Sound Dynamic Deadlock Prediction in Linear Time

HÜNKAR CAN TUNÇ, Aarhus University, Denmark
UMANG MATHUR, National University of Singapore, Singapore
ANDREAS PAVLOGIANNIS, Aarhus University, Denmark
MAHESH VISWANATHAN, University of Illinois, Urbana Champaign, USA

Deadlocks are one of the most notorious concurrency bugs, and significant research has focused on detecting them efficiently. *Dynamic predictive analyses* work by observing concurrent executions, and reason about alternative interleavings that can witness concurrency bugs. Such techniques offer scalability and sound bug reports, and have emerged as an effective approach for concurrency bug detection, such as data races. Effective dynamic deadlock prediction, however, has proven a challenging task, as no deadlock predictor currently meets the requirements of *soundness*, *high-precision*, and *efficiency*.

In this paper, we first formally establish that this tradeoff is unavoidable, by showing that (a) sound and complete deadlock prediction is intractable, in general, and (b) even the seemingly simpler task of determining the presence of *potential* deadlocks, which often serve as unsound witnesses for actual predictable deadlocks, is intractable. The main contribution of this work is a new class of predictable deadlocks, called *sync(hronization)-preserving* deadlocks. Informally, these are deadlocks that can be predicted by reordering the observed execution while preserving the relative order of conflicting critical sections. We present two algorithms for *sound* deadlock prediction based on this notion. Our first algorithm `SPDOffline` detects all sync-preserving deadlocks, with running time that is linear per *abstract deadlock pattern*, a novel notion also introduced in this work. Our second algorithm `SPDOnline` predicts all sync-preserving deadlocks that involve two threads in a *strictly online* fashion, runs in overall linear time, and is better suited for a runtime monitoring setting.

We implemented both our algorithms and evaluated their ability to perform offline and online deadlock-prediction on a large dataset of standard benchmarks. Our results indicate that our new notion of sync-preserving deadlocks is highly effective, as (i) it can characterize the vast majority of deadlocks and (ii) it can be detected using an online, sound, complete and highly efficient algorithm.

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

Additional Key Words and Phrases: concurrency, runtime analyses, predictive analyses

ACM Reference Format:

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 (June 2023), 26 pages. https://doi.org/10.1145/3591291

Authors' addresses: Hünkar Can Tunç, Aarhus University, Denmark, tunc@cs.au.dk; Umang Mathur, National University of Singapore, Singapore, umathur@comp.nus.edu.sg; Andreas Pavlogiannis, Aarhus University, Denmark, pavlogiannis@cs.au.dk; Mahesh Viswanathan, University of Illinois, Urbana Champaign, USA, vmahesh@illinois.edu.

This work is licensed under a Creative Commons Attribution-NoDerivatives 4.0 International License.
© 2023 Copyright held by the owner/author(s).
2475-1421/2023/6-ART177
https://doi.org/10.1145/3591291

Corrected Version of Record. V.1.1. Published July 18, 2023.

1 INTRODUCTION

The verification of concurrent programs is a major challenge due to the non-deterministic behavior intrinsic to them. Certain scheduling patterns may be unanticipated by the programmers, which may then lead to introducing concurrency bugs. Such bugs are easy to introduce during development but can be very hard to reproduce during in-house testing, and have been notoriously called *heisenbugs* [Musuvathi et al. 2008]. Among the most notorious concurrency bugs are deadlocks, occurring when the system blocks its execution because each thread is waiting for another thread to finish a task in a circular fashion. Deadlocks account for a large fraction of concurrency bugs in the wild across various programming languages [Lu et al. 2008; Tu et al. 2019] while they are often introduced accidentally when fixing other concurrency bugs [Yin et al. 2011].

Deadlock-detection techniques can be broadly classified into static and dynamic techniques. As usual, static techniques analyze source code and have the potential to prove the absence of deadlocks [Liu et al. 2021; Naik et al. 2009; Ng and Yoshida 2016]. However, as static analyses face simultaneously two dimensions of non-determinism, namely in inputs and scheduling, they lead to poor performance in terms of scalability and false positives, and are less suitable when the task at hand is to help software developers proactively find bugs. Dynamic analyses, on the other hand, have the more modest goal of discovering deadlocks by analyzing program executions, allowing for better scalability and few (or no) false positives. Although dynamic analyses cannot prove the absence of bugs, they offer *statistical* and *coverage* guarantees. These advantages have rendered dynamic techniques a standard practice in principled testing for various bugs, such as data races, atomicity violations, deadlocks, and others [Bensalem and Havelund 2005; Biswas et al. 2014; Flanagan and Freund 2009; Flanagan et al. 2008; Mathur and Viswanathan 2020; Pozniansky and Schuster 2003; Savage et al. 1997; Serebryany and Iskhodzhanov 2009]. A recent trend in this direction advocates for *predictive analysis* [Flanagan et al. 2008; Genç et al. 2019; Huang 2018; Huang et al. 2014; Kalhauge and Palsberg 2018; Kini et al. 2017; Smaragdakis et al. 2012], where the goal is to enhance coverage by additionally reasoning about alternative reorderings of the observed execution trace that *could* have taken place and also manifest the bug.

Due to the difficulty of the problem, many dynamic deadlock analyses focus on detecting *deadlock patterns*, broadly defined as cyclic lock-acquisition patterns in the observed execution trace. One of the earliest works in this direction is the Goodlock algorithm [Havelund 2000]. As deadlock patterns are necessary but insufficient conditions for the presence of deadlocks, subsequent work has focused on refining this notion in order to reduce false-positives [Agarwal et al. 2006; Bensalem and Havelund 2005]. Further techniques reduce the size of the lock graph to improve scalability [Cai and Chan 2012; Cai et al. 2020]. To further address the unsoundness (false positives) problem, various works propose controlled-scheduling techniques that attempt to realize deadlock warnings via program re-execution [Bensalem et al. 2006; Joshi et al. 2009; Samak and Ramanathan 2014a,b; Sorrentino 2015] and exhaustive exploration of all reorderings [Joshi et al. 2010; Sen et al. 2005].

Fully sound deadlock prediction has traditionally relied on explicitly [Joshi et al. 2010; Sen et al. 2005] or symbolically (SMT-based) [Eslamimehr and Palsberg 2014; Kalhauge and Palsberg 2018] producing all sound witness reorderings. The heavyweight nature of such techniques limits their applicability to executions of realistic size, which is often in the order of millions of events. The first steps for sound, polynomial-time deadlock prediction were made recently with `SeqCheck` [Cai et al. 2021], an extension of M2 [Pavlogiannis 2019] that targets data races.

This line of work highlights the need for a most-efficient sound deadlock predictor, approaching the golden standard of *linear time*. Moreover, dynamic analyses are often employed as runtime monitors, and must thus operate *online*, reporting bugs as soon as they occur. Unfortunately, most

	t_1	t_2
1	acq(ℓ_1)	
2	**acq(ℓ_2)**	
3	w(x)	
4	rel(ℓ_2)	
5	rel(ℓ_1)	
6		acq(ℓ_2)
7		r(x)
8		**acq(ℓ_1)**
9		rel(ℓ_1)
10		rel(ℓ_2)

(a) A trace σ_1 with no predictable deadlock.

	t_1	t_2	t_3	t_4
1	acq(ℓ_1)			
2	rel(ℓ_1)			
3		acq(ℓ_2)		
4		**acq(ℓ_3)**		
5		w(z)		
6		rel(ℓ_3)		
7		rel(ℓ_2)		
8				acq(ℓ_1)
9				w(y)
10				r(z)

	t_1	t_2	t_3	t_4
11				rel(ℓ_1)
12	acq(ℓ_3)			
13	w(x)			
14	r(y)			
15	rel(ℓ_3)			
16			acq(ℓ_3)	
17			r(x)	
18			**acq(ℓ_2)**	
19			rel(ℓ_2)	
20			rel(ℓ_3)	

(b) A trace σ_2 with a sync-preserving deadlock, stalling t_2 on e_4 and t_3 on e_{18}.

Fig. 1. Traces with no predictable deadlock (a), and with a sync-preserving deadlock (b).

existing online algorithms only report *deadlock patterns*, thus suffering false positives. The lack of such a deadlock predictor is even more pronounced when contrasted to the problem of dynamic race prediction, which has seen a recent surge of sound, online, *linear-time* predictors (e.g., [Kini et al. 2017; Roemer et al. 2020]), and highlights the bigger challenges that deadlocks entail. We address these challenges in this work, by presenting the first high-precision, sound dynamic deadlock-prediction algorithm that works online and in linear time.

The task of checking if a potential deadlock is a real predictable deadlock, in general, involves searching for the reordering of the original execution that witnesses the deadlock. The first ingredient towards our technique is the notion of *synchronization-preserving reorderings* [Mathur et al. 2021] that help systematize this search space. *Synchronization-preserving deadlocks* are then those predictable deadlocks that can be witnessed in some synchronization-preserving reordering. We illustrate synchronization-preserving deadlocks using an example in Section 1.1.

This notion of synchronization-preservation, by itself, is not sufficient when it comes to deadlock detection as the prerequisite step towards predicting deadlocks also involves identifying *potential deadlock patterns*. Unlike data races, where *potential races* can be identified in polynomial-time, the identification of deadlock patterns is in general, intractable; we prove this in Section 3. As a result, an approach that works by explicitly enumerating cycles in a *lock graph* and then checking if any of these cycles is realizable to a deadlock is likely to be not scalable. To tackle this, we propose the novel notion of *abstract deadlock patterns* which, informally, represent clusters of deadlock patterns of the same signature. Intuitively, a set of deadlock patterns have the same signature if the threads and locks that participate in the patterns are the same. Our next *key observation* is that a single abstract deadlock pattern can be checked for sync-preserving deadlocks in *linear total time* in the length of the execution, *regardless* of how many concrete deadlock patterns it represents. Our first deadlock prediction algorithm `SPDOffline` builds upon this — it enumerates all abstract deadlock patterns in a first phase and then checks their realizability in a second phase, while running in linear time per abstract deadlock pattern. Since the number of abstract deadlock patterns is typically *far smaller* than the number of (concrete) deadlock patterns (see Table 1 in Section 6), this approach achieves high scalability. Our second algorithm `SPDOnline` works in a single streaming pass — it computes abstract deadlock patterns that involve only two threads and checks their realizability *on-the-fly* simultaneously in overall linear time in the length of the execution.

1.1 Synchronization-Preserving Deadlocks

Consider the trace σ_1 in Figure 1a consisting of 10 events and two threads. We use e_i to denote the i-th event of σ_1. The events e_2 and e_8 form a *deadlock pattern*: they respectively acquire the locks ℓ_2 and ℓ_1 while holding the locks ℓ_1 and ℓ_2, and no common lock protects these operations.

A deadlock pattern is a necessary but insufficient condition for an actual deadlock: a sound algorithm must examine whether it can be realized to a deadlock via a witness. A witness is a reordering ρ of (a slice of) σ_1 that is also a valid trace, and such that e_2 and e_8 are locally enabled in their respective threads at the end of ρ. In general, the problem of checking if a deadlock pattern can be realized is intractable (Theorem 3.3). In this work we focus on checking whether a given deadlock pattern forms a *sync-preserving deadlock*, which is a subclass of the class of all predictable deadlocks.

A deadlock pattern is said to be sync-preserving deadlock if it can be witnessed in a *sync-preserving reordering*. A reordering ρ^{SP} of a trace σ is said to be sync-preserving if it preserves the control flow taken by the original observed trace σ, and further it preserves the mutual order of any two critical sections (on the same lock) that appear in the reordering ρ^{SP}. Consider, for example, the sequence $\rho_1 = e_1..e_3\, e_6..e_7$ where $e_i..e_j$ denote the contiguous sequence of events that starts from e_i and ends at e_j. We call ρ_1 a *correct reordering* of σ_1, being a slice of σ_1 closed under the thread order and preserving the writer of each read in σ_1; the precise definition is presented in Section 2. In this case, however, ρ_1 does not witness the deadlock as the event e_2 is not *enabled* in ρ_1. In fact, due to the dependency between the events e_3 and e_7, there are no correct reorderings of σ_1 which make both e_2 and e_8 enabled. This makes the deadlock pattern $\langle e_2, e_8\rangle$ non-predictable. Consider now σ_2 in Figure 1b, and the sequence $\rho_2 = e_3..e_7\, e_8..e_{11}\, e_1 e_2$. Observe that ρ_2 is also a correct reordering. However, ρ_2 is not sync-preserving as the order of the two critical sections on lock ℓ_1 in ρ_2 is different from their original order in σ_2. On the other hand, $\rho_3 = e_1 e_2 e_3 e_8 e_9\, e_{12}..e_{15}\, e_{16} e_{17}$ is a correct reordering that is also sync-preserving — all pairs of critical sections on the same lock appear in the same order in ρ_3 as they did in σ_2. Further, ρ_3 also witnesses the deadlock as the events e_4 and e_{18} are both *enabled* in ρ_3. This makes the deadlock pattern $\langle e_4, e_{18}\rangle$ a sync-preserving deadlock.

In this work we show that sync-preserving deadlocks enjoy two remarkable properties. First, all sync-preserving deadlocks of a given *abstract* deadlock pattern can be checked in linear time. Second, our extensive experimental evaluation on standard benchmarks indicates that sync-preservation captures a vast majority of deadlocks in practice. In combination, these two benefits suggest that sync-preservation is the right notion of deadlocks to be targeted by dynamic deadlock predictors.

1.2 Our Contributions

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

(1) **Complexity of Deadlock Prediction.** Perhaps surprisingly, the complexity of detecting deadlock patterns, as well as predicting deadlocks has remained elusive. Our first contribution resolves such questions. Given a trace σ of size $\mathcal{N}$ and $\mathcal{T}$ threads, we first show that detecting even one deadlock *pattern* of length k is $\mathsf{W}[1]$-hard in k. This establishes that the problem is NP-hard, and further rules out algorithms that are fixed-parameter-tractable in k, i.e., with running time of the form $f(k) \cdot \mathrm{poly}(\mathcal{N})$, for some function f. We next show that even with just $\mathcal{T} = 2$ threads, the problem of detecting a single deadlock *pattern* (of size $k = 2$) admits a quadratic lower bound, i.e., it cannot be solved in time $O(\mathcal{N}^{2-\epsilon})$, no matter what $\epsilon > 0$ we choose. These two results shed light on the difficulty in identifying deadlock patterns — a task that might otherwise appear easier than the core task of prediction. These hardness results, in particular the fine-grained lower bound result, are based on novel constructions, and results from fine-grained complexity [Williams 2018]. Our third result is about confirming predictable deadlocks — even for a deadlock pattern of size $k = 2$, checking whether it yields a *predictable deadlock* is $\mathsf{W}[1]$-hard in the number of threads $\mathcal{T}$ (and thus again NP-hard), and is inspired from an analogous result in the context of data race prediction [Mathur et al. 2020]. These results capture the intractability of deadlock prediction in general, even for the class of parametrized algorithms.

