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:

Authors’ addresses: Hünkar Can Tunç, Aarhus University, Denmark, tunç@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
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 SPD0ff1ine 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 SPD0n1ine 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 $\sigma_1$ in Figure 1a consisting of 10 events and two threads. We use $e_i$ to denote the $i$-th event of $\sigma_1$. 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 $\rho$ of (a slice of) $\sigma_1$ 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 $\rho$. 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 $\rho^{\text{SP}}$ of a trace $\sigma$ is said to be sync-preserving if it preserves the control flow taken by the original observed trace $\sigma$, and further it preserves the mutual order of any two critical sections (on the same lock) that appear in the reordering $\rho^{\text{SP}}$. Consider, for example, the sequence $\rho_1 = e_1 .. e_3 e_6 .. e_7$ where $e_1 .. e_7$ denote the contiguous sequence of events that starts from $e_1$ and ends at $e_7$. We call $\rho_1$ a correct reordering of $\sigma_1$, being a slice of $\sigma_1$ closed under the thread order and preserving the writer of each read in $\sigma_1$; the precise definition is presented in Section 2. In this case, however, $\rho_1$ does not witness the deadlock as the event $e_2$ is not enabled in $\rho_1$. In fact, due to the dependency between the events $e_3$ and $e_7$, there are no correct reorderings of $\sigma_1$ which make both $e_2$ and $e_8$ enabled. This makes the deadlock pattern $(e_2, e_8)$ non-predictable. Consider now $\sigma_2$ in Figure 1b, and the sequence $\rho_2 = e_3 .. e_7 e_8 .. e_{11} e_1 e_2$. Observe that $\rho_2$ is also a correct reordering. However, $\rho_2$ is not sync-preserving as the order of the two critical sections on lock $l_1$ in $\rho_2$ is different from their original order in $\sigma_2$. 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 $\rho_3$ as they did in $\sigma_2$. Further, $\rho_3$ also witnesses the deadlock as the events $e_4$ and $e_{18}$ are both enabled in $\rho_3$. This makes the deadlock pattern $(e_4, e_{18})$ 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 $\sigma$ of size $N$ and $T$ threads, we first show that detecting even one deadlock pattern of length $k$ is $W[1]$-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 $f(k) \cdot \text{poly}(N)$, for some function $f$. We next show that even with just $T = 2$ 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 $O(N^{2-\epsilon})$, 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 $W[1]$-hard in the number of threads $T$ (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 $\tilde{O}(N)$. 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 $N$, 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 $\sigma$ is a linear arrangement of events $\text{Events}_\sigma$. An event $e \in \text{Events}_\sigma$ is tuple $e = (i, t, o)$, 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 ($o = r(x)$ or $o = w(x)$) operation to some variable $x$, or an acquire or release ($o = \text{acq}(\ell)$ or $o = \text{rel}(\ell)$) operation on some lock $\ell$. For the sake of simplicity, we often omit $i$ when referring to an event. We use thread$(e)$ and op$(e)$ to respectively denote the thread identifier and the operation performed in the event $e$. We use $\text{Threads}_\sigma$, $\text{Vars}_\sigma$ and $\text{Locks}_\sigma$ to denote the set of thread, variable and lock identifiers in $\sigma$.

We restrict our attention to well-formed traces $\sigma$, that abide to shared-memory semantics. That is, if a lock $\ell$ is acquired at an event $e$ by thread $t$, then any later acquisition event $e'$ of the same lock $\ell$ must be preceded by an event $e''$ that releases lock $\ell$ 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 $e = \text{match}_\sigma(e'')$ and $e'' = \text{match}_\sigma(e)$. 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 $\sigma$ implicitly defines some relations. The trace-order $\preceq_{tr} \subseteq \text{Events}_\sigma \times \text{Events}_\sigma$ orders the events of $\sigma$ in a total order based on their order of occurrence in

---

*We use $\tilde{O}$ to ignore polynomial appearance of trace parameters typically much smaller than $N$ (e.g., number of threads).*
the sequence $\sigma$. The \textit{thread-order} $\leq_{\text{TO}}$ is the unique partial order over $\text{Events}_\sigma$ such that $e \leq_{\text{TO}} e'$ iff $\text{thread}(e) = \text{thread}(e')$ and $e \leq_{\nu} e'$. We say $e <_{\text{TO}} e'$ if $e \leq_{\text{TO}} e'$ but $e \neq e'$. The \textit{reads-from function} $\text{rf}_\sigma$ is a map from the read events to the write events in $\sigma$. Under sequential consistency, for a read event $e$ on variable $x$, we have that $e' = \text{rf}_\sigma(e)$ be the latest write event on the same variable $x$ such that $e' \leq_{\nu} e$. We say that a lock $\ell \in \text{Locks}_\sigma$ is held at an event $e \in \text{Events}_\sigma$ if there is an event $e'$ such that (i) $\text{op}(e') = \text{acq}(\ell)$, (ii) $e' <_{\text{TO}} e$, and (iii) either $\text{match}_\sigma(e')$ does not exist in $\sigma$, or $e \leq_{\text{TO}} \text{match}_\sigma(e')$. We use $\text{HeldLks}_\sigma(e)$ to denote the set of all the locks that are held by $\text{thread}(e)$ right before $e$. The lock nesting depth of $\sigma$ is $\max_{e \in \text{Events}_\sigma} |\text{HeldLks}_\sigma(e)| + 1$ where $\text{op}(e) = \text{acq}(\ell)$.

