].

2 PRELIMINARIES

In this section, we discuss preliminary notation and the formal definition of the problem of dynamic data race prediction. Next, we briefly recall the notion of *sync-preserving* data races [47] and discuss some of the limitations of this notion, paving the way to our algorithm OSR.

Trace and events. An execution trace (or simply trace) σ of a concurrent program is a sequence of events $\sigma = e_1 e_2 \ldots e_{\mathcal{N}}$. An event is a tuple $e = \langle i, t, op \rangle$, where i is a unique identifier for e, t is the thread that performs e and op is the operation corresponding to e; often the identifier i will be clear from context and we will drop it. We use $\mathsf{th}(e)$ and $\mathsf{op}(e)$ to denote the thread and operation of e. Operations are $\mathsf{r}(x)$, $\mathsf{w}(x)$ (read or write access of memory location or variable x) or $\mathsf{acq}(\ell)$, $\mathsf{rel}(\ell)$ (acquire or release of lock ℓ); fork and join operations are omitted from presentation but not from our implementation. For a trace σ, we will use $\mathsf{Events}(\sigma)$, $\mathsf{Threads}(\sigma)$, $\mathsf{Vars}(\sigma)$, $\mathsf{Locks}(\sigma)$ to denote respectively the set of all events, threads, variables and locks appearing in σ.

Well-formedness. We assume that traces are well-formed, in that they do not violate *lock semantics*. In particular, for a well formed trace σ, we require that for each lock $\ell \in \mathsf{Locks}(\sigma)$, the sequence of operations on ℓ alternate between acquires and releases, where each release event is preceded by a matching acquire event of the same thread. For an acquire (resp. release) event e, we use the notation $\mathsf{match}_\sigma(e)$ to denote the matching release (resp. acquire) event of e in σ if one exists; otherwise we say $\mathsf{match}_\sigma(e) = \perp$.

Trace order, thread order and reads-from. The trace order $\leq^\sigma_{\mathsf{tr}}$ of a trace σ is the total order induced by the sequence of events in σ, i.e., $e_1 \leq^\sigma_{\mathsf{tr}} e_2$ iff either $e_1 = e_2$ or e_1 appears earlier than e_2 in σ. The thread order $\leq^\sigma_{\mathsf{TO}}$ is a partial order on $\mathsf{Events}(\sigma)$ such that for any two events e_1, e_2, we have $e_1 \leq^\sigma_{\mathsf{TO}} e_2$ iff $\mathsf{th}(e_1) = \mathsf{th}(e_2)$ and $e_1 \leq^\sigma_{\mathsf{tr}} e_2$. When looking for predictable data races, we often look for reorderings of a given trace that preserve its control flow, and determine this using the *reads-from* function. For a read event $r \in \mathsf{Events}(\sigma)$ with $\mathsf{op}(r) = \mathsf{r}(x)$ for some variable x, the *writer* of r, denoted $w = \mathsf{rf}_\sigma(r)$ is the last write event on x before r, i.e., $\mathsf{op}(w) = \mathsf{w}(x)$, $w \leq^\sigma_{\mathsf{tr}} r$ and $\neg(\exists w' \neq w, \mathsf{op}(w') = \mathsf{w}(x) \wedge w \leq^\sigma_{\mathsf{tr}} w' \leq^\sigma_{\mathsf{tr}} r)$. Without loss of generality, we will assume that $\mathsf{rf}_\sigma(e)$ is always defined for each read event e. Given a set $S \subseteq \mathsf{Events}(\sigma)$, we say that S is $(\leq^\sigma_{\mathsf{TO}}, \mathsf{rf}_\sigma)$-closed if (a) for all events $e_1, e_2 \in \mathsf{Events}(\sigma)$ if $(e_1 \leq^\sigma_{\mathsf{TO}} e_2 \wedge e_2 \in S)$, then $e_1 \in S$, and (b) for all events $\forall e_1, e_2 \in \mathsf{Events}(\sigma)$, if $(e_1 = \mathsf{rf}_\sigma(e_2) \wedge e_2 \in S)$, then $e_1 \in S$. We use $\mathsf{TRClosure(S)}$ to denote the smallest set S' such that $S \subseteq S'$ and S' is $(\leq^\sigma_{\mathsf{TO}}, \mathsf{rf}_\sigma)$-closed.

Correct reordering. Predictive race detection, given a trace σ, asks if an alternate execution trace ρ witnesses a data race, and

	t_1	t_2
1	$\mathsf{w}(x)$	
2	$\mathsf{acq}(\ell)$	
3	$\mathsf{rel}(\ell)$	
4		$\mathsf{acq}(\ell)$
5		$\mathsf{r}(x)$
6		$\mathsf{rel}(\ell)$

(a) Trace σ_2

	t_1	t_2
1		$\mathsf{acq}(\ell)$
2	$\mathsf{w}(x)$	
3		$\mathsf{r}(x)$
4		$\mathsf{rel}(\ell)$
5	$\mathsf{acq}(\ell)$	
6	$\mathsf{rel}(\ell)$	

(b) Reordering ρ_2

	t_1	t_2
1		$\mathsf{acq}(\ell)$
2	$\mathsf{w}(x)$	
3		$\mathsf{r}(x)$
4		$\mathsf{rel}(\ell)$

(c) Reordering ρ_2'

Figure 2: The two write events $e_1 = \langle t_1, \mathsf{w}(x)\rangle$ and $e_5 = \langle t_2, \mathsf{w}(x)\rangle$ in σ_2 are conflicting. (e_1, e_5) is not a data race but a predictable data race of σ_2, witnessed by correct reorderings ρ_2 and ρ_2'.

more importantly, ρ can be *inferred* from σ. The notion of correct reorderings precisely formalizes this. Given well-formed traces σ and ρ, with $\mathsf{Events}(\rho) \subseteq \mathsf{Events}(\sigma)$, we say that ρ is a *correct reordering* of σ if ρ respects the thread order and reads-from relations of σ. This means that (1) $\mathsf{Events}(\rho)$ is $(\leq^{\sigma}_{\mathsf{TO}}, \mathsf{rf}_{\sigma})$-closed, (2) for any two events $e_1, e_2 \in \mathsf{Events}(\rho)$, if $e_1 \leq^{\sigma}_{\mathsf{TO}} e_2$, then $e_1 \leq^{\rho}_{\mathsf{TO}} e_2$, and (3) for any two events $e_1, e_2 \in \mathsf{Events}(\rho)$, if $e_1 = \mathsf{rf}_{\sigma}(e_2)$, then $e_1 = \mathsf{rf}_{\rho}(e_2)$.

Data races and predictable data races. A pair of events (e, e') in σ is said to be a conflicting pair, denoted $e \bowtie e'$, if both are access events to the same variable, and at least one of them is a write event, i.e., $(\mathsf{op}(e), \mathsf{op}(e')) \in \{(\mathsf{w}(x), \mathsf{w}(x)), (\mathsf{w}(x), \mathsf{r}(x)), (\mathsf{r}(x), \mathsf{w}(x))\}$ for some $x \in \mathsf{Vars}(\sigma)$. For a trace π with $\mathsf{Events}(\pi) \subseteq \mathsf{Events}(\sigma)$, we say that event e is σ-enabled in π if $e \notin \mathsf{Events}(\pi)$ but all thread-predecessors of e are in π, i.e., $\{e' \in \mathsf{Events}(\sigma) \mid e' \neq e, e' \leq^{\sigma}_{\mathsf{TO}} e\} \subseteq \mathsf{Events}(\pi)$. A conflicting pair (e, e') is said to be a *data race* of σ if there is a prefix π of σ such that both e and e' are σ-enabled in π. Finally, a conflicting pair (e, e') is a *predictable data race* of σ if there is a correct reordering ρ of σ such that both e and e' are σ-enabled in some prefix of ρ. In this case, we say that ρ witnesses the data race (e, e').

Example 1. Consider trace σ_2 in Figure 2a containing 6 events performed by two threads t_1 and t_2. As before, we use e_i to denote the i^{th} event of σ_2. The two events $e_1 = \langle t_1, \mathsf{w}(x)\rangle$ and $e_5 = \langle t_2, \mathsf{w}(x)\rangle$ are conflicting (i.e., $e_1 \bowtie e_5$). The pair (e_1, e_5) is not a data race in σ_2 as no prefix of σ_2 has both these events simultaneously enabled. Consider the trace ρ_2 in Figure 2b; it is a correct reordering of σ_2 because it preserves both the thread order and reads-from relation of σ_2. For the same reason, ρ_2' is a correct reordering of σ_2 (and also of ρ_2). Now, observe that (e_1, e_5) is a data race in ρ_2 (and also in ρ_2') because in the prefix $\pi = \langle t_2, \mathsf{acq}(\ell)\rangle$, both e_1 and e_5 are ρ_2-enabled (resp. ρ_2'-enabled) and thus σ_2-enabled. Thus, while (e_1, e_5) is not a data race in σ_2, it is a predictable data race of σ_2.

The problem of predicting data races — given an execution trace σ, determine if there is a predictable data race of σ — has been studied before [33, 36, 56, 59, 62, 67] and is known to be an intractable problem [46]. This means that any sound and complete algorithm for predicting data races is unlikely to scale to real world software applications whose execution traces can have billions of events. To cater to this, practical data race predictors resort to incomplete but sound algorithms that run in polynomial time. In the next section, we discuss the recently proposed SyncP algorithm that employs the principle of *synchronization preservation* for predicting data races whose theoretical complexity is linear.

2.1 Sync-Preserving Data Races

Our work is closer in spirit to the work of [47] which presents the SyncP algorithm that works in linear time and is the current state-of-the-art race prediction algorithm. The principle employed by SyncP is to focus on a special class of reorderings and the data races witnessed by such reorderings; we discuss these next.

