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 non-determinism 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 \( \sigma \) of an execution of a concurrent program, can \( \sigma \) 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, 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:

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].

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 \( \sigma \), and then look for a reordering \( \rho \) in this space, that can serve as a witness to a bug such as a data race. Consider the execution \( \sigma_1 \) 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 \( \sigma_1 \). However, the correct reordering \( \rho_1 \) of \( \sigma_1 \), in which the two write events appear consecutively, shows that it is nevertheless, a predictable data race of \( \sigma_1 \). Indeed any program \( P \) that generates \( \sigma_1 \) will also generate \( \rho_1 \) 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 synchronization. The data race \((e_1,e_2)\) 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 \(t\). 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 \(O(N^2)\), spending \(O(N)\) time for processing each event in the given execution trace \(\sigma\). Here, \(N\) is the number of events in \(\sigma\) and \(O\) 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 \(O(N)\) 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) \(\sigma\) of a concurrent program is a sequence of events \(\sigma = e_1 \ldots e_N\). An event is a tuple \(e = (i, t, op)\), 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 \(\text{th}(e)\) and \(\text{op}(e)\) to denote the thread and operation of \(e\). Operations are \(r(x), \text{w}(x)\) (read or write access of memory location or variable \(x\)) or \(\text{acq}(t), \text{rel}(t)\) (acquire or release of lock \(t\)); fork and join operations are omitted from presentation but not from our implementation. For a trace \(\sigma\), we will use \(\text{Events}(\sigma)\), \(\text{Threads}(\sigma)\), \(\text{Vars}(\sigma)\), \(\text{Locks}(\sigma)\) to denote respectively the set of all events, threads, variables and locks appearing in \(\sigma\).

### Well-formedness.
We assume that traces are well-formed, in that they do not violate lock semantics. In particular, for a well formed trace \(\sigma\), we require that for each lock \(t \in \text{Locks}(\sigma)\), the sequence of operations on \(t\) 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 \(\text{match}_a(e)\) to denote the matching release (resp. acquire) event of \(e\) if one exists; otherwise we say \(\text{match}_a(e) = \perp\).

### Trace order, thread order and reads-from.
The trace order \(\preceq_T\) of a trace \(\sigma\) is the total order induced by the sequence of events in \(\sigma\), i.e., \(e_1 \preceq_T e_2\) if \(e_1\) appears earlier than \(e_2\) in \(\sigma\). The thread order \(\preceq_T\) is a partial order on \(\text{Events}(\sigma)\) such that for any two events \(e_1, e_2\), we have \(e_1 \preceq_T e_2\) if \(\text{th}(e_1) = \text{th}(e_2)\) and \(e_1 \preceq_T e_2\). 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 \(r \in \text{Events}(\sigma)\) with \(\text{op}(r) = r(x)\) for some variable \(x\), the writer of \(r\), denoted \(w = \text{rf}_\sigma(r)\) is the last write event on \(x\) before \(r\), i.e., \(\text{op}(w) = w(x), w \preceq_T r\) and \(\neg(\exists w' : w \neq w' \land \text{op}(w') = w(x) \land w \preceq_T w' \preceq_T r)\). Without loss of generality, we will assume that \(\text{rf}_\sigma(e)\) is always defined for each read event \(e\). Given a set \(S \subseteq \text{Events}(\sigma)\), we say that \(S\) is \((\leq_T, \text{rf}_\sigma)\)-closed if (a) for all events \(e_1, e_2 \in \text{Events}(\sigma)\) if \(e_1 \leq_T e_2 \land e_2 \in S\), then \(e_1 \in S\); and (b) for all events \(\forall e_1, e_2 \in \text{Events}(\sigma)\), if \(e_1 = \text{rf}_\sigma(e_2) \land e_2 \in S\), then \(e_1 \in S\). We use \(\text{TRC}(S)\) to denote the smallest set \(S'\) such that \(S \subseteq S'\) and \(S'\) is \((\leq_T, \text{rf}_\sigma)\)-closed.

### Correct reordering.
Predictive race detection, given a trace \(\sigma\), asks if an alternate execution trace \(\rho\) witnesses a data race, and
more importantly, $p$ can be inferred from $\sigma$. The notion of correct reorderings precisely formalizes this. Given well-formed traces $\sigma$ and $p$, with $\text{Events}(p) \subseteq \text{Events}(\sigma)$, we say that $p$ is a correct reordering of $\sigma$ if $p$ respects the thread order and reads-from relations of $\sigma$. This means that (1) $\text{Events}(p)$ is $(\leq_{\text{TO}}^{r_{\text{el}}})$-closed, (2) for any two events $e_1, e_2 \in \text{Events}(p)$, if $e_1 \leq_{\text{TO}}^{r_{\text{el}}} e_2$, then $e_1 \leq_{\text{TO}}^{r_{\text{el}}} e_2$; and (3) for any two events $e_1, e_2 \in \text{Events}(p)$, if $e_1 = r_{\sigma}(e_2)$, then $e_1 = r_{p}(e_2)$.