(2) **Sync-preserving Deadlock Prediction and Abstract Deadlock Patterns.** Given the above hardness of predicting arbitrary deadlocks, we define a novel notion of sync(hronization)-preserving deadlocks, illustrated in Section 1.1. We develop `SPDOnline`, an *online, sound* deadlock predictor that takes as input a trace and reports *all* sync-preserving deadlocks of size 2 in *linear time* $\widetilde{O}(\mathcal{N})$[*]. As most deadlocks in practice involve only two threads [Lu et al. 2008], restricting `SPDOnline` to size 2 deadlocks leads to linear-time deadlock prediction with small impact on its coverage. We also develop our more general algorithm, `SPDOffline`, that detects *all* sync-preserving deadlocks of all sizes. `SPDOffline` operates in two phases. In the first phase, it detects all *abstract deadlock patterns*. An abstract deadlock pattern is a novel notion that serves as a succinct representation of the class of deadlock patterns having the same signature. In the second phase, `SPDOffline` executes `SPDOnline` on each abstract pattern to decide whether a deadlock is formed. The running time of `SPDOffline` remains linear in $\mathcal{N}$, but increases by a factor proportional to the number of abstract deadlock patterns in the lock graph.

(3) **Implementation and Evaluation.** We have evaluated `SPDOnline` and `SPDOffline` in terms of performance and predictive power on a large dataset of standard benchmarks. In the offline setting, `SPDOffline` finds the same number of deadlocks as the recently introduced `SeqCheck`, while achieving a speedup of $> 200\times$ on the most demanding benchmarks, and $21\times$ overall. In the online setting, `SPDOnline` achieved a significant improvement in deadlock discovery and deadlock-hit-rate compared to the random scheduling based controlled concurrency testing technique of `DeadlockFuzzer` [Joshi et al. 2009]. Our experiments thus support that the notion of sync-preserving deadlocks is suitable: (i) it captures the vast majority of the deadlocks in practice, and (ii) sync-preserving deadlocks can be detected online and optimally — that is, soundly, completely and in linear time, (iii) it can enhance the deadlock detection capability of controlled concurrency testing techniques, (iv) with reasonable runtime overhead.

2 PRELIMINARIES

Here we set up our model and develop relevant notation, following related work in predictive analyses of concurrent programs [Kini et al. 2017; Roemer et al. 2020; Smaragdakis et al. 2012].

Execution traces. A dynamic analysis observes traces generated by a concurrent program, and analyzes them to determine the presence of a bug. Each such trace σ is a linear arrangement of events Events_σ. An event $e \in \mathsf{Events}_\sigma$ is tuple $e = \langle i, t, o \rangle$, where i is a unique identifier of e, t is the unique identifier of the thread performing e, and o is either a read or write ($o = \mathtt{r}(x)$ or $o = \mathtt{w}(x)$) operation to some variable x, or an acquire or release ($o = \mathtt{acq}(\ell)$ or $o = \mathtt{rel}(\ell)$) operation on some lock ℓ. For the sake of simplicity, we often omit i when referring to an event. We use $\mathrm{thread}(e)$ and $\mathrm{op}(e)$ to respectively denote the thread identifier and the operation performed in the event e. We use $\mathsf{Threads}_\sigma$, Vars_σ and Locks_σ to denote the set of thread, variable and lock identifiers in σ.

We restrict our attention to *well-formed* traces σ, that abide to shared-memory semantics. That is, if a lock ℓ is acquired at an event e by thread t, then any later acquisition event e' of the same lock ℓ must be preceded by an event e'' that releases lock ℓ in thread t in between the occurrence of e and e'. Taking e'' to be the earliest such release event, we say that e and e'' are matching acquire and release events, and denote this by $e = \mathsf{match}_\sigma(e'')$ and $e'' = \mathsf{match}_\sigma(e)$. Moreover, every read event has at least one preceding write event on the same location, that it reads its value from.

Functions and relations on traces. A trace σ implicitly defines some relations. The *trace-order* $\leq^\sigma_{\mathrm{tr}} \subseteq \mathsf{Events}_\sigma \times \mathsf{Events}_\sigma$ orders the events of σ in a total order based on their order of occurrence in

[*]We use $\widetilde{O}$ to ignore *polynomial* appearance of trace parameters typically much smaller than $\mathcal{N}$ (e.g., number of threads).

the sequence σ. The *thread-order* $\leq^{\sigma}_{\mathsf{TO}}$ is the unique partial order over Events_σ such that $e \leq^{\sigma}_{\mathsf{TO}} e'$ iff $\mathsf{thread}(e) = \mathsf{thread}(e')$ and $e \leq^{\sigma}_{\mathsf{tr}} e'$. We say $e <^{\sigma}_{\mathsf{TO}} e'$ if $e \leq^{\sigma}_{\mathsf{TO}} e'$ but $e \neq e'$. The *reads-from* function rf_σ is a map from the read events to the write events in σ. Under sequential consistency, for a read event e on variable x, we have that $e' = \mathsf{rf}_\sigma(e)$ be the latest write event on the same variable x such that $e' \leq^{\sigma}_{\mathsf{tr}} e$. We say that a lock $\ell \in \mathsf{Locks}_\sigma$ is held at an event $e \in \mathsf{Events}_\sigma$ if there is an event e' such that (i) $\mathsf{op}(e') = \mathsf{acq}(\ell)$, (ii) $e' <^{\sigma}_{\mathsf{TO}} e$, and (iii) either $\mathsf{match}_\sigma(e')$ does not exist in σ, or $e \leq^{\sigma}_{\mathsf{TO}} \mathsf{match}_\sigma(e')$. We use $\mathsf{HeldLks}_\sigma(e)$ to denote the set of all the locks that are held by $\mathsf{thread}(e)$ right before e. The lock nesting depth of σ is $\max\limits_{e \in \mathsf{Events}_\sigma} |\mathsf{HeldLks}_\sigma(e)| + 1$ where $\mathsf{op}(e) = \mathsf{acq}(\ell)$.

Deadlock patterns. A deadlock pattern[†] of size k in a trace σ is a sequence $D = \langle e_0, e_1, \ldots, e_{k-1} \rangle$, with k distinct threads $t_0, \ldots, t_{k-1}$ and k distinct locks $\ell_0, \ldots, \ell_{k-1}$ such that $\mathsf{thread}(e_i) = t_i$, $\mathsf{op}(e_i) = \mathsf{acq}(\ell_i)$, $\ell_i \in \mathsf{HeldLks}_\sigma(e_{(i+1)\%k})$, and further, $\mathsf{HeldLks}_\sigma(e_i) \cap \mathsf{HeldLks}_\sigma(e_j) = \varnothing$ for every i, j such that $i \neq j$ and $0 \leq i, j < k$. A deadlock pattern is a necessary but insufficient condition of an actual deadlock, due to subtle synchronization or control and data flow in the underlying program.

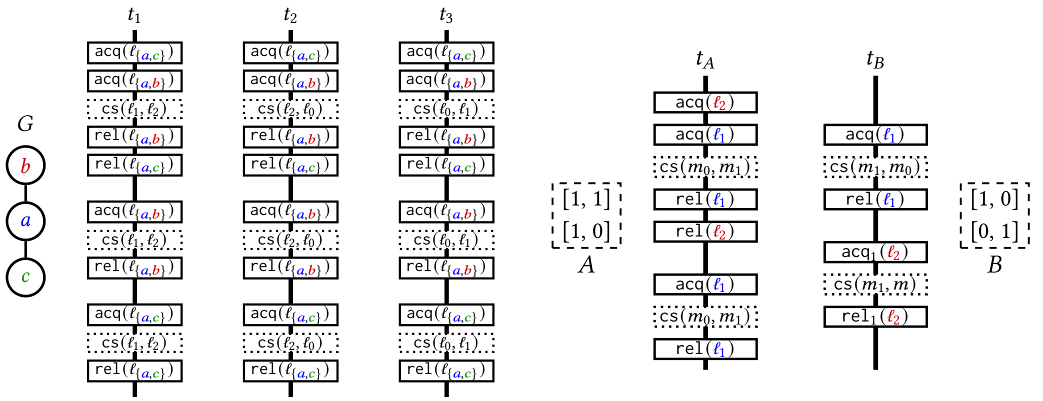
Dynamic predictive analysis and correct reorderings. Dynamic analyses aim to expose bugs by observing traces σ of a concurrent program, often without accessing the source code. While such purely dynamic approaches enjoy the benefits of scalability, simply detecting bugs that manifest on σ offer poor coverage and are bound to miss bugs that appear in select thread interleavings [Musuvathi et al. 2008]. Therefore, for better coverage, *predictive* dynamic techniques are developed. Such techniques predict the occurrence of bugs in alternate executions that can be *inferred* from σ, irrespective of the program that produced σ. The notion of such inferred executions is formalized by the notion of correct reorderings [Sen et al. 2005; Şerbănută et al. 2013; Smaragdakis et al. 2012].

A trace ρ is a *correct reordering* of a trace σ if (1) $\mathsf{Events}_\rho \subseteq \mathsf{Events}_\sigma$, (2) for every $e, f \in \mathsf{Events}_\sigma$ with $e \leq^{\sigma}_{\mathsf{TO}} f$, if $f \in \mathsf{Events}_\rho$, then $e \in \mathsf{Events}_\rho$ and $e \leq^{\rho}_{\mathsf{TO}} f$, and (3) for every read event $r \in \mathsf{Events}_\rho$, we have $\mathsf{rf}_\sigma(r) \in \mathsf{Events}_\rho$ and $\mathsf{rf}_\rho(r) = \mathsf{rf}_\sigma(r)$. Intuitively, a correct reordering ρ of σ is a permutation of σ that respects the thread order and preserves the values of each read and write that occur in ρ. This ensures a key property — every program that generated σ is also capable of generating ρ (possibly under a different thread schedule), and thus ρ serves as a true witness of a bug.

Predictable deadlocks. We say that an event e is σ-enabled in a correct reordering ρ of σ if $e \in \mathsf{Events}_\sigma$, $e \notin \mathsf{Events}_\rho$ and for every $f \in \mathsf{Events}_\sigma$ if $f <^{\sigma}_{\mathsf{TO}} e$, then $f \in \mathsf{Events}_\rho$. A deadlock pattern $D = \langle e_0, e_1 \ldots e_{k-1} \rangle$ of size k in trace σ is said to be a predictable deadlock if there is a correct reordering ρ of σ such that each of $e_0, \ldots, e_{k-1}$ are σ-enabled in ρ. This notion guarantees that the witness ρ is a valid execution of *any* concurrent program that produced σ. Analogous definitions have also been widely used for other predictable bugs [Huang et al. 2014; Smaragdakis et al. 2012]. We call a deadlock-prediction algorithm *sound* if for every input trace σ, all deadlock reports on σ are predictable deadlocks of σ (i.e., no false positives), and *complete* if all predictable deadlocks of σ are reported by the algorithm (i.e., no false negatives). This is in line with the previous works on the topic of predictive analyses [Cai et al. 2021; Kalhauge and Palsberg 2018; Mathur et al. 2021; Pavlogiannis 2019]. We remark that other domains sometimes use this terminology reversed.

Example 1. Let us illustrate these definitions on the trace σ_2 in Fig. 1b, with e_i denoting the i^{th} event in the figure. The set of events, threads, variables and locks of σ_2 are respectively $\mathsf{Events}_{\sigma_2} = \{e_i\}_{i=1}^{20}$, $\mathsf{Threads}_{\sigma_2} = \{t_1, t_2, t_3, t_4\}$, $\mathsf{Vars}_{\sigma_2} = \{x, y, z\}$ and $\mathsf{Locks}_{\sigma_2} = \{\ell_1, \ell_2, \ell_3\}$. The trace order yields $e_i \leq^{\sigma_2}_{\mathsf{tr}} e_j$ iff $i \leq j$, and some examples of thread-ordered events are $e_1 <^{\sigma_2}_{\mathsf{TO}} e_2 <^{\sigma_2}_{\mathsf{TO}} e_{15}$

[†] Similar notions have been used in the literature, sometimes under the term *deadlock potential* [Havelund 2000].



(a) Reduction of INDEPENDENT-SET(3) on G to detecting a deadlock pattern of size 3.

(b) Reduction for OV-hardness proof from an instance of size $n = 2$ and $d = 2$.

Fig. 2. Construction of W[1]-hardness (a) and OV-hardness (b) results. We use the shortcut $\texttt{cs}(\ell, \ell')$ to denote two nested critical sections on ℓ and ℓ'. That is, $\texttt{cs}(\ell, \ell') = \texttt{acq}(\ell) \cdot \texttt{acq}(\ell') \cdot \texttt{rel}(\ell') \cdot \texttt{rel}(\ell)$.

and $e_{16} <^{\sigma_2}_{\mathsf{TO}} e_{18} <^{\sigma_2}_{\mathsf{TO}} e_{20}$. The reads-from function is as follows: $\mathsf{rf}_{\sigma_2}(e_{10}) = e_5$, $\mathsf{rf}_{\sigma_2}(e_{14}) = e_9$ and $\mathsf{rf}_{\sigma_2}(e_{17}) = e_{13}$. The lock nesting depth of σ_2 is 2. The sequence $D = \langle e_4, e_{18} \rangle$ forms a deadlock pattern because of the cyclic acquisition of locks ℓ_2 and ℓ_3 without simultaneously holding a common lock. The trace $\rho_4 = e_3..e_7\, e_8..e_{11}\, e_1 e_2 e_{12}..e_{15}\, e_{16} e_{17}$ is a correct reordering of σ_2; even though it differs from σ_2 in the relative order of the critical sections of lock ℓ_1, and contains only a prefix of thread t_3, it is consistent with rf_{σ_2} and $<^{\sigma_2}_{\mathsf{TO}}$. However, ρ_4 does not witness $\langle e_4, e_{18} \rangle$ as a deadlock, as only e_{18} is σ_2-enabled in ρ_4. On the other hand, the trace $\rho_3 = e_1 e_2 e_3 e_8 e_9\, e_{12}..e_{15}\, e_{16} e_{17}$ is a correct reordering of σ_2 in which e_4 and e_{18} are σ_2-enabled, witnessing D as a predictable deadlock of σ_2.