Sync-preserving reorderings and data races. A correct reordering ρ of a trace σ is said to be *sync(hronization)*-preserving if for any two critical sections of σ (on the same lock) that are both present in ρ, their relative order is the same, That is, for every lock $\ell \in \mathsf{Locks}(\sigma)$ and for any two acquire events $a_1, a_2 \in \mathsf{Events}(\sigma)$ such that $\mathsf{op}(a_1) = \mathsf{op}(a_2) = \mathsf{acq}(\ell)$, if $a_1, a_2 \in \mathsf{Events}(\rho)$, then we have: $a_1 \leq^{\rho}_{\mathsf{tr}} a_2$ iff $a_1 \leq^{\sigma}_{\mathsf{tr}} a_2$. A pair of conflicting events (e, e') in $\mathsf{Events}(\sigma)$ is said to be a sync-preserving data race of σ if there is a sync-preserving correct reordering ρ of σ that witnesses this race.

Example 2. Consider again, the trace σ_2 and recall from Example 1 that the pair (e_1, e_5) is not a data race of σ_2 but a predictable race witnessed by the correct reordering ρ_2. Observe however that ρ_2 is not a sync-preserving reordering of σ_2 because it flips the order of the two critical sections on lock ℓ. Nevertheless, (e_1, e_5) is a sync-preserving race of σ_2. This is because the reordering ρ_2' is, in fact, a sync-preserving reordering of σ_2 (even though it is a prefix of the non-sync-preserving reordering ρ_2); there is only one critical section in ρ_2' and thus vacuously, the relative order on critical sections is the same as in σ_2.

Limited predictive power of SyncP. While the SyncP algorithm runs in overall linear time, it can miss data races which are not synchronization-preserving. These are precisely those conflicting pairs (e, e') such that any correct reordering that witnesses a race on e and e' necessarily reverses the relative order of two critical sections on a common lock. We illustrate this next, and remark that, in general, reasoning about even a single reversal is intractable [47].

Example 3. Let us again consider the trace σ in Figure 1a (Section 1). The two conflicting events $e_1 = \langle t_1, \mathsf{w}(x)\rangle$ and $e_{12} = \langle t_4, \mathsf{w}(x)\rangle$, are a predictable data race of σ_1 as witnessed by the correct reordering ρ_1 in Figure 1b, which is not a sync-preserving correct reordering of ρ_1. In fact, consider any correct reordering α of σ_1 that witnesses the race (e_1, e_{12}). Then α must include the events e_{10} and e_{11}, and thus the corresponding write events e_4 and e_8, together with the thread predecessors $e_3 = \langle t_2, \mathsf{acq}(\ell)\rangle$ and $e_7 = \langle t_3, \mathsf{acq}(\ell)\rangle$. Next, for well-formedness, at least one of the matching releases $e_6 = \langle t_2, \mathsf{rel}(\ell)\rangle$ as well as $e_9 = \langle t_3, \mathsf{rel}(\ell)\rangle$ must also be present in α. However, including e_6 in α would enforce that $e_5 = \langle t_2, \mathsf{r}(z)\rangle$, and its write event $e_2 = \langle t_1, \mathsf{w}(y)\rangle$ are present in α, and then, the event e_1 must also be present in the reordering making it no longer enabled in α. This, therefore, means that $e_6 \notin \mathsf{Events}(\alpha)$, and thus, the only other available release event e_9 must be present in α (for well-formedness). Further, to ensure well-formedness, e_3 must appear after e_9 in α. Thus, any reordering α witnessing the race between $e_1$and e_{12} must reverse the order of the critical sections.

3 OPTIMISTIC REASONING FOR REVERSALS

Given that reasoning about synchronization reversals is computationally hard, how do we identify such races efficiently? At a high level, the intractability in data race prediction arises because a search for a correct reordering entails (1) a search for an appropriate set of events (amongst exponentially many sets) and further, (2) given an appropriate set of events, a search for a linear order (amongst exponentially many linear orders) on this set which is well-formed, is a correct reordering and witnesses the race. We propose (1) a new notion of data races called *optimistic sync(hronization) reversal* races which can be predicted by opting for an *optimistic* approach to resolve both these steps, and (2) an algorithm OSR to detect all such data races in $\widetilde{O}(\mathcal{N}^2)$ time. In this section, we discuss this notion of data races and discuss our algorithm in Section 4.

3.1 Optimistic Sync-Reversal Races

A crucial aspect of choosing the correct set of events is to ensure that multiple acquire events on the same lock do not stay unmatched; otherwise, the set cannot be linearized to a well-formed trace. In general, adding a matching release event may lead to recursive addition of further events. Some choices may (recursively) at times lead to the addition of one of the two focal events e, e' (candidate data race), leading to them being no longer enabled. We define a simple and tractable notion of *optimistic lock-closure*, which, instead of considering all choices, simply includes all matching release events as long as the two focal events are not included. In the following, we fix a trace σ.

Optimistic lock-closure. Let $e_1, e_2 \in \mathsf{Events}(\sigma)$. We say that a set $S \subseteq \mathsf{Events}(\sigma)$ is *optimistically lock-closed* with respect to (e_1, e_2) if (a) $e_1, e_2 \notin S$ and $\mathsf{prev}_\sigma(e_1), \mathsf{prev}_\sigma(e_2) \in S$, (b) S is $(\leq^\sigma_{\mathsf{TO}}, \mathsf{rf}_\sigma)$-closed, and (c) for every acquire event $a \in S$, if $e_1, e_2 \notin \mathsf{TRClosure}(\mathsf{match}_\sigma(a))$, then $\mathsf{match}_\sigma(a) \in S$. We denote the smallest set that contains S and is optimistically lock-closed set, as $\mathsf{OLClosure}(S, e_1, e_2)$

Example 4. Let us recall trace σ_1 from Figure 1 and consider the set $S_1 = \{e_3, e_4, e_7, e_8, e_9, e_{10}, e_{11}\}$. Observe that S_1 is optimistically lock-closed with respect to (e_1, e_{12}), because (1) S_1 doesn't include either of e_1, e_{12}, (2) S_1 is $(\leq^\sigma_{\mathsf{TO}}, \mathsf{rf}_\sigma)$-closed, and finally, (3) $e_1, e_{12} \notin \mathsf{TRClosure}(e_9)$. Note that $e_1 \in \mathsf{TRClosure}(e_6)$ but $e_6 \notin S_1$.

Even though the notion of optimistically lock-closed set is simple, in general, checking if such a set can be linearized into a correct reordering that witnesses a data race, is an intractable problem, as we show next (Theorem 3.1).

Theorem 3.1. Let σ be a trace, let e_1, e_2 be conflicting events and let $S \subseteq \mathsf{Events}(\sigma)$ be an optimistically lock-closed set with respect to (e_1, e_2). The problem of determining whether there is a correct reordering ρ such that $\mathsf{Events}(\rho) = S$ is NP-hard.

The proof of Theorem 3.1 is presented in appendix A.1. Given the above result, we also define the following more tractable notion of *optimistic reordering* that ensures that there are no memory reversals, and moreover, critical sections are reversed only when absolutely required, i.e., that unmatched critical sections appear later than matched ones.

Optimistic correct reordering. A trace ρ is said to be an optimistic correct reordering of σ if (a) ρ is a correct reordering of σ, (b) for all pairs of conflicting memory access events $e_1 \bowtie e_2$ in $\mathsf{Events}(\rho)$, $e_1 \leq^\rho_{\mathsf{tr}} e_2$ iff $e_1 \leq^\sigma_{\mathsf{tr}} e_2$, and (c) for any lock ℓ and for any two acquire events $a_1 \neq a_2$ (with $\mathsf{op}(a_1) = \mathsf{op}(a_2) = \mathsf{acq}(\ell)$), if a_1 and a_2 are both matched in ρ (i.e., $\mathsf{match}_\sigma(a_i) \in \mathsf{Events}(\rho)$ for both $i \in \{1, 2\}$), then we must have $a_1 \leq^\rho_{\mathsf{tr}} a_2$ iff $a_1 \leq^\sigma_{\mathsf{tr}} a_2$.

We now formalize *optimistic sync-reversal* data races.

Definition 1 (Optimistic Sync-Reversal Race)**.** Let σ be a trace and let (e_1, e_2) be a pair of conflicting events in σ. We say that (e_1, e_2) is an optimistic sync-reversal data race if there is an optimistic correct reordering ρ of σ such that $\mathsf{Events}(\rho)$ is optimistically lock-closed with respect to (e_1, e_2) and both e_1 and e_2 are σ-enabled in ρ.

Example 5. In Figure 1, the pair (e_1, e_{12}) is an optimistic sync-reversal race, because the prefix ρ'_1 with first 7 events of ρ_1 is an optimistic reordering of the optimistically lock closed set S_1, outlined in Example 4, (in which e_1 and e_{12} are σ_1-enabled). This is because, all conflicting accesses of ρ'_1 have the same relative order as in σ_1, and further, the unmatched acquire event is positioned after all closed critical sections. Similarly, for the trace σ_2 of Figure 2, the linearization $\rho'_2 = \langle t_2, \mathsf{acq}(\ell) \rangle$ of the set S_2 (outlined in Example 4) is trivially an optimistic correct reordering.

3.2 Comparison with other techniques

Here, we qualitatively compare our proposed class of races with those reported by other sound predictive race detection techniques proposed in the literature, namely SyncP [47] and M2 [56] and illustrate how the set of races reported by OSR is neither a strict subset, nor a strict super set of those detected by each.

Example 6. Recall again the execution trace σ_1 in Figure 1. In Example 5 we established that the pair (e_1, e_{12}) is an optimistic sync-reversal race, while in Example 3, we showed that it is not a sync-preserving data race. When determining if (e_1, e_{12}) can be declared a predictive data race, the M2 algorithm computes the set $S = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_{10}, e_{11}\}$ to be the candidate set that witnesses the race. Observe however, this set contains the event e_1 and thus cannot witness the race (e_1, e_{12}) since one of these events is not enabled in S. Thus, some optimistic sync-reversal races are neither sync-preserving races, nor can be detected by M2.