### Data races and predictable data races

A pair of events $(e, e')$ in $\sigma$ is said to be a conflicting pair, denoted $e \Rightarrow e'$, if both are access events to the same variable, and at least one of them is a write event, i.e., $(\text{op}(e), \text{op}(e')) \in \{(w(x), w(x)), (w(x), r(x)), (r(x), w(x))\}$ for some $x \in \text{Vars}(\sigma)$. For a trace $\pi$ with $\text{Events}(\pi) \subseteq \text{Events}(\sigma)$, we say that event $e$ is $\sigma$-enabled in $\pi$ if $e \not\in \text{Events}(\pi)$ but all thread-predecessors of $e$ are in $\pi$, i.e., $(e' \in \text{Events}(\sigma) \mid e' \neq e, e' \leq_{\text{TO}}^{r_{\text{el}}} e) \subseteq \text{Events}(\pi)$. A conflicting pair $(e, e')$ is said to be a data race of $\sigma$ if there is a prefix $\pi$ of $\sigma$ such that both $e$ and $e'$ are $\sigma$-enabled in $\pi$. Finally, a conflicting pair $(e, e')$ is a predictable data race of $\sigma$ if there is a correct reordering $p$ of $\sigma$ such that both $e$ and $e'$ are $\sigma$-enabled in some prefix of $p$. In this case, we say that $p$ witnesses the data race $(e, e')$.

### Example 1

Consider trace $\sigma_2$ in Figure 2a containing 6 events performed by two threads $t_1$ and $t_2$. As before, we use $t_i$ to denote the $i$th event of $\sigma_2$. The two events $e_1 = (t_1, w(x))$ and $e_5 = (t_2, w(x))$ are conflicting (i.e., $e_1 \Rightarrow e_5$). The pair $(e_1, e_5)$ is not a data race in $\sigma_2$ as no prefix of $\sigma_2$ has both these events simultaneously enabled. Consider the trace $p_2$ in Figure 2h; it is a correct reordering of $\sigma_2$ because it preserves both the thread order and read-from relation of $\sigma_2$. For the same reason, $p_2'$ is a correct reordering of $\sigma_2$ (and also of $p_2$). Now, observe that $(e_1, e_5)$ is a data race in $p_2$ (and also in $p_2'$) because in the prefix $\pi = (t_2, \text{acq}(f))$, both $e_1$ and $e_5$ are $p_2$-enabled (resp. $p_2'$-enabled) and thus $\sigma_2$-enabled. Thus, while $(e_1, e_5)$ is not a data race in $\sigma_2$, it is a predictable data race of $\sigma_2$.

### Problem of predicting data races

The problem of predicting data races — given an execution trace $\sigma$, determine if there is a predictable data race of $\sigma$ — 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-preserven 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 the data races witnessed by such reorderings; we discuss these next.

#### Sync-preserving reorderings and data races

A correct reordering $p$ of a trace $\sigma$ is said to be sync-preserving if for any two critical sections of $\sigma$ (on the same lock) that are both present in $p$, their relative order is the same. That is, for every lock $\ell \in \text{Locks}(\sigma)$ and for any two acquire events $a_1, a_2 \in \text{Events}(\sigma)$ such that $\text{op}(a_1) = \text{op}(a_2) = \text{acq}(\ell)$, if $a_1, a_2 \in \text{Events}(p)$, then we have: $a_1 \leq_{\text{TO}}^{r_{\text{el}}} a_2$ iff $a_1 \leq_{\text{TO}}^{r_{\text{el}}} a_2$. A pair of conflicting events $(e, e')$ in $\text{Events}(\sigma)$ is said to be a sync-preserving data race of $\sigma$ if there is a sync-preserving correct reordering $p$ of $\sigma$ that witnesses this race.

#### Example 2

Consider again, the trace $\sigma_2$ and recall from Example 1 that the pair $(e_1, e_5)$ is not a data race of $\sigma_2$ but a predictable race witnessed by the correct reordering $p_2$. Observe however that $p_2$ is not a sync-preserving reordering of $\sigma_2$ because it flips the order of the two critical sections on lock $\ell$. Nevertheless, $(e_1, e_5)$ is a sync-preserving race of $\sigma_2$. This is because the reordering $p_2'$ is, in fact, a sync-preserving reordering of $\sigma_2$ (even though it is a prefix of the non-sync-preserving reordering $p_2$); there is only one critical section in $p_2'$ and thus vacuously, the relative order on critical sections is the same as in $\sigma_2$.