**Deadlock patterns.** A deadlock pattern $\top$ of size $k$ in a trace $\sigma$ 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 $l_0, \ldots, l_{k-1}$ such that $\text{thread}(e_i) = t_i$, $\text{op}(e_i) = \text{acq}(l_i)$, $l_i \in \text{HeldLks}_\sigma(e_i)$, and further, $\text{HeldLks}_\sigma(e_i) \cap \text{HeldLks}_\sigma(e_j) = \emptyset$ 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 $\sigma$ 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 $\sigma$ offer poor coverage and are bound to miss bugs that appear in select thread interleavings [Musuvathi et al. 2008]. Therefore, for better coverage, \textit{predictive} dynamic techniques are developed. Such techniques predict the occurrence of bugs in alternate executions that can be \textit{inferred} from $\sigma$, irrespective of the program that produced $\sigma$. The notion of such inferred executions is formalized by the notion of correct reorderings [Sen et al. 2005; Şerbanuţă et al. 2013; Smaragdakis et al. 2012].

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

**Predictable deadlocks.** We say that an event $e$ is $\sigma$-enabled in a correct reordering $\rho$ of $\sigma$ if $e \in \text{Events}_\sigma$ and $e \notin \text{Events}_\rho$ and for every $f \in \text{Events}_\sigma$ with $f <_{\text{TO}} e$, then $f \in \text{Events}_\rho$. A deadlock pattern $D = \langle e_0, e_1, \ldots, e_{k-1} \rangle$ of size $k$ in trace $\sigma$ is said to be a predictable deadlock if there is a correct reordering $\rho$ of $\sigma$ such that each of $e_0, \ldots, e_{k-1}$ are $\sigma$-enabled in $\rho$. This notion guarantees that the witness $\rho$ is a valid execution of any concurrent program that produced $\sigma$. 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 \textit{sound} if for every input trace $\sigma$, all deadlock reports on $\sigma$ are predictable deadlocks of $\sigma$ (i.e., no false positives), and \textit{complete} if all predictable deadlocks of $\sigma$ 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 $\sigma_2$ in Fig. 1b, with $e_i$ denoting the $i^{\text{th}}$ event in the figure. The set of events, threads, variables and locks of $\sigma_2$ are respectively $\text{Events}_{\sigma_2} = \{e_i\}_{i=1}^{20}$, $\text{Threads}_{\sigma_2} = \{t_1, t_2, t_3, t_4\}$, $\text{Vars}_{\sigma_2} = \{x, y, z\}$ and $\text{Locks}_{\sigma_2} = \{l_1, l_2, l_3\}$. The trace order yields $e_i \leq_{\text{TO}} e_j$ iff $i \leq j$, and some examples of thread-ordered events are $e_1 <_{\text{TO}} e_2 <_{\text{TO}} e_3 <_{\text{TO}} e_5$.

\footnote{Similar notions have been used in the literature, sometimes under the term \textit{deadlock potential} [Havelund 2000].}
The trace $\text{rf}_{\sigma_2}(e_{10}) = e_5$, $\text{rf}_{\sigma_2}(e_{14}) = e_9$ and $\text{rf}_{\sigma_2}(e_{17}) = e_{13}$. The lock nesting depth of $\sigma_2$ is 2. The sequence $D = \langle e_4, e_{18} \rangle$ forms a deadlock pattern because of the cyclic acquisition of locks $t_2$ and $t_3$ without simultaneously holding a common lock. The trace $\rho_4 = e_3, e_7, e_8, e_{11}, e_{12}, e_{15}, e_{16}, e_{17}$ is a correct reordering of $\sigma_2$; even though it differs from $\sigma_2$ in the relative order of the critical sections of lock $t_1$, and contains only a prefix of thread $t_3$, it is consistent with $\text{rf}_{\sigma_2}$ and $<_{TO}^{\sigma_2}$. However, $\rho_4$ does not witness $\langle e_4, e_{18} \rangle$ as a deadlock, as only $e_{18}$ is $\sigma_2$-enabled in $\rho_4$. On the other hand, the trace $\rho_5 = e_1, e_2, e_3, e_5, e_{12}, e_{15}, e_{16}, e_{17}$ is a correct reordering of $\sigma_2$ in which $e_4$ and $e_{18}$ are $\sigma_2$-enabled, witnessing $D$ as a predictable deadlock of $\sigma_2$.