3 THE COMPLEXITY OF DYNAMIC DEADLOCK PREDICTION

Detecting deadlock patterns and predictable deadlocks is clearly a problem in NP, as any witness for either problem can be verified in polynomial time. However, little has been known about the hardness of the problem in terms of rigorous lower bounds. Here we settle these questions, by proving strong intractability results. Due to space constraints, we state and explain the main results here, and refer to our technical report [Tunç et al. 2023a] for the full proofs.

Parametrized hardness for detecting deadlock patterns. We show that the basic problem of checking the existence of a deadlock *pattern* is itself hard parameterized by the size k of the pattern.

Theorem 3.1. *Checking if a trace σ contains a deadlock pattern of size k is* W[1]*-hard in the parameter k. Moreover, the problem remains* NP*-hard even when the lock-nesting depth of σ is constant.*

Proof. We show that there is a polynomial-time fixed parameter tractable reduction from INDEPENDENT-SET(c) to the problem of checking the existence of deadlock-patterns of size c. Our reduction takes as input an undirected graph G and outputs a trace σ such that G has an independent set of size c iff σ has a deadlock pattern of size c.

Construction. Let $V = \{v_1, v_2, \ldots, v_n\}$. We assume a total ordering $<_E$ on the set of edges E. The trace σ we construct is a concatenation of c sub-traces: $\sigma = \sigma^{(1)} \cdot \sigma^{(2)} \cdots \sigma^{(c)}$ and uses c threads $\{t_1, t_2, \ldots t_c\}$ and $|E| + c$ locks $\{\ell_{\{u,v\}}\}_{\{u,v\} \in E} \uplus \{\ell_0, \ell_1 \ldots, \ell_{c-1}\}$. The i^{th} sub-trace $\sigma^{(i)}$ is a sequence of events performed by thread t_i, and is obtained by concatenation of $n = |V|$ sub-traces:

$\sigma^{(i)} = \sigma_1^{(i)} \cdot \sigma_2^{(i)} \cdots \sigma_n^{(i)}$. Each sub-trace $\sigma_j^{(i)}$ with $(i \leq c, j \leq n)$ comprises of nested critical sections over locks of the form $\ell_{\{v_j,u\}}$, where u is a neighbor of v_j. Inside the nested block we have critical sections on locks $\ell_{i\%c}$ and $\ell_{(i+1)\%c}$. Formally, let $\{v_j, v_{k_1}\}, \ldots, \{v_j, v_{k_d}\}$ be the neighboring edges of v_j (ordered according to $<_E$). Then, $\sigma_j^{(i)}$ is the unique string generated by the grammar having $d+1$ non-terminals $S_0, S_1, \ldots, S_d$, start symbol S_d and the following production rules:

- $S_0 \longrightarrow \langle t_i, \mathtt{acq}(\ell_{i\%c})\rangle \cdot \langle t_i, \mathtt{acq}(\ell_{(i+1)\%c})\rangle \cdot \langle t_i, \mathtt{rel}(\ell_{(i+1)\%c})\rangle \cdot \langle t_i, \mathtt{rel}(\ell_{i\%c})\rangle$.
- for each $1 \leq r \leq d$, $S_r \longrightarrow \langle t_i, \mathtt{acq}(\ell_{\{v_j,v_{k_r}\}})\rangle \cdot S_{r-1} \cdot \langle t_i, \mathtt{rel}(\ell_{\{v_j,v_{k_r}\}})\rangle$.

Fig. 2a illustrates this construction for a graph with 3 nodes and parameter $c = 3$. Finally, observe that the lock-nesting depth in σ is bounded by 2 + the degree of G. □

Theorem 3.1 implies that the problem is not only NP-hard, but also unlikely to be *fixed parameter tractable* in the size k of the deadlock pattern. In fact, under the well-believed Exponential Time Hypothesis (ETH), the parametrized problem INDEPENDENT-SET(c) cannot be solved in time $f(c) \cdot n^{o(c)}$ [Chen et al. 2006]. The above reduction preserves the parameter $k = c$, thus under ETH, detecting deadlock patterns of size k is unlikely to be solvable in time complexity $f(k) \cdot \mathcal{N}^{g(k)}$, where $g(k)$ is $o(k)$ (such as $g(k) = \sqrt{k}$ or even $g(k) = k/\log(k)$). The problem of checking the existence of deadlock patterns is, intuitively, a precursor to the deadlock prediction problem. Thus, an approach for deadlock prediction that first identifies the existence of arbitrary deadlock patterns and then verifying their feasibility is unlikely to be tractable. In practice, the synchronization patterns corresponding to the hard instances are uncommon in executions, and our proposed algorithms (Section 4 and Section 5) can effectively expose predictable deadlocks (Section 6).

Fine-grained hardness for deadlock pattern detection. We next consider the problem of detecting deadlock patterns of size 2, as these form the most common case in practice [Lu et al. 2008]. Observe that Theorem 3.1 has no implications on this case, as here k is fixed. The problem admits a folklore $O(\mathcal{N}^2)$ time algorithm, by iterating over all pairs of lock-acquisition events of the input trace, and checking whether any such pair forms a deadlock pattern. Perhaps surprisingly, here we show that, despite its simplicity, this algorithm is optimal, i.e., we cannot hope to improve over this quadratic bound. This result is based on a reduction from the popular Orthogonal Vectors (OV) problem. Given two sets of d-dimensional vectors $A, B \subseteq \{0,1\}^d$ of cardinality $|A| = |B| = n$, the OV problem asks if there are $a \in A, b \in B$ such that $a \cdot b = \sum_i a[i] \cdot b[i] = 0$. The OV hypothesis states that for any $\epsilon > 0$, there is no $O(n^{2-\epsilon} \cdot \text{poly}(d))$ algorithm for solving OV. This is also a consequence of the famous Strong Exponential Time Hypothesis (SETH) [Williams 2005]. We next show that detecting deadlock patterns of size 2 is at least as hard as solving OV.

Theorem 3.2. *Given a trace σ of size $\mathcal{N}$ and $\mathcal{L}$ locks, for any $\epsilon > 0$, there is no algorithm that determines in $O(\mathcal{N}^{2-\epsilon} \cdot poly(\mathcal{L}))$ time whether σ has a deadlock pattern of size 2, under the OV hypothesis.*

Proof. We show a fine-grained reduction from the Orthogonal Vectors Problem to the problem of checking for deadlock patterns of size 2. For this, we start with two sets $A, B \subseteq \{0,1\}^d$ of d-dimensional vectors with $|A| = |B| = n$. We write the i^{th} vector in A as A_i and that in B as B_i.

Construction. We will construct a trace σ such that σ has a deadlock pattern of length 2 iff (A, B) is a positive OV instance. The trace σ is of the form $\sigma = \sigma^A \cdot \sigma^B$ and uses 2 threads $\{t_A, t_B\}$ and $d+2$ distinct locks $\ell_1, \ldots, \ell_d, m_0, m_1$. Intuitively, σ^A and σ^B encode the given sets of vectors A and B. The sub-traces $\sigma^A = \sigma_1^A \cdot \sigma_2^A \cdots \sigma_n^A$ and $\sigma^B = \sigma_1^B \cdot \sigma_2^B \cdots \sigma_n^B$ are defined as follows. For each

$i \in \{1, 2, \ldots, n\}$ and $Z \in \{A, B\}$, the sub-trace σ_i^Z is the unique string generated by the grammar having $d+1$ non-terminals $S_0, S_1, \ldots, S_d$, start symbol S_d and the following production rules:

- $S_0 \longrightarrow \langle t_Z, \texttt{acq}(m)\rangle \cdot \langle t_Z, \texttt{acq}(m')\rangle \cdot \langle t_Z, \texttt{rel}(m')\rangle \cdot \langle t_Z, \texttt{rel}(m)\rangle$, where $(m, m') = (m_0, m_1)$ if $Z = A$, and $(m, m') = (m_1, m_0)$.
- for each $1 \leq j \leq d$, $S_j \longrightarrow S_{j-1}$ if $Z_i[j] = 0$. Otherwise (if $Z_i[j] = 1$), $S_j \longrightarrow \langle t_Z, \texttt{acq}(\ell_j)\rangle \cdot S_{j-1} \cdot \langle t_Z, \texttt{rel}(\ell_j)\rangle$.

In words, all events of σ^A are performed by thread t_A and those in σ^B are performed by t_B. Next, the i^{th} sub-trace of σ^A, denoted σ_i^A corresponds to the vector A_i as follows — σ_i^A is a nested block of critical sections, with the innermost critical section being on lock ℓ', which is immediately enclosed in a critical section on lock ℓ. Further, in the sub-trace σ_i^A, the lock ℓ_j occurs iff $A_i[j] = 1$. The sub-traces σ_i^B is similarly constructed, except that the order of the two innermost critical sections is inverted. Fig. 2b illustrates the construction for an OV-instance with $n = 2$ and $d = 2$. □

The complexity of deadlock prediction. Finally, we settle the complexity of the prediction problem for deadlocks, and show that, even for deadlock patterns of size 2, the problem is W[1]-hard parameterized by the number of threads. In contrast, recall that the W[1]-hardness of Theorem 3.1 concerns deadlock patterns of arbitrary size. Our result is based on a similar hardness that was established recently for predicting data races [Mathur et al. 2020].

Theorem 3.3. *The problem of checking if a trace σ has a predictable deadlock of size* 2 *is* W[1]*-hard in the number of threads $\mathcal{T}$ appearing in σ, and thus is also* NP*-hard.*

4 SYNCHRONIZATION-PRESERVING DEADLOCKS AND THEIR PREDICTION

Having established the intractability of general deadlock prediction in Section 3, we now define the subclass of predictable deadlocks called synchronization-preserving (*sync-preserving*, for short) in Section 4.1. The key benefit of sync-preserving deadlocks is that, unlike arbitrary deadlocks, they can be detected efficiently; we develop our algorithm `SPDOffline` for this task in Sections 4.2-4.5. Our experiments later indicate that most predictable deadlocks are actually sync-preserving, hence the benefit of fast detection comes at the cost of little-to-no precision loss in practice.

Overview of the algorithm. There are several insights behind our algorithm. First, given a deadlock pattern, one can verify if it is a sync-preserving deadlock in linear time (Section 4.3); this is based on our sound and complete characterization of sync-preserving deadlocks (Section 4.2). Next, instead of verifying single deadlock patterns one-by-one, we consider *abstract deadlock patterns*, which are essentially collections of deadlock patterns that share the same signature; the formal definition is given in Section 4.4. We show that our basic algorithm can be extended to *incrementally* verify *all* the concretizations of an abstract deadlock pattern in linear time (Section 4.4), in a single pass (Lemma 4.3). Finally, we feed this algorithm all the abstract deadlock patterns of the input trace, by constructing an *abstract lock graph* and enumerating cycles in it (Section 4.5).

4.1 Synchronization-Preserving Deadlocks

Our notion of sync-preserving deadlocks builds on the recently introduced concept of sync-preserving correct reorderings [Mathur et al. 2021].

Definition 1 (Sync-preserving Correct Reordering). A correct reordering ρ of a trace σ is *sync-preserving* if for every lock $\ell \in \mathsf{Locks}_\rho$ and every two acquire events $e_1 \neq e_2 \in \mathsf{Events}_\rho$ with $\mathrm{op}(e_1) = \mathrm{op}(e_2) = \texttt{acq}(\ell)$, the order of e_1 and e_2 is the same in σ and ρ, i.e., $e_1 \leq_{\mathrm{tr}}^{\rho} e_2$ iff $e_1 \leq_{\mathrm{tr}}^{\sigma} e_2$.

	t_1	t_2	t_3
1	acq(ℓ_1)		
2	acq(ℓ_2)		
3	rel(ℓ_2)		
4	acq(ℓ_2)		
5	w(y)		
6	rel(ℓ_2)		
7	rel(ℓ_1)		
8		acq(ℓ_3)	
9		w(x)	
10		r(y)	
11		rel(ℓ_3)	
12			acq(ℓ_2)
13			acq(ℓ_3)
14			r(x)
15			rel(ℓ_3)
16			acq(ℓ_1)
17			w(v)
18			rel(ℓ_1)
19			acq(ℓ_1)
20			rel(ℓ_1)
21			rel(ℓ_2)
22		acq(ℓ_4)	
23		acq(ℓ_1)	
24		w(z)	
25		r(v)	
26		rel(ℓ_1)	
27		rel(ℓ_4)	
28	acq(ℓ_1)		
29	acq(ℓ_2)		
30	r(z)		
31	rel(ℓ_2)		
32	rel(ℓ_1)		

$$\eta_1 = \langle t_1, \ell_2, \{\ell_1\}, [e_2, e_4, e_{29}]\rangle$$
$$\eta_2 = \langle t_2, \ell_1, \{\ell_4\}, [e_{23}]\rangle$$
$$\eta_3 = \langle t_3, \ell_1, \{\ell_2\}, [e_{16}, e_{19}]\rangle$$
$$\eta_4 = \langle t_3, \ell_3, \{\ell_2\}, [e_{13}]\rangle$$

$$D^{\text{abs}} = \langle \eta_1, \eta_3\rangle$$

$$D_1 = \langle e_2, e_{16}\rangle \quad D_4 = \langle e_4, e_{19}\rangle$$
$$D_2 = \langle e_2, e_{19}\rangle \quad D_5 = \langle e_{29}, e_{16}\rangle$$
$$D_3 = \langle e_4, e_{16}\rangle \quad D_6 = \langle e_{29}, e_{19}\rangle$$

Fig. 3. A trace σ_3, its abstract acquires η_i, unique abstract deadlock pattern D^{abs}, concrete patterns $D_i \in D^{\text{abs}}$.

A sync-preserving correct reordering preserves the order of those critical sections (on the same lock) that actually appear in the reordering, but allows intermediate critical sections to be dropped completely. This style of reasoning is more permissive than the space of reorderings induced by the Happens-Before partial order [Lamport 1978], that implicitly enforces that all intermediate critical sections on a lock be present. Sync-preserving deadlocks can now be defined naturally.

Definition 2 (Sync-preserving Deadlocks)**.** Let σ be a trace and $D = \langle e_0, e_1, \ldots, e_{k-1}\rangle$ be a deadlock pattern. We say that D is a sync-preserving deadlock of σ if there is a sync-preserving correct reordering ρ of σ such that each of $e_0, \ldots, e_{k-1}$ is σ-enabled in ρ.