#### 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 $\sigma_1$ in Figure 1a (Section 1). The two conflicting events $e_1 = (t_1, w(x))$ and $e_12 = (t_4, w(x))$, are a predictable data race of $\sigma_1$ as witnessed by the correct reordering $p_1$ in Figure 1b, which is not a sync-preserving correct reordering of $p_1$. In fact, consider any correct reordering $a$ of $\sigma_1$ that witnesses the race $(e_1, e_12)$. Then $a$ 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 $e_3 = (t_2, \text{acq}(f))$ and $e_7 = (t_3, \text{acq}(f))$. Next, for well-formedness, at least one of the matching releases $e_6 = (t_2, \text{rel}(f))$ as well as $e_9 = (t_3, \text{rel}(f))$ must also be present in $a$. However, including $e_6$ in $a$ would enforce that $e_5 = (t_2, r(z))$, and its write event $e_2 = (t_1, w(y))$ are present in $a$, and then, the event $e_1$ must also be present in the reordering making it no longer enabled in $a$. This, therefore, means that $e_6 \not\in \text{Events}(a)$, and thus, the only other available release event $e_6$ must be present in $a$ (for well-formedness). Further, to ensure well-formedness, $e_3$ must appear after $e_9$ in $a$. Thus, any reordering $\sigma_1$ 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 $O(N^3)$ 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 $\sigma$.

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

Example 4. Let us recall trace $\sigma_1$ 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 $(e_1, e_2)$, because (1) $S_1$ doesn’t include either of $e_1, e_2$, (2) $S_1$ is $(\leq, \text{TR}^\prime)$-closed, and finally, (3) $e_1, e_2 \notin \text{TRClosure}(e_9)$. Note that $e_1 \in \text{TRClosure}(e_9)$ 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 $\sigma$ be a trace, let $e_1, e_2$ be conflicting events and let $S \subseteq \text{Events}(\sigma)$ be an optimistically lock-closed set with respect to $(e_1, e_2)$. The problem of determining whether there is a correct reordering $\rho$ such that $\text{Events}(\rho) = S$ is $\text{NP}$-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 $\rho$ is said to be an optimistic correct reordering of $\sigma$ if (a) $\rho$ is a correct reordering of $\sigma$, (b) for all pairs of conflicting memory access events $e_1 \Rightarrow e_2$ in $\text{Events}(\rho)$, $e_1 \not\leq \rho e_2$ if $e_1 \not\leq \sigma e_2$, and (c) for any lock $f$ and for any two acquire events $a_1 \neq a_2$ (with $\text{op}(a_1) = \text{op}(a_2) = \text{acq}(f)$), if $a_1$ and $a_2$ are both matched in $\rho$ (i.e., $\text{match}_\sigma(a_1) \in \text{Events}(\rho)$ for both $i \in \{1, 2\}$), then we must have $a_1 \leq \sigma a_2$, $a_2 \leq \rho a_1$.

We now formally optimize sync-reversal data races.

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

Example 5. In Figure 1, the pair $(e_1, e_{12})$ is an optimistic sync-reversal race, because the prefix $\rho_1'$ with first 7 events of $\rho_1$ is an optimistic reordering of the optimistically lock-closed set $S_1$, outlined in Example 4, (in which $e_1$ and $e_{12}$ are $\sigma_1$-enabled). This is because, all conflicting accesses of $\rho_1'$ have the same relative order as $e_1$ and further, the unmatched acquire event is positioned after all closed critical sections. Similarly, for the trace $\sigma_2$ of Figure 2, the linearization $\rho_2' = \{t_2, \text{acq}(f)\}$ 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 $\sigma_3$ 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 $S = \{e_1, e_2, e_4, e_5, e_6, e_7, e_8, e_{10}, e_{11}\}$ 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 $S_{\text{OSR}} = \{e_{13}, e_{12}, e_{14}, e_{17}, e_{20}\}$, where $e_{ij}$ is shorthand for $e_i, e_{i+1}, \ldots, e_{j-1}, e_j$. Observe that $S_{\text{OSR}}$ contains two unmatched acquire events of lock $e_2$ 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
We now describe our algorithm OSR witness of race in (a). The other one (c) can be detected by M2, but not OSR. Trace in (a) is directly cited from M2 paper [56] without modification. (d) is the witness of race in (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 O(N) time, where N is the number of events in the given trace. Naively, it can be used to report all optimistic sync-reversal data races in O(N^2) time, by enumerating all O(N^2) pairs of conflicting events and checking each of them in O(N) time. Instead, OSR runs in overall O(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 OLClosure(e_1, e_2) to denote the set OLClosure(S_{e_1,e_2}, e_1, e_2), where S_{e_1,e_2} = \{prev_{e_1}(e_2)\} \cup \{prev_{e_2}(e_1)\}. Here, prev_{e}(e) is the last event f such that f \leq e^{TO} e, if no such event exists, we say prev_{e}(e) = \perp, in which case \{prev_{e}(e)\} = \emptyset.