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 [Tunc 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 $\sigma$ 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 $\sigma$ 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 $\sigma$ such that $G$ has an independent set of size $c$ iff $\sigma$ 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 $\sigma$ 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 $\{l_{(u,v)}\}_{(u,v) \in E} \cup \{l_0, t_1, \ldots, t_{c-1}\}$. 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 \( \ell_{(u_j, u_j)} \), where \( u \) is a neighbor of \( v_j \). Inside the nested block we have critical sections on locks \( \ell_{v_j \ell_{v_j}} \) and \( \ell_{(i+1) \ell_{v_j}} \). Formally, let \( \{v_j, v_{k_1}, \ldots, v_j, v_{k_d}\} \) be the neighboring edges of \( v_j \) (ordered according to \( \prec_E \)). Then, \( \sigma_i^{(i)} \) 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:

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

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

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 \( \text{INDEPENDENT-SET}(c) \) 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 \( f(k) \cdot N^{g(k)} \), 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 \( O(N^2) \) 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 \( \sigma \) of size \( N \) and \( L \) locks, for any \( \epsilon > 0 \), there is no algorithm that determines in \( O(N^{2-\epsilon} \cdot \text{poly}(L)) \) time whether \( \sigma \) 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 \( \sigma \) such that \( \sigma \) has a deadlock pattern of length 2 iff \( (A, B) \) is a positive OV instance. The trace \( \sigma \) 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, \( \sigma^A \) and \( \sigma^B \) 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-\( \sigma^Z \) 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:

- \( S_0 \rightarrow \langle t_z, \text{acq}(m) \rangle \cdot \langle t_z, \text{rel}(m') \rangle \cdot \langle t_z, \text{acq}(m') \rangle \cdot \langle t_z, \text{rel}(m) \rangle \), where \((m, m') = (m_0, m_1)\) if \( Z = A \), and \((m, m') = (m_1, m_0)\).
- for each \( 1 \leq j \leq d \), \( S_j \rightarrow S_{j-1} \) if \( Z_j[j] = 0 \). Otherwise (if \( Z_j[j] = 1 \)), \( S_j \rightarrow \langle t_z, \text{acq}(t_j) \rangle \cdot S_{j-1} \cdot \langle t_z, \text{rel}(t_j) \rangle \).

In words, all events of \( \sigma^A \) are performed by thread \( t_A \) and those in \( \sigma^B \) are performed by \( t_B \). Next, the \( i^{th} \) sub-\( \sigma^A \), denoted \( \sigma^A_i \), corresponds to the vector \( A_i \) as follows — \( \sigma^A_i \) is a nested block of critical sections, with the innermost critical section being on lock \( \ell' \), which is immediately enclosed in a critical section on lock \( \ell \). Further, in the sub-\( \sigma^A_i \), the lock \( t_j \) occurs iff \( A_i[j] = 1 \). The sub-traces \( \sigma^B_i \) 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 \( \sigma \) has a predictable deadlock of size 2 is \( W[1] \)-hard in the number of threads \( T \) appearing in \( \sigma \), 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 \( \rho \) of a trace \( \sigma \) is sync-preserving if for every lock \( \ell \in \text{Locks}_\rho \) and every two acquire events \( e_1 \neq e_2 \in \text{Events}_\rho \) with \( \text{op}(e_1) = \text{op}(e_2) = \text{acq}(\ell) \), the order of \( e_1 \) and \( e_2 \) is the same in \( \sigma \) and \( \rho \), i.e., \( e_1 \leq_{\ell} e_2 \) iff \( e_1 \leq_{\ell}^\rho e_2 \).
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 \( \sigma \) 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 \( \sigma \) if there is a sync-preserving correct reordering \( \rho \) of \( \sigma \) such that each of \( e_0, \ldots, e_{k-1} \) is \( \sigma \)-enabled in \( \rho \).

**Example 2.** Consider the trace \( \sigma_2 \) in Fig. 1b. The deadlock pattern \( D = \langle e_4, e_{16} \rangle \) is a sync-preserving deadlock, witnessed by the sync-preserving correct reordering \( \rho_3 = e_1 e_2 e_3 e_8 e_9 e_{12} e_{15} e_{16} e_{17} \). Now consider the trace \( \sigma_3 \) from Fig. 3 and the deadlock pattern \( D_3 = \langle e_{29}, e_{16} \rangle \). This is a predictable deadlock, witnessed by the correct reordering \( \rho_3 = e_1 e_2 e_3 \ldots e_{11} e_{12} e_{15} e_{28} \). Observe that \( \rho_3 \) is a sync-preserving reordering, which makes \( D_3 \) a sync-preserving deadlock. A key aspect in \( \rho_3 \) is that the events \( e_{22} \ldots e_{27} \) are dropped, as otherwise \( e_{16} \) cannot be \( \sigma_3 \)-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 \( e_8 \ldots e_{11} \) and \( e_1 \ldots e_6 \). This prevents the deadlocks from becoming realizable as the events \( e_2 \) or \( e_3 \) that appear in these deadlock patterns are no longer \( \sigma_3 \)-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 \( \sigma \) be a trace and \( S \subseteq \text{Events}_\sigma \). The sync-preserving closure of \( S \), denoted \( \text{SPClosure}_\sigma(S) \), is the smallest set \( S' \) such that (a) \( S \subseteq S' \), (b) for every
\(e, e' \in \text{Events}_\sigma\) such that \(e \ll_{TO} e'\) or \(e = \text{rf}_\sigma(e')\), if \(e' \in S'\), then \(e \in S'\), and (c) for every lock \(\ell\) and every two distinct events \(e, e' \in S'\) with \(\text{op}(e) = \text{op}(e') = \text{acq}(\ell)\), if \(e \leq_{\ell} e'\) then \(\text{match}_\sigma(e) \in S'\).

