Sound Dynamic Deadlock Prediction in Linear Time

Umang Mathur
University of Illinois, Urbana Champaign
USA
umathur3@illinois.edu

Matthew S. Bauer
Galois, Inc.
USA
mbauer@galois.com

Mahesh Viswanathan
University of Illinois, Urbana Champaign
USA
vmahesh@illinois.edu

Abstract

We consider the problem of dynamic deadlock prediction, i.e., inferring the presence of deadlocks in programs by analyzing executions. Traditional dynamic deadlock detection techniques rely on constructing a *lock graph* and checking for cycles in it. However, a cycle in a lock graph only implies a *potential* deadlock that can be a false alarm. In order to guarantee soundness (i.e. the absence of false positives), deadlock detectors must confirm potential deadlocks through program re-execution or constraint solving. The former technique requires heuristically controlling the thread scheduler in the hope that a deadlock is encountered while the latter doesn't scale to large programs. We propose a partial order DCP (Deadlock Causal Precedence) and a vector-clock algorithm that can identify the presence of deadlocks in a program using DCP. Our technique is sound and the algorithm runs in linear time, utilizing a single-pass through a program's trace. The experimental evaluation of our algorithm shows that it is significantly faster and more effective than existing state-of-the-art sound deadlock detection tools.

1 Introduction

Concurrent programs use shared resources (such as locks) and communication primitives (such as wait and notify) to synchronize operations performed in threads that otherwise evolve asynchronously. When these mechanisms are used improperly, they can introduce deadlocks. *Resource deadlocks* arise when a set of threads are waiting to acquire a lock that is held by another thread in the same set, and commonly result from developers adding synchronization mechanisms to prevent other concurrency bugs such as data races.

There are two main approaches to discovering deadlocks in software. Methods derived from static analysis and model checking analyze source code to over-approximate a set of potential deadlocks. These include the use of type systems [5], flow-sensitive and interprocedural analysis [11], flow and context sensitive analysis [33], theorem provers and decision procedures [13, 14], and may alias analysis and reachability [28]. However, the principal drawback of these approaches is that either they are too conservative and generate many false alarms, or if they are precise, then they don't scale to large software. The other popular approach is a hybrid approach [1, 3, 6, 12, 18, 32, 34], where the trace of a program is analyzed using lock graphs [15, 16] that reveal the nesting structure of critical sections to identify potential deadlocks. Unfortunately, the potential deadlocks identified in this manner may not be real deadlocks. Therefore, the first phase of identifying potential deadlocks is coupled with a second phase where the program is re-executed in an attempt to identify a schedule that witnesses a potential deadlock [2, 7–10, 25]. However, re-execution to schedule a potential deadlock has significant drawbacks. Not only is finding the schedule like searching for the proverbial needle in a haystack, re-execution requires knowing the original inputs used in the first run which may not have been recorded. In addition, it relies on the ability to control the thread scheduler which is challenging when the software uses third party libraries.

The drawback of the hybrid approach to deadlock detection can be avoided if the dynamic analysis is *sound* and *predictive*. In other words, whenever the dynamic analysis claims the presence of a deadlock, one can prove that there is some execution of the program (not necessarily the one that was analyzed) where some of the program threads are deadlocked. Recently, the first sound, predictive deadlock detection algorithm was proposed [20]. The approach taken in this paper is to identify potential deadlocks using a graph based analysis, and then, instead of re-executing the program to schedule the deadlock, a set of constraints are identified that correspond to a valid candidate execution of the program in which the deadlock is reached. If the constraints are satisfiable, then the deadlock is guaranteed to be scheduled. Satisfiability of the constructed constraints are then checked using an off-the-shelf SMT-solver.

The main disadvantage of the approach in [20] is that satisfiability checking is computationally expensive — though SAT solvers have made impressive advances, the fundamental problem remains intractable. This means that for a long trace, the constraints identifying candidate schedules for a deadlock are too large for a solver to handle. Thus, one is forced to use a "windowing strategy" [17], where the trace is broken up into smaller subtraces, and constraints are constructed for each subtrace. The windowing strategy allows the SMT-solver based approach to scale to large traces, at the expense of possibly failing to identify deadlocks.

In this paper, we present a philosophically different approach to sound, predictive deadlock detection. We rely on

PLDI'19, June 22–28, 2019, Phoenix, AZ, USA
2019.

	t_1	t_2
1	acq(ℓ)	
2	acq(m)	
3	rel(m)	
4	rel(ℓ)	
5		acq(m)
6		acq(ℓ)
7		rel(ℓ)
8		rel(m)

Figure 1. Trace ρ_1 of a program that has a deadlock.

partial orders. Partial order based dynamic analysis has been extensively used in the context of race detection [21, 24, 30] to obtain sound, precise, and scalable algorithms. The idea behind these algorithms is to identify a partial order P on the events of a trace with special properties that ensure that some of the events unordered by P are "concurrent". That is, any program that exhibits the trace being analyzed, will generate a trace where the unordered events are consecutive. Partial order based analysis algorithms are typically "streaming" (that is, events of the trace are processed as they appear and the algorithm does not rely on random access to them), and run in *linear* time, which allows them to scale to traces from industrial size programs. The possible downside of these approaches is that the partial order P is typically conservative, and is therefore, theoretically, likely to identify fewer anomalies than SMT-solver based approaches.