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

**Algorithm 1:** Computing optimistic lock closure

1. procedure ComputeOLClosure(S_0, e_1, e_2)
2. \begin{align*}
S & \leftarrow S_0 \cup TRClosure(prev_{\sigma}(e_1)) \cup TRClosure(prev_{\sigma}(e_2)) \\
\text{while } S \text{ changes do} & \begin{align*}
\text{if } (3a \in Acqs(S), \text{match}_{\sigma}(a) \notin S \land e_1, e_2 \notin TRClosure(\text{match}_{\rho}(a)) ) \text{ then} & \\
S & \leftarrow S \cup TRClosure(\text{match}_{\rho}(a))
\end{align*}
\text{return } S
\end{align*}

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 OLClosure(e_1, e_2), we must set S_0 = \emptyset; 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 TRClosure(prev_{\sigma}(e_1)) \cup TRClosure(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; Acqs(S) denotes the set of acquire events in the set S. The algorithm ensures that the set is (\leq^{TO}, r) closed at each step, and runs in O(T^2 N) = O(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 acquire OAcqs(S, \ell) = \{a \in Acqs(S) \mid \text{match}_{\sigma}(a) \notin S\} for each lock \ell is either singleton or empty:

\text{lockFeasible}(S) \equiv \forall \ell \in \text{Locks}(\sigma), |OAcqs(S, \ell)| \leq 1
Observe that if lockFeasible(S) does not hold, then every lineariza-
ion 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-reorder-
ing-graph $G^\text{Opt}_S = (V^\text{Opt}_S, E^\text{Opt}_S)$, where $V^\text{Opt}_S = S$, and $E^\text{Opt}_S =$
$E^\text{Opt}_S \subseteq \sigma_{<=10} \cup E^\text{match} \cup E^\text{match} \cup E^\text{match}$.
Here, $E^\text{match}$ is the set of edges $(r, a')$ such that $r \leq \sigma_{tr}$ $a'$
and there is no intermediate event in $S$ that conflicts with both. The set
$E^\text{match}$ consists of all pairs $(r, a')$ such that $r \leq \sigma_{tr}$ $a'$ and there is a
common lock $l$ for which op(r) = rel(l), op(a') = acq(l), 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 sec-
tions before unmatched ones, i.e., $E^\text{match}$ = $\{(r, a') \mid \exists l, op(r) =$
$rel(l), op(a') = acq(l), match(l, a') \notin S\}$. Since optimistic reorder-
ings 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 acyclicity of $G^\text{Opt}_S$, so that
the existence of witness is guaranteed.

**Lemma 4.2.** Let $S$ be a trace and let $S \subseteq \text{Events}(\sigma)$ such that $S$ is
$(\leq_{\sigma_{tr}} rf_{\sigma})$-closed and also lock-feasible. Then, there is an optimistic
reordering $\rho$ of $\sigma$ on the set $S$ iff the graph $G^\text{Opt}_S$ is acyclic.

**Example 9.** For trace $\sigma_1$ in Figure 1a, we have
 $\text{OLClosure}(e_1, e_{12}) = \{e_1, e_4, e_7, e_8, e_9, e_{10}, e_{11}\}$. The optimistic-
reordering-graph over $S_1 = \text{OLClosure}(e_1, e_{12})$ 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 $\rho_1$ from Figure 1b that witnesses the race $(e_1, e_{12})$. Let
us now consider $\sigma_2$, Figure 5a. The optimistic lock-closure with
respect to $(e_4, e_9)$ is $S_2 = \text{OLClosure}(e_4, e_9) = \{e_1, e_2, e_3, e_6, e_7, e_8\}$.
The optimistic reordering graph over $S_2$, shown in Figure 5b,
contains a cycle. Indeed, $(e_4, e_9)$ is not a predictable race.

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

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

### 4.2 Incremental Race Detection

**Overview.** Recall that there are $O(N^2)$ 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 $\tilde{O}(N^2)$ time. We achieve this by spending
$\tilde{O}(N)$ time per (read/write) event $e \in \text{Events}(\sigma)$, and determine
in overall $\tilde{O}(N)$ 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
$G^\text{Opt}(\sigma)$ (Section 4.1) cannot be computed in an incremental fash-
ion, 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 $\tilde{O}(1)$ (instead of $\tilde{O}(N)$) 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 $\tilde{O}(1)$ 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,
as we fix a thread $t$, and scan the events of $t$ from earliest to latest
events, we can reuse prior computations. In fact, Al-
gorithm 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 $e_1, e_2, e'_2 \in \text{Events}(\sigma)$ be events in trace $\sigma$
with $e_2 \leq_{\text{TO}} e'_2$. Let $S = \text{OLClosure}(e_1, e_2)$ and let $S' =$
$\text{OLClosure}(e_1, e'_2)$. We have the following: (1) $S \subseteq S'$. (2) $S =$
$\text{ComputeOLClosure}(e_1, e_2, \phi)$, and further this call in (Algorithm 1)
takes $\tilde{O}(|S|)$ time. (3) $S' = \text{ComputeOLClosure}(e_1, e'_2, S)$, and
further this call in (Algorithm 1) takes $\tilde{O}(|S'| - |S|)$ time.