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 \(\sigma\) be a trace and let \(S \subseteq \text{Events}_\sigma\). For any sync-preserving correct reordering \(\rho\) of \(\sigma\), if \(S \subseteq \text{Events}_\rho\), then \(\text{SPClosure}_\sigma(S) \subseteq \text{Events}_\rho\). Further, there is a sync-preserving correct reordering \(\rho\) of \(\sigma\) such that \(\text{Events}_\rho = \text{SPClosure}_\sigma(S)\).

For an intuition, consider again Figure 3 and the sync-preserving correct reordering \(\rho_5 = e_1, e_7, e_8, e_9, e_{12}, e_{15}, e_{28}\) computed in Example 2. According to Lemma 4.1, \(\text{SPClosure}_{\sigma_1}(S) \subseteq \text{Events}_{\rho_5}\) holds for all \(S\) such that \(S \subseteq \text{Events}_{\rho_5}\). For example, if we take \(S = \{e_1, e_{15}\}\) then observe that \(S \subseteq \text{Events}_{\rho_5}\) and \(\text{SPClosure}_{\sigma_1}(S) = \{e_1, \ldots, e_6, e_9, e_{12}, \ldots, e_{17}\}\) holds.

Based on Lemma 4.1, we present a sound and complete characterization of sync-preserving deadlocks (Lemma 4.2). For a set \(S \subseteq \text{Events}_\sigma\), we let \(\text{pred}_\sigma(S)\) denote the set of immediate thread predecessors of events in \(S\). That is, \(\text{pred}_\sigma(S) = \{e \in \text{Events}_\sigma \mid \exists f \in S, e \prec_{\text{TO}} f \text{ and } \forall e' \prec_{\text{TO}} f, e' \not<_e f\}\).

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

**Example 3.** Consider the trace \(\sigma_2\) in Fig. 1b, and the deadlock pattern \(D = \langle e_4, e_{18} \rangle\). We have \(\text{SPClosure}_{\sigma_2}(\text{pred}_{\sigma_2}(\{e_4, e_{18}\})) = \{e_1, e_2, e_3, e_5, e_9, e_{12}, \ldots, e_{17}\}\). Since we have that \(e_4, e_{18} \notin \text{SPClosure}_{\sigma_2}(\text{pred}_{\sigma_2}(\{e_4, e_{18}\}))\), \(D\) is a sync-preserving deadlock. Now consider the trace \(\sigma_3\) in Fig. 3, and the deadlock patterns \(D_1 = \langle e_2, e_{16} \rangle, D_3 = \langle e_{29}, e_{16} \rangle, \text{ and } D_6 = \langle e_{29}, e_{19} \rangle\). We have \(\text{SPClosure}_{\sigma_3}(\text{pred}_{\sigma_3}(\{e_2, e_{16}\})) = \{e_1, \ldots, e_8, \ldots, e_{13}\}\), \(\text{SPClosure}_{\sigma_3}(\text{pred}_{\sigma_3}(\{e_{29}, e_{19}\})) = \{e_1, \ldots, e_9, e_{12}, \ldots, e_{28}\}\). Since \(e_2 \in \text{SPClosure}_{\sigma_3}(\text{pred}_{\sigma_3}(\{e_2, e_{16}\}))\), \(D_1\) is not a sync-preserving deadlock. However, \(e_{29}, e_{19} \notin \text{SPClosure}_{\sigma_3}(\text{pred}_{\sigma_3}(\{e_{29}, e_{19}\}))\), and \(e_{29}, e_{19} \notin \text{SPClosure}_{\sigma_3}(\text{pred}_{\sigma_3}(\{e_{29}, e_{19}\}))\), thus \(D_3\) 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 \(\leq_{\text{TO}}\) predecessors of the events in the deadlock pattern, and (ii) iteratively add \(\leq_{\text{TO}}\) and \(\text{rf}\) 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 \(\text{Threads}\) of threads, a timestamp is simply a mapping \(T : \text{Threads} \to \mathbb{N}\). 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 \(\leq_{\text{TRF}}\) be the reflexive transitive closure of the relation \((\leq_{\text{TO}} \cup \{\text{rf}_\sigma(e), e\}) \cup \{x \in \text{Vars}_\sigma, \text{op}(e) = \text{rf}(x)\}\); observe that \(\leq_{\text{TRF}}\) is a partial order. We define the timestamp \(T^{\sigma}_S(e)\) of an event \(e\) in \(\sigma\) to be a \(\text{Threads}_\sigma\)-indexed timestamp as follows: \(T^{\sigma}_S(e) = \{|f \mid f \leq_{\text{TRF}} e\}|\). This ensures that for two events \(e, e' \in \text{Events}_\sigma\), \(e \leq_{\text{TRF}} e'\) iff \(T^{\sigma}_S(e) \subseteq T^{\sigma}_S(e')\). For a set \(S \subseteq \text{Events}_\sigma\), we overload the notation and say the timestamp of \(S\) is \(T^{\sigma}_S = \bigsqcup \{T^{\sigma}_S(e) \mid e \in S\} \).
Given a trace $\sigma$ with $N$ events and $T$ threads we can compute these timestamps for all the events in $O(N \cdot T)$ 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 $\leq^\sigma_{\text{TRF}}$-closure. In order to identify such events efficiently, for every thread $t$ and lock $l$, we maintain a FIFO queue $\text{CSHist}_{t,l}$ (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 $O(|S| \cdot T + T \cdot A)$, where $T$ and $A$ are respectively the number of threads and acquire events in $\sigma$.