Example 2. Consider the trace σ_2 in Fig. 1b. The deadlock pattern $D = \langle e_4, e_{18}\rangle$ is a sync-preserving deadlock, witnessed by the sync-preserving correct reordering $\rho_3 = e_1e_2e_3e_8e_9\, e_{12}..e_{15}\, e_{16}e_{17}$. Now consider the trace σ_3 from Fig. 3 and the deadlock pattern $D_5 = \langle e_{29}, e_{16}\rangle$. This is a predictable deadlock, witnessed by the correct reordering $\rho_5 = e_1..e_7e_8..e_{11}e_{12}..e_{15}\, e_{28}$. Observe that ρ_5 is a sync-preserving reordering, which makes D_5 a sync-preserving deadlock. A key aspect in ρ_5 is that the events $e_{22}..e_{27}$ are dropped, as otherwise e_{16} cannot be σ_3-enabled. A similar reasoning applies for the deadlock pattern D_6, and it is also a sync-preserving deadlock. The other deadlock patterns (D_1, D_2, D_3, D_4) are not predictable deadlocks. Intuitively, the reason for this is that realizing these deadlock patterns require executing the read event e_{14}, which then enforces to execute the events $e_8..e_{11}$ and $e_1..e_6$. This prevents the deadlocks from becoming realizable as the events e_2 or e_4 that appear in these deadlock patterns are no longer σ_3-enabled. This point is detailed in Example 3.

4.2 Characterizing Sync-Preserving Deadlocks

There are two fundamental tasks in searching for a correct reordering that witnesses a deadlock — (i) determining the set of events in the correct reordering, and (ii) identifying a total order on such events — both of which are intractable [Mathur et al. 2020]. On the contrary, for sync-preserving deadlocks, we show that (a) the search for a correct reordering can be reduced to the problem of checking if some well-defined set of events (Definition 3) does not contain the events appearing in the deadlock pattern (Lemma 4.2), and that (b) this set can be constructed efficiently.

Definition 3 (Sync-Preserving Closure)**.** Let σ be a trace and $S \subseteq \mathsf{Events}_\sigma$. The sync-preserving closure of S, denoted $\mathsf{SPClosure}_\sigma(S)$ is the smallest set S' such that (a) $S \subseteq S'$, (b) for every

$e, e' \in \mathsf{Events}_\sigma$ such that $e <^\sigma_{\mathsf{TO}} e'$ or $e = \mathsf{rf}_\sigma(e')$, if $e' \in S'$, then $e \in S'$, and (c) for every lock ℓ and every two distinct events $e, e' \in S'$ with $\mathsf{op}(e) = \mathsf{op}(e') = \mathsf{acq}(\ell)$, if $e \leq^\sigma_{\mathsf{tr}} e'$ then $\mathsf{match}_\sigma(e) \in S'$.

Definition 3 resembles the notion of correct reorderings (Definition 1). Indeed, Lemma 4.1 justifies using this set — it is both a necessary and a sufficient set for sync-preserving correct reorderings.

Lemma 4.1. *Let σ be a trace and let $S \subseteq \mathsf{Events}_\sigma$. For any sync-preserving correct reordering ρ of σ, if $S \subseteq \mathsf{Events}_\rho$, then $\mathsf{SPClosure}_\sigma(S) \subseteq \mathsf{Events}_\rho$. Further, there is a sync-preserving correct reordering ρ of σ such that $\mathsf{Events}_\rho = \mathsf{SPClosure}_\sigma(S)$.*

For an intuition, consider again Figure 3 and the sync-preserving correct reordering $\rho_5 = e_1..e_7e_8..e_{11}e_{12}..e_{15}\,e_{28}$ computed in Example 2. According to Lemma 4.1, $\mathsf{SPClosure}_{\sigma_3}(S) \subseteq \mathsf{Events}_{\rho_5}$ holds for all S such that $S \subseteq \mathsf{Events}_{\rho_5}$. For example, if we take $S = \{e_1, e_{15}\}$ then observe that $S \subseteq \mathsf{Events}_{\rho_5}$ and $\mathsf{SPClosure}_{\sigma_3}(S) = \{e_1, \ldots, e_6,\ e_8, \ldots e_{15}\} \subseteq \mathsf{Events}_{\rho_5}$ holds.

Based on Lemma 4.1, we present a sound and complete characterization of sync-preserving deadlocks (Lemma 4.2). For a set $S \subseteq \mathsf{Events}_\sigma$, we let $\mathsf{pred}_\sigma(S)$ denote the set of immediate thread predecessors of events in S. That is, $\mathsf{pred}_\sigma(S) = \{e \in \mathsf{Events}_\sigma \mid \exists f \in S, e <^\sigma_{\mathsf{tr}} f \text{ and } \forall e' <^\sigma_{\mathsf{tr}} f, e' \leq^\sigma_{\mathsf{TO}} e\}$.

Lemma 4.2. *Let σ be a trace and let $D = \langle e_0, \ldots, e_{k-1}\rangle$ be a deadlock pattern of size k in σ. D is a sync-preserving deadlock of σ iff $\mathsf{SPClosure}_\sigma(\mathit{pred}_\sigma(S)) \cap S = \emptyset$, where $S = \{e_0, \ldots, e_{k-1}\}$.*

Example 3. Consider the trace σ_2 in Fig. 1b, and the deadlock pattern $D = \langle e_4, e_{18}\rangle$. We have $\mathsf{SPClosure}_{\sigma_2}(\mathsf{pred}_{\sigma_2}(\{e_4, e_{18}\})) = \{e_1, e_2, e_3, e_8, e_9, e_{12}, \ldots, e_{17}\}$. Since we have that $e_4, e_{18} \notin \mathsf{SPClosure}_{\sigma_2}(\mathsf{pred}_{\sigma_2}(\{e_4, e_{18}\}))$, D is a sync-preserving deadlock. Now consider the trace σ_3 in Fig. 3, and the deadlock patterns $D_1 = \langle e_2, e_{16}\rangle$, $D_5 = \langle e_{29}, e_{16}\rangle$, and $D_6 = \langle e_{29}, e_{19}\rangle$. We have $\mathsf{SPClosure}_{\sigma_3}(\mathsf{pred}_{\sigma_3}(\{e_2, e_{16}\})) = \{e_1, \ldots, e_6, e_8, \ldots, e_{15}\}$, $\mathsf{SPClosure}_{\sigma_3}(\mathsf{pred}_{\sigma_3}(\{e_{29}, e_{16}\})) = \{e_1, \ldots, e_{15}, e_{28}\}$, and $\mathsf{SPClosure}_{\sigma_3}(\mathsf{pred}_{\sigma_3}(\{e_{29}, e_{19}\})) = \{e_1, \ldots, e_{18}, e_{28}\}$. Since $e_2 \in \mathsf{SPClosure}_{\sigma_3}(\mathsf{pred}_{\sigma_3}(\{e_2, e_{16}\}))$, D_1 is not a sync-preserving deadlock. However, $e_{29}, e_{16} \notin \mathsf{SPClosure}_{\sigma_3}(\mathsf{pred}_{\sigma_3}(\{e_{29}, e_{16}\}))$, and $e_{29}, e_{19} \notin \mathsf{SPClosure}_{\sigma_3}(\mathsf{pred}_{\sigma_3}(\{e_{29}, e_{19}\}))$, thus D_5 and D_6 are sync-preserving deadlocks (as we also concluded in Example 2).

4.3 Verifying Deadlock Patterns

Given a deadlock pattern, we check if it constitutes a sync-preserving deadlock by constructing the sync-preserving closure (Lemma 4.2) in linear time. Based on Definition 3, this can be done in an iterative manner. We (i) start with the set of $\leq_{\mathsf{TO}}$ predecessors of the events in the deadlock pattern, and (ii) iteratively add $\leq_{\mathsf{TO}}$ and rf predecessors of the current set of events. Additionally, we identify and add the release events that must be included in the set. We utilize *timestamps* to ensure that the *entire* fixpoint computation is performed in linear time.

Thread-read-from timestamps. Given a set Threads of threads, a timestamp is simply a mapping $T : \mathsf{Threads} \to \mathbb{N}$. Given timestamps T_1, T_2, we use the notations $T_1 \sqsubseteq T_2$ and $T_1 \sqcup T_2$ for pointwise comparison and pointwise maximum, respectively. For a set U of timestamps, we write $\bigsqcup U$ to denote the pointwise maximum over all elements of U. Let $\leq^\sigma_{\mathsf{TRF}}$ be the reflexive transitive closure of the relation $(\leq^\sigma_{\mathsf{TO}} \cup \{(\mathsf{rf}_\sigma(e), e) \mid \exists x \in \mathsf{Vars}_\sigma, \mathsf{op}(e) = \mathsf{r}(x)\})$; observe that $\leq^\sigma_{\mathsf{TRF}}$ is a partial order. We define the timestamp TS^e_σ of an event e in σ to be a $\mathsf{Threads}_\sigma$-indexed timestamp as follows: $\mathsf{TS}^e_\sigma(t) = |\{f \mid f \leq^\sigma_{\mathsf{TRF}} e\}|$. This ensures that for two events $e, e' \in \mathsf{Events}_\sigma$, $e \leq^\sigma_{\mathsf{TRF}} e'$ iff $\mathsf{TS}^e_\sigma \sqsubseteq \mathsf{TS}^{e'}_\sigma$. For a set $S \subseteq \mathsf{Events}_\sigma$, we overload the notation and say the timestamp of S is $\mathsf{TS}^S_\sigma = \bigsqcup \{\mathsf{TS}^e_\sigma\}_{e \in S}$.

Given a trace σ with $\mathcal{N}$ events and $\mathcal{T}$ threads we can compute these timestamps for all the events in $O(\mathcal{N} \cdot \mathcal{T})$ time, using a simple vector clock algorithm [Fidge 1991; Mattern 1989].

Computing sync-preserving closures. Recall the basic template of the fixpoint computation. In each iteration, we identify the set of release events that must be included in the set, together with their $\leq^{\sigma}_{\mathsf{TRF}}$-closure. In order to identify such events efficiently, for every thread t and lock ℓ, we maintain a FIFO queue $\mathsf{CSHist}_{t,\ell}$ (*critical section history* of t and ℓ) to store the sequence of events that acquire ℓ in thread t. In each iteration, we traverse each list to determine the last acquire event that belongs to the current set. For a given lock, we need to add the matching release events of all thus identified events to the closure, except possibly the matching release event of the latest acquire event (see Definition 3). This computation is performed using timestamps, as shown in Algorithm 1. Starting with a set S, the algorithm runs in time $O(|S| \cdot \mathcal{T} + \mathcal{T} \cdot \mathcal{A})$, where $\mathcal{T}$ and $\mathcal{A}$ are respectively the number of threads and acquire events in σ.

Algorithm 1: CompSPClosure: Computing sync-preserving closure.

Input: Trace σ, Timestamp T_0

- **let** $\{\mathsf{CSHist}_{t,\ell}\}_{\ell \in \mathsf{Locks}_\sigma, t \in \mathsf{Threads}_\sigma}$ be the lock-acquisition histories in σ
- $T \leftarrow T_0$
- **repeat**
  - **for** $\ell \in \mathsf{Locks}$ **do**
    - **foreach** $t \in \mathsf{Threads}$ **do**
      - **let** e_t be the last event in $\mathsf{CSHist}_{t,\ell}$ with $\mathsf{TS}^{e_t}_{\sigma} \sqsubseteq T$
      - Remove all earlier events in $\mathsf{CSHist}_{t,\ell}$
    - **let** e_{t*} be the last event in $\{e_t\}_{t \in \mathsf{Threads}_\sigma}$ according to $\leq^{\sigma}_{\mathsf{tr}}$
    - $T := T \sqcup \bigsqcup\{\mathsf{TS}^{\mathsf{match}_\sigma(e_t)}_{\sigma} \mid e_t \neq e_{t*}\}$
- **until** T *does not change*
- **return** T

Algorithm 2: CheckAbsDdlck: Checking an abstract deadlock pattern.

Input: Trace σ, D^{abs} of length k

- **let** $F_0, \ldots, F_{k-1}$ be the sequences of acquires in D^{abs}
- **let** $n_0, \ldots, n_{k-1}$ be the lengths of $F_0, \ldots, F_{k-1}$
- **foreach** $j \in \{0, \ldots, k-1\}$ **do** $i_j \leftarrow 1$
- $T \leftarrow \lambda t, 0$ **while** $\bigwedge_{j=0}^{k-1} i_j < n_j$ **do**
  - **let** $e_0 = F_0[i_0], \ldots, e_{k-1} = F_{k-1}[i_{k-1}]$
  - $S \leftarrow \mathsf{pred}_\sigma\{e_0, \ldots, e_{k-1}\}$
  - $T \leftarrow \mathtt{CompSPClosure}(\sigma, T \sqcup \mathsf{TS}^{S}_{\sigma})$
  - **if** $\forall j < k, \mathsf{TS}^{e_j}_{\sigma} \sqsubseteq T$ **then**
    - **report** pattern $D = e_0, \ldots, e_{k-1}$ and **exit**
  - **foreach** $j \in \{0, \ldots, k-1\}$ **do**
    - $i_j = \min\{l \leq n_j \mid \mathsf{TS}^{F_j[l]}_{\sigma} \not\sqsubseteq T\}$

Checking a deadlock pattern. After computing the timestamp T of the closure (output of Algorithm 1, starting with the set of events in the given deadlock pattern), determining whether a given deadlock pattern $D = e_0, \ldots, e_{k-1}$ is a sync-preserving deadlock can be performed in time $O(k \cdot \mathcal{T})$ — simply check if $\forall i, \mathsf{TS}_\sigma(e_i) \not\sqsubseteq T$. This gives an algorithm for checking if a deadlock pattern of length k is sync-preserving that runs in time $O(\mathcal{T} \cdot \mathcal{N} + k \cdot \mathcal{T} + \mathcal{T} \cdot \mathcal{A}) = O(\mathcal{N} \cdot \mathcal{T})$.