**Abstract optimistic-reordering-graph.** For a set $S \subseteq \text{Events}(\sigma)$,
the abstract optimistic-reordering-graph is a tuple $G^\text{Abs}_{\sigma} =$
$(V^\text{Abs}_{\sigma}, E^\text{Abs}_{\sigma})$, where the vertices and edges are defined as follows.
(1) $V^\text{Abs}_{\sigma} = \{\ell \mid \ell \in \text{Locks}(\sigma) \mid \text{LastRel}(S, \ell)\} \cup \text{OAcqs}(S, \ell)$, where
lastRel(S, $\ell$) is the last release event on lock $\ell$ (according to $\leq_{\text{tr}}$) which
is present in $S$. (2) $(e, e') \in E^\text{Abs}_{\sigma}$ if there is a path from $e$
to $e'$ in the graph $G^\text{Opt}_{\sigma}$. In other words, $G^\text{Abs}_{\sigma}$ only contains $O(L)$
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

Reordering graph

Abstract graph

Figure 5: In $\sigma_3$, $(e_4, e_9)$ 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 $G^{Abs}_S$, because any cycle in $G^{Opt}_S$ must involve a ‘backward’ edge from a matched release and an unmatched acquire event. $G^{Abs}_S$ can thus be used to check for the existence of an optimistic reordering using an $O(1)$ check instead of an $O(N)$ check based on Lemma 4.2.

Lemma 4.4. Let $S$ be a trace and let $S \subseteq \text{Events}(\sigma)$ be a $(\leq_{TO}, rf_{\sigma})$-closed set. $G^{Opt}_S$ has a cycle iff $G^{Abs}_S$ has a cycle.

Example 10. Figure 4b shows the abstract optimistic reordering graph for trace $\sigma_1$ in Figure 1a, corresponding to the set $S_1 = \text{OLClosure}(e_1, e_{12})$, and contains the last release of lock $t$ 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 $e_2 \rightarrow e_3 \rightarrow e_7 \rightarrow e_8$ of Figure 5b with a direct edge $e_2 \rightarrow e_8$, thereby preserving the cycle.

Constructing vertices and backward edges of $G^{Abs}_S$. Recall that $S$ is a $(\leq_{TO}, rf_{\sigma})$-closed subset of $\text{Events}(\sigma)$. The set of vertices of this graph can be determined in $O(L)$ 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 $t$, and $r = \text{lastRel}(S, t)$ but $a \leq_{TO} r$ — can be computed in $O(L)$ 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 $e_1 \in \text{Events}(\sigma)|t_1 = \{e \in \text{Events}(\sigma) | \text{th}(e) = t_1\}$, the event EIS$_{t_1, t_2}(e_1)$ is the earliest immediate successor of $e_1$ in thread $t_2$ in the full graph $G^{Opt}_{\text{Events}(\sigma)}$; observe the subscript Events($\sigma$) instead of an arbitrary set $S$. EIS$_{t_1, t_2}$ can be computed as a pre-processing step in $O(TN) = O(N)$ time and stored as an array, indexed by the events of thread $t_1$.

Determining forward edges of $G^{Abs}_S$. The forward edges of $G^{Abs}_S$ summarize paths in $G^{Opt}_S$ and are computed as follows. Recall that we are given a $(\leq_{TO}, rf_{\sigma})$-closed subset $S$ of $\text{Events}(\sigma)$, and the path between two events must only be contained with the events of $S$, thus the arrays $\{\text{EIS}_{t_1, t_2} \forall t_1, t_2 \in \text{Threads}(\sigma)\}$ cannot be used as is to efficiently determine paths. However, a combination of range minima queries [7] and shortest path computation can nevertheless still be used to determine path information efficiently. Let us use $\text{succ}_{e,t}^S$ to denote the earliest event in thread $t$ that has a path from event $e$, using only forward edges of $G^{Opt}_S$. The event $\text{succ}_{e,t}^S$ can be computed using a Bellman-Ford-Moore [14, 29, 50] style shortest path computation, as shown in Algorithm 2. This algorithm performs rangeMin($A[a, b]$) queries which return the earliest event (according to $\leq_{TO}$) in the segment of the array $A$ starting at index $a$ and ending at index $b$. With $O(N)$ time and space pre-processing, each range minimum query takes $O(1)$ time [7, 30]. Thus, the task of determining $\{\text{succ}_{e,t}^S \forall e \in \text{Threads}(\sigma)\}$ takes $O(T^2)$ time. Now, in the graph $G^{Abs}_S$, we add an edge from $e$ to $e'$ if $\text{succ}_{e, \text{th}(e')}^S \leq_{TO} e'$. Thus, we add all forward edges of the graph in overall $O(T^2 L)$ 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 $\{\text{EIS}_{t_1, t_2} \forall t_1, t_2 \in \text{Threads}(\sigma)\}$ using a linear scan of the trace $\sigma$, and also do pre-processing for fast performing range minima queries, spending overall time $O(TN)$. Then, we iterate over each event $e'$ of thread $t$ that conflict with $e$, starting from the earlier 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 $G^{Abs}_S$ and check if it is acyclic, and report a race if so.
Algorithm 4: Detecting optimistic sync-reversal races in \( \sigma \)