The most difficult challenge in coming up with a partial order based dynamic analysis is defining an appropriate partial order. Let us focus our attention on two thread deadlocks, which is the most common form of deadlocks in programs — 97% of all deadlock bugs in software have been emprically observed to be two thread deadlocks [27]. Consider the program shown in Figure 1. In all traces in this paper we follow the conventions of representing events top down, with temporally earlier events appearing above later events, and we denote the i^{th} event in the trace by e_i. Also, we use $\texttt{acq}(\ell)/\texttt{rel}(\ell)$ to denote the acquire/release of lock ℓ. In trace ρ_1 (of Figure 1), the program does not deadlock. But we could reorder the events of ρ_1 to obtain a deadlock—execute e_1 followed by e_5. The goal of a partial order based deadlock detection approach would be identify a partial order that does not order e_2 and e_6, to argue that they can be "concurrent". The most commonly used partial order, namely *happens before* [24], is too strong to be able to detect deadlocks even in this simple example. Happens before orders all critical sections on the same lock, and therefore, event e_2 is ordered before e_5 and hence also before e_6. To detect deadlocks, we need a *weaker* partial order that will not order events e_2 and e_6 of ρ_1 and can reason about alternate traces where critical sections on the same lock maybe reordered.

The inspiration for our partial order based deadlock detection alorithm is the order weak causal precedence (WCP) introduced in [21], which is the weakest sound ordering we are aware of. However, while WCP is a sound ordering for detecting data races, it is inadequate to reason about deadlocks. The reason is because the soundness guarantee (that unordered events can be concurrently scheduled) only applies to the first pair of unordered data access events, and it is not clear how to extend the proof to other events in the presence of data races. Therefore, we introduce a new partial order called *deadlock causal precedence (DCP)* that, though similar in spirit to WCP, is appropriate for deadlock detection. Our main results states that a trace σ has a predictable deadlock, if there are a pair of threads t_1 and t_2, and events $e = \texttt{acq}(\ell)$ and $f = \texttt{acq}(m)$ performed by threads t_1 and t_2, respectively, such that the following properties hold: (a) e and f are unordered by DCP; (b) locks held by t_1 at e is disjoint from the locks held by t_2 at f; and (c) lock m is held by t_1 at e and lock ℓ is held by t_2 at f. The proof to establish the soundness of DCP is nontrivial. It subtly exploits the soundness guarantees of WCP — given a trace σ, we show that one can construct another trace σ' such that the soundness of DCP for deadlock detection in σ follows from the soundness of WCP for race detection in σ'.

Next, we show that deadlock detection using DCP has a *streaming, linear time* algorithm. Our algorithm is a vector clock based algorithm that computes the DCP partial order on events of the trace. It is similar in structure to the vector clock algorithm for WCP [21], and it could, in the worst case, use linear space. We prove that our algorithm has optimal resource bounds from a couple of perspectives. First we show that any linear time algorithm computing the partial order DCP must use linear space. Second we prove that *any* sound, predictive deadlock detection algorithm running in linear time, must use at least linear space. Notice that our lower bound applies to any deadlock detection algorithm and not just to those that are DCP-based or (more generally) partial order based. Our lower bound proofs rely on lifting results from communication complexity [23] to this context. Therefore, our DCP-based deadlock detection algorithm is the best one can hope for in terms of asymptotic complexity.

Our first DCP-based deadlock detection algorithm makes very strong assumptions about data dependency — we assume that every value read and written during the execution can influence the control flow of the program. What if the execution explicitly tags branching events, and one can make weaker data dependency assumptions? Can one detect more deadlocks? Is it is easy to incorporate such information to get a new algorithm? We answer all these questions in the affirmative and present a modified DCP-based algorithm that incorporates data flow information to obtain a sound algorithm that is more precise. Our modified DCP-based algorithm is a two pass algorithm running in linear time.

Finally, the DCP based algorithm has been implemented and tested on standard benchmark programs. Our evaluation demonstrates the power of a linear-time sound deadlock prediction algorithm — DCP allows us to scale to traces with billions of events, without compromising prediction power. We observed that our approach is significantly faster than existing contemporary techniques that rely on SMT solvers — with speed-ups as high as $380,000\times$, with a median speedup of more than $6\times$.

The rest of the paper is organized as follows. Basic definitions and notations are introduced in Section 2. Our partial order DCP is defined in Section 3, and we present illustrative examples that highlight features of its definition and demonstrate its predictive power. In Section 4, we give details of our vector clock algorithm for predictive deadlock detection. Data flow based dynamic analysis is introduced in Section 5. Our algorithms have been implemented in our tool DeadTrack . We present an experimental comparison of our algorithm with other deadlock detection approaches on standard benchmark examples in Section 6. Conclusions and future work is presented in Section 7.

2 Preliminaries

Traces and Events. We assume the sequential consistency model for shared memory concurrent programs. Under this assumption, a program execution, or trace, can be seen as a sequence of events. We will use $\sigma, \sigma', \rho, \rho_1, \rho_2 \ldots$ to denote traces. An event of a trace σ is a pair $e = \langle t, op \rangle$[1], where t is the thread that performs the event e and op is the operation performed in the event and can be one of $\mathtt{r}(x)$ (read from memory location x), $\mathtt{w}(x)$ (write to x), $\mathtt{acq}(\ell)$ (acquire of lock ℓ), $\mathtt{rel}(\ell)$ (release of ℓ), $\mathtt{fork}(u)$ (fork of thread u) or $\mathtt{join}(u)$ (join of thread u). We will use $\mathsf{thr}(e)$ and $\mathsf{op}(e)$ to denote the thread and the operation performed by the event e. For a trace σ, we denote the set of threads in σ by $\mathsf{Threads}_\sigma$, the set of memory locations (or variables) accessed by σ as Vars_σ, and the set of locks acquired or released as Locks_σ. The set of all events in σ will be denoted by Events_σ. We denote the set of events that read from and write to a memory location $x \in \mathsf{Vars}_\sigma$ by $\mathsf{Reads}_\sigma(x)$ and $\mathsf{Writes}_\sigma(x)$ respectively and set $\mathsf{Accesses}_\sigma(x) = \mathsf{Reads}_\sigma(x) \cup \mathsf{Writes}_\sigma(x)$. We use $\mathsf{Reads}_\sigma = \bigcup_{x \in \mathsf{Vars}_\sigma} \mathsf{Reads}_\sigma(x)$, $\mathsf{Writes}_\sigma = \bigcup_{x \in \mathsf{Vars}_\sigma} \mathsf{Writes}_\sigma(x)$ and $\mathsf{Accesses}_\sigma = \bigcup_{x \in \mathsf{Vars}_\sigma} \mathsf{Accesses}_\sigma(x)$. We will use $\mathsf{Acquires}_\sigma(\ell)$ (resp. $\mathsf{Releases}_\sigma(\ell)$) to denote the set of events that acquire (resp. release) a lock $\ell \in \mathsf{Locks}_\sigma$. Further, $\mathsf{Acquires}_\sigma = \bigcup_{\ell \in \mathsf{Locks}_\sigma} \mathsf{Acquires}_\sigma(\ell)$ and $\mathsf{Releases}_\sigma = \bigcup_{\ell \in \mathsf{Locks}_\sigma} \mathsf{Releases}_\sigma(\ell)$. Traces are assumed to respect lock semantics—a lock ℓ that is acquired by a thread t cannot be acquired by another thread t' until t releases the lock. To keep the presentation simple, we assume none of the locks are re-entrant, i.e., a lock cannot be reacquired by an owning thread until it is released. However, the results can be easily generalized to executions with re-entrant locking. For a release event e, the matching acquire event for e in trace σ will be denoted by $\mathsf{match}_\sigma(e)$. Similarly, we will denote by $\mathsf{match}_\sigma(e)$ to be the matching release event (if any) corresponding to the acquire event e, and set it to $\bot$ if it does not exist. Further, each thread is assumed to be forked and joined at most once.