4.4 Verifying Abstract Deadlock Patterns

Abstract acquires and abstract deadlock patterns. Given a thread t, a lock ℓ and a set of locks $L \subseteq \mathsf{Locks}_\sigma \neq \emptyset$ with $\ell \notin L$, we define the *abstract acquire* $\eta = \langle t, \ell, L, F\rangle$, where $F = [e_1, \ldots, e_n]$ is the sequence of all events $e_i \in \mathsf{Events}_\sigma$ (in trace-order) such that for each i, we have (i) $\mathsf{thread}(e_i) = t$, (ii) $\mathsf{op}(e_i) = \mathtt{acq}(\ell)$, and (iii) $\mathsf{HeldLks}_\sigma(e_i) = L$. In words, the abstract acquire η contains the sequence of all acquire events of a specific thread, that access a specific lock and hold the same set of locks when executed, ordered as per thread order. An *abstract deadlock pattern* of size k in a trace σ is a sequence $D^{\mathsf{abs}} = \eta_0, \ldots, \eta_{k-1}$ of abstract acquires $\eta_i = \langle t_i, \ell_i, L_i, F_i\rangle$ such that $t_0, \ldots, t_{k-1}$ are distinct threads, $\ell_0, \ldots, \ell_{k-1}$ are distinct locks, and $L_0, L_1, \ldots, L_{k-1} \subseteq \mathsf{Locks}_\sigma$ are such that $\ell_i \notin L_i$,

$\ell_i \in L_{(i+1)\%k}$ for every i, and $L_i \cap L_j = \emptyset$ for every $i \neq j$. Thus, an abstract deadlock pattern D^{abs} succinctly encodes all concrete deadlock patterns $F_0 \times F_1 \times \cdots \times F_{k-1}$, called *instantiations* of D^{abs}. We also write $D \in D^{\text{abs}}$ to denote that $D \in F_0 \times F_1 \times \cdots \times F_{k-1}$. We say that D^{abs} contains a sync-preserving deadlock if there exists some instantiation $D \in D^{\text{abs}}$ that is a sync-preserving deadlock. See Fig. 3 for an example. Our next result is stated below, followed by its proof idea.

Lemma 4.3. *Consider a trace σ with $\mathcal{N}$ events and $\mathcal{T}$ threads, and an abstract deadlock pattern D^{abs} of σ. We can determine if D^{abs} contains a sync-preserving deadlock in $O(\mathcal{T} \cdot \mathcal{N})$ time.*

An abstract deadlock pattern of length $k \geq 2$ can have $\mathcal{N}^k$ instantiations, giving a naive enumerate-and-check algorithm running in time $O(\mathcal{T} \cdot \mathcal{N}^{k+1})$, which is prohibitively large. Instead, we exploit (i) the monotonicity properties of the sync-preserving closure (Proposition 4.4) and (ii) instantiations of an abstract pattern (Corollary 4.5) that allow for an *incremental* algorithm that iteratively checks successive instantiations of a given abstract deadlock pattern, while spending total $O(\mathcal{N} \cdot \mathcal{T})$ time. The first observation allows us to re-use a prior computation when checking later deadlock patterns.

Proposition 4.4. *For a trace σ and sets $S, S' \subseteq \mathsf{Events}_\sigma$. If for every event $e \in S$, there is an event $e' \in S'$ such that $e \leq^{\sigma}_{\mathsf{TO}} e'$, then $\mathsf{SPClosure}_\sigma(S) \subseteq \mathsf{SPClosure}_\sigma(S')$.*

Consider σ_3 in Figure 3 and let $S = \mathsf{pred}_{\sigma_3}(\{e_{29}, e_{16}\})$, and $S' = \mathsf{pred}_{\sigma_3}(\{e_{29}, e_{19}\})$. The sets S, S' satisfy the conditions of Proposition 4.4, hence $\mathsf{SPClosure}_{\sigma_3}(S) \subseteq \mathsf{SPClosure}_{\sigma_3}(S')$, as computed in Example 3. Next, we extend Proposition 4.4 to avoid redundant computations when a sync-preserving deadlock is not found and later deadlock patterns must be checked. Given two deadlock patterns $D_1 = e_0, \ldots, e_{k-1}$ and $D_2 = f_0, \ldots, f_{k-1}$ of the same length k, we say $D_1 \prec D_2$ if they are instantiations of a common abstract pattern D^{abs} (i.e., $D_1, D_2 \in D^{\text{abs}}$) and for every $i < k$, $e_i \leq^{\sigma}_{\mathsf{TO}} f_i$.

Corollary 4.5. *Let σ be a trace and let $D_1 = e_0, \ldots, e_{k-1}$ and $D_2 = f_0, \ldots f_{k-1}$ be deadlock patterns of size k in σ such that $D_1 \prec D_2$. Let $S_1 = \{e_0, \ldots, e_{k-1}\}$ and $S_2 = \{f_0, \ldots, f_{k-1}\}$. If $\mathsf{SPClosure}_\sigma(pred_\sigma(S_1)) \cap S_2 \neq \emptyset$, then $\mathsf{SPClosure}_\sigma(pred_\sigma(S_2)) \cap S_2 \neq \emptyset$.*

We now describe how Proposition 4.4 and Corollary 4.5 are used in our algorithms, and illustrate them later in Example 4. Algorithm 2 checks if an abstract deadlock pattern contains a sync-preserving deadlock. The algorithm iterates over the sequences $F_0, \ldots, F_{k-1}$ of acquires (one for each abstract acquire) in trace order. For this, it maintains indices $i_0, \ldots, i_{k-1}$ that point to entries in $F_0, \ldots, F_{k-1}$. At each step, it determines whether the current deadlock pattern $D = e_0, \ldots, e_{k-1}$ constitutes a sync-preserving deadlock by computing the sync-preserving closure of the thread-local predecessors of the events of the deadlock pattern. The algorithm reports a deadlock if the sync-preserving closure does not contain any of $e_0, \ldots, e_{k-1}$. Otherwise, it looks for the next eligible deadlock pattern, which it determines based on Corollary 4.5. In particular, it advances the pointer i_j all the way until an entry which is outside of the closure computed so far. Observe that the timestamp T of the closure computed in an iteration is being used in later iterations; this is a consequence of Proposition 4.4. Furthermore, in the call to the Algorithm 1 at Line 7, we ensure that the list of acquires $\mathsf{CSHist}_{t,\ell}$, used in the function `CompSPClosure` is reused across iterations, and not re-assigned to the original list of all acquire events. The correctness of this optimization follows from Proposition 4.4. Let us now calculate the running time of Algorithm 2. Each of the $\mathsf{CSHist}_{t,\ell}$ in `CompSPClosure` is traversed at most once. Next, each element of the sequences $F_0, \ldots, F_{k-1}$ is also traversed at most once. For each of these acquires, the algorithm spends $O(\mathcal{T})$ time for vector clock updates. The total time required is thus $O(\mathcal{N} \cdot \mathcal{T})$. This concludes the proof of Lemma 4.3.

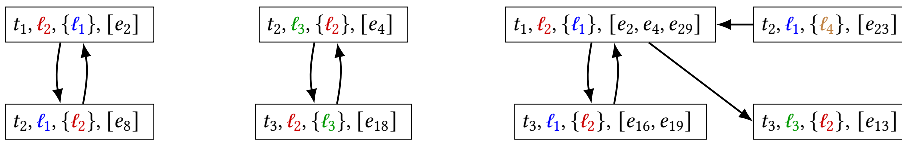


Fig. 4. Abstract lock graphs of the traces from Fig. 1a (left), Fig. 1b (middle) and Fig. 3 (right).

4.5 The Algorithm `SPDOffline`

We now present the final ingredients of `SPDOffline`. We construct the *abstract lock graph*, enumerate cycles in it, check whether any cycle is an abstract deadlock pattern, and if so, whether it contains sync-preserving deadlocks.

Abstract lock graph. The abstract lock graph of σ is a directed graph $\mathsf{ALG}_\sigma = (V_\sigma, E_\sigma)$, where

- $V_\sigma = \{\langle t_1, \ell_1, L_1, F_1\rangle, \dots, \langle t_k, \ell_k, L_k, F_k\rangle\}$ is the set of abstract acquires of σ, and
- for every $\eta_1 = \langle t_1, \ell_1, L_1, F_1\rangle, \eta_2 = \langle t_2, \ell_2, L_2, F_2\rangle \in V_\sigma$, we have $(\eta_1, \eta_2) \in E_\sigma$ iff $t_1 \neq t_2$, $\ell_1 \in L_2$, and $L_1 \cap L_2 = \emptyset$.

A node $\langle t_1, \ell_1, L_1, F_1\rangle$ signifies that there is an event $\mathtt{acq}_1(\ell_1)$ performed by thread t_1 while holding the locks in L_1. The last component F_1 is a list which contains all such events $\mathtt{acq}_1$ in order of appearance in σ. An edge (η_1, η_2) signifies that the lock ℓ_1 acquired by each of the events $\mathtt{acq}_1 \in F_1$ was held by t_2 when it executed each of $\mathtt{acq}_2 \in F_2$ while not holding a common lock. The abstract lock graph can be constructed incrementally as new events appear in σ. For $\mathcal{N}$ events, $\mathcal{L}$ locks and nesting depth d, the graph has $|V_\sigma| = O(\mathcal{T} \cdot \mathcal{L}^d)$ vertices, $|E_\sigma| = O(|V_\sigma| \cdot \mathcal{L}^{d-1})$ edges and can be constructed in $O(\mathcal{N} \cdot d)$ time. See Fig. 4 for examples. In the left graph, the cycle marks an abstract deadlock pattern and its single concrete deadlock pattern $D^{\mathsf{abs}} = \{e_2\} \times \{e_8\}$, and similarly for the middle graph where $D^{\mathsf{abs}} = \{e_4\} \times \{e_{18}\}$. In the right graph, there is a unique cycle which marks an abstract deadlock pattern of 6 concrete deadlock patterns $D^{\mathsf{abs}} = \{e_2, e_4, e_{29}\} \times \{e_{16}, e_{19}\}$.

Algorithm 3: Algorithm `SPDOffline`.

Input: A trace σ.
Output: All abstract deadlock patterns of σ that contain a sync-preserving deadlock.

```
Construct the abstract lock graph ALG_σ
foreach cycle C = ⟨η_0, ..., η_{k-1}⟩ in G do
    Let η_i = ⟨t_i, ℓ_i, L_i, F_i⟩
    if ∀i ≠ j we have t_i ≠ t_j and ℓ_i ≠ ℓ_j and L_i ∩ L_j = ∅ then // C is an abstract deadlock pattern
        if CheckAbsDdlck(C) then Report that C contains a sync-preserving deadlock
```

Algorithm `SPDOffline`. It is straightforward to verify that every abstract deadlock pattern of σ appears as a (simple) cycle in ALG_σ. However, the opposite is not true. A cycle $C = \eta_0, \eta_1, \dots, \eta_{k-1}$ of ALG_σ, where $\eta_i = \langle t_i, \ell_i, L_i, F_i\rangle$ defines an abstract deadlock pattern if additionally every thread t_i is distinct, all every lock ℓ_i is distinct, and all sets L_i are pairwise disjoint. This gives us a simple recipe for enumerating all abstract deadlock patterns, by using Johnson's algorithm [Johnson 1975] to enumerate every simple cycle C in ALG_σ, and check whether C is an abstract deadlock pattern. We thus arrived at our offline algorithm `SPDOffline` (Algorithm 3). The running time depends linearly on the length of σ and the number of cycles in ALG_σ.

THEOREM 4.6. *Consider a trace σ of $\mathcal{N}$ events, $\mathcal{T}$ threads and Cyc_σ cycles in ALG_σ. The algorithm `SPDOffline` reports all sync-preserving deadlocks of σ in time $O(\mathcal{N} \cdot \mathcal{T} \cdot Cyc_\sigma)$.*

Although, in principle, we can have exponentially many cycles in ALG_σ, because the nodes of ALG_σ are *abstract* acquire events (as opposed to *concrete*), we expect that the number of cycles (and thus abstract deadlock patterns) in ALG_σ remains small, even though the number of *concrete* deadlock patterns can grow exponentially. Since `SPDOffline` spends linear time per abstract deadlock pattern, we have an efficient procedure overall for constant $\mathcal{T}$ and $\mathcal{L}$. We evaluate Cyc_σ experimentally in Section 6, and confirm that it is very small compared to the number of concrete deadlock patterns in σ. Nevertheless, Cyc_σ can become exponential when $\mathcal{T}$ and $\mathcal{L}$ are large, making Algorithm 3 run in exponential time. Note that this barrier is unavoidable in general, as proven in Theorem 3.1.

Example 4. We illustrate how the lock graph is integrated inside `SPDOffline`. Consider the trace σ_3 in Fig. 3. It contains 6 concrete deadlock patterns $D_1 \dots D_6$. A naive algorithm would enumerate each pattern explicitly until it finds a deadlock. However, the tight interplay between the abstract lock graph and sync-preservation enables a more efficient procedure. `SPDOffline` starts by computing the sync-preserving closure of D_1, $\mathsf{SPClosure}_{\sigma_3}(\mathsf{pred}_{\sigma_3}(\{e_2, e_{16}\})) = \{e_1, \dots, e_6, e_8, \dots, e_{15}\}$. As $e_2 \in \mathsf{SPClosure}_{\sigma_3}(\mathsf{pred}_{\sigma_3}(\{e_2, e_{16}\}))$, we conclude that D_1 is not a sync-preserving deadlock. The algorithm further deduces that the deadlock patterns D_2, D_3 and D_4 are also not sync-preserving deadlocks, as follows. $D_2 = \langle e_2, e_{19} \rangle$ shares a common event e_2 with D_1 but contains the event e_{19} instead of e_{16}, while $e_5 \in \mathsf{SPClosure}_{\sigma_3}(\mathsf{pred}_{\sigma_3}(\{e_2, e_{16}\}))$. Since $e_{16} \leq^{\sigma_3}_{\mathsf{TO}} e_{19}$, and the sync-preserving closure grows monotonically (Proposition 4.4), the sync-preserving closure of e_2 and e_{19} will also contain e_5 (and thus e_2). Therefore, D_2 cannot be a sync-preserving deadlock. This reasoning is formalized in Corollary 4.5, and also applies to D_3 and D_4. Next, the algorithm proceeds with D_5. The above reasoning does not hold for D_5 as $\mathsf{SPClosure}_{\sigma_3}(\mathsf{pred}_{\sigma_3}(\{e_2, e_{16}\})) \cap S_5 = \varnothing$ where $S_5 = \{e_{29}, e_{16}\}$. The algorithm then computes the sync-preserving closure of D_5, reports a deadlock (Example 3) and stops analyzing this abstract deadlock pattern. In the end, we have only explicitly enumerated the deadlock patterns D_1 and D_5.

Remark 1. Although the concept of lock graphs exists in the literature [Bensalem and Havelund 2005; Cai and Chan 2014; Cai et al. 2020; Havelund 2000], our notion of *abstract* lock graphs is novel and tailored to sync-preserving deadlocks. The closest concept to abstract lock graphs is that of equivalent cycles [Cai and Chan 2014]. However, equivalent cycles unify all the concrete patterns of a given abstract pattern and lead to unsound deadlock detection, which was indeed their use.