**Algorithm 1: CompSPClosure:**

Computing sync-preserving closure.

**Input:** Trace $\sigma$, Timestamp $I_0$

1. let $\{\text{CSHist}_{t,l} | t \in \text{Locks}_\sigma, l \in \text{Threads}_\sigma\}$ be the lock-acquisition histories in $\sigma$
2. $T \leftarrow I_0$
3. repeat
   4. for $l \in \text{Locks}_\sigma$ do
      5. foreach $t \in \text{Threads}_\sigma$ do
         6. let $e_t$ be the last event in $\text{CSHist}_{t,l}$
            with $TS^l_{\sigma,t} \subseteq T$
         7. Remove all earlier events in $\text{CSHist}_{t,l}$
         8. let $e_t$, be the last event in
            \[
            \{e_t | e_t \in \text{Threads}_\sigma \text{ according to } \leq^\sigma_{\text{trf}}
            \}
            \]
9. $T := T \cup \{TS^\sigma_{\text{match}^\sigma(e_t)} | e_t \neq e_t\}$
10. until $T$ does not change
11. return $T$

**Algorithm 2: CheckAbsDdlck:**

Checking an abstract deadlock pattern.

**Input:** Trace $\sigma$, $D^\text{abs}$ of length $k$

1. let $F_0, \ldots, F_{k-1}$ be the sequences of acquires in $D^\text{abs}$
2. let $n_0, \ldots, n_{k-1}$ be the lengths of $F_0, \ldots, F_{k-1}$
3. foreach $j \in \{0, \ldots, k - 1\}$ do
   $ij \leftarrow 1$
4. $T \leftarrow \lambda t, 0 \text{ while } \bigwedge_j ij < n_j$
5. let $e_0 = F_0[0], \ldots, e_{k-1} = F_{k-1}[i_{k-1}]$
6. $S \leftarrow \text{pred}_\sigma(e_0, \ldots, e_{k-1})$
7. $T \leftarrow \text{CompSPClosure}(\sigma, T \cup TS^S_{\sigma})$
8. if $\forall j < k, TS^l_{\sigma,j} \subseteq T$ then
   9. $\text{report pattern } D = e_0, \ldots, e_{k-1} \text{ and exit}$
10. foreach $j \in \{0, \ldots, k-1\}$ do
    $ij = \min\{l \leq n_j | TS^l_{\sigma,l} \not\subseteq T\}$

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 $O(k \cdot T)$ — simply check if $\forall i, TS^\sigma(e_i) \not\subseteq T$. This gives an algorithm for checking if a deadlock pattern of length $k$ is sync-preserving that runs in time $O(T \cdot N + k \cdot T + T \cdot A) = O(N \cdot T)$.

### 4.4 Verifying Abstract Deadlock Patterns

**Abstract acquires and abstract deadlock patterns.** Given a thread $t$, a lock $l$ and a set of locks $L \subseteq \text{Locks}_\sigma \neq \emptyset$ with $l \notin L$, we define the abstract acquire $\eta = (t, l, L, F)$, where $F = [e_1, \ldots, e_n]$ is the sequence of all events $e_i \in \text{Events}_\sigma$ (in trace-order) such that for each $i$, we have (i) $\text{thread}(e_i) = t$, (ii) $\text{op}(e_i) = \text{acq}(t)$, and (iii) $\text{HeldLks}_\sigma(e_i) = L$. In words, the abstract acquire $\eta$ 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 $\sigma$ is a sequence $D^\text{abs} = \eta_0, \ldots, \eta_{k-1}$ of abstract acquires $\eta_i = (t_i, l_i, L_i, F_i)$ such that $t_0, \ldots, t_{k-1}$ are distinct threads, $l_0, \ldots, l_{k-1}$ are distinct locks, $L_0, L_1, \ldots, L_{k-1} \subseteq \text{Locks}_\sigma$ are such that $l_i \notin L_i$, $\forall i$. 

are used in our algorithms, and illustrate for an example. Our next result is stated below, followed by its proof idea.

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

An abstract deadlock pattern of length $k \geq 2$ can have $N^k$ instantiations, giving a naive enumerate-and-check algorithm running in time $O(T \cdot N^{k+1})$, 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 $O(N \cdot T)$ time. The first observation allows us to re-use a prior computation when checking later deadlock patterns.

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

Consider $\sigma_3$ in Figure 3 and let $S = \text{pred}_{\sigma_3}(\{e_{29}, e_{16}\})$, and $S' = \text{pred}_{\sigma_3}(\{e_{29}, e_{19}\})$. The sets $S, S'$ satisfy the conditions of Proposition 4.4, hence $\text{SPClosure}_{\sigma_3}(S) \subseteq \text{SPClosure}_{\sigma_3}(S')$, 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 $D_1 < D_2$ if they are instantiations of a common abstract pattern $D^{\text{abs}}$ (i.e., $D_1, D_2 \in D^{\text{abs}}$) and for every $i < k$, $e_i \leq_{\sigma} f_i$.