[1]Formally, each event has an associated *unique event identifier*. Thus, two events performed by the same thread and performing the same operation are considered *different* events. However, to reduce notational overhead we will not formally introduce these identifiers and implicitly assume that each event is unique.

	t_1	t_2	t_3
1	$\mathtt{w}(x)$		
2	$\mathtt{acq}(\ell)$		
3	$\mathtt{fork}(t_2)$		
4		$\mathtt{acq}(m)$	
5		$\mathtt{rel}(m)$	
6		$\mathtt{r}(x)$	
7	$\mathtt{join}(t_2)$		
8	$\mathtt{rel}(\ell)$		
9			$\mathtt{acq}(m)$
10			$\mathtt{r}(x)$
11			$\mathtt{acq}(\ell)$
12			$\mathtt{rel}(\ell)$

Figure 2. Trace ρ_2 for illustration.

Orders on Traces. For a trace σ and events $e_1, e_2 \in \mathsf{Events}_\sigma$, we say e_1 is *trace-ordered* before e_2, denoted $e_1 \leq^\sigma_{\mathsf{tr}} e_2$ if e_1 occurs before (or is the same as) e_2 in the sequence σ. For a trace σ, the *thread order* of σ, denoted $\leq^\sigma_{\mathsf{TO}}$ is the smallest partial order such that for any two events $e_1, e_2 \in \mathsf{Events}_\sigma$ with $e_1 \leq^\sigma_{\mathsf{tr}} e_2$, if either (a) $\mathsf{thr}(e_1) = \mathsf{thr}(e_2)$ or , (b) $\mathsf{op}(e_1) = \mathtt{fork}(\mathsf{thr}(e_2))$ or, (c) $\mathsf{op}(e_2) = \mathtt{join}(\mathsf{thr}(e_1))$, then $e_1 \leq^\sigma_{\mathsf{TO}} e_2$. We will use $e <^\sigma_{\mathsf{tr}} e'$ (resp. $e <^\sigma_{\mathsf{TO}} e'$) when we have $e \leq^\sigma_{\mathsf{tr}} e'$ (resp. $e \leq^\sigma_{\mathsf{TO}} e'$) and $e \neq e'$. We say that a trace σ *respects* a partial order $\leq_{\mathsf{PO}}$ on some set of events $E \supseteq \mathsf{Events}_\sigma$ if for each pair of events $e_1, e_2 \in E$ with $e_1 \leq_{\mathsf{PO}} e_2$, whenever $e_2 \in \mathsf{Events}_\sigma$, then we have $e_1 \in \mathsf{Events}_\sigma$ and $e_1 \leq^\sigma_{\mathsf{tr}} e_2$. Given a trace σ and a partial order $\leq^\sigma_{\mathsf{PO}}$ over Events_σ, events $e_1, e_2 \in \mathsf{Events}_\sigma$ are said to be *concurrent* according to $\leq^\sigma_{\mathsf{PO}}$ if neither $e_1 \leq^\sigma_{\mathsf{PO}} e_2$, nor $e_2 \leq^\sigma_{\mathsf{PO}} e_1$. This is denoted by $e_1 \,||^\sigma_{\mathsf{PO}}\, e_2$.

Example 1. Let us illustrate the definitions introduced so far on the trace ρ_2 shown in Figure 2. Recall that we denote the i^{th} event in ρ_2 by e_i. Here, $\mathsf{Threads}_{\rho_2} = \{t_1, t_2, t_3\}$, $\mathsf{Vars}_{\rho_2} = \{x\}$ and $\mathsf{Locks}_{\rho_2} = \{\ell, m\}$. Further, $\mathsf{Events}_{\rho_2} = \{e_i\}_{1 \leq i \leq 12}$, $\mathsf{Reads}_{\rho_2} = \mathsf{Reads}_{\rho_2}(x) = \{e_6, e_{10}\}$, $\mathsf{Writes}_{\rho_2} = \mathsf{Writes}_{\rho_2}(x) = \{e_1\}$, $\mathsf{Acquires}_{\rho_2}(\ell) = \{e_2, e_{11}\}$ and $\mathsf{Releases}_{\rho_2} = \{e_5, e_8, e_{12}\}$. Finally, $\mathsf{match}_{\rho_2}(e_7) = e_2$ and $\mathsf{match}_{\rho_2}(e_8) = \bot$. Let us now illustrate the different orders using σ. Here, $e_i \leq^{\rho_2}_{\mathsf{tr}} e_j$ iff $i \leq j$. $e_1 \leq^{\rho_2}_{\mathsf{TO}} e_3$ because both e_1 and e_3 are performed by t_1. Further, $e_1 \leq^{\rho_2}_{\mathsf{TO}} e_5$ because of the event e_3 which forks t_2, and, $e_7 \,||^{\rho_2}_{\mathsf{TO}}\, e_{12}$. The sequence $\rho_2' = e_9 e_1 e_2 e_3 e_4$ is a valid trace that preserves $\leq^{\rho_2}_{\mathsf{TO}}$ but does not preserve the total order $\leq^{\rho_2}_{\mathsf{tr}}$. On the other hand, the trace $\rho_2'' = e_1 e_2 e_4$ does not preserve $\leq^{\rho_2}_{\mathsf{TO}}$ because $e_4 \in \mathsf{Events}_{\rho_2''}$, $e_3 \leq^{\rho_2}_{\mathsf{TO}} e_4$ but $e_3 \notin \mathsf{Events}_{\rho_2''}$.