5 ON-THE-FLY DEADLOCK PREDICTION

Although `SPDOffline` is efficient, both theoretically (Theorem 4.6) and in practice (Table 1), it runs in two passes, akin to other predictive deadlock-detection methods [Cai et al. 2021; Kalhauge and Palsberg 2018]. In a runtime monitoring setting, it is desirable to operate in an *online* fashion. Recall that `CheckAbsDdlck(·)` indeed operates online (Section 4.4), while the offline nature of `SPDOffline` is tied to the offline construction of the abstract lock graph ALG_σ. To achieve the golden standard of online, linear-time, sound deadlock prediction, we focus on deadlocks of size 2. This focus is barely restrictive as most deadlocks in the wild have size 2 [Lu et al. 2008]. Further, deadlocks of size 2 enjoy the following computational benefits: (a) cycles of length 2 can be detected instantaneously without performing graph traversals, and (b) *every* cycle of length 2 is an abstract deadlock pattern.

Algorithm `SPDOnline`. The algorithm `SPDOnline` maintains all abstract acquires of the form $\eta = \langle t, \ell_1, \{\ell_2\}, F \rangle$, i.e., we only focus on one lock ℓ_2 that is protecting each such acquire. When a new acquire event $e = \langle t, \mathsf{acq}(\ell_1) \rangle$ is encountered, the algorithm iterates over all the locks $\ell_2 \in \mathsf{HeldLks}_\sigma(e)$ that are held in e, and append the event e to the sequence F of the corresponding abstract acquire $\eta = \langle t, \ell_1, \{\ell_2\}, F \rangle$; F is maintained as FIFO queue. Recall that we use timestamps on the acquire events in F to determine membership in our closure computation. Our online algorithm

Algorithm 4: `SPDOnline`.

1 **function** `checkDeadlock`$(Lst, I, \langle t_1, \ell_1, t_2, \ell_2 \rangle)$
2   **while not** $Lst \cdot$ `isEmpty()` **do**
3     $(C_{\text{pred}}, C) := Lst \cdot$ `first()`
4     $I :=$ `CompSPClosure`$(I \sqcup C_{pred}, \langle t_1, \ell_1, t_2, \ell_2 \rangle)$
5   **if** $C \not\sqsubseteq I$ **then** **declare** 'Deadlock' and **break**
6     $Lst \cdot$ `removeFirst()`
7   **return** I
8 **handler** `write`(t, x)
9   $\mathbb{LW}_x := \mathbb{C}_t$
10   $\mathbb{C}_t := \mathbb{C}_t[t \mapsto \mathbb{C}_t(t) + 1]$
11 **handler** `release`(t, ℓ)
12   $\mathbb{C}_t := \mathbb{C}_t[t \mapsto \mathbb{C}_t(t) + 1]$
13   **foreach** $t_1, t_2 \in$ Threads, $\ell_1, \ell_2 \in$ Locks **do**
14     $\mathsf{CSHist}_{t,\ell}^{\langle t_1,\ell_1,t_2,\ell_2 \rangle} \cdot$ `last()` $\cdot$ `updateRelease`$(\mathbb{C}_t)$
15 **handler** `acquire`(t, ℓ)
16   $C_{\text{pred}} := \mathbb{C}_t$
17   $\mathbb{C}_t := \mathbb{C}_t[t \mapsto \mathbb{C}_t(t) + 1]$
18   $g_\ell := g_\ell + 1$
19   **foreach** $t_1, t_2 \in$ Threads, $\ell_1, \ell_2 \in$ Locks **do**
20     $\mathsf{CSHist}_{t,\ell}^{\langle t_1,\ell_1,t_2,\ell_2 \rangle} \cdot$ `addLast`$((g_\ell, \mathbb{C}_t, \bot))$
21   **foreach** $u \in$ Threads, $\ell' \in$ HeldLks **do**
22     $\mathsf{AcqHist}_{t,\ell,\ell'}^{\langle u \rangle} \cdot$ `addLast`$((C_{\text{pred}}, \mathbb{C}_t))$
23   **foreach** $u \neq t \in$ Threads, $\ell' \in$ HeldLks **do**
24     $I := \mathbb{I}^{\langle u,\ell' t,\ell \rangle} \sqcup C_{\text{pred}}$
25     $\mathbb{I}^{\langle u,\ell' t,\ell \rangle} :=$ `checkDeadlock`$(\mathsf{AcqHist}_{u,\ell',\ell}^{\langle t \rangle}, I, \langle u, \ell' t, \ell \rangle)$
26 **handler** `read`(t, x)
27   $\mathbb{C}_t := \mathbb{C}_t \sqcup \mathbb{LW}_x$

computes these timestamps on-the-fly and stores them in these queues together with the events. Then the algorithm calls `CheckAbsDdlck`(C) on the abstract deadlock pattern C formed between η and $\eta' = \langle t' \neq t, \ell_2, \{\ell_1\}, F' \rangle$, in order to check for sync-preserving deadlocks between the deadlock patterns in $F \times F'$. If `CheckAbsDdlck`(C) reports no deadlock, the contents of F' are emptied as we are guaranteed that F' will not cause a sync-preserving deadlock with any further acquire of thread t on lock ℓ_1. For a trace with $\mathcal{N}$ events, $\mathcal{T}$ threads and $\mathcal{L}$ locks. The algorithm calls `CheckAbsDdlck` for each of the $O(\mathcal{T}^2 \cdot \mathcal{L}^2)$ abstract deadlock patterns of size 2, each call taking $O(\mathcal{N} \cdot \mathcal{T})$ time.

`SPDOnline` is shown in detail in Algorithm 4. The pseudocode contains handlers for processing the different events of σ in a streaming fashion, as well as a helper function for checking deadlocks. The main data structures of the algorithm are (i) vector clocks $\mathbb{C}_t$, $\mathbb{LW}_x$, and $\mathbb{I}^{\langle t_1,\ell_1,t_2,\ell_2 \rangle}$, (ii) scalar g_ℓ, and (iii) FIFO queues of vector clocks $\mathsf{CSHist}_{t,\ell}^{\langle t_1,\ell_1,t_2,\ell_2 \rangle}$ and $\mathsf{AcqHist}_{t,\ell_1,\ell_2}^{\langle u \rangle}$, where t, t_1, t_2, u range over threads, x ranges over variables, and ℓ, ℓ_1, ℓ_2 range over locks. $\mathbb{C}_t$ stores the timestamp TS_σ^e where e is the last event in thread t. $\mathbb{LW}_x$ keeps track of the TS_σ^e where e is the last event such that $\mathsf{op}(e) = \mathtt{w}(x)$. $\mathbb{I}^{\langle t_1,\ell_1,t_2,\ell_2 \rangle}$ stores the computed sync-preserving closures for every tuple $\langle t_1, \ell_1, t_2, \ell_2 \rangle$. The scalar variable g_ℓ keeps track of the index of the last acquire event on lock ℓ. Similar to Algorithm 1, the FIFO queue $\mathsf{CSHist}_{t,\ell}^{\langle t_1,\ell_1,t_2,\ell_2 \rangle}$ is maintained to keep track of the critical section history of thread t and lock ℓ. Lastly, for an acquire event e, $\mathsf{AcqHist}_{t,\ell_1,\ell_2}^{\langle u \rangle}$ maintains a queue of tuples of the form $\langle C_{\text{pred}}, \mathbb{C}_t \rangle$ where C_{pred} and $\mathbb{C}_t$ are the timestamps of $\mathsf{pred}_\sigma(e)$ and e, respectively These tuples are utilized when checking for deadlocks (Line 25).

Theorem 5.1. *Consider a trace σ of $\mathcal{N}$ events, $\mathcal{T}$ threads and $\mathcal{L}$ locks. The online* `SPDOnline` *algorithm reports all sync-preserving deadlocks of size 2 of σ in $O(\mathcal{N} \cdot \mathcal{T}^3 \cdot \mathcal{L}^2)$ time.*

6 EXPERIMENTAL EVALUATION

We first evaluated our algorithms in an offline setting (Section 6.1), where we record execution traces and evaluate different approaches on the *same* input. This eliminates biases due to non-deterministic thread scheduling. Next, we consider an online setting (Section 6.2), where we instrument programs and perform the analyses during runtime. We conducted all our experiments on a standard laptop with 1.8 GHz Intel Core i7 processor and 16 GB RAM.

6.1 Offline Experiments

Experimental setup. The goal of the first set of experiments is to evaluate SPDOffline, and compare it against prior algorithms for dynamic deadlock prediction. In order for our evaluation to be precise we evaluate all algorithms on the *same* execution trace. We implemented SPDOffline in Java inside the RAPID analysis tool [Mathur 2019], following closely the pseudocode in Algorithm 3. RAPID takes as input execution traces, as defined in Section 2. These also include fork, join, and lock-request events. We compare SPDOffline with two state-of-the-art, theoretically-sound albeit computationally more expensive, deadlock predictors, SeqCheck [Cai et al. 2021] and Dirk [Kalhauge and Palsberg 2018], both of which also work on execution traces.

On the theoretical side, the complexity of SeqCheck is $\widetilde{O}(\mathcal{N}^4)$, as opposed to the $\widetilde{O}(\mathcal{N})$ complexity of SPDOffline. Moreover, SeqCheck only predicts deadlocks of size 2, and though it could be extended to handle deadlocks of any size, this would degrade performance further. SeqCheck may miss sync-preserving deadlocks even of size 2, but can detect deadlocks that are not sync-preserving. Thus SeqCheck and SPDOffline are theoretically incomparable in their detection capability. We refer to [Tunç et al. 2023a] for examples. We noticed that SeqCheck fails on traces with non-well-nested locks — we encountered one such case in our dataset. Dirk's algorithm is theoretically complete, i.e., it can find all predictable deadlocks in a trace. In addition, it can find deadlocks beyond the predictable ones, by reasoning about event values. However, Dirk relies on heavyweight SMT-solving and employs windowing techniques to scale to large traces. Due to windowing, it can miss deadlocks between events that are outside the given window. As with previous works [Cai et al. 2021; Kalhauge and Palsberg 2018], we set a window size of 10K for Dirk.

Our dataset consists of several benchmarks from standard benchmark suites — IBM Contest suite [Farchi et al. 2003], Java Grande suite [Smith et al. 2001], DaCapo [Blackburn et al. 2006], and SIR [Do et al. 2005] — and recent literature [Cai et al. 2021; Joshi et al. 2009; Jula et al. 2008; Kalhauge and Palsberg 2018]. Each benchmark was instrumented with RV-Predict [Rosu 2018] or Wiretap [Kalhauge and Palsberg 2018] and executed in order to log a single execution trace.

Evaluation. Table 1 presents our results. A bug identifies a unique tuple of source code locations corresponding to events participating in the deadlock. Trace lengths vary vastly from 39 to about 241M, while the number of threads ranges from 3 to about 800, which are representative features of real-world settings. Hsqldb contains critical sections that are not well nested, and SeqCheck was not able to handle this benchmark; our algorithm does not have such a restriction.

Abstract vs Concrete Patterns. Columns 7-9 present statistics on the abstract lock graph ALG_σ of each trace σ. Many traces have a large number of concrete deadlock patterns but much fewer abstract deadlock patterns; a single abstract deadlock pattern can comprise up to an order of 10^4 more concrete patterns (Column 8 v/s Column 9). Unlike all prior sound techniques, our algorithms analyze abstract deadlock patterns, instead of concrete ones. We thus expect our algorithms to be much more scalable in practice.

Deadlock-detection capability. In total, both SeqCheck and SPDOffline reported 40 deadlocks. SeqCheck misses a deadlock of size 5 in DiningPhil, which is detected by SPDOffline, and SPDOffline misses a deadlock in jigsaw which is detected by SeqCheck. As SPDOffline is complete for sync-preserving deadlocks, we conclude that there are no more such deadlocks in our dataset. Overall, SPDOffline and SeqCheck miss only three deadlocks reported by Dirk. On closer inspection, we found that these deadlocks are not witnessed by correct reorderings, and require reasoning about event values. On the other hand, Dirk struggles to analyze even moderately-sized benchmarks and times out in 3 of them. This results in Dirk failing to report 5 deadlocks after 9 hours, all of which are reported by SPDOffline in under a minute. Similar conclusions were recently