1. \textbf{procedure} \textsc{OSR}(\( \sigma \))
2. \hspace{1em} \textbf{for} \( e \in \text{Events}(\sigma) \) \textbf{s.t.} \( e \) is a memory access event \textbf{do}
3. \hspace{2em} \textbf{for} \( t' \in \text{Threads}(\sigma) \) \textbf{do}
4. \hspace{3em} \text{incrementalRaceDetection}(e, t')

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

4.3 Detecting All Optimistic Sync-Reversal Races

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

Theorem 4.3. Given a trace \( \sigma \), the problem of checking if \( \sigma \) has an optimistic sync-reversal data race, can be solved in time \( O(\mathcal{T}^{2} + \mathcal{L})N^{2} = \tilde{O}(N^{2}) \) 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 \textsc{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 \( \sigma \), the problem of determining if \( \sigma \) has an \textsc{OSR} race cannot be solved in time \( \tilde{O}(N^{2} - \epsilon) \) (where \( N = |\text{Events}(\sigma)| \)) for every \( \epsilon > 0 \).

5 EVALUATION

We implemented \textsc{OSR} in Java, using the \textsc{RAPID} dynamic analysis framework [4]. We evaluate the performance and precision of \textsc{OSR}, on 153 benchmarks and compare it with prior state-of-the-art sound predictive race detection algorithms. We discuss our experimental setup 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++ (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 OpenSCR [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 \textsc{ThreadSanitizer} [66] (for Category-2) and using \textsc{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 \textsc{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 \textsc{OSR} with state-of-the-art sound predictive algorithms: \textsc{WCP} [36], \textsc{SHB} [44], \textsc{M2} [56] and \textsc{SyncP} [47]. Amongst these, \textsc{SHB} and \textsc{WCP} are partial order based methods and run in linear time. \textsc{M2} and \textsc{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. \textsc{SyncP} works in linear time while \textsc{M2} has higher polynomial complexity of \( \tilde{O}(N^{4}\log(N)) \) [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 \textsc{SeqC} [20] was not possible because the implementation of \textsc{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 \textsc{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 \textsc{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 – 100x 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 \textsc{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  is racy if there is a conflicting event earlier in the trace, such that  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  is racy if there is a racy event  that accesses . 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 synchronization-reversal races in principle, and reports much fewer races than OSR—essentially, M2, and often runs faster than more exhaustive techniques.

**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 1.5x. 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 2.5x 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 2.5x and 2.7x more races than SHB (2nd highest) and WCP (3rd highest). SyncP and M2 report 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 2.5x and 2.7x more races than SHB (2nd highest) and WCP (3rd highest). SyncP and M2 report 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 2.5x and 2.7x more races than SHB (2nd highest) and WCP (3rd highest).