Well Nesting. We will assume that all lock acquires and releases in a trace follow the well-nesting principle that intuitively states that a thread holding multiple locks must release them in the future in reverse order of how they were acquired. In the presence of forks and joins this definition becomes subtle, and we define it precisely as follows. A trace σ is well nested if for any acquire $e \in \mathsf{Acquires}_\sigma$ such that $\mathsf{match}_\sigma(e)$ exists in σ and for any event f with $e \leq^\sigma_{\mathsf{TO}} f$ the following two conditions hold: (a) either $f \leq^\sigma_{\mathsf{TO}} \mathsf{match}_\sigma(e)$ or $\mathsf{match}_\sigma(e) \leq^\sigma_{\mathsf{TO}} f$, i.e., f is thread ordered in relation to $\mathsf{match}_\sigma(e)$, and (b) if $f \in \mathsf{Acquires}_\sigma$ with $f \leq^\sigma_{\mathsf{TO}} \mathsf{match}_\sigma(e)$, then $\mathsf{match}_\sigma(f)$ exists in σ and $\mathsf{match}_\sigma(f) \leq^\sigma_{\mathsf{TO}} \mathsf{match}_\sigma(e)$.

Conflicting Events. Two events $e_1, e_2 \in \mathsf{Events}_\sigma$ are said to be *conflicting* if $e_1, e_2 \in \mathsf{Accesses}_\sigma(x)$ for some $x \in \mathsf{Vars}_\sigma$, $\mathsf{w}(x) \in \{\mathsf{op}(e_1), \mathsf{op}(e_2)\}$, and $e_1 \,||^\sigma_{\mathsf{TO}}\, e_2$. We use $e_1 \asymp e_2$ to denote that e_1 and e_2 are conflicting events.

Critical Sections, Locks Held and Next Events. For an acquire event $e \in \mathsf{Acquires}_\sigma$, we say that the critical section of e, denoted $\mathsf{CS}_\sigma(e)$ is the set $\{f \mid e \leq^\sigma_{\mathsf{TO}} f \leq^\sigma_{\mathsf{TO}} \mathsf{match}_\sigma(e)\}$ if e has a matching release, and $\{f \mid e \leq^\sigma_{\mathsf{TO}} f\}$ otherwise. Similarly, for a release event e, $\mathsf{CS}_\sigma(e) = \mathsf{CS}_\sigma(\mathsf{match}_\sigma(e))$. The outermost acquire and release events (if they exist) for a critical section C will be denoted $\mathtt{acq}(C)$ and $\mathtt{rel}(C)$. The set of locks held at an event $e \in \mathsf{Events}_\sigma$ is the set $\mathsf{locksHeld}_\sigma(e) = \{\ell \mid \exists e' \in \mathsf{Acquires}_\sigma(\ell) \text{ such that } e \in \mathsf{CS}_\sigma(e')\}$. For an event $e \in \mathsf{Events}_\sigma$, the set of *next events* of e (denoted $\mathsf{next}_\sigma(e)$) are the ones that are scheduled immediately after e as per thread order. That is, $\mathsf{next}_\sigma(e) = \{e' \mid e <^\sigma_{\mathsf{TO}} e', \text{ and } \nexists e'' \cdot e <^\sigma_{\mathsf{TO}} e'' <^\sigma_{\mathsf{TO}} e'\}$. Clearly, if e is a fork event, then the size of the set $\mathsf{next}_\sigma(e)$ can either be 0, 1 or 2. In all other cases, $\mathsf{next}_\sigma(e)$ is either empty or is singleton.

Example 2. In the trace ρ_2 from Figure 2, the pair (e_1, e_6) is not a conflicting pair of events since $e_1 \leq^{\rho_2}_{\mathsf{TO}} e_6$. However, e_1 conflicts with e_{10} in ρ_2 (i.e., $e_1 \asymp e_{10}$). The critical sections in ρ_2 are $\mathsf{CS}_{\rho_2}(e_2) = \mathsf{CS}_{\rho_2}(e_8) = \{e_2, e_3, e_4, e_5, e_6, e_7, e_8\}$, $\mathsf{CS}_{\rho_2}(e_4) = \mathsf{CS}_{\rho_2}(e_5) = \{e_4, e_5\}$, $CS(e_9) = \{e_9, e_{10}, e_{11}, e_{12}\}$ and $\mathsf{CS}_{\rho_2}(e_{11}) = \mathsf{CS}_{\rho_2}(e_{12}) = \{e_{11}, e_{12}\}$. In this trace, the set of locks held at e_1 is $\varnothing$. On the other hand, $\mathsf{locksHeld}_{\rho_2}(e_2) = \mathsf{locksHeld}_{\rho_2}(e_3) = \{\ell\}$ and $\mathsf{locksHeld}_{\rho_2}(e_{10}) = \{m\}$. Finally, $\mathsf{next}_{\rho_2}(e_1) = \{e_2\}$, $\mathsf{next}_{\rho_2}(e_3) = \{e_4, e_7\}$, $\mathsf{next}_{\rho_2}(e_6) = \{e_7\}$ and $\mathsf{next}_{\rho_2}(e_{12}) = \varnothing$.