Example 7. Consider the trace in Figure 3a. The pair (e_1, e_{21}) is a sync-preserving data race as witnessed by the correct reordering shown in Figure 3b. This pair, however is not an optimistic sync-reversal data race since the smallest optimistically lock-closed set capable of witnessing the race is the set $S_{\mathsf{OSR}} = \{e_{[3,9]}, e_{[12,14]}, e_{17,20}\}$, where $e_{i,j}$ is shorthand for $e_i, e_{i+1}, \ldots, e_{j-1}, e_j$. Observe that S_{OSR}contains two unmatched acquire events of lock ℓ_2, and adding either matching release will bring e_1 in the set. Likewise, M2 computes the set containing all events but e_{21}, and thus contains e_1. Thus, there are sync-preserving races which are neither optimistic sync-reversal races, nor can be detected by M2.

Example 8. Finally, consider the trace in Figure 3c, derived from [56]. Here, the pair (e_{10}, e_{19}) is a data race that M2 can predict (also see Figure 3d for the witnessing execution). We remark that any correct reordering witnessing this race must reverse the order

	t_1	t_2	t_3	t_4	t_5
1	$w(x)$				
2	$w(z_1)$				
3		$acq(\ell_1)$			
4		$w(z_2)$			
5		$rel(\ell_1)$			
6			$acq(\ell_1)$		
7			$w(z_3)$		
8		$acq(\ell_2)$			
9		$w(z_4)$			
10		$r(z_1)$			
11		$rel(\ell_2)$			
12			$r(z_4)$		
13					$acq(\ell_2)$
14					$w(z_5)$
15					$r(z_1)$
16					$rel(\ell_2)$
17			$r(z_5)$		
18			$rel(\ell_1)$		
19				$r(z_2)$	
20				$r(z_3)$	
21				$w(x)$	

(a) SyncP (but not M2 or OSR) race

	t_1	t_2	t_3	t_4
1		$acq(\ell_1)$		
2		$w(z_2)$		
3		$rel(\ell_1)$		
4			$acq(\ell_1)$	
5			$w(z_3)$	
6				$r(z_2)$
7				$r(z_3)$
8	$w(x)$			
9				$w(x)$

(b) Witness reordering of (a)

	t_1	t_2
1	$w(x_1)$	
2	$w(x_2)$	
3	$w(x_3)$	
4	$acq(\ell)$	
5	$w(x_4)$	
6	$r(x_1)$	
7	$rel(\ell)$	
8	$acq(\ell)$	
9	$r(x_4)$	
10	$w(x)$	
11	$rel(\ell)$	
12		$w(x_3)$
13		$acq(\ell)$
14		$w(x_4)$
15		$rel(\ell)$
16		$w(x_1)$
17		$r(x_2)$
18		$r(x_3)$
19		$r(x)$

(c) M2 (but not SyncP/OSR) race

	t_1	t_2
1	$w(x_1)$	
2	$w(x_2)$	
3	$w(x_3)$	
4		$w(x_3)$
5		$acq(\ell)$
6		$w(x_4)$
7		$rel(\ell)$
8	$acq(\ell)$	
9	$w(x_4)$	
10	$r(x_1)$	
11	$rel(\ell)$	
12	$acq(\ell)$	
13	$r(x_4)$	
14		$w(x_1)$
15		$r(x_2)$
16		$r(x_3)$
17	$w(x)$	
18		$r(x)$

(d) Witness reordering of (c)

Figure 3: Two traces containing two predictable races. One of them (a) can be detected by SyncP, but not M2 nor OSR. (b) is the witness of race in (a). The other one (c) can be detected by M2, but not SyncP nor OSR. Trace in (c) is directly cited from M2 paper [56] without modification. (d) is the witness of race in (c).

of the two acquire events e_8 and e_{13}, as well as the order of conflicting memory access events e_9 and e_{14}. Consequently, this is an example of a race reported by M2 that is neither a sync-preserving race, nor an optimistic sync-reversal race.

4 THE OSR ALGORITHM

We now describe our algorithm OSR that detects optimistic sync-reversal data races. For ease of presentation, we will first discuss how to check if a given pair (e_1, e_2) of conflicting events is an optimistic sync-reversal data race (Section 4.1), in $\widetilde{O}(\mathcal{N})$ time, where $\mathcal{N}$ is the number of events in the given trace. Naively, it can be used to report all optimistic sync-reversal data races in $\widetilde{O}(\mathcal{N}^3)$ time, by enumerating all $O(\mathcal{N}^2)$ pairs of conflicting events and checking each of them in $\widetilde{O}(\mathcal{N})$ time. Instead, OSR runs in overall $\widetilde{O}(\mathcal{N}^2)$ time and is based on interesting insights that enable it to perform incremental computation over the entire trace (Section 4.2). We present our overall algorithm and its optimality in Section 4.3.

4.1 Checking Race On A Given Pair Of Events

Based on Definition 1, the task of checking if a given pair (e_1, e_2) of conflicting events is an optimistic sync-reversal data race entails examining all optimistic lock-closed sets and checking if any of these can be linearized.

Constructing optimistically lock-closed set. Our algorithm, however, exploits the following observation (Lemma 4.1), and focuses on only a single set, namely the smallest such set. In the following, we will abuse the notation and use $\mathsf{OLClosure}(e_1, e_2)$ to denote the set $\mathsf{OLClosure}(S_{e_1,e_2}, e_1, e_2)$, where $S_{e_1,e_2} = \{\mathsf{prev}_\sigma(e_1)\} \cup \{\mathsf{prev}_\sigma(e_2)\}$. Here, $\mathsf{prev}_\sigma(e)$ is the last event f such that $f \leq^\sigma_{\mathsf{TO}} e$; if no such event exists, we say $\mathsf{prev}_\sigma(e) = \bot$, in which case $\{\mathsf{prev}_\sigma(e)\} = \varnothing$.

Lemma 4.1. Let e_1, e_2 be conflicting events in trace σ. If (e_1, e_2) is an optimistic sync-reversal race, then it can be witnessed in an optimistic correct reordering ρ such that $\mathsf{Events}(\rho) = \mathsf{OLClosure}(e_1, e_2)$.

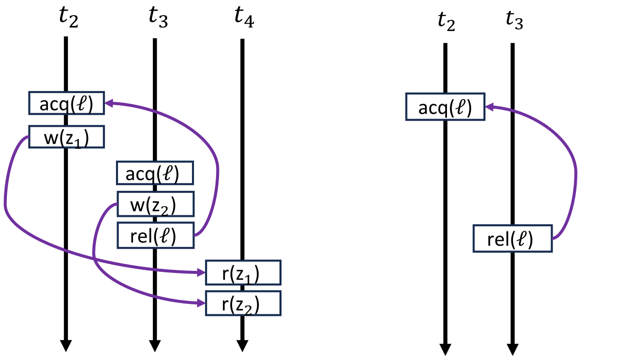
Algorithm 1: Computing optimistic lock closure

```
procedure ComputeOLClosure(S_0, e_1, e_2)
    S ← S_0 ∪ TRClosure(prev_σ(e_1)) ∪ TRClosure(prev_σ(e_2))
    while S changes do
        if (∃a ∈ Acqs(S), match_σ(a) ∉ S ∧
            e_1, e_2 ∉ TRClosure(match_σ(a)) ) then
            S ← S ∪ TRClosure(match_σ(a))
    return S
```

In Algorithm 1, we outline our algorithm to compute the smallest set that we identified in Lemma 4.1. It takes 3 arguments — the two events e_1, e_2 and a set S_0; for computing $\mathsf{OLClosure}(e_1, e_2)$, we must set $S_0 = \varnothing$; later in Section 4.2 this set will be used to enable incremental computation. This algorithm performs a fixpoint computation starting from the set $S_0 \cup \mathsf{TRClosure}(\mathsf{prev}_\sigma(e_1)) \cup \mathsf{TRClosure}(\mathsf{prev}_\sigma(e_2))$, and identifies an unmatched acquire event a and checks if its matching release r can be added without adding e_1 or e_2; if so, r is added; $\mathsf{Acqs}(S)$ denotes the set of acquire events in the set S. The algorithm ensures that the set is $(\leq^\sigma_{\mathsf{TO}}, \mathsf{rf}_\sigma)$-closed at each step, and runs in $O(\mathcal{T}^2\mathcal{N}) = \widetilde{O}(\mathcal{N})$ time.

Checking optimistic reordering. First, we check if the set S constructed by Algorithm 1 is *lock-feasible*, i.e., the set of unmatched acquires $\mathsf{OAcqs}(S, \ell) = \{a \in \mathsf{Acqs}(S) \mid \mathsf{match}_\sigma(a) \notin S\}$ for each lock ℓ is either singleton or empty:

$$\mathsf{lockFeasible}(S) \equiv \forall \ell \in \mathsf{Locks}(\sigma), |\mathsf{OAcqs}(S, \ell)| \leq 1$$



(a) Opt. reord. graph of σ_1 **(b) Abst. opt. reord. graph of σ_1**

Figure 4: Optimistic and abstract optimistic reordering graphs of σ_1 from Figure 1a are acyclic

Observe that if $\mathsf{lockFeasible}(S)$ does not hold, then every linearization of S will have more than one critical sections (on some lock) that overlap, making it a non-well-formed trace. Next, inspired from the notion of optimistic reordering, we construct the *optimistic-reordering-graph* $G_S^{\mathsf{Opt}} = (V_S^{\mathsf{Opt}}, E_S^{\mathsf{Opt}})$, where $V_S^{\mathsf{Opt}} = S$, and $E_S^{\mathsf{Opt}} = E_{S,\leq_{\mathsf{TO}}^{\sigma}}^{\mathsf{Opt}} \cup E_{S,\bowtie}^{\mathsf{Opt}} \cup E_{S,\mathsf{match}}^{\mathsf{Opt}} \cup E_{S,\mathsf{unmatch}}^{\mathsf{Opt}}$. Here, $E_{S,\leq_{\mathsf{TO}}^{\sigma}}^{\mathsf{Opt}}$ is the set of edges (e, e'), where $e = \mathsf{prev}_{\sigma}(e')$. The set $E_{S,\bowtie}^{\mathsf{Opt}}$ consists of all immediate conflict edges, i.e., all pairs (e, e') in S such that $e \bowtie e'$, $e \leq_{\mathsf{tr}}^{\sigma} e'$ and there is no intermediate event in σ that conflicts with both. The set $E_{S,\mathsf{match}}^{\mathsf{Opt}}$ consists of all pairs (r, a') such that $r \leq_{\mathsf{tr}}^{\sigma} a'$ and there is a common lock ℓ for which $\mathsf{op}(r) = \mathsf{rel}(\ell), \mathsf{op}(a') = \mathsf{acq}(\ell)$, both r and a' are matched in S, and there is no intermediate critical section on ℓ. Finally, the remaining set of edges order matched critical sections before unmatched ones, i.e., $E_{S,\mathsf{unmatch}}^{\mathsf{Opt}} = \{(r, a') \mid \exists \ell, \mathsf{op}(r) = \mathsf{rel}(\ell), \mathsf{op}(a') = \mathsf{acq}(\ell), \mathsf{match}_{\sigma}(a') \notin S\}$. Since optimistic reorderings forbid reversal in the order of conflicting memory accesses, as well as in the order of same-lock critical sections that are completely matched, it suffices to check the acycliclity of G^{Opt}, so that the existence of witness is guaranteed.