**Corollary 4.5.** Let $\sigma$ 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 $\sigma$ such that $D_1 < D_2$. Let $S_1 = \{e_0, \ldots, e_{k-1}\}$ and $S_2 = \{f_0, \ldots, f_{k-1}\}$. If $\text{SPClosure}_{\sigma}(\text{pred}_{\sigma}(S_1)) \cap S_2 \neq \emptyset$, then $\text{SPClosure}_{\sigma}(\text{pred}_{\sigma}(S_2)) \cap S_2 \neq \emptyset$.

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 $\text{CSHist}_{i, f}$, used in the function $\text{CompSPClosure}$ 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 $\text{CSHist}_{i, f}$ in $\text{CompSPClosure}$ 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 $O(T)$ time for vector clock updates. The total time required is thus $O(N \cdot T)$. This concludes the proof of Lemma 4.3.
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 $\sigma$ is a directed graph $ALG_\sigma = (V_\sigma, E_\sigma)$, where

- $V_\sigma = \{(t_1, t_2, L_1, F_1), \ldots, (t_k, t_k, L_k, F_k)\} = \{\text{abstract acquires of } \sigma\}$, and
- for every $\eta_1 = (t_1, t_2, L_1, F_1), \eta_2 = (t_2, t_2, L_2, F_2) \in V_\sigma$, 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 $(t_1, t_2, L_1, F_1)$ signifies that there is an event $\text{acq}_1(t_1)$ 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 $\text{acq}_1$, ordered in order of appearance in $\sigma$. An edge $(\eta_1, \eta_2)$ signifies that the lock $t_1$ acquired by each of the events $\text{acq}_1 \in F_1$ was held by $t_2$ when it executed each of $\text{acq}_2 \in F_2$ while not holding a common lock. The abstract lock graph can be constructed incrementally as new events appear in $\sigma$. For $N$ events, $L$ locks and nesting depth $d$, the graph has $|V_\sigma| = O(T \cdot L^d)$ vertices, $|E_\sigma| = O(|V_\sigma| \cdot L^{d-1})$ edges and can be constructed in $O(N \cdot d)$ time. See Fig. 4 for examples. In the left graph, the cycle marks an abstract deadlock pattern and its single concrete deadlock pattern $D^{\text{abs}} = \{e_2\} \times \{e_3\}$, and similarly for the middle graph where $D^{\text{abs}} = \{e_2\} \times \{e_1\}$. In the right graph, there is a unique cycle which marks an abstract deadlock pattern of 6 concrete deadlock patterns $D^{\text{abs}} = \{e_2, e_4, e_29\} \times \{e_{16}, e_{19}\}$.

Algorithm SPDOffline. It is straightforward to verify that every abstract deadlock pattern of $\sigma$ appears as a (simple) cycle in $ALG_\sigma$. However, the opposite is not true. A cycle $C = \eta_0, \eta_1, \ldots, \eta_{k-1}$ of $ALG_\sigma$, where $\eta_1 = (t_1, t_2, L_1, F_1)$ defines an abstract deadlock pattern if additionally every thread $t_j$ is distinct, all every lock $\ell_i$ 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 $ALG_\sigma$, 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 $\sigma$ and the number of cycles in $ALG_\sigma$.

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

---

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 𝑇 and 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 𝑇 and 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 \ldots 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, SPClosure_σ₁ (pred_σ₁ (\{e₂, e₁₆\})) = \{e₁, \ldots , e₆, e₈, \ldots , e₁₅\}. As e₂ \in SPClosure_σ₁ (pred_σ₁ (\{e₂, e₁₆\})), 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 = ⟨e₂, e₁₉⟩ shares a common event e₂ with D_1 but contains the event e₁₉ instead of e₁₆, while e₅ \in SPClosure_σ₁ (pred_σ₁ (\{e₂, e₁₆\})). Since e₁₆ ≤ᵦ₁₆ e₁₉, and the sync-preserving closure grows monotonically (Proposition 4.4), the sync-preserving closure of e₂ and e₁₉ will also contain e₅ (and thus e₂). 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 SPClosure_σ₁ (pred_σ₁ (\{e₂, e₁₆\})) \cap S₅ = ∅ where S₅ = \{e₂₉, e₁₆\}. 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 D₁ and D₅.

**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 𝜂 = ⟨t, ℓ₁, {ℓ₂}, F⟩, i.e., we only focus on one lock ℓ₂ that is protecting each such acquire. When a new acquire event e = ⟨t, acq(ℓ₁)⟩ is encountered, the algorithm iterates over all the locks ℓ₂ ∈ HeldLks_σ (e) that are held in e, and append the event e to the sequence F of the corresponding abstract acquire 𝜂 = ⟨t, ℓ₁, {ℓ₂}, F⟩; 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
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.

THEOREM 5.1. Consider a trace \( \sigma \) of \( N \) events, \( T \) threads and \( L \) locks. The online SPDOnline algorithm reports all sync-preserving deadlocks of size 2 of \( \sigma \) in \( O(N \cdot T^3 \cdot L^2) \) time.