**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.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>(N)</th>
<th>(T)</th>
<th>SHB Races</th>
<th>WCP Races</th>
<th>SyncP Races</th>
<th>M2 Races</th>
<th>OSR Races</th>
<th>SHB Time (s)</th>
<th>WCP Time (s)</th>
<th>SyncP Time (s)</th>
<th>M2 Time (s)</th>
<th>OSR Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>11</td>
<td>3</td>
<td>0(0)</td>
<td>0(0)</td>
<td>0(0)</td>
<td>0(0)</td>
<td>0(0)</td>
<td>0.05</td>
<td>0.08</td>
<td>0.06</td>
<td>0.03</td>
<td>0.09</td>
</tr>
<tr>
<td>critical</td>
<td>11</td>
<td>4</td>
<td>3(3)</td>
<td>0.04</td>
<td>1(1)</td>
<td>0.05</td>
<td>3(3)</td>
<td>0.07</td>
<td>0.02</td>
<td>0.07</td>
<td>0.02</td>
<td>0.07</td>
</tr>
<tr>
<td>account</td>
<td>15</td>
<td>4</td>
<td>3(1)</td>
<td>0.04</td>
<td>3(1)</td>
<td>0.06</td>
<td>3(1)</td>
<td>0.06</td>
<td>0.02</td>
<td>0.07</td>
<td>0.02</td>
<td>0.08</td>
</tr>
<tr>
<td>airtickets</td>
<td>18</td>
<td>5</td>
<td>8(3)</td>
<td>0.05</td>
<td>5(2)</td>
<td>0.08</td>
<td>8(3)</td>
<td>0.06</td>
<td>0.03</td>
<td>0.07</td>
<td>0.03</td>
<td>0.08</td>
</tr>
<tr>
<td>pingpong</td>
<td>24</td>
<td>7</td>
<td>8(3)</td>
<td>0.04</td>
<td>8(3)</td>
<td>0.07</td>
<td>8(3)</td>
<td>0.06</td>
<td>0.03</td>
<td>0.07</td>
<td>0.03</td>
<td>0.08</td>
</tr>
<tr>
<td>twostage</td>
<td>83</td>
<td>12</td>
<td>4(1)</td>
<td>0.06</td>
<td>4(1)</td>
<td>0.10</td>
<td>4(1)</td>
<td>0.14</td>
<td>0.05</td>
<td>0.07</td>
<td>0.05</td>
<td>0.10</td>
</tr>
<tr>
<td>wronglock</td>
<td>122</td>
<td>22</td>
<td>12(2)</td>
<td>0.07</td>
<td>3(2)</td>
<td>0.11</td>
<td>25(2)</td>
<td>0.22</td>
<td>0.18</td>
<td>0.25</td>
<td>0.18</td>
<td>0.13</td>
</tr>
<tr>
<td>bbuffer</td>
<td>9</td>
<td>3</td>
<td>3(1)</td>
<td>0.05</td>
<td>1(1)</td>
<td>0.06</td>
<td>3(1)</td>
<td>0.05</td>
<td>0.02</td>
<td>0.07</td>
<td>0.02</td>
<td>0.10</td>
</tr>
<tr>
<td>prodcons</td>
<td>246</td>
<td>8</td>
<td>1(1)</td>
<td>0.07</td>
<td>1(1)</td>
<td>0.13</td>
<td>1(1)</td>
<td>0.16</td>
<td>0.06</td>
<td>0.11</td>
<td>0.06</td>
<td>0.12</td>
</tr>
<tr>
<td>clean</td>
<td>867</td>
<td>59(4)</td>
<td>0.11</td>
<td>82(4)</td>
<td>0.23</td>
<td>60(4)</td>
<td>0.26</td>
<td>110(4)</td>
<td>0.65</td>
<td>110(4)</td>
<td>0.20</td>
<td></td>
</tr>
<tr>
<td>mergesort</td>
<td>167</td>
<td>5</td>
<td>1(1)</td>
<td>0.09</td>
<td>1(1)</td>
<td>0.13</td>
<td>3(1)</td>
<td>0.10</td>
<td>5(2)</td>
<td>0.04</td>
<td>5(2)</td>
<td>0.12</td>
</tr>
<tr>
<td>bubblesort</td>
<td>1.7K</td>
<td>13</td>
<td>269(5)</td>
<td>0.15</td>
<td>100(5)</td>
<td>0.30</td>
<td>269(5)</td>
<td>2.29</td>
<td>374(5)</td>
<td>8.40</td>
<td>374(5)</td>
<td>2.88</td>
</tr>
<tr>
<td>lang</td>
<td>1.8K</td>
<td>7</td>
<td>400(1)</td>
<td>0.17</td>
<td>400(1)</td>
<td>0.26</td>
<td>400(1)</td>
<td>0.34</td>
<td>0.01</td>
<td>0.54</td>
<td>0.01</td>
<td>0.22</td>
</tr>
<tr>
<td>readwrite</td>
<td>9.8K</td>
<td>5</td>
<td>150(4)</td>
<td>0.09</td>
<td>150(4)</td>
<td>0.17</td>
<td>150(4)</td>
<td>0.18</td>
<td>0.02</td>
<td>0.21</td>
<td>0.02</td>
<td>0.09</td>
</tr>
<tr>
<td>raytracer</td>
<td>1.7K</td>
<td>11</td>
<td>69(21)</td>
<td>1.18</td>
<td>70(21)</td>
<td>0.99</td>
<td>85(21)</td>
<td>6.01</td>
<td>85(21)</td>
<td>2.43</td>
<td>85(21)</td>
<td>0.79</td>
</tr>
<tr>
<td>prodcons</td>
<td>246</td>
<td>8</td>
<td>1(1)</td>
<td>0.07</td>
<td>1(1)</td>
<td>0.13</td>
<td>1(1)</td>
<td>0.16</td>
<td>0.06</td>
<td>1(1)</td>
<td>0.12</td>
<td></td>
</tr>
<tr>
<td>clean</td>
<td>867</td>
<td>59(4)</td>
<td>0.11</td>
<td>82(4)</td>
<td>0.23</td>
<td>60(4)</td>
<td>0.26</td>
<td>110(4)</td>
<td>0.65</td>
<td>110(4)</td>
<td>0.20</td>
<td></td>
</tr>
<tr>
<td>mergesort</td>
<td>167</td>
<td>5</td>
<td>1(1)</td>
<td>0.09</td>
<td>1(1)</td>
<td>0.13</td>
<td>3(1)</td>
<td>0.10</td>
<td>5(2)</td>
<td>0.04</td>
<td>5(2)</td>
<td>0.12</td>
</tr>
<tr>
<td>bubblesort</td>
<td>1.7K</td>
<td>13</td>
<td>269(5)</td>
<td>0.15</td>
<td>100(5)</td>
<td>0.30</td>
<td>269(5)</td>
<td>2.29</td>
<td>374(5)</td>
<td>8.40</td>
<td>374(5)</td>
<td>2.88</td>
</tr>
<tr>
<td>lang</td>
<td>1.8K</td>
<td>7</td>
<td>400(1)</td>
<td>0.17</td>
<td>400(1)</td>
<td>0.26</td>
<td>400(1)</td>
<td>0.34</td>
<td>0.01</td>
<td>0.54</td>
<td>0.01</td>
<td>0.22</td>
</tr>
<tr>
<td>readwrite</td>
<td>9.8K</td>
<td>5</td>
<td>150(4)</td>
<td>0.09</td>
<td>150(4)</td>
<td>0.17</td>
<td>150(4)</td>
<td>0.18</td>
<td>0.02</td>
<td>0.21</td>
<td>0.02</td>
<td>0.09</td>
</tr>
<tr>
<td>raytracer</td>
<td>1.7K</td>
<td>11</td>
<td>69(21)</td>
<td>1.18</td>
<td>70(21)</td>
<td>0.99</td>
<td>85(21)</td>
<td>6.01</td>
<td>85(21)</td>
<td>2.43</td>
<td>85(21)</td>
<td>0.79</td>
</tr>
<tr>
<td>prodcons</td>
<td>246</td>
<td>8</td>
<td>1(1)</td>
<td>0.07</td>
<td>1(1)</td>
<td>0.13</td>
<td>1(1)</td>
<td>0.16</td>
<td>0.06</td>
<td>1(1)</td>
<td>0.12</td>
<td></td>
</tr>
</tbody>
</table>