Correct Reordering. The goal of this paper is to perform predictive deadlock detection, i.e., given a trace σ we want to check if either σ or another valid rescheduling of the events of σ exhibit a deadlock. *Correct reorderings* of a trace σ, identify the space of "valid reschedules" that are semantically sound; any program producing the trace σ could produce any of the correct reorderings under a different thread schedule. For a read event $e \in \mathsf{Reads}_\sigma(x)$ on some $x \in \mathsf{Vars}_\sigma$, we denote by $\mathsf{lw}_\sigma(e)$ the last write event $e' \in \mathsf{Writes}_\sigma(x)$ before e in the sequence σ if it exists; and $\mathsf{lw}_\sigma(e) = \bot$ otherwise. A trace σ' is a *correct reordering* of a trace σ if (a) $\mathsf{Events}_{\sigma'} \subseteq \mathsf{Events}_\sigma$, (b) σ' respects $\leq^\sigma_{\mathsf{TO}}$, and (c) for every $e \in \mathsf{Reads}_{\sigma'}$, $\mathsf{lw}_{\sigma'}(e) = \mathsf{lw}_\sigma(e)$. To ensure that programs producing σ can produce each of its correct reorderings, we require correct reorderings of σ preserve intra-thread order, fork and join dependencies. Further, in a correct reordering every read event sees the same *value* as in σ, and as a result, any conditionals in any branches encountered in the underlying program still evaluate to the same value.

Predictable Deadlock. A deadlock happens when a group of threads are waiting for each other in a cyclic dependency. Predictable deadlocks happen when a trace can be reordered to exhibit a deadlock. Formally, a trace σ is said to exhibit a *predictable deadlock* of size k if there is a correct reordering σ' of σ, k distinct events $e_0, e_1, \ldots, e_{k-1} \in \mathsf{Events}_\sigma$, and k distinct locks $\ell_0, \ell_1, \ldots, \ell_{k-1} \in \mathsf{Locks}_\sigma$ such that for every $0 \leq i \leq k-1$, we have

(a) $\ell_i \in \mathsf{locksHeld}_{\sigma'}(e_i)$, and
(b) there is an event $f_i \in \mathsf{next}_\sigma(e_i)$ such that $\mathsf{op}(f_i) = \mathsf{acq}(\ell_{(i+1)\%k})$.

The witness for a predictable deadlock, therefore, is a correct reordering σ' of σ such that ℓ_i is acquired but not released when e_i is performed in σ' (condition (a)) and each thread $t_i = \mathsf{thr}(e_i)$ is waiting for the lock $\ell_{(i+1)\%k}$ to be released (condition (b)).

Deadlock Pattern. A *deadlock pattern* of size k in a trace σ is a tuple of $2k$ events $\langle e_0, f_0, e_1, f_1, \ldots, e_{k-1}, f_{k-1}\rangle$ such that there are distinct locks $\ell_0, \ell_1 \ldots, \ell_{k-1} \in \mathsf{Locks}_\sigma$ for which $\mathsf{op}(e_i) = \mathsf{acq}(\ell_i)$, $f_i \in \mathsf{CS}_\sigma(e_i)$, $\mathsf{op}(f_i) = \mathsf{acq}(\ell_{(i+1)\%k})$, and

$$(\mathsf{locksHeld}_\sigma(f_i) \setminus \{\ell_{(i+1)\%k}\}) \cap (\mathsf{locksHeld}_\sigma(f_j) \setminus \{\ell_{(j+1)\%k}\}) = \varnothing$$

for every $0 \leq i, j \leq k-1$. Observe that if a trace σ has a predictable deadlock, then it also has a deadlock pattern of the same size. However, the converse is not true. That is, a trace exhibiting a deadlock pattern may not have any predictable deadlock. Algorithms such as Goodlock [16] and its generalizations [1] report deadlock patterns (also known as *potential deadlocks*) in executions by constructing and analyzing a *lock graph* whose nodes correspond to locks and there is an edge from a node ℓ_1 to ℓ_2 if ℓ_2 is acquired in a critical section of lock ℓ_1. Since deadlock patterns can be false positives, existing approaches [2, 25, 29] resort to re-executing the original program hoping to encounter a correct reordering of the original trace that witnesses the deadlock. We remark that existing Goodlock based algorithms often miss reporting deadlock patterns $\langle e_0, f_0, \ldots, e_{k-1}, f_{k-1}\rangle$ if, for some i, $\mathsf{thr}(e_i) \neq \mathsf{thr}(f_i)$ (eventhough f_i belongs to $\mathsf{CS}_\sigma(e_i)$ because of fork/join dependencies, like in Figure 2). This means that other sound techniques such as [19, 20] that use the reported patterns to confirm deadlocks can also miss simple deadlocks arising because of such patterns.