6 EXPERIMENTAL EVALUATION
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 $\tilde{O}(N^4)$, as opposed to the $\tilde{O}(N)$ 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.


**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 $ALG_\sigma$ of each trace $\sigma$. 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.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>N</th>
<th>T</th>
<th>V</th>
<th>L</th>
<th>A/R</th>
<th>Dlk</th>
<th>Dirk</th>
<th>SeqCheck</th>
<th>SPDOffLine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deadlock</td>
<td>39</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>8</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.02</td>
</tr>
<tr>
<td>NotADeadlock</td>
<td>60</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>16</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.02</td>
</tr>
<tr>
<td>Picklock</td>
<td>66</td>
<td>3</td>
<td>6</td>
<td>6</td>
<td>20</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0.02</td>
</tr>
<tr>
<td>Bensalem</td>
<td>68</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>22</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0.02</td>
</tr>
<tr>
<td>Transfer</td>
<td>72</td>
<td>3</td>
<td>11</td>
<td>4</td>
<td>12</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.02</td>
</tr>
<tr>
<td>Test-Dimmunix</td>
<td>73</td>
<td>3</td>
<td>9</td>
<td>7</td>
<td>26</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0.02</td>
</tr>
<tr>
<td>StringBuffer</td>
<td>74</td>
<td>3</td>
<td>14</td>
<td>4</td>
<td>16</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>0.02</td>
</tr>
<tr>
<td>Test-Callfuzz</td>
<td>165</td>
<td>5</td>
<td>16</td>
<td>6</td>
<td>48</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0.02</td>
</tr>
<tr>
<td>DiningPhil</td>
<td>277</td>
<td>6</td>
<td>21</td>
<td>6</td>
<td>100</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>177.18</td>
</tr>
<tr>
<td>HashTable</td>
<td>318</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>174</td>
<td>1</td>
<td>2</td>
<td>43</td>
<td>0.19</td>
</tr>
<tr>
<td>Account</td>
<td>706</td>
<td>6</td>
<td>47</td>
<td>7</td>
<td>134</td>
<td>3</td>
<td>1</td>
<td>12</td>
<td>0.19</td>
</tr>
<tr>
<td>Log4j</td>
<td>1K</td>
<td>4</td>
<td>334</td>
<td>11</td>
<td>43</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.65</td>
</tr>
<tr>
<td>Dbcp1</td>
<td>2K</td>
<td>3</td>
<td>768</td>
<td>5</td>
<td>56</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>F</td>
</tr>
<tr>
<td>Dbcp2</td>
<td>2K</td>
<td>3</td>
<td>592</td>
<td>10</td>
<td>76</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>F</td>
</tr>
<tr>
<td>Derby2</td>
<td>3K</td>
<td>3</td>
<td>1K</td>
<td>4</td>
<td>16</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.23</td>
</tr>
<tr>
<td>RayTracer</td>
<td>31K</td>
<td>5</td>
<td>5K</td>
<td>15</td>
<td>976</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>F</td>
</tr>
<tr>
<td>jigsaw</td>
<td>149K</td>
<td>21</td>
<td>8K</td>
<td>2K</td>
<td>67K</td>
<td>172</td>
<td>12</td>
<td>70</td>
<td>F</td>
</tr>
<tr>
<td>elevator</td>
<td>246K</td>
<td>5</td>
<td>727</td>
<td>52</td>
<td>48K</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1.65</td>
</tr>
<tr>
<td>hdedc</td>
<td>410K</td>
<td>7</td>
<td>109K</td>
<td>8</td>
<td>32</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2.09</td>
</tr>
<tr>
<td>JDBCMysql-1</td>
<td>442K</td>
<td>3</td>
<td>73K</td>
<td>11</td>
<td>13K</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>28.45</td>
</tr>
<tr>
<td>JDBCMysql-2</td>
<td>442K</td>
<td>3</td>
<td>73K</td>
<td>11</td>
<td>13K</td>
<td>4</td>
<td>4</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>JDBCMysql-3</td>
<td>443K</td>
<td>3</td>
<td>73K</td>
<td>13</td>
<td>13K</td>
<td>5</td>
<td>8</td>
<td>16</td>
<td>31.23</td>
</tr>
<tr>
<td>JDBCMysql-4</td>
<td>443K</td>
<td>3</td>
<td>73K</td>
<td>14</td>
<td>13K</td>
<td>5</td>
<td>10</td>
<td>18</td>
<td>2.51</td>
</tr>
<tr>
<td>cache4j</td>
<td>775K</td>
<td>2</td>
<td>46K</td>
<td>20</td>
<td>35K</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5.86</td>
</tr>
<tr>
<td>ArrayList</td>
<td>3M</td>
<td>801</td>
<td>121K</td>
<td>802</td>
<td>176K</td>
<td>9</td>
<td>3</td>
<td>672</td>
<td>3.87</td>
</tr>
<tr>
<td>IdentityHashMap</td>
<td>3M</td>
<td>801</td>
<td>496K</td>
<td>802</td>
<td>162K</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>443.93</td>
</tr>