Lemma 4.2. Let σ be a trace and let $S \subseteq \mathsf{Events}(\sigma)$ such that S is $(\leq_{\mathsf{TO}}^{\sigma}, \mathsf{rf}_{\sigma})$-closed and also lock-feasible. Then, there is an optimistic reordering ρ of σ on the set S iff the graph G_S^{Opt} is acyclic.

Example 9. For trace σ_1 in Figure 1a, we have $\mathsf{OLClosure}(e_1, e_{12}) = \{e_3, e_4, e_7, e_8, e_9, e_{10}, e_{11}\}$. The optimistic-reordering-graph over $S_1 = \mathsf{OLClosure}(e_1, e_{12})$ is shown in Figure 4a; Observe that there is no cycle. Indeed, as guaranteed by Lemma 4.2, there is an optimistic reordering, namely the 7 length prefix of ρ_1 from Figure 1b that witnesses the race (e_1, e_{12}). Let us now consider σ_3, Figure 5a. The optimistic lock-closure with respect to (e_4, e_9) is $S_3 = \mathsf{OLClosure}(e_4, e_9) = \{e_1, e_2, e_3, e_6, e_7, e_8\}$. The optimistic reordering graph over S_3, shown in Figure 5b, contains a cycle. Indeed, (e_4, e_9) is not a predictable race.

We remark that G^{Opt} can be constructed and checked for cycles in time $O(\mathcal{T}\mathcal{N}) = \widetilde{O}(\mathcal{N})$. Thus the overall algorithm for checking if given (e_1, e_2) is an optimistic sync-reversal race is — first compute $\mathsf{OLClosure}(e_1, e_2)$ in $\widetilde{O}(\mathcal{N})$ time, check lock-feasibility in $O(\mathcal{L}\mathcal{T}) = \widetilde{O}(1)$ time and perform graph construction and cycle detection in $\widetilde{O}(\mathcal{N})$ time. We thus have the following theorem.

Theorem 4.1. Let σ be a trace and let e_1, e_2 be conflicting events in σ. The problem of determining if (e_1, e_2) is an optimistic sync-reversal race can be solved in time $O(\mathcal{T}^2\mathcal{N}) = \widetilde{O}(\mathcal{N})$ time.

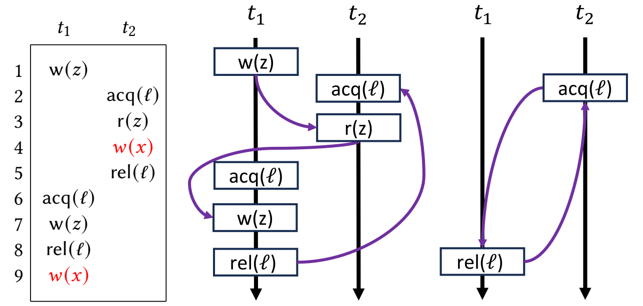
4.2 Incremental Race Detection

Overview. Recall that there are $O(\mathcal{N}^2)$ pairs of conflicting events, and instead of naively examining each of them, we develop an incremental algorithm that determines the existence of an optimistic sync-reversal race in total $\widetilde{O}(\mathcal{N}^2)$ time. We achieve this by spending $\widetilde{O}(\mathcal{N})$ time per (read/write) event $e \in \mathsf{Events}(\sigma)$, and determine in overall $\widetilde{O}(\mathcal{N})$ time if there is some event e' such that (e', e) is a race, by scanning the trace from earliest to latest events. To do so, our algorithm exploits several novel insights. Let us fix one of the events e. First, we show that the optimistic lock closure can be computed incrementally from previously computed sets, instead of computing it from scratch for each e'. Even though the closure sets can be computed incrementally, the optimistic-reordering-graph G^{Opt} (Section 4.1) cannot be computed in an incremental fashion, because the edges in this graph depend upon precisely which events are present in the set. In particular, a previously unmatched acquire event may become matched in a larger set, and thus, we may have fewer edges in the larger graph. Our second insight caters to this — we represent the graph succinctly as an *abstract optimistic-reordering-graph* which has $\widetilde{O}(1)$ (instead of $\widetilde{O}(\mathcal{N})$) nodes, and moreover, can be computed by pre-populating an appropriate data structure and performing *range minima queries* over it, to determine reachability information in the abstract graph in $\widetilde{O}(1)$ time.

Incrementally constructing optimistic lock closure. The incremental closure computation relies on the observation that the closure is monotonic with respect to thread-order (Lemma 4.3). Thus, if we fix a thread t, and scan the events of t from earliest to latest events, then we can reuse prior computations. In fact, Algorithm 1 already works in this fashion — it builds on top of the given input set S. Lemma 4.3 establishes the correctness and time complexity of closure computation.

Lemma 4.3. Let $e_1, e_2, e_2' \in \mathsf{Events}(\sigma)$ be events in trace σ with $e_2 \leq_{\mathsf{TO}}^{\sigma} e_2'$. Let $S = \mathsf{OLClosure}(e_1, e_2)$ and let $S' = \mathsf{OLClosure}(e_1, e_2')$. We have the following: (1) $S \subseteq S'$. (2) $S = \mathtt{ComputeOLClosure}(e_1, e_2, \varnothing)$, and further this call (in Algorithm 1) takes $\widetilde{O}(|S|)$ time. (3) $S' = \mathtt{ComputeOLClosure}(e_1, e_2', S)$, and further this call (in Algorithm 1) takes $\widetilde{O}(|S'| - |S|)$ time.

Abstract optimistic-reordering-graph. For a set $S \subseteq \mathsf{Events}(\sigma)$, the abstract optimistic-reordering-graph is a tuple $G_S^{\mathsf{Abs}} = (V_S^{\mathsf{Abs}}, E_S^{\mathsf{Abs}})$, where the vertices and edges are defined as follows. (1) $V_S^{\mathsf{Abs}} = \bigcup_{\ell \in \mathsf{Locks}(\sigma)} \{\mathsf{lastRel}(S, \ell)\} \cup \mathsf{OAcqs}(S, \ell)$, where $\mathsf{lastRel}(S, \ell)$ is the last release event on lock ℓ (according to $\leq_{\mathsf{tr}}^{\sigma}$) which is present in S. (2) $(e, e') \in E^{\mathsf{Abs}}$ if there is a path from e to e' in the graph G_S^{Opt}. In other words, G_S^{Abs} only contains $O(\mathcal{L})$ vertices, corresponding to the last release events, and acquire events that are unmatched in S, and preserves the reachability information between these events. Lemma 4.4 formalizes the intuition behind



(a) Trace σ_3 (b) Reordering graph (c) Abstract graph

Figure 5: In σ_3, (e_4, e_9) is not a predictable race. The optimistic reordering graph and the abstract optimistic reordering graph are cyclic.

this graph — it preserves the cyclicity information of the larger graph G_S^{Opt}, because any cycle in G_S^{Opt} must involve a 'backward' edge from a matched release and an unmatched acquire event. G_S^{Abs} can thus be used to check for the existence of an optimistic reordering using an $\widetilde{O}(1)$ check instead of an $\widetilde{O}(\mathcal{N})$ check based on Lemma 4.2.

Lemma 4.4. Let σ be a trace and let $S \subseteq \mathsf{Events}(\sigma)$ be a $(\leq^\sigma_{\mathsf{TO}}, \mathsf{rf}_\sigma)$-closed set. G_S^{Opt} has a cycle iff G_S^{Abs} has a cycle.

Example 10. Figure 4b shows the abstract optimistic reordering graph for trace σ_1 in Figure 1a, corresponding to the set $S_1 = \mathsf{OLClosure}(e_1, e_{12})$, and contains the last release of lock ℓ in S_1 as well as the only open acquire in S_1. This graph, like the graph in Figure 4a is acyclic. In Figure 5, the abstract graph (Figure 5c) captures the path $e_2 \to e_3 \to e_7 \to e_8$ of Figure 5b with a direct edge $e_2 \to e_8$, thereby preserving the cycle.

Constructing vertices and backward edges of G_S^{Abs}. Recall that S is a $(\leq^\sigma_{\mathsf{TO}}, \mathsf{rf}_\sigma)$-closed subset of $\mathsf{Events}(\sigma)$. The set of vertices of this graph can be determined in $O(\mathcal{L})$ time by maintaining the last event of every thread present in S. This information can be inductively maintained as S is being computed incrementally. The 'backward' edges — namely those pairs (r, a) where $a \in S$ is an unmatched acquire on some lock ℓ, and $r = \mathsf{lastRel}(\mathsf{S}, \ell)$ but $a \leq^\sigma_{\mathsf{tr}} r$ — can be computed in $O(\mathcal{L})$ time.