Example 3. In the trace ρ_2 from Figure 2, $\mathsf{lw}_{\rho_2}(e_6) = \mathsf{lw}_{\rho_2}(e_{10}) = e_1$. Let us now consider the trace $\rho_2^{\mathsf{CR}} = e_1 e_2 e_3 e_9 e_{10}$, where e_i refers to the ith event of ρ_2. ρ_2^{CR} is a correct reordering of the trace ρ_2. This is because it respects the thread-order $\leq_{\mathsf{TO}}^{\rho_2}$ of the trace ρ_2 and the last write event corresponding to the $\mathtt{w}(x)$ in t_3 in ρ_2^{CR} is the same as that in ρ_2. Next, this trace also has a deadlock pattern of size 2, namely, $\langle e_2, e_4, e_9, e_{11}\rangle$. Further, ρ_2 also has a predictable deadlock of size 2 because of the correct reordering ρ_2^{CR} that witnesses the deadlock. The events e_3 and e_5 in ρ_2^{CR} correspond respectively to the events e_3 and e_{10} in ρ_2. Here, $\ell \in \mathsf{locksHeld}_{\rho_2^{\mathsf{CR}}}(e_3)$, $m \in \mathsf{locksHeld}_{\rho_2^{\mathsf{CR}}}(e_5)$, $e_4 = \langle t_2, \mathtt{acq}(m)\rangle \in \mathsf{next}_{\rho_2}(e_3)$ and $e_{11} = \langle t_3, \mathtt{acq}(\ell)\rangle \in \mathsf{next}_{\rho_2}(e_{10})$.

3 Sound Deadlock Prediction

Partial orders are routinely used to detect predictable data races in executions. The advantage of partial order based dynamic analysis is that they typically have linear time (or at least polynomial time) algorithms. However, they have thus far not been successfully used to soundly predict deadlocks in traces. The reason is because the design of partial order is subtle and needs to capture reasoning about concurrent behavior. We present the first partial order that is conducive to sound deadlock prediction. We define the partial order $\leq_{\mathsf{DCP}}$ in Section 3.1, and state its soundness guarantees (Section 3.2). We illustrate the subtle decisions made in defining $\leq_{\mathsf{DCP}}$ and demonstrate its effectiveness in reasoning about deadlocks through examples in Section 3.3. $\leq_{\mathsf{DCP}}$, we believe, carefully navigates the competing goals of increased predictive power and sound reasoning.

3.1 Deadlock Causal Precedence

We first define a partial order $\leq_{\mathsf{CHB}}$, inspired by the happens-before partial order [24].

Definition 1 (Conflict-HB). For a trace σ, $\leq_{\mathsf{CHB}}^{\sigma}$ (read 'conflict HB') is the smallest partial order on Events_σ such that for any two events $e_1 \leq_{\mathsf{tr}}^{\sigma} e_2$ if either (a) $e_1 \leq_{\mathsf{TO}}^{\sigma} e_2$, or (b) $e_1 \in \mathsf{Releases}_\sigma(\ell)$ and $e_2 \in \mathsf{Acquires}_\sigma(\ell)$ for some lock $\ell \in \mathsf{Locks}_\sigma$, or (c) $e_1 \asymp e_2$, then $e_1 \leq_{\mathsf{CHB}}^{\sigma} e_2$.

In the following definition, the composition of a binary relation $R_1 \subseteq E \times E$ with another binary relation $R_2 \subseteq E \times E$, denoted by $R_1 \circ R_2$ (resp. $R_2 \circ R_1$) is the binary relation $\{(a, c) \mid \exists b \cdot (a, b) \in R_1 \text{ and } (b, c) \in R_2\}$.

Definition 2 (Deadlock Causal Precedence). For a trace σ, $\prec_{\mathsf{DCP}}^{\sigma}$ is the smallest relation such that the following hold.

(a) For $e_1 <_{\mathsf{tr}}^{\sigma} e_2$ such that $e_1 \asymp e_2$, we have $e_1 \prec_{\mathsf{DCP}}^{\sigma} e_2$,

(b) Let C_1 and C_2 be two critical sections on some lock $\ell \in \mathsf{Locks}_\sigma$ such that $\mathtt{rel}(C_1) \leq_{\mathsf{tr}}^{\sigma} \mathtt{acq}(C_2)$ and $\mathtt{rel}(C_1) \not\leq_{\mathsf{TO}}^{\sigma} \mathtt{acq}(C_2)$. Further, let $e_1 \in C_1$ and $e_2 \in C_2$ be events such that $e_1 \prec_{\mathsf{DCP}}^{\sigma} e_2$. Then, $\mathtt{rel}(C_1) \prec_{\mathsf{DCP}}^{\sigma} \mathtt{rel}(C_2)$.

(c) $\prec_{\mathsf{DCP}}^{\sigma}$ is closed under left and right composition with $\leq_{\mathsf{CHB}}^{\sigma}$. That is, $\prec_{\mathsf{DCP}}^{\sigma} \circ \leq_{\mathsf{CHB}}^{\sigma} \subseteq \prec_{\mathsf{DCP}}^{\sigma}$ and $\leq_{\mathsf{CHB}}^{\sigma} \circ \prec_{\mathsf{DCP}}^{\sigma} \subseteq \prec_{\mathsf{DCP}}^{\sigma}$.

	t_1	t_2
1	$\mathtt{acq}(\ell)$	
2	$\mathtt{acq}(m)$	
3	$\mathtt{rel}(m)$	
4	$\mathtt{rel}(\ell)$	
5	$\mathtt{w}(x)$	
6		$\mathtt{r}(x)$
7		$\mathtt{acq}(m)$
8		$\mathtt{acq}(\ell)$
9		$\mathtt{rel}(\ell)$
10		$\mathtt{rel}(m)$

	t_1	t_2
1	$\mathtt{acq}(n)$	
2	$\mathtt{w}(x)$	
3	$\mathtt{acq}(\ell)$	
4	$\mathtt{acq}(m)$	
5	$\mathtt{rel}(m)$	
6	$\mathtt{rel}(\ell)$	
7	$\mathtt{rel}(n)$	
8		$\mathtt{acq}(n)$
9		$\mathtt{r}(x)$
10		$\mathtt{rel}(n)$
11		$\mathtt{acq}(m)$
12		$\mathtt{acq}(\ell)$
13		$\mathtt{rel}(\ell)$
14		$\mathtt{rel}(m)$

Figure 3. (a) Trace ρ_3 on the left; (b) Trace ρ_4 on the right.

We define the partial order $\leq_{\mathsf{DCP}}^{\sigma} = \prec_{\mathsf{DCP}}^{\sigma} \cup \leq_{\mathsf{TO}}^{\sigma}$.