<tr>
<td>Stack</td>
<td>3M</td>
<td>801</td>
<td>118K</td>
<td>2K</td>
<td>405K</td>
<td>9</td>
<td>3</td>
<td>481</td>
<td>1</td>
</tr>
<tr>
<td>Sor</td>
<td>3M</td>
<td>301</td>
<td>2K</td>
<td>3</td>
<td>719K</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>15.89</td>
</tr>
<tr>
<td>LinkedList</td>
<td>3M</td>
<td>801</td>
<td>290K</td>
<td>802</td>
<td>176K</td>
<td>9</td>
<td>3</td>
<td>10K</td>
<td>4.7K</td>
</tr>
<tr>
<td>HashMap</td>
<td>3M</td>
<td>801</td>
<td>555K</td>
<td>802</td>
<td>169K</td>
<td>1</td>
<td>3</td>
<td>10K</td>
<td>4.4K</td>
</tr>
<tr>
<td>WeakHashMap</td>
<td>3M</td>
<td>801</td>
<td>540K</td>
<td>802</td>
<td>169K</td>
<td>1</td>
<td>3</td>
<td>10K</td>
<td>-</td>
</tr>
<tr>
<td>Swing</td>
<td>4M</td>
<td>8</td>
<td>31K</td>
<td>739</td>
<td>2M</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.72</td>
</tr>
<tr>
<td>Vector</td>
<td>4M</td>
<td>3</td>
<td>15</td>
<td>4800</td>
<td>1</td>
<td>1</td>
<td>1B</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>LinkedHashMap</td>
<td>4M</td>
<td>801</td>
<td>617K</td>
<td>802</td>
<td>169K</td>
<td>1</td>
<td>3</td>
<td>10K</td>
<td>2.40</td>
</tr>
<tr>
<td>montecarlo</td>
<td>8M</td>
<td>3</td>
<td>850K</td>
<td>3</td>
<td>26</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2.6K</td>
</tr>
<tr>
<td>TreeMap</td>
<td>9M</td>
<td>801</td>
<td>493K</td>
<td>802</td>
<td>169K</td>
<td>1</td>
<td>3</td>
<td>10K</td>
<td>2.105</td>
</tr>
<tr>
<td>hsqldb</td>
<td>20M</td>
<td>46</td>
<td>945K</td>
<td>403</td>
<td>419K</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>sunflow</td>
<td>21M</td>
<td>16</td>
<td>2M</td>
<td>12</td>
<td>1K</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>F</td>
</tr>
<tr>
<td>jsipider</td>
<td>22M</td>
<td>11</td>
<td>5M</td>
<td>15</td>
<td>10K</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>F</td>
</tr>
<tr>
<td>tradesoap</td>
<td>42M</td>
<td>236</td>
<td>3M</td>
<td>6K</td>
<td>245K</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>F</td>
</tr>
<tr>
<td>tradebeans</td>
<td>42M</td>
<td>236</td>
<td>3M</td>
<td>6K</td>
<td>245K</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>F</td>
</tr>
<tr>
<td>eclipse</td>
<td>64M</td>
<td>15</td>
<td>10M</td>
<td>5K</td>
<td>377K</td>
<td>9</td>
<td>5</td>
<td>280</td>
<td>-</td>
</tr>
<tr>
<td>TestPerf</td>
<td>80M</td>
<td>50</td>
<td>599K</td>
<td>9</td>
<td>197K</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>795.04</td>
</tr>
<tr>
<td>Groovy2</td>
<td>120M</td>
<td>13</td>
<td>13M</td>
<td>10K</td>
<td>69K</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1.7K</td>
</tr>
<tr>
<td>Tsp</td>
<td>200M</td>
<td>6</td>
<td>24K</td>
<td>3</td>
<td>882</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7.6K</td>
</tr>
<tr>
<td>lusarch</td>
<td>203M</td>
<td>7</td>
<td>3M</td>
<td>98</td>
<td>273K</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1.3K</td>
</tr>
<tr>
<td>biojava</td>
<td>221M</td>
<td>6</td>
<td>121K</td>
<td>79</td>
<td>16K</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>F</td>
</tr>
<tr>
<td>graphenei</td>
<td>241M</td>
<td>20</td>
<td>25M</td>
<td>61</td>
<td>1K</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>F</td>
</tr>
</tbody>
</table>

|   | 1B | 7K | 70M | 37K | 8M | 256 | 93 | 1B | >18h | 40 | 2801 | 40 | 135 |

| Totals | 1B | 7K | 70M | 37K | 8M | 256 | 93 | 1B | >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 $\sim 21 \times$ faster than SeqCheck. On the most demanding benchmarks, such as `HashMap` and `TreeMap`, SPDOffline is more than $200 \times$ 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 $\text{pred}(D)$ with respect to $\leq_{\text{TO}}$ and $\text{rf}$, 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 $\text{acq}_1$, $\text{acq}_2$ participating in the deadlock pattern, each $\text{acq}_i$ is preceded by a critical section on a lock that appears in $\text{HeldLks}(\text{acq}_{i-1})$, 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 SPDOffline 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 attempts to write on a shared variable 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
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 [Șerbânută 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


Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. 2012. Sound Predictive Race Detection


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