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 \(\text{NP}\)-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].
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.

<table>
<thead>
<tr>
<th>Benchmark Group</th>
<th>N</th>
<th>( T )</th>
<th>SHB</th>
<th>WCP</th>
<th>SyncP</th>
<th>M2</th>
<th>OSR</th>
</tr>
</thead>
<tbody>
<tr>
<td>CoMD (8)</td>
<td>[2.5M, 117M]</td>
<td>707M</td>
<td>16, 56</td>
<td>41.6K</td>
<td>29.1</td>
<td>247K</td>
<td>73.2</td>
</tr>
<tr>
<td>SimpleMOC (1)</td>
<td>[19M, 19M]</td>
<td>19M</td>
<td>16, 16</td>
<td>380</td>
<td>0.1</td>
<td>388</td>
<td>23.8</td>
</tr>
<tr>
<td>OMPRacer (15)</td>
<td>[0.7M, 157M]</td>
<td>625M</td>
<td>16, 58</td>
<td>1.2M</td>
<td>17.4</td>
<td>0.7M</td>
<td>84.1</td>
</tr>
<tr>
<td>DRACC (13)</td>
<td>[0.5k, 104M]</td>
<td>694M</td>
<td>16, 16</td>
<td>2247</td>
<td>8.4</td>
<td>2247</td>
<td>105.7</td>
</tr>
<tr>
<td>DRB (33)</td>
<td>[0.5k, 900M]</td>
<td>5.7B</td>
<td>16, 56</td>
<td>50.4k</td>
<td>169.5</td>
<td>54.5k</td>
<td>0.6k</td>
</tr>
<tr>
<td>HPC (46)</td>
<td>[1k, 335M]</td>
<td>3.8B</td>
<td>16, 56</td>
<td>6.5M</td>
<td>174.2</td>
<td>6.4M</td>
<td>775.4</td>
</tr>
<tr>
<td>misc (7)</td>
<td>[1k, 29M]</td>
<td>49M</td>
<td>[4, 219]</td>
<td>8548</td>
<td>0.9</td>
<td>8481</td>
<td>182.2</td>
</tr>
<tr>
<td><strong>Total (123)</strong></td>
<td></td>
<td>11.6B</td>
<td></td>
<td>7.9M</td>
<td>6.7h</td>
<td>7.4M</td>
<td>30.3h</td>
</tr>
</tbody>
</table>

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

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 VIL-LUM 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.