Remark. Definition 2 could have been defined differently without changing its semantics. In rule (c), we could have used HB as opposed to CHB. However, we would then have to define $\prec_{\mathsf{DCP}}$ as the smallest *transitive* relation satisfying rules (a), (b), and (modified) (c), as opposed to simply the smallest relation satisfying the conditions in Definition 2. The vector clock algorithm we will present, follows the current definition closely, and hence the choice.

3.2 Using $\leq_{\mathsf{DCP}}$ For Deadlock Prediction

In order to check if two acquire events e_1, e_2 can participate in a deadlock, we need to check if e_1 and e_2 are *concurrent*, i.e., if there is no causal relationship between e_1 and e_2. We will use $\leq_{\mathsf{DCP}}$ to determine the absence of such a causal dependence and Theorem 1 establishes the soundness of $\leq_{\mathsf{DCP}}$ for the purpose of predicting deadlocks.

Theorem 1. *Let σ be a trace with a deadlock pattern $\langle e, f, e', f'\rangle$ of size 2 such that $f \parallel_{\mathsf{DCP}}^{\sigma} f'$. Then, σ exhibits a predictable deadlock.*

The proof of Theorem 1 exploits the soundness guarantee of the WCP [21] partial order and is presented in Appendix A.

3.3 Illustrative Examples

Let us now illustrate different aspects of Definition 2 and Theorem 1 via some example traces.

Example 4. Figure 3a illustrates the intuition behind rule (a) of $\prec_{\mathsf{DCP}}$ in Definition 2. The trace ρ_3 has a *deadlock pattern* of size 2 (with events e_1, e_2, e_7 and e_8 participating in the pattern). Since e_5 and e_6 are ordered, any correct reordering of ρ_3 is a prefix of ρ_3. Hence, ρ_3 does not have a predictable deadlock. $\prec_{\mathsf{DCP}}^{\rho_3}$ ensures soundness by ordering $e_5 \prec_{\mathsf{DCP}}^{\sigma} e_6$ (rule (a)) and this implies that $e_2 \leq_{\mathsf{DCP}}^{\sigma} e_8$ because of composition with $\leq_{\mathsf{CHB}}^{\rho_3}$. As a result, Theorem 1 does not report any predictable deadlock in ρ_3.

Based on Example 4 above, one might naïvely conclude that simply ordering all pairs of conflicting events is enough to ensure soundness. The next example illustrates why this reasoning is flawed.

Example 5. Consider the trace ρ_4 in Figure 3b. In this trace, the tuple $\langle e_3, e_4, e_{11}, e_{12}\rangle$ constitute a deadlock pattern, and Goodlock-based algorithms report this pattern as a potential deadlock. Further, naïvely ordering conflicting events will only order e_2 before e_9 and thus the two nesting sequences remain unordered. However, the deadlock pattern does not constitute a real predictable deadlock. To understand why, observe that any correct reordering ρ'_4 of ρ_4 that schedules the deadlock will execute e_1, e_2, e_8, e_9 and e_{10}. Further, in order to preserve the last write of e_9, it must also ensure $e_2 \leq^{\rho'_4}_{\mathsf{tr}} e_9$. Finally, to preserve lock semantics, e_7 must be executed in ρ'_4 before e_8. Hence, any such correct reordering ρ'_4 of ρ_4 will completely execute the nested critical section e_3, e_4, e_5, e_6 and then proceed executing the nested section $e_{11}, e_{12}, e_{13}, e_{14}$, thereby not resulting in a deadlock. The rule (b) of $\prec_{\mathsf{DCP}}$ in Definition 2, in fact, orders $e_7 \prec^{\rho_4}_{\mathsf{DCP}} e_{10}$ because of the rule (a) ordering $e_2 \prec^{\rho_4}_{\mathsf{DCP}} e_9$. As a result we have $e_4 \leq^{\rho_4}_{\mathsf{DCP}} e_{12}$. Therefore, Theorem 1 does not report any predictable deadlock in ρ_4.

While DCP is inspired from prior work on race detection, DCP is subtly different from these in important ways. One of these differences is the statement of rule (b) in the definition of $\prec_{\mathsf{DCP}}$. In particular, for two critical sections C_1 and C_2 on some common lock ℓ which contain events $e_1 \in C_1$ and $e_2 \in C_2$ already ordered by $\prec_{\mathsf{DCP}}$, the releases $r_1 = \mathtt{rel}(C_1)$ and $r_2 = \mathtt{rel}(C_2)$ are not ordered by $\prec_{\mathsf{DCP}}$ if $r_1 \leq_{\mathsf{TO}} r_2$ (but of course are ordered by $\leq_{\mathsf{DCP}} = \prec_{\mathsf{DCP}} \cup \leq_{\mathsf{TO}}$). This is in sharp contrast with both CP [30] and WCP [21] where r_1 and r_2 are ordered by the irreflexive versions of the partial orders CP and WCP. The following example demonstrates the importance of this difference from earlier partial orders.