Pre-computing earliest immediate successor. For constructing forward edges, we first pre-compute a map (for each pair of threads t_1, t_2), EIS_{t_1,t_2} such that, for every $e_1 \in \mathsf{Events}(\sigma)|_{t_1} = \{e \in \mathsf{Events}(\sigma) \mid \mathsf{th}(e) = t\}$, the event $\mathsf{EIS}_{t_1,t_2}(e_1)$ is the earliest immediate successor of e_1 in thread t_2, in the full graph $G^{\mathsf{Opt}}_{\mathsf{Events}(\sigma)}$; observe the subscript $\mathsf{Events}(\sigma)$ instead of an arbitrary set S. EIS_{t_1,t_2} can be computed as a pre-processing step in $O(\mathcal{T}\mathcal{N}) = \widetilde{O}(\mathcal{N})$ time and stored as an array, indexed by the events of thread t_1.

Determining forward edges of G_S^{Abs}. The forward edges of G_S^{Abs} summarize paths in G_S^{Opt} and are computed as follows. Recall that we are given a $(\leq^\sigma_{\mathsf{TO}}, \mathsf{rf}_\sigma)$-closed subset S of $\mathsf{Events}(\sigma)$, and the path between two events must only be contained with the events of S, thus the arrays $\{\mathsf{EIS}_{t_1,t_2}\}_{t_1,t_2 \in \mathsf{Threads}(\sigma)}$ cannot be used as is to efficiently determine paths. However, a combination of range

Algorithm 2: Earliest successors of event e within set S

procedure `getSuccessors`(e, S)
  let $t_e = \mathsf{th}(e)$, $\mathsf{visitedThr} \leftarrow \varnothing$
  let last^S_t be the last event by t in S, for $t \in \mathsf{Threads}(\sigma)$
  for $t \in \mathsf{Threads}(\sigma)$ **do**
    $\mathsf{succ}^S_{e,t} \leftarrow \mathsf{rangeMin}(\mathsf{EIS}_{t_e,t})[e, \mathsf{last}^S_{t_e}]$
  while $\mathsf{visitedThr} \neq \mathsf{Threads}(\sigma)$ **do**
    let t_1 be s.t. $t_1 \notin \mathsf{visitedThr}$ and succ^S_{e,t_1} is the earliest in $\leq^\sigma_{\mathsf{tr}}$ from $\{\mathsf{succ}^S_{e,t}\}_{t \notin \mathsf{visitedThr}}$
    for $t_2 \in \mathsf{Threads}(\sigma)$ **do**
      $newSucc \leftarrow \mathsf{rangeMin}(\mathsf{EIS}_{t_1,t_2})[\mathsf{succ}^S_{e,t_1}, \mathsf{last}^S_{t_1}]$
      if $newSucc \leq^\sigma_{\mathsf{TO}} \mathsf{succ}^S_{e,t_2}$ **then**
        $\mathsf{succ}^S_{e,t_2} \leftarrow newSucc$
    $\mathsf{visitedThr} \leftarrow \mathsf{visitedThr} \cup \{t_1\}$
  return $\{\mathsf{succ}^S_{e,t}\}_{t \in \mathsf{Threads}(\sigma)}$

Algorithm 3: Detecting races between e and thread t

procedure `incrementalRaceDetection`(e, t)
  $S \leftarrow \varnothing$
  for $e' \in \mathsf{Events}(\sigma)|_t$ *s.t.* $e \bowtie e'$ *and* $e' \leq^\sigma_{\mathsf{tr}} e$ **do**
    $S \leftarrow$ `ComputeOLClosure`(e, e', S)
    if $\mathsf{lockFeasible}(S)$ *and* G_S^{Abs} *is acyclic* **then**
      declare (e', e) as race.

minima queries [7] and shortest path computation can nevertheless still be used to determine path information efficiently. Let us use $\mathsf{succ}^S_{e,t}$ to denote the earliest event in thread t that has a path from event e, using only forward edges of G_S^{Opt}. The event $\mathsf{succ}^S_{e,t}$ can be computed using a Bellman-Ford-Moore [14, 29, 50] style shortest path computation, as shown in Algorithm 2. This algorithm performs $\mathsf{rangeMin}(A)[a, b]$ queries which return the earliest event (according to $\leq^\sigma_{\mathsf{TO}}$) in the segment of the array A starting at index a and ending at index b. With $\widetilde{O}(\mathcal{N})$ time and space pre-processing, each range minimum query takes $O(1)$ time [7, 30], Thus, the task of determining $\{\mathsf{succ}^S_{e,t}\}_{t \in \mathsf{Threads}(\sigma)}$ takes $O(\mathcal{T}^2)$ time. Now, in the graph G_S^{Abs}, we add an edge from e to e' if $\mathsf{succ}^S_{e,\mathsf{th}(e')} \leq^\sigma_{\mathsf{TO}} e'$. Thus, we add all forward edges of the graph in overall $O(\mathcal{T}^2\mathcal{L})$ time.

Checking if a given event e is in race with some event. We now have all the ingredients to describe our overall incremental algorithm to check if event e is in optimistic-sync-reversal race with some event of a given thread t (Algorithm 3). For this, we first initialize all the arrays $\{\mathsf{EIS}_{t_1,t_2}\}_{t_1,t_2 \in \mathsf{Threads}(\sigma)}$ using a linear scan of the trace σ, and also do pre-processing for fast performing range minima queries, spending overall time $O(\mathcal{T}\mathcal{N})$. Then, we iterate over each event e' of thread t that conflict with e, starting from the earliest to the latest. For each event, we incrementally update the optimistic lock-closure set S and check if it is lock-feasible. If so, we construct the abstract optimistic-reordering-graph G_S^{Abs} and check if it is acyclic, and report a race if so.

Algorithm 4: Detecting optimistic sync-reversal races in σ

```
procedure OSR(σ)
    for e ∈ Events(σ) s.t. e is a memory access event do
        for t′ ∈ Threads(σ) do
            incrementalRaceDetection(e, t′)
```

Theorem 4.2. Let σ be an execution, $e \in \mathsf{Events}(\sigma)$ be a read or write event and let $t \in \mathsf{Threads}(\sigma)$. The problem of checking if there is an event e' with $\mathsf{th}(e') = t$ such that (e, e') is an optimistic-sync-reversal race, can be solved in time $O\big((\mathcal{T}^2 + \mathcal{L})\mathcal{L}\mathcal{N}\big)$.

4.3 Detecting All Optimistic Sync-Reversal Races

Given a trace σ, all the optimistic sync-reversal races in σ can now be detected by enumerating all events e and threads t and checking if `incrementalRaceDetection`(e, t) reports a race. Our resulting algorithm OSR (Algorithm 4) runs in time $O\big(\mathcal{T}\mathcal{L}(\mathcal{T}^2 + \mathcal{L})\mathcal{N}^2\big)$.

Theorem 4.3. Given a trace σ, the problem of checking if σ has an optimistic sync-reversal data race, can be solved in time $O\big(\mathcal{T}\mathcal{L}(\mathcal{T}^2 + \mathcal{L})\mathcal{N}^2\big) = \widetilde{O}(\mathcal{N}^2)$ time.

Hardness of detecting optimistic sync-reversal races. We have, thus far, established that the problem of checking the existence of optimistic sync-reversal data races can be solved in quadratic time. In the following, we also show a matching quadratic time lower bound, thus establishing that our algorithm OSR is indeed optimal. The lower bound is conditioned on the Strong Exponential Time Hypothesis (SETH), which is a widely believed conjecture. We use fine-grained reductions to establish a reduction from the *orthogonal vectors problem* which holds true under SETH [72]. The full proof of the following result is presented in Appendix B.8.

Theorem 4.4. Assume SETH holds. Given an arbitrary trace σ, the problem of determining if σ has an OSR race cannot be solved in time $O(\mathcal{N}^{2-\epsilon})$ (where $\mathcal{N} = |\mathsf{Events}(\sigma)|$) for every $\epsilon > 0$.

5 EVALUATION

We implemented OSR in Java, using the Rapid dynamic analysis framework [4]. We evaluate the performance and precision of OSR, on 153 benchmarks and compare it with prior state-of-the-art sound predictive race detection algorithms. We discuss our experimental set up in Section 5.1 and evaluation results in Section 5.2, Section 5.3 and Section 5.4. The source code and traces are open sourced at [8].

5.1 Experimental Setup

Benchmarks. Our evaluation subjects are both Java (`Category-1`) as well as C/C++/OpenMP (`Category-2`) benchmarks. `Category-1`, derived from [47], contains 30 Java programs from the IBM Contest benchmark suite [26], the Java Grande forum benchmark suite [65], DaCapo [15], SIR [24] and other standalone benchmarks. `Category-2` contains 123 benchmarks from OmpSCR [25], DataRaceBench [41] DataRaceOnAccelerator [64], NAS parallel benchmarks [13], CORAL [5, 6], ECP proxy applications [1] and the Mantevo project [2]. For an apples-to-apples comparison, we evaluate all compared techniques on the same execution trace to remove bias due to thread-scheduler. For this, we generate traces out of these programs using ThreadSanitizer [66] (for `Category-2`) and using RVPredict [49] (for `Category-1`). For Java programs, we generate one trace per program and for C/C++ programs, we generate multiple traces of the same program with different thread number and input parameters. All compared methods then evaluate each generated trace 3 times. We did not exclude any traces from the benchmarks, except one corrupted trace.

As part of our evaluation, we also explored synthetically created benchmark traces from RaceInjector [3, 71], that uses SMT solving to inject data races into existing traces. However, the traces in [3] are short, could not be used to distinguish most compared methods and were not useful for a conclusive evaluation. Our evaluation on these traces is deferred to Appendix C (Table 4). As observed in prior works [20, 28, 47, 56], a large fraction of events in traces are thread-local, and do not affect the precision or soundness of race detection algorithms, but can significantly slow down race detection. Therefore, we filter out these thread-local events, as with prior work [36, 47, 56].