Table 1. Trace characteristics, abstract lock graph statistics and performance comparison. Columns 2-6 show the number of events, threads, variables, locks and total number of lock acquire and request events. Columns 7-9 show the number of cycles, abstract and concrete deadlock patterns in the abstract lock graph. Columns 10 - 15 show the number of deadlocks reported and the times (in seconds) taken. by `Dirk`, `SeqCheck`, and `SPDOffline`. Time out (T.O) was set to 3h. F stands for technical failure.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Benchmark	$\mathcal{N}$	$\mathcal{T}$	$\mathcal{V}$	$\mathcal{L}$	$\mathcal{A}/\mathcal{R}$	A. Lock Graph			`Dirk`		`SeqCheck`		`SPDOffline`	
						|Cyc|	A. P.	C. P.	**Dlk**	**Time**	**Dlk**	**Time**	**Dlk**	**Time**
Deadlock	39	3	4	3	8	1	1	1	1	0.02	0	0.09	0	0.16
NotADeadlock	60	3	4	5	16	1	1	1	0	0.02	0	0.09	0	0.16
Picklock	66	3	6	6	20	2	2	2	1	0.02	1	0.10	1	0.18
Bensalem	68	4	5	5	22	2	2	2	1	0.02	1	0.12	1	0.16
Transfer	72	3	11	4	12	1	1	1	1	0.02	0	0.09	0	0.15
Test-Dimmunix	73	3	9	7	26	2	2	2	2	0.02	2	0.10	2	0.17
StringBuffer	74	3	14	4	16	1	3	6	2	0.02	2	0.12	2	0.19
Test-Calfuzzer	168	5	16	6	48	2	1	1	1	0.02	1	0.12	1	0.17
DiningPhil	277	6	21	6	100	1	1	3K	1	1.60	0	0.09	1	0.17
HashTable	318	3	5	3	174	1	2	43	2	0.19	2	0.12	2	0.19
Account	706	6	47	7	134	3	1	12	0	0.19	0	0.09	0	0.18
Log4j2	1K	4	334	11	43	1	1	1	1	0.65	1	0.11	1	0.20
Dbcp1	2K	3	768	5	56	2	2	3	-	F	2	0.11	2	0.19
Dbcp2	2K	3	592	10	76	1	2	4	-	F	0	0.10	0	0.18
Derby2	3K	3	1K	4	16	1	1	1	1	0.23	1	0.10	1	0.17
RayTracer	31K	5	5K	15	976	0	0	0	-	F	0	0.15	0	0.19
jigsaw	143K	21	8K	2K	67K	172	12	70	-	F	2	0.36	1	1.55
elevator	246K	5	727	52	48K	0	0	0	0	1.65	0	0.33	0	0.27
hedc	410K	7	109K	8	32	0	0	0	0	2.09	0	0.50	0	0.24
JDBCMySQL-1	442K	3	73K	11	13K	2	4	6	2	28.45	2	0.24	2	0.48
JDBCMySQL-2	442K	3	73K	11	13K	4	4	9	1	3.37	1	0.22	1	0.33
JDBCMySQL-3	443K	3	73K	13	13K	5	8	16	1	31.23	1	0.25	1	0.45
JDBCMySQL-4	443K	3	73K	14	13K	5	10	18	2	5.51	2	0.28	2	0.49
cache4j	775K	2	46K	20	35K	0	0	0	0	5.86	0	0.46	0	0.39
ArrayList	3M	801	121K	802	176K	9	3	672	3	8.7K	3	21.98	3	1.68
IdentityHashMap	3M	801	496K	802	162K	1	3	4	1	443.93	1	8.51	1	1.45
Stack	3M	801	118K	2K	405K	9	3	481	1	T.O	3	25.34	3	2.94
Sor	3M	301	2K	3	719K	0	0	0	0	15.89	0	44.12	0	0.61
LinkedList	3M	801	290K	802	176K	9	3	10K	3	4.7K	3	48.02	3	2.06
HashMap	3M	801	555K	802	169K	1	3	10K	3	4.4K	2	504.36	2	1.65
WeakHashMap	3M	801	540K	802	169K	1	3	10K	-	T.O	2	499.68	2	1.70
Swing	4M	8	31K	739	2M	0	0	0	-	F	0	0.72	0	0.88
Vector	4M	3	15	4	800K	1	1	1B	-	T.O	1	1.52	1	1.90
LinkedHashMap	4M	801	617K	802	169K	1	3	10K	2	40.74	2	492.87	2	1.69
montecarlo	8M	3	850K	3	26	0	0	0	0	2.6K	0	1.81	0	0.79
TreeMap	9M	801	493K	802	169K	1	3	10K	2	105.45	2	480.11	2	1.92
hsqldb	20M	46	945K	403	419K	0	0	0	-	F	-	-	0	2.38
sunflow	21M	16	2M	12	1K	0	0	0	-	F	0	8.35	0	1.62
jspider	22M	11	5M	15	10K	0	0	0	-	F	0	8.49	0	1.95
tradesoap	42M	236	3M	6K	245K	2	1	4	-	F	0	108.16	0	7.06
tradebeans	42M	236	3M	6K	245K	2	1	4	-	F	0	116.23	0	7.26
eclipse	64M	15	10M	5K	377K	9	5	280	-	F	0	26.67	0	9.90
TestPerf	80M	50	599	9	197K	0	0	0	0	795.04	0	47.56	0	4.30
Groovy2	120M	13	13M	10K	69K	0	0	0	0	1.7K	0	38.06	0	8.92
Tsp	200M	6	24K	3	882	0	0	0	0	7.6K	0	72.62	0	12.70
lusearch	203M	7	3M	98	273K	0	0	0	0	1.3K	0	75.88	0	14.44
biojava	221M	6	121K	79	16K	0	0	0	-	F	0	63.79	0	12.65
graphchi	241M	20	25M	61	1K	0	0	0	-	F	0	102.05	0	25.25
Totals	**1B**	**7K**	**70M**	**37K**	**8M**	**256**	**93**	**1B**	**35**	**>18h**	**40**	**2801**	**40**	**135**

made in [Cai et al. 2021]. Overall, our results strongly indicate that the notion of sync-preservation characterizes most deadlocks that other tools are able to predict.

Unsoundness of `Dirk`. In our evaluation, we discovered that the soundness guarantee underlying `Dirk` [Kalhauge and Palsberg 2018] is broken, resulting in it reporting false positives. First, its constraint formulation [Kalhauge and Palsberg 2018] does not rule out deadlock patterns when acquire events in the pattern hold common locks, in which case mutual exclusion disallows such a pattern to be a real predictable deadlock. Second, `Dirk` also models conditional statements, allowing it to reason about witnesses beyond correct reorderings. While this relaxation allows `Dirk` to predict additional deadlocks in `Transfer`, `Deadlock` and `HashMap`, its formalization is not precise and its implementation is erroneous. We elaborate these aspects further in [Tunç et al. 2023a].

Running time. Our experimental results indicate that `Dirk`, backed by SMT solving, is the least efficient technique in terms of running time — it takes considerably longer or times out on large benchmark instances. `SPDOffline` analyzed the entire set of traces ~21× faster than `SeqCheck`. On the most demanding benchmarks, such as HashMap and TreeMap, `SPDOffline` is more than 200× faster than `SeqCheck`. Although `SeqCheck` employs a polynomial-time algorithm for deadlock prediction, and thus significantly faster than the SMT-based `Dirk`, the large polynomial complexity in its running time hinders scalability on execution traces coming from benchmarks that are more representative of realistic workloads. In contrast, the linear time guarantees of `SPDOffline` are realized in practice, allowing it to scale on even the most challenging inputs. More importantly, the improved performance comes while preserving essentially the same precision.

False negatives. Our benchmark set contains 93 abstract deadlock patterns, 40 of which are confirmed sync-preserving deadlocks. We inspected the remaining 53 abstract patterns to see if any of them are predictable deadlocks missed by our sync-preserving criterion, independently of the compared tools. 48 of these 53 patterns are in fact not predictable deadlocks — for every such pattern D, the set S_D of events in the downward-closure of $\mathsf{pred}(D)$ with respect to $\leq_{\mathsf{TO}}$ and rf, already contains an event from D, disallowing any correct reordering (sync-preserving or not) in which D can be enabled. Of the remaining, 4 deadlock patterns obey the following scheme: there are two acquire events $\mathsf{acq}_1, \mathsf{acq}_2$ participating in the deadlock pattern, each acq_i is preceded by a critical section on a lock that appears in $\mathsf{HeldLks}(\mathsf{acq}_{3-i})$, again disallowing a correct reordering that witnesses the pattern. Thus, *only one* predictable deadlock is not sync-preserving in our whole dataset. This analysis supports that the notion of sync-preservation is not overly conservative in practice.

The above analysis concerns false negatives wrt. predictable deadlocks. Some deadlocks are beyond the common notion of predictability we have adopted here, as they can only be exposed by reasoning about event values and control-flow dependencies, a problem that is NP-hard even for 3 threads [Gibbons and Korach 1997]. We noticed 3 such deadlocks in our dataset, found by `Dirk`, though, as mentioned above, `Dirk`'s reasoning for capturing such deadlocks is unsound in practice.

6.2 Online Experiments

Experimental setup. The objective of our second set of experiments is to evaluate the performance of our proposed algorithms in an *online* setting. For this, we implemented our `SPDOnline` algorithm inside the framework of `DeadlockFuzzer` [Joshi et al. 2009] following closely the pseudocode in Algorithm 4. This framework instruments a concurrent program so that it can perform analysis on-the-fly while executing it. If a deadlock occurs during execution, it is reported and the execution halts. However, if a deadlock is predicted in an alternate interleaving, then this deadlock is reported and the execution continues to search further deadlocks. We used the same dataset as in Section 6.1, after discarding some benchmarks that could not be instrumented by `DeadlockFuzzer`.

To the best of our knowledge, all prior deadlock prediction techniques work offline. For this reason, we only compared our online tool with the randomized scheduling technique of [Joshi et al. 2009] already implemented inside the same `DeadlockFuzzer` framework. At a high level, this random scheduling technique works as follows. Initially, it (i) executes the input program with a random scheduler, (ii) constructs a *lock dependency relation*, and (iii) runs a cycle detection algorithm to discover deadlock patterns. For each deadlock pattern thus found, it spawns new executions that attempt to realize it as an actual deadlock. To increase the likelihood of hitting the deadlock, `DeadlockFuzzer` biases the random scheduler by pausing threads at specific locations.

The second, confirmation phase of [Joshi et al. 2009] acts as a best-effort proxy for sound deadlock prediction. On the other hand, `SPDOnline` is already sound and predictive, and thus does not require additional confirmation runs, making it more efficient. Towards effective prediction, we also implemented a simple bias to the scheduler. If a thread t attempts to write on a shared variable x while holding a lock, then our procedure randomly decides to pause this operation for a short duration. This effectively explores race conditions in different orders. Overall, implementing `SPDOnline` inside `DeadlockFuzzer` provided the added advantage of supplementing a powerful prediction technique with a biased randomized scheduler. To our knowledge, our work is the first to effectively combine these two orthogonal techniques. We also remark that such a bias is of no benefit to `DeadlockFuzzer` itself since it does not employ any predictive reasoning.

For this experiment, we run `DeadlockFuzzer` on each benchmark, and for each deadlock pattern found in the initial execution, we let it spawn 3 new executions trying to realize the deadlock, as per standard (https://github.com/ksen007/calfuzzer). We repeated this process 50 times and recorded the total time taken. Then, we allocated the same time for `SPDOnline` to repeatedly execute the same program and perform deadlock prediction. We measured all deadlocks found by each technique.

Evaluation. Table 2 presents our experimental results. Columns 2-3 of the table display the total number of bug hits, which is the total number of times a bug was predicted by `SPDOnline` in the entire duration, or was confirmed in any trial of `DeadlockFuzzer`. Columns 4-6 display the unique bugs (i.e., unique tuples of source code locations leading to a deadlock) found by the techniques. The employed techniques are able to find a maximum of 3 unique bugs for each benchmark in our benchmark set. Respectively, columns 7-12 display the detailed information on the number of times a particular bug was found by each technique. Runtime overheads are displayed in the columns 13-16, with `−I` denoting the instrumentation phase only.

Deadlock-detection capability. `DeadlockFuzzer` had 2076 bug reports in total, and it found 42 unique bugs. In contrast, `SPDOnline` flagged 7633 bug reports, corresponding to 49 unique bugs. In more detail, `DeadlockFuzzer` missed 9 bugs reported by `SPDOnline` whereas `SPDOnline` missed 2 bugs reported by `DeadlockFuzzer`. Also, `SPDOnline` significantly outperformed `DeadlockFuzzer` in total number of bugs hits. Our experiments again support that the notion of sync-preservation captures most deadlocks that occur in practice, to the extent that other state-of-the-art techniques can capture. A further observation is that in the offline experiments, `SPDOffline` was not able to find deadlocks in `Transfer` and `Deadlock`. However, the random scheduling procedure allowed `SPDOnline` to navigate to executions from which deadlocks can be predicted. This demonstrates the potential of combining predictive dynamic techniques with controlled concurrency testing.

Runtime overhead. We have also measured the runtime overhead of both `SPDOnline` and `DeadlockFuzzer`, both as incurred by instrumentation, as well as by the deadlock analysis. The latter is the time taken by Algorithm 4 for the case of `SPDOnline`, and the overhead introduced due to the new executions in the second confirmation phase for the case of `DeadlockFuzzer`. Our results show that the instrumentation overhead of `SPDOnline` is, in fact, comparable to that of

Table 2. Performance comparison of `SPDOnline` (SPD) and `DeadlockFuzzer` (DF). Columns 2-3 show the total number of bug reports. Columns 4-6 show the total number of unique bugs found by each tool, and their union. Columns 7-12 show the hit rate on each bug. Columns 13-16 show the runtime overhead of the tools.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Benchmark	Bug Hits		Unique Bugs			Bug 1		Bug 2		Bug 3		Runtime Overhead			
	SPD	DF	SPD	DF	All	SPD	DF	SPD	DF	SPD	DF	SPD-I	SPD	DF-I	DF
Deadlock	50	50	1	1	1	50	50	-	-	-	-	2×	3×	2×	4×
Picklock	227	97	2	1	2	226	97	1	0	-	-	2×	2×	2×	5×
Bensalem	355	32	2	1	2	8	0	347	32	-	-	2×	2×	2×	6×
Transfer	54	50	1	1	1	54	50	-	-	-	-	2×	2×	1×	4×
Test-Dimmunix	702	0	2	0	2	351	0	351	0	-	-	2×	2×	2×	4×
StringBuffer	153	131	2	2	2	128	118	25	13	-	-	-	-	-	-
Test-Calfuzzer	177	44	1	1	1	177	44	-	-	-	-	2×	2×	2×	4×
DiningPhil	162	100	1	1	1	162	100	-	-	-	-	-	-	-	-
HashTable	169	120	2	2	2	82	21	87	99	-	-	-	-	-	-
Account	19	188	1	1	1	19	188	-	-	-	-	2×	8×	2×	16×
Log4j2	290	100	2	1	2	145	100	145	0	-	-	-	-	-	-
Dbcp1	265	138	2	2	2	264	61	1	77	-	-	-	-	-	-
Dbcp2	129	126	2	2	2	86	99	43	27	-	-	-	-	-	-
RayTracer	0	0	0	0	0	-	-	-	-	-	-	122×	124×	109×	111×
Tsp	0	0	0	0	0	-	-	-	-	-	-	47×	60×	37×	40×
jigsaw	1189	1	1	1	2	1189	0	0	1	-	-	-	-	-	-
elevator	0	0	0	0	0	-	-	-	-	-	-	2×	2×	2×	2×
JDBCMySQL-1	349	117	2	3	3	1	21	0	4	348	92	3×	4×	2×	13×
JDBCMySQL-2	559	73	1	1	1	559	73	-	-	-	-	2×	4×	2×	18×
JDBCMySQL-3	560	224	1	1	1	560	224	-	-	-	-	2×	5×	2×	24×
JDBCMySQL-4	1717	101	3	1	3	95	0	834	0	788	101	3×	5×	2×	31×
hedc	0	0	0	0	0	-	-	-	-	-	-	2×	2×	1×	2×
cache4j	0	0	0	0	0	-	-	-	-	-	-	2×	2×	2×	2×
lusearch	0	0	0	0	0	-	-	-	-	-	-	16×	17×	13×	16×
ArrayList	47	45	3	3	3	20	22	3	5	24	18	50×	69×	18×	79×
Stack	44	27	3	3	3	18	13	8	4	18	10	69×	91×	64×	86×
IdentityHashMap	68	62	2	2	2	13	47	55	15	-	-	4×	8×	3×	10×
LinkedList	48	26	3	2	3	21	17	7	0	20	9	16×	28×	14×	32×
Swing	0	0	0	0	0	-	-	-	-	-	-	5×	6×	4×	6×
Sor	0	0	0	0	0	-	-	-	-	-	-	2×	7×	2×	2×
HashMap	46	44	2	2	2	18	11	28	33	-	-	7×	11×	4×	8×
Vector	126	50	1	1	1	126	50	-	-	-	-	2×	2×	2×	3×
LinkedHashMap	57	43	2	2	2	22	10	35	33	-	-	10×	10×	4×	8×
WeakHashMap	29	40	2	2	2	6	11	23	29	-	-	7×	12×	4×	8×
montecarlo	0	0	0	0	0	-	-	-	-	-	-	16×	100×	13×	126×
TreeMap	42	47	2	2	2	16	15	26	32	-	-	9×	12×	5×	9×
eclipse	0	0	0	0	0	-	-	-	-	-	-	2×	2×	2×	2×
TestPerf	0	0	0	0	0	-	-	-	-	-	-	2×	2×	2×	2×
Total	**7633**	**2076**	**49**	**42**	**51**	-	-	-	-	-	-	-	-	-	-

`DeadlockFuzzer`, though somewhat larger. This is expected, as `SPDOnline` needs to also instrument memory access events, while `DeadlockFuzzer` only instruments lock events, but at the same time surprising because the number of memory access events is typically much larger than the number of lock events. On the other hand, the analysis overhead is often larger for `DeadlockFuzzer`, even though it reports fewer bugs. It was not possible to measure the runtime overhead in certain benchmarks as either they were always deadlocking or the computation was running indefinitely.