Example 6. Consider the trace ρ_5 in Figure 4. The only deadlock pattern here is $\langle e_1, e_2, e_{17}, e_{18}\rangle$. In this trace, we have $e_8 \prec^{\rho_5}_{\mathsf{DCP}} e_9$ and $e_{14} \prec^{\rho_5}_{\mathsf{DCP}} e_{15}$, giving us $e_8 \prec^{\rho_5}_{\mathsf{DCP}} e_{15}$ after composing with $e_9 \leq^{\rho_5}_{\mathsf{CHB}} e_{14}$. This means that the two critical sections $C_1 = \mathsf{CS}_{\rho_5}(e_7)$ and $C_2 = \mathsf{CS}_{\rho_5}(e_{13})$ (on the same lock n) contain events e_8 and e_{15} respectively ordered by $\prec^{\rho_5}_{\mathsf{DCP}}$. In the absence of the additional condition $(\mathtt{rel}(C_1) \not\leq^{\mathsf{acq}}_{\mathsf{TO}} (C_2))$ in rule (b), we would have ordered $e_{12} \prec^{\rho_5}_{\mathsf{DCP}} e_{16}$. This in turn, composes with the CHB edge $e_2 \leq^{\rho_5}_{\mathsf{CHB}} e_6 \leq^{\rho_5}_{\mathsf{CHB}} e_{10} \leq^{\rho_5}_{\mathsf{CHB}} e_{12} \prec^{\rho_5}_{\mathsf{DCP}} e_{16} \leq^{\rho_5}_{\mathsf{CHB}} e_{18}$ (due to rule (c)) giving us $e_2 \prec^{\rho_5}_{\mathsf{DCP}} e_{18}$. However, the additional condition $(\mathtt{rel}(C_1) \not\leq^{\mathsf{acq}}_{\mathsf{TO}} (C_2))$ ensures that $e_2 \,||^{\rho_5}_{\mathsf{DCP}}\, e_{18}$. Indeed, the trace $\rho_5^{\mathsf{CR}} = e_7e_8e_9e_{10}e_{11}e_{12}e_{13}e_{14}e_{15}e_{16}e_1e_{17}$, is a correct reordering of ρ_5 and witnesses a deadlock. Hence, Theorem 1 correctly identifies a predictable deadlock in ρ_5.

	t_1	t_2	t_3
1	acq(ℓ)		
2	acq(m)		
3	rel(m)		
4	rel(ℓ)		
5	acq(k)		
6	rel(k)		
7		acq(n)	
8		w(x)	
9			r(x)
10		acq(k)	
11		rel(k)	
12		rel(n)	
13		acq(n)	
14			w(y)
15		r(y)	
16		rel(n)	
17		acq(m)	
18		acq(ℓ)	
19		rel(ℓ)	
20		rel(m)	

Figure 4. Trace ρ_5.

Observe that $\leq_{\mathsf{TO}}$ as defined in Section 2 is more subtle than intra-thread order (also popularly called program order in the literature). Our definition incorporates dependencies due to `fork` and `join` events. Notice, however, that one could, alternatively, define $\leq_{\mathsf{TO}}$ to be only program order and instead incorporate fork and join dependencies in the partial order $\prec_{\mathsf{DCP}}$ as base rules (like rule (a)). In the following, we will demonstrate how our choice of defining $\leq_{\mathsf{TO}}$ allows us to detect deadlocks that would have been missed otherwise, while remaining sound.

	t_1	t_2	t_3
1	fork(t_2)		
2		acq(ℓ)	
3		acq(m)	
4		rel(m)	
5		rel(ℓ)	
6		fork(t_3)	
7			acq(m)
8			acq(ℓ)
9			rel(ℓ)
10			rel(m)

Figure 5. Trace ρ_6.

Example 7. Consider the trace ρ_6 in Figure 5, with a deadlock pattern $\langle e_2, e_3, e_7, e_8\rangle$. However, this is not a predictable deadlock because any event of t_3 can only be executed after e_6, which in turn is only after the nesting sequence e_2, e_3, e_4, e_5. Now let us see the corresponding DCP reasoning. The events e_3 and e_8 are unordered by $\prec^{\rho_6}_{\mathsf{DCP}}$ but are ordered by $\leq^{\rho_6}_{\mathsf{DCP}} = \prec^{\rho_6}_{\mathsf{DCP}} \cup \leq^{\rho_6}_{\mathsf{TO}}$. Therefore, Theorem 1 reports no predictable deadlock.

	t_1	t_2	t_3
1	acq(ℓ)		
2	acq(m)		
3	rel(m)		
4	rel(ℓ)		
5	fork(t_2)		
6		acq(m)	
7		rel(m)	
8			acq(m)
9			acq(ℓ)
10			rel(ℓ)
11			rel(m)

Figure 6. Trace ρ_7.

Example 8. The trace ρ_7 in Figure 6 has a deadlock pattern of size 2, namely, $\langle e_1, e_2, e_8, e_9 \rangle$. Further, the correct reordering $\rho_7^{\mathsf{CR}} = e_1 e_8$ of ρ_7 also witnesses the deadlock. If $\prec_{\mathsf{DCP}}$ included fork/join dependencies, then we would have had $e_5 \prec^{\rho_7}_{\mathsf{DCP}} e_6$, $e_2 \leq^{\rho_7}_{\mathsf{CHB}} e_5$ and $e_6 \leq^{\rho_7}_{\mathsf{CHB}} e_9$ thus giving $e_2 \prec^{\rho_7}_{\mathsf{DCP}} e_9$. That is, including the fork dependency in $\prec_{\mathsf{DCP}}$ forces the two acquires e_2 and e_9 to get ordered, thereby missing the predictable deadlock witnessed in ρ_7^{CR}. This justifies our choice of not including such dependencies in $\prec_{\mathsf{DCP}}$. There are no conflicting events in ρ_7 and thus no two events are ordered by $\prec^{\rho_7}_{\mathsf{DCP}}$. This in turn means that e_2 and e_9 remain unordered according to $\leq^{\rho_7}_{\mathsf{DCP}}$, and Theorem 1 correctly declares a predictable deadlock in ρ_7.

Example 9. Let us again consider the trace ρ_2 from Figure 2. This trace has a deadlock pattern $\langle e_2, e_4, e_9, e_{11} \rangle$. First observe that while $e_2 \leq^{\rho_2}_{\mathsf{TO}} e_4$, $\mathsf{thr}(e_2) \neq \mathsf{thr}(e_4)$. As described in Section 2, existing deadlock detection tools, both sound [20] or unsound [3] that rely on traditional Goodlock style construction of a lock graph do not identify a deadlock pattern here. On the other hand, our definition of $\leq_{\mathsf{