Compared methods. We compare OSR with state-of-the-art sound predictive algorithms: WCP [36], SHB [44], M2 [56] and SyncP [47]. Amongst these, SHB and WCP are partial order based methods and run in linear time. M2 and SyncP are closer in spirit to ours — they first identify a set of events and then a linearization of this set that can witness a data race. SyncP works in linear time while M2 has higher polynomial complexity of $\widetilde{O}(\mathcal{N}^4 \log(\mathcal{N}))$ [56]. For all these algorithms, we use the publicly available source codes [36, 44, 47, 56]. To achieve fair comparison, we modify each of them, so that (1) each algorithm reports on the same criteria (events v/s memory locations v/s program locations) (2) any redundant operations not relevant to the reporting criteria are removed. A comparison with recent work SeqC [20] was not possible because the implementation of SeqC is neither publicly available nor could be obtained even after contacting the authors. Our evaluation didn't include comparison with solver-aided race predictors, such as RVPredict [33]. Based on prior work [36], such predictors are known to not scale, have unpredictable race reports and typically have lower predictive power than the simplest of race prediction algorithms, thanks to the windowing strategy they implement.

Machine configuration and evaluation settings. The experiments are conducted on a 2.0GHz 64-bit Linux machine. For `Category-1` (Java) benchmarks, we set the heap size of JVM to be 60GB and timeout to be 2 hours; this set up is similar to previous works [36, 47], except for the larger heap space, mandated by the larger memory requirement of M2. For `Category-2` (C/C++) benchmarks, we set the heap size to be 400GB and timeout to be 3 hours, since these are much more challenging — the number of events, locks and variables in these are typically $10 - 100\times$ more than traces in `Category-1`. All experiments are repeated 3 times and the times reported are averaged over these 3 runs.

Reported metrics. Our evaluation aims to understand the prediction power (precision) as well as the scalability of OSR and assess how it compares against existing state-of-the-art race prediction techniques. For each execution trace , we report key characteristics (number of events, threads, locks, read events, write events, acquire events and release events) to estimate how challenging each benchmark is. Next, we measure and report the following :

Running time. For each algorithm, we report the average running time (over 3 trials) for processing the entire execution. This is aimed to understand if the worst case quadratic complexity of `OSR` affects its performance in practice, or it is on par with other linear time methods such as WCP, SHB and SyncP.

Race reports in `Category-1`*.* For benchmarks in `Category-1`, we report the number of racy events reported; an event e_2 is racy if there is a conflicting event e_1 earlier in the trace, such that (e_1, e_2) is a race. We also report the number of distinct source code lines for these racy events. We note here one racy source code line could correspond to many racy events.

Race reports in `Category-2`*.* For benchmarks in `Category-2`, we report the number of variables (memory locations) that are racy. A variable x is racy if there is a racy event e that accesses x. The number of racy events in the C/C++ benchmarks is typically very large, and reporting each racy event throttles nearly all algorithms. If a compared method times out, we report the number of racy variables found before timing out. This enables us to better evaluate their ability to find races in a more reasonable setting. Besides, most algorithms report many races before they timeout.

Scaling behavior of `OSR`*.* `OSR` runs in worst case quadratic time. We empirically evaluate how `OSR` scales with trace length, for a small set of benchmarks to gauge its in-practice behavior.

5.2 Evaluation Results For Java Benchmarks

Table 1 summarizes the results for `Category-1`.

Prediction power. OSR reports the largest number of races on each trace; it reports about 200 more racy events and 3 extra racy locations over the second most predictive method (SyncP); we remark that any extra data race can be an insidious bug [17] and deserves rigorous attention by developers. Although WCP can detect sync-reversal races in principle, and reports much fewer races than OSR (and also misses races reported by SyncP). M2 takes much more memory and time than OSR, and times out on two benchmarks (linkedlist and lufact), while runs out of memory on the benchmark tsp. On other benchmarks, OSR demonstrates the same prediction power as M2. Overall M2 detects 29.2k less races. In terms of racy source code locations, OSR also reports 24, 47, 3, 13 more than SHB, WCP, SyncP and M2, respectively. We remark that this class of benchmarks does not bring out the full potential of OSR— even if OSR reports the highest number of races individually for each benchmark, at least one other method also reports this number of races. `Category-2` though does better justice to OSR.

Running time. SHB and WCP are lightweight partial order-based linear time algorithms and finish fastest. On the other hand, M2 performs an expensive computation, times out on some large traces and takes more than 6 hours to finish. SyncP runs in linear time, but our algorithm OSR outperforms it by about 1.5×. We note that the linkedlist benchmark is especially challenging, with large number of variables, as a result of which SyncP allocates a large memory to account for its heavy data structure usage.

Thus, for `Category-1` benchmarks, OSR demonstrates highest race coverage, and runs faster than the state-of-the-art SyncP.

5.3 Evaluation Results For C/C++ Benchmarks

Table 2 summarizes our evaluation over `Category-2` (C/C++) benchmarks. In Appendix C, we present detailed statistics of these benchmarks (see Table 6 and Table 5).

Prediction power. OSR displays high race coverage on this set of traces. Overall, OSR reports 2.5× more races than the second most predictive method (SHB). On all, except 5, of the 118 benchmarks, OSR reports the highest number of racy variables. Each of the remaining 5 benchmark traces have a large number of events, and only the lightweight algorithms (SHB and WCP) finish within the 3 hour time limit. In terms of total races found, OSR reports 2.5× and 2.7× more races than SHB (2$^{\text{nd}}$ highest) and WCP (3$^{\text{rd}}$ highest). SyncP and M2 time out on most benchmarks. We speculate that this is because both these methods have high memory requirement and result in large time spent in garbage collection. OSR, therefore, has the highest race coverage even for the C/C++ benchmarks.

We remark that the number of racy variables in this class of benchmarks is very high. We speculate this is because our instrumentation using ThreadSanitizer does not explicitly tag atomic operations. Further many benchmarks perform matrix operations, giving rise to many distinct memory locations. Nevertheless, we choose to report all races because data races can render these programs potentially non-robust, and under weak memory consistency, data races can lead to undefined semantics.

Running time. Overall, SHB runs the fastest. SyncP and M2, on the other hand, frequently time out. The difference in the performance between SyncP, M2 and OSR gets exacerbated on the C/C++ benchmarks because these contain much larger execution traces than Java benchmarks. The performance of OSR (total running time of 42 hours) is close to WCP (30 hours). OSR, therefore, achieves an optimal balance between predictive power and scalability — OSR has the highest predictive power and outperforms SHB, WCP, SyncP, M2, and often runs faster than more exhaustive techniques.

5.4 Scalability

In this section, we take a closer look at the run-time behavior of OSR to understand its unexpected high scalability on some benchmarks. We select the most challenging benchmarks from each of the following groups: HPCBench, CoMD, DataRaceBench, OMPRacer in `Category-2`. For these benchmarks, we measure the time to process every million events and report it in Figure 6. We observe that on these four benchmarks, OSR scales linearly for a large prefix, while gradually slows down on two of them. The near-linear behavior of OSR is likely an artefact of the fact that, many of these benchmarks traces have large number of data races, thus the race check for a single event succeeds quickly instead of the worst case linear time requirement. Therefore, instead of spending overall quadratic time, OSR spends linear time on average.

6 RELATED WORK

Dynamic predictive analysis. Happens-before (HB) [39] based race detection [28, 57] has been adopted by mature tools [51, 66],

Table 1: Evaluation on `Category-1` (Java benchmarks). Columns 1-3 denote the name, number of events and number of threads for each benchmark. Columns 4-13 are the number of racy events (and racy program locations) reported and average running time of each algorithm.