7 RELATED WORK

Dynamic techniques for detecting deadlock patterns, like the GoodLock algorithm [Havelund 2000] have been improved in performance [Cai et al. 2020; Zhou et al. 2017] and precision [Bensalem and Havelund 2005], sometimes using re-executions to verify potential deadlocks [Bensalem et al. 2006; Joshi et al. 2009; Samak and Ramanathan 2014a,b; Sorrentino 2015]. Predictive analyses directly infer concurrency bugs in alternate executions [Şerbănută et al. 2013] and are typically *sound* (no false positives). This approach has been successfully applied for detecting bugs such as data races [Huang et al. 2014; Kini et al. 2017; Mathur et al. 2021; Pavlogiannis 2019; Roemer et al. 2020; Said et al. 2011; Smaragdakis et al. 2012], use-after-free vulnerabilities [Huang 2018], and more recently for deadlocks [Cai et al. 2021; Eslamimehr and Palsberg 2014; Kalhauge and Palsberg 2018].

The notion of sync-preserving deadlocks has been inspired by a similar notion pertaining to data races [Mathur et al. 2021]. However, sync-preserving deadlock prediction rests on some further novelties. First, unlike data races, deadlocks can involve more than 2 events. Generalizing sync-preserving ideals of sets of events of arbitrary size, as well as establishing the monotonicity properties (Proposition 4.4 and Corollary 4.5) for arbitrarily many events is non-trivial. Second, our notions of abstract deadlock patterns (Section 4.4) and abstract lock graphs (Section 4.5) are novel and carefully crafted to leverage these monotonicity properties in the deadlock setting. Indeed, the linear-time sync-preserving verification of each abstract deadlock pattern is the cornerstone of our approach, for the first linear-time, sound and precise deadlock predictor.

Although the basic principles of data-race and deadlock prediction are similar, there are notable differences. First, identifying potential deadlocks is theoretically intractable, whereas, potential races are identified easily. Second, popular partial-order based techniques [Flanagan and Freund 2009; Kini et al. 2017] for data races are likely to require non-trivial modifications for deadlocks, as they typically order critical sections, which may hide a deadlock. Nevertheless, bridging prediction techniques between data races and deadlocks is an interesting and relatively open direction.

Predicting deadlocks is an intractable problem, the complexity of which we have characterized in this work. Prior works have also focused on the complexity of predicting data races [Kulkarni et al. 2021; Mathur et al. 2020] and atomicity violations [Farzan and Madhusudan 2009].

8 CONCLUSION

We have studied the complexity of deadlock prediction and introduced the new tractable notion of sync-preserving deadlocks, along with sound, complete and efficient algorithms for detecting them. Our experiments show that the majority of deadlocks occurring in practice are indeed sync-preserving, and our algorithm `SPDOffline` is the first deadlock predictor that achieves sound and high coverage, while also spending only linear time to process its input. Our online algorithm `SPDOnline` enhances the bug detection capability of controlled concurrency testing techniques like [Joshi et al. 2009], at close runtime overheads. Interesting future work includes incorporating static checks [Rhodes et al. 2017] and exploring ways for deeper integration of controlled concurrency testing with predictive techniques. Another step is to extend the coverage of sync-preserving deadlocks while maintaining efficiency, for example, by reasoning about program control flow.

ACKNOWLEDGMENTS

Andreas Pavlogiannis was partially supported by a research grant (VIL42117) from VILLUM FONDEN. Umang Mathur was partially supported by the Simons Institute for the Theory of Computing, and by a Singapore Ministry of Education (MoE) Academic Research Fund (AcRF) Tier 1 grant. Mahesh Viswanathan was partially supported by NSF SHF 1901069 and NSF CCF 2007428.

DATA AND SOFTWARE AVAILABILITY STATEMENT

The artifact developed for this work is available [Tunç et al. 2023b], which contains all source codes and experimental data necessary to reproduce our evaluation in Section 6, excluding the results of SeqCheck.

REFERENCES

Rahul Agarwal, Liqiang Wang, and Scott D. Stoller. 2006. Detecting Potential Deadlocks with Static Analysis and Runtime Monitoring. In *Proceedings of the Parallel and Distributed Systems: Testing and Debugging (PADTAD) Track of the 2005 Haifa Verification Conference (Lecture Notes in Computer Science, Vol. 3875)*. Springer-Verlag, 191–207. https://doi.org/10.1007/11678779_14

Saddek Bensalem, Jean-Claude Fernandez, Klaus Havelund, and Laurent Mounier. 2006. Confirmation of Deadlock Potentials Detected by Runtime Analysis. In *Proceedings of the 2006 Workshop on Parallel and Distributed Systems: Testing and Debugging* (Portland, Maine, USA) *(PADTAD '06)*. Association for Computing Machinery, New York, NY, USA, 41–50. https://doi.org/10.1145/1147403.1147412

Saddek Bensalem and Klaus Havelund. 2005. Dynamic Deadlock Analysis of Multi-Threaded Programs. In *Proceedings of the First Haifa International Conference on Hardware and Software Verification and Testing* (Haifa, Israel) *(HVC'05)*. Springer-Verlag, Berlin, Heidelberg, 208–223. https://doi.org/10.1007/11678779_15

Swarnendu Biswas, Jipeng Huang, Aritra Sengupta, and Michael D. Bond. 2014. DoubleChecker: Efficient Sound and Precise Atomicity Checking. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. ACM, New York, NY, USA, 28–39. https://doi.org/10.1145/2594291.2594323

Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA) *(OOPSLA '06)*. Association for Computing Machinery, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

Yan Cai and W.K. Chan. 2014. Magiclock: Scalable Detection of Potential Deadlocks in Large-Scale Multithreaded Programs . *IEEE Transactions on Software Engineering* 40, 3 (2014), 266–281. https://doi.org/10.1109/TSE.2014.2301725

Yan Cai and W. K. Chan. 2012. MagicFuzzer: Scalable Deadlock Detection for Large-Scale Applications. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press, 606–616. https://doi.org/10.1109/ICSE.2012.6227156

Yan Cai, Ruijie Meng, and Jens Palsberg. 2020. Low-Overhead Deadlock Prediction. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1298–1309. https://doi.org/10.1145/3377811.3380367

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* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 255–267. https://doi.org/10.1145/3468264.3468549

Jianer Chen, Xiuzhen Huang, Iyad A. Kanj, and Ge Xia. 2006. Strong computational lower bounds via parameterized complexity. *J. Comput. System Sci.* 72, 8 (2006), 1346–1367. https://doi.org/10.1016/j.jcss.2006.04.007

Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. 2005. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Software Engineering* 10, 4 (2005), 405–435. https://doi.org/10.1007/s10664-005-3861-2

Mahdi Eslamimehr and Jens Palsberg. 2014. Sherlock: Scalable Deadlock Detection for Concurrent Programs. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 353–365. https://doi.org/10.1145/2635868.2635918

Eitan Farchi, Yarden Nir, and Shmuel Ur. 2003. Concurrent Bug Patterns and How to Test Them. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing (IPDPS '03)*. IEEE Computer Society, Washington, DC, USA, 286.2. https://doi.org/10.1109/IPDPS.2003.1213511

Azadeh Farzan and Parthasarathy Madhusudan. 2009. The Complexity of Predicting Atomicity Violations. In *Tools and Algorithms for the Construction and Analysis of Systems*, Stefan Kowalewski and Anna Philippou (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 155–169. https://doi.org/10.1007/978-3-642-00768-2_14

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

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

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

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

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

Klaus Havelund. 2000. Using Runtime Analysis to Guide Model Checking of Java Programs. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*. Springer-Verlag, Berlin, Heidelberg, 245–264. https://doi.org/10.1007/10722468_15

Jeff Huang. 2018. UFO: Predictive Concurrency Use-after-Free Detection. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 609–619. https://doi.org/10.1145/3180155.3180225

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

Donald B. Johnson. 1975. Finding All the Elementary Circuits of a Directed Graph. *SIAM J. Comput.* 4, 1 (1975), 77–84. https://doi.org/10.1137/0204007

Pallavi Joshi, Mayur Naik, Koushik Sen, and David Gay. 2010. An Effective Dynamic Analysis for Detecting Generalized Deadlocks. 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, 327–336. https://doi.org/10.1145/1882291.1882339

Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. 2009. A Randomized Dynamic Program Analysis Technique for Detecting Real Deadlocks . 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, 110–120. https://doi.org/10.1145/1542476.1542489

Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, and George Candea. 2008. Deadlock Immunity: Enabling Systems to Defend against Deadlocks. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 295–308.

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

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

Rucha Kulkarni, Umang Mathur, and Andreas Pavlogiannis. 2021. Dynamic Data-Race Detection Through the Fine-Grained Lens. In *32nd International Conference on Concurrency Theory (CONCUR 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 203)*, Serge Haddad and Daniele Varacca (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 16:1–16:23. https://doi.org/10.4230/LIPIcs.CONCUR.2021.16

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

Bozhen Liu, Peiming Liu, Yanze Li, Chia-Che Tsai, Dilma Da Silva, and Jeff Huang. 2021. When Threads Meet Events: Efficient and Precise Static Race Detection with Origins . In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 725–739. https://doi.org/10.1145/3453483.3454073

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

Umang Mathur. 2019. RAPID. https://github.com/umangm/rapid. Accessed: 2022-10-14.

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

Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. 2021. Optimal Prediction of Synchronization-Preserving Races. *Proc. ACM Program. Lang.* 5, POPL, Article 36 (2021), 29 pages. https://doi.org/10.1145/3434317

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

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

Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, Berkeley, CA, USA, 267–280.

Mayur Naik, Chang-Seo Park, Koushik Sen, and David Gay. 2009. Effective Static Deadlock Detection. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, USA, 386–396. https://doi.org/10.1109/ICSE.2009.5070538

Nicholas Ng and Nobuko Yoshida. 2016. Static Deadlock Detection for Concurrent Go by Global Session Graph Synthesis . In *Proceedings of the 25th International Conference on Compiler Construction* (Barcelona, Spain) *(CC 2016)*. Association for Computing Machinery, New York, NY, USA, 174–184. https://doi.org/10.1145/2892208.2892232

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

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

Dustin Rhodes, Cormac Flanagan, and Stephen N. Freund. 2017. BigFoot: Static Check Placement for Dynamic Race Detection. 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, 141–156. https://doi.org/10.1145/3062341.3062350

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

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

Mahmoud Said, Chao Wang, Zijiang Yang, and Karem Sakallah. 2011. Generating Data Race Witnesses by an SMT-Based Analysis. In *NASA Formal Methods*, Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 313–327. https://doi.org/10.1007/978-3-642-20398-5_23

Malavika Samak and Murali Krishna Ramanathan. 2014a. Multithreaded Test Synthesis for Deadlock Detection. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. Association for Computing Machinery, New York, NY, USA, 473–489. https://doi.org/10.1145/2660193.2660238

Malavika Samak and Murali Krishna Ramanathan. 2014b. Trace Driven Dynamic Deadlock Detection and Reproduction. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Orlando, Florida, USA) *(PPoPP '14)*. Association for Computing Machinery, New York, NY, USA, 29–42. https://doi.org/10.1145/2555243.2555262

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

Koushik Sen, Grigore Roşu, and Gul Agha. 2005. Detecting Errors in Multithreaded Programs by Generalized Predictive Analysis of Executions. In *Formal Methods for Open Object-Based Distributed Systems*, Martin Steffen and Gianluigi Zavattaro (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 211–226. https://doi.org/10.1007/11494881_14

Traian Florin Şerbănută, Feng Chen, and Grigore Roşu. 2013. Maximal Causal Models for Sequentially Consistent Systems. In *Runtime Verification*, Shaz Qadeer and Serdar Tasiran (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 136–150. https://doi.org/10.1007/978-3-642-35632-2_16

Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: Data Race Detection in Practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications* (New York, New York, USA) *(WBIA '09)*. Association for Computing Machinery, New York, NY, USA, 62–71. https://doi.org/10.1145/1791194.1791203

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

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

Francesco Sorrentino. 2015. PickLock: A Deadlock Prediction Approach under Nested Locking. In *Proceedings of the 22nd International Symposium on Model Checking Software - Volume 9232* (Stellenbosch, South Africa) *(SPIN 2015)*. Springer-Verlag, Berlin, Heidelberg, 179–199. https://doi.org/10.1007/978-3-319-23404-5_13

Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiying Zhang. 2019. Understanding Real-World Concurrency Bugs in Go. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 865–878. https://doi.org/10.1145/3297858.3304069

Hünkar Can Tunç, Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. 2023a. Sound Dynamic Deadlock Prediction in Linear Time. arXiv:2304.03692

Hünkar Can Tunç, Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. 2023b. *Sound Dynamic Deadlock Prediction in Linear Time*. https://doi.org/10.5281/zenodo.7809600 Artifact.

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).

Virginia Vassilevska Williams. 2018. On some fine-grained questions in algorithms and complexity. In *Proceedings of the International Congress of Mathematicians: Rio de Janeiro 2018*. World Scientific, 3447–3487. https://doi.org/10.1142/9789813272880_0188

Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. 2011. How Do Fixes Become Bugs?. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 26–36. https://doi.org/10.1145/2025113.2025121

Jinpeng Zhou, Sam Silvestro, Hongyu Liu, Yan Cai, and Tongping Liu. 2017. UNDEAD: Detecting and Preventing Deadlocks in Production Software. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE '17)*. IEEE Press, 729–740. https://doi.org/10.1109/ASE.2017.8115684

Received 2022-11-10; accepted 2023-03-31