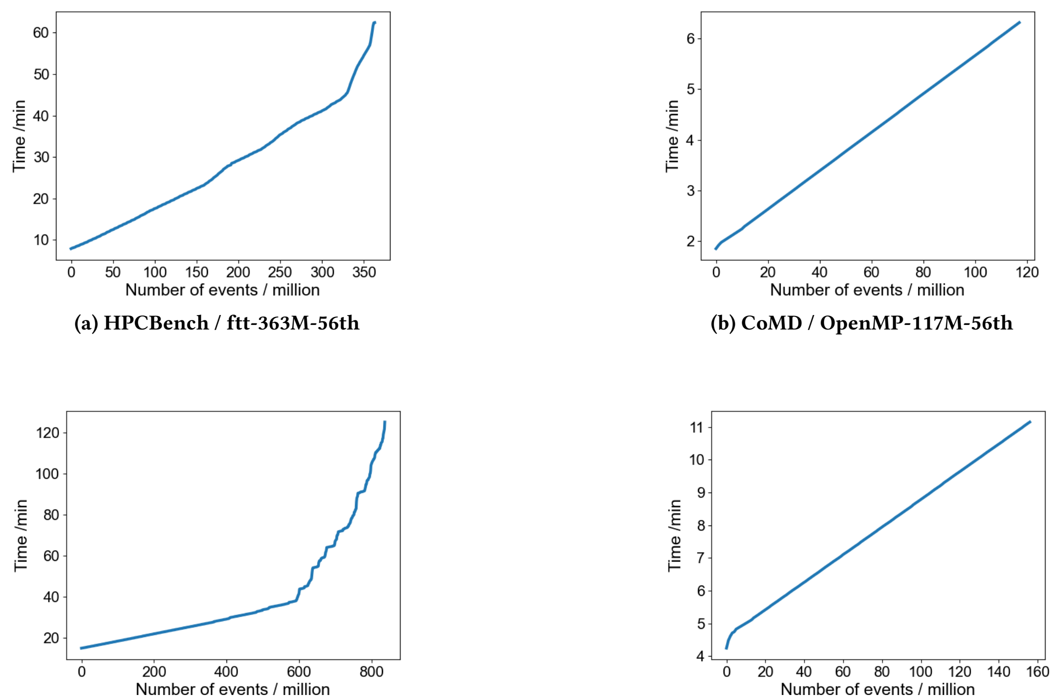
1	2	3	4	5	6	7	8	9	10	11	12	13
Benchmarks	$\mathcal{N}$	$\mathcal{T}$	SHB		WCP		SyncP		M2		OSR	
			Races	Time (s)	Races	Time (s)	Races	Time (s)	Races	Time (s)	Races	Time (s)
array	11	3	0(0)	0.05	0(0)	0.08	0(0)	0.06	0(0)	0.03	0(0)	0.09
critical	11	4	3(3)	0.04	1(1)	0.05	3(3)	0.07	3(3)	0.02	3(3)	0.07
account	15	4	3(1)	0.04	3(1)	0.06	3(1)	0.06	3(1)	0.02	3(1)	0.08
airtickets	18	5	8(3)	0.05	5(2)	0.08	8(3)	0.06	8(3)	0.03	8(3)	0.08
pingpong	24	7	8(3)	0.04	8(3)	0.07	8(3)	0.06	8(3)	0.03	8(3)	0.08
twostage	83	12	4(1)	0.06	4(1)	0.10	4(1)	0.14	8(2)	0.05	8(2)	0.10
wronglock	122	22	12(2)	0.07	3(2)	0.11	25(2)	0.22	25(2)	0.18	25(2)	0.13
bbuffer	9	3	3(1)	0.05	1(1)	0.06	3(1)	0.05	3(1)	0.02	3(1)	0.10
prodcons	246	8	1(1)	0.07	1(1)	0.13	1(1)	0.16	1(1)	0.06	1(1)	0.12
clean	867	8	59(4)	0.11	82(4)	0.23	60(4)	0.26	110(4)	0.65	110(4)	0.20
mergesort	167	5	1(1)	0.89	1(1)	0.13	3(1)	0.10	5(2)	0.04	5(2)	0.12
bubblesort	1.7K	13	269(5)	0.15	100(5)	0.30	269(5)	2.29	374(5)	8.40	374(5)	0.28
lang	1.8K	7	400(1)	0.17	400(1)	0.26	400(1)	0.33	400(1)	0.54	400(1)	0.22
readwrite	9.8K	5	92(4)	0.27	92(4)	0.63	199(4)	0.81	228(4)	9.00	228(4)	0.69
raytracer	526	3	8(4)	0.10	8(4)	0.17	8(4)	0.15	8(4)	0.09	8(4)	0.15
bufwriter	10K	6	8(4)	0.29	8(4)	0.77	8(4)	0.75	8(4)	0.52	8(4)	0.49
ftpserver	17K	11	69(21)	1.18	70(21)	0.99	85(21)	6.01	85(21)	2.43	85(21)	0.79
moldyn	21K	3	103(3)	1.03	103(3)	0.73	103(3)	0.79	103(3)	31.43	103(3)	0.46
linkedlist	910K	12	6.0K(4)	3.77	6.0K(3)	6.80	7.1K(4)	378.25	0(0)	7200	7.1K(4)	6.56
derby	75K	4	29(10)	0.94	28(10)	2.30	29(10)	19.08	30(11)	5.66	30(11)	3.67
jigsaw	3.2K	8	4(4)	0.17	4(4)	0.39	6(6)	2.90	6(6)	0.23	6(6)	0.35
sunflow	3.3K	17	84(6)	0.17	69(6)	0.39	119(7)	2.53	130(7)	1.10	130(7)	0.35
cryptorsa	1.3M	7	11(5)	5.95	11(5)	10.87	35(7)	156.19	35(7)	20.39	35(7)	173.74
xalan	672K	7	31(10)	3.22	21(7)	12.07	37(12)	160.62	37(12)	6.56	37(12)	230.03
lufact	892K	5	22.0K(3)	3.39	22.0K(3)	7.16	22.0K(3)	62.10	0(0)	7200	22.0K(3)	4.15
batik	131	7	10(2)	0.09	10(2)	0.11	10(2)	0.12	10(2)	0.04	10(2)	0.12
lusearch	751K	8	232(44)	2.86	119(27)	7.94	232(44)	9.26	232(44)	50.4	232(44)	3.65
tsp	15M	10	143(6)	33.63	140(6)	66.07	143(6)	146.24	0(0)	7200	143(6)	160.39
luindex	16K	3	1(1)	0.38	2(2)	0.68	15(15)	0.71	15(15)	0.53	15(15)	0.49
sor	1.9M	5	0(0)	4.79	0(0)	9.92	0(0)	13.16	0(0)	10.61	0(0)	38.0
Sum			29.5K(157)	64.0	29.2K(134)	129.7	30.9K(178)	961.9	1.9K(168)	6.0h	31.1K(181)	625.7

and has subsequently been strengthened to SHB [44] so that all races reported are sound. Causal Precedence (CP) [67] and Weak Causal Precedence (WCP) [36] weaken HB in favor of predictive power, and run in polynomial and linear time, respectively. Other works such as DC [58, 60] and SDP [31] are also partial order based methods that are either sound by design or perform graph-based analysis to regain soundness. SyncP [47], M2 [56], SeqCheck [20] work similar to OSR, by constructing an appropriate set of events and appropriate linearization over this set. SMT solver backed approaches [33, 62] aim for sound and complete race prediction but do not scale to moderately large execution traces. The complexity of data race prediction was extensively studied in [46] and was shown to be NP-hard and also W[1]-hard, implying that an FPT algorithm (parameterized by the number of threads) for race prediction is unlikely. The fine-grained complexity of HB and SyncP was studied in [38]; in practice, HB can be sped up using the tree clock data structure [45]. Predictive analyses have also been developed for deadlocks [35, 70], atomicity violations [48, 68], for more general temporal specifications [12] and more recently has been investigated from the lens of generalizing trace equivalence [27].

Other concurrency testing approaches. Static analysis techniques employ forms of lockset style reasoning [63] to detect data races [16, 40, 53, 75] to report data races, but are known to report false positives. Model checking techniques for concurrent software [10, 37, 54] have been employed to detect concurrency bugs [32, 55]. Another class of systematic exploration techniques include controlled concurrency testing [11, 23], including those that employ randomization [19, 42, 74] and state-based learning [52]. More recently, feedback driven randomized techniques have been

Table 2: Evaluation summary on `Category-2` (C/C++ benchmarks). Benchmarks are grouped based on their source, and each row corresponds to one group. Column 1 denotes the source and size of each group. Columns 2 and 3 respectively denote the range and the total number of events in each group. Column 4 denotes the range of number of threads in the benchmarks. Column 5-14 denote the total number of racy memory locations, and average running time (in minutes) reported by each algorithm.

1	2	3	4	5	6	7	8	9	10	11	12	13	14
Benchmark Group	$\mathcal{N}$		$\mathcal{T}$	SHB		WCP		SyncP		M2		OSR	
	Range	Total		Races	Time	Races	Time	Races	Time	Races	Time	Races	Time
CoMD (8)	[2.5M, 117M]	707M	[16, 56]	41.6k	29.1	247k	72.3	32	1440	672	1440	441k	32.4
SimpleMOC (1)	[19M, 19M]	19M	[16, 16]	380	0.1	388	23.8	32	180	32	180	32	180
OMPRacer (15)	[0.7M, 157M]	625M	[16, 58]	1.2M	17.4	0.7M	84.1	3.3k	2.4k	1.9k	2.1k	1.3M	35.8
DRACC (13)	[0.5k, 104M]	694M	[16, 16]	2247	8.4	2247	105.7	2442	1440.5	361	990.1	2450	66.6
DRB (33)	[0.5k, 900M]	5.7B	[16, 56]	50.4k	169.5	54.5k	0.6k	1.5k	5.4k	0.9k	4.9k	47.4k	1.4k
HPC (46)	[1k, 335M]	3.8B	[16, 56]	6.5M	174.2	6.4M	775.4	102k	7.7k	3305	6.8k	18.3M	574.8
misc (7)	[1k, 29M]	49M	[4, 219]	8548	0.9	8481	182.2	479	900.9	895	444.6	4289	183.4
Total (123)		11.6B		7.9M	6.7h	7.4M	30.3h	109k	324.4h	8.1k	280.5h	20.1M	41.2h



(a) HPCBench / ftt-363M-56th

(b) CoMD / OpenMP-117M-56th

(c) DataRaceBench / DRB177-837M-16th

(d) OMPRacer / Lulesh-157M-56th

Figure 6: Time spent to process every million events for 4 selected traces.

employed for testing concurrent programs [34, 73] Randomization has also been shown to reduce time overhead of dynamic data race detection [18, 43, 69].

7 CONCLUSIONS AND FUTURE WORK

We propose OSR, a sound polynomial time race prediction algorithm that identifies data races that can be witnessed by *optimistically reversing synchronization operations*. OSR significantly advances the state-of-the-art in sound dynamic data race prediction. OSR-style reasoning can be helpful for exposing other concurrency bugs such as deadlocks [35, 70] and atomicity violations.

ACKNOWLEDGMENTS

This work is partially supported by the National Research Foundation, Singapore, and Cyber Security Agency of Singapore under its National Cybersecurity R&D Programme (Fuzz Testing <NRF-NCR25-Fuzz-0001>) and by a research grant (VIL42117) from VILLUM FONDEN. Any opinions, findings and conclusions, or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore, and Cyber Security Agency of Singapore.

REFERENCES

[1] [n. d.]. ECP Proxy Applications. https://proxyapps.exascaleproject.org/. Accessed: 2021-08-01.

[2] [n. d.]. Mantevo Project. https://mantevo.org/. Accessed: 2021-08-01.

[3] [n. d.]. RaceInjector traces. https://github.com/ALFA-group/RaceInjector-counterexamples/tree/main. Accessed: 2023-07-14.

[4] [n. d.]. RAPID. https://github.com/umangm/rapid. Accessed: 2023-07-06.

[5] 2014. *CORAL Benchmarks*. Accessed: 2021-08-01.

[6] 2014. *CORAL2 Benchmarks*. Accessed: 2021-08-01.

[7] 2023. Range Minima Query Solutions. https://en.wikipedia.org/wiki/Range_minimum_query. Accessed: 2023-07-18.

[8] 2024. OSR implementation. https://zenodo.org/records/10437347. Accessed: 2024-01-11.

[9] 2024. OSR technical report. https://arxiv.org/abs/2401.05642. Accessed: 2024-01-12.

[10] Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. 2014. Optimal Dynamic Partial Order Reduction. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 373–384. https://doi.org/10.1145/2535838.2535845

[11] Udit Agarwal, Pantazis Deligiannis, Cheng Huang, Kumseok Jung, Akash Lal, Immad Naseer, Matthew Parkinson, Arun Thangamani, Jyothi Vedurada, and Yunpeng Xiao. 2021. Nekara: Generalized Concurrency Testing. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 679–691. https://doi.org/10.1109/ASE51524.2021.9678838

[12] Zhendong Ang and Umang Mathur. 2024. Predictive Monitoring against Pattern Regular Languages. *Proc. ACM Program. Lang.* 8, POPL, Article 73 (jan 2024). https://doi.org/10.1145/3632915

[13] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. 1991. The NAS parallel benchmarks—summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*. 158–165.

[14] RICHARD BELLMAN. 1958. ON A ROUTING PROBLEM. *Quart. Appl. Math.* 16, 1 (1958), 87–90. http://www.jstor.org/stable/43634538

[15] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. 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*. 169–190.

[16] Sam Blackshear, Nikos Gorogiannis, Peter W O'Hearn, and Ilya Sergey. 2018. RacerD: compositional static race detection. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–28.

[17] 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*. 9–14.

[18] 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)*. Association for Computing Machinery, New York, NY, USA, 255–268. https://doi.org/10.1145/1806596.1806626

[19] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. 2010. A randomized scheduler with probabilistic guarantees of finding bugs. *ACM SIGARCH Computer Architecture News* 38, 1 (2010), 167–178.

[20] Yan Cai, Hao Yun, Jinqiu Wang, Lei Qiao, and Jens Palsberg. 2021. Sound and efficient concurrency bug prediction. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 255–267.

[21] Milind Chabbi and Murali Krishna Ramanathan. 2022. A Study of Real-World Data Races in Golang. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 474–489. https://doi.org/10.1145/3519939.3523720

[22] Lijie Chen and Ryan Williams. 2019. An equivalence class for orthogonal vectors. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 21–40.

[23] Pantazis Deligiannis, Aditya Senthilnathan, Fahad Nayyar, Chris Lovett, and Akash Lal. 2023. Industrial-Strength Controlled Concurrency Testing for C# Programs with COYOTE. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 433–452.

[24] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. 2005. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering* 10 (2005), 405–435.

[25] Antonio J Dorta, Casiano Rodriguez, and Francisco de Sande. 2005. The OpenMP source code repository. In *13th Euromicro Conference on Parallel, Distributed and Network-Based Processing*. IEEE, 244–250.

[26] Eitan Farchi, Yarden Nir, and Shmuel Ur. 2003. Concurrent bug patterns and how to test them. In *Proceedings international parallel and distributed processing symposium*. IEEE, 7–pp.

[27] Azadeh Farzan and Umang Mathur. 2024. Coarser Equivalences for Causal Concurrency. *Proc. ACM Program. Lang.* 8, POPL, Article 31 (jan 2024). https://doi.org/10.1145/3632873

[28] 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)*. Association for Computing Machinery, New York, NY, USA, 121–133. https://doi.org/10.1145/1542476.1542490

[29] Lester Randolph Ford. 1956. Network flow theory. (1956).

[30] Harold N Gabow, Jon Louis Bentley, and Robert E Tarjan. 1984. Scaling and related techniques for geometry problems. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*. 135–143.

[31] Kaan Genç, Jake Roemer, Yufan Xu, and Michael D Bond. 2019. Dependence-aware, unbounded sound predictive race detection. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–30.

[32] Patrice Godefroid. 2005. Software model checking: The VeriSoft approach. *Formal Methods in System Design* 26 (2005), 77–101.

[33] 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*. 337–348.

[34] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. 2019. Razzer: Finding kernel race bugs through fuzzing. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 754–768.

[35] 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

[36] 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)*. Association for Computing Machinery, New York, NY, USA, 157–170. https://doi.org/10.1145/3062341.3062374

[37] Michalis Kokologiannakis and Viktor Vafeiadis. 2021. GenMC: A model checker for weak memory models. In *International Conference on Computer Aided Verification*. Springer, 427–440.

[38] Rucha Kulkarni, Umang Mathur, and Andreas Pavlogiannis. 2021. Dynamic Data-Race Detection Through the Fine-Grained Lens. In *32nd International Conference on Concurrency Theory*.

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

[40] Yanze Li, Bozhen Liu, and Jeff Huang. 2019. Sword: A scalable whole program race detector for java. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 75–78.

[41] Chunhua Liao, Pei-Hung Lin, Joshua Asplund, Markus Schordan, and Ian Karlin. 2017. DataRaceBench: a benchmark suite for systematic evaluation of data race detection tools. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[42] Weiyu Luo and Brian Demsky. 2021. C11Tester: a race detector for C/C++ atomics. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 630–646.

[43] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. 2009. LiteRace: Effective sampling for lightweight data-race detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 134–143.

[44] 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. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–29.

[45] Umang Mathur, Andreas Pavlogiannis, Hünkar Can Tunç, and Mahesh Viswanathan. 2022. A Tree Clock Data Structure for Causal Orderings in Concurrent Executions. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Lausanne Switzerland, 710–725. https://doi.org/10.1145/3503222.3507734

[46] 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*. 713–727.

[47] Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. 2021. Optimal prediction of synchronization-preserving races. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–29.

[48] 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

[49] Patrick Meredith and Grigore Roşu. 2010. Runtime verification with the RV system. In *International Conference on Runtime Verification*. Springer, 136–152.

[50] Edward F. Moore. 1959. The shortest path through a maze. In *Proc. Internat. Sympos. Switching Theory 1957, Part II*. Harvard Univ. Press, Cambridge, Mass., 285–292.

[51] Arndt Müehlenfeld and Franz Wotawa. 2007. Fault Detection in Multi-threaded C++ Server Applications. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Jose, California, USA) *(PPoPP '07)*. ACM, New York, NY, USA, 142–143. https://doi.org/10.1145/1229428.1229457

[52] Suvam Mukherjee, Pantazis Deligiannis, Arpita Biswas, and Akash Lal. 2020. Learning-based controlled concurrency testing. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–31.

[53] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective static race detection for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 308–319.

[54] Brian Norris and Brian Demsky. 2013. CDSchecker: Checking Concurrent Data Structures Written with C/C++ Atomics. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (Indianapolis, Indiana, USA) *(OOPSLA '13)*. Association for Computing Machinery, New York, NY, USA, 131–150. https://doi.org/10.1145/2509136.2509514

[55] Jonas Oberhauser, Rafael Lourenco de Lima Chehab, Diogo Behrens, Ming Fu, Antonio Paolillo, Lilith Oberhauser, Koustubha Bhat, Yuzhong Wen, Haibo Chen, Jaeho Kim, et al. 2021. VSync: push-button verification and optimization for synchronization primitives on weak memory models. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 530–545.

[56] Andreas Pavlogiannis. 2019. Fast, sound, and effectively complete dynamic race prediction. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–29.

[57] 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*. 179–190.

[58] 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*. 747–762.

[59] 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)*. Association for Computing Machinery, New York, NY, USA, 374–389. https://doi.org/10.1145/3192366.3192385

[60] 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)*. Association for Computing Machinery, New York, NY, USA, 374–389. https://doi.org/10.1145/3192366.3192385

[61] Caitlin Sadowski and Jaeheon Yi. 2014. How Developers Use Data Race Detection Tools. In *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools* (Portland, Oregon, USA) *(PLATEAU '14)*. Association for Computing Machinery, New York, NY, USA, 43–51. https://doi.org/10.1145/2688204.2688205

[62] Mahmoud Said, Chao Wang, Zijiang Yang, and Karem Sakallah. 2011. Generating data race witnesses by an SMT-based analysis. In *NASA Formal Methods Symposium*. Springer, 313–327.

[63] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)* 15, 4 (1997), 391–411.

[64] Adrian Schmitz, Joachim Protze, Lechen Yu, Simon Schwitanski, and Matthias S Müller. 2019. DataRaceOnAccelerator–a micro-benchmark suite for evaluating correctness tools targeting accelerators. In *European Conference on Parallel Processing*. Springer, 245–257.

[65] Koushik Sen, Grigore Roşu, and Gul Agha. 2005. Detecting errors in multithreaded programs by generalized predictive analysis of executions. In *Formal Methods for Open Object-Based Distributed Systems: 7th IFIP WG 6.1 International Conference, FMOODS 2005, Athens, Greece, June 15-17, 2005. Proceedings 7*. Springer, 211–226.

[66] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: data race detection in practice. In *Proceedings of the workshop on binary instrumentation and applications*. 62–71.

[67] 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)*. Association for Computing Machinery, New York, NY, USA, 387–400. https://doi.org/10.1145/2103656.2103702

[68] 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)*. Association for Computing Machinery, New York, NY, USA, 37–46. https://doi.org/10.1145/1882291.1882300

[69] Mosaad Al Thokair, Minjian Zhang, Umang Mathur, and Mahesh Viswanathan. 2023. Dynamic Race Detection with O (1) Samples. *Proceedings of the ACM on Programming Languages* 7, POPL (2023), 1308–1337.

[70] Hünkar Can Tunç, Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. 2023. Sound Dynamic Deadlock Prediction in Linear Time. *Proc. ACM Program. Lang.* 7, PLDI, Article 177 (jun 2023), 26 pages. https://doi.org/10.1145/3591291

[71] Michael Wang, Shashank Srikant, Malavika Samak, and Una-May O'Reilly. 2023. RaceInjector: Injecting Races to Evaluate and Learn Dynamic Race Detection Algorithms. In *Proceedings of the 12th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*. 63–70.

[72] Ryan Williams. 2005. A new algorithm for optimal 2-constraint satisfaction and its implications. *Theoretical Computer Science* 348, 2 (2005), 357–365. https://doi.org/10.1016/j.tcs.2005.09.023 Automata, Languages and Programming: Algorithms and Complexity (ICALP-A 2004).

[73] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. 2020. Krace: Data race fuzzing for kernel file systems. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1643–1660.

[74] Xinhao Yuan, Junfeng Yang, and Ronghui Gu. 2018. Partial order aware concurrency sampling. In *Computer Aided Verification: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II 30*. Springer, 317–335.

[75] Sheng Zhan and Jeff Huang. 2016. ECHO: instantaneous in situ race detection in the IDE. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 775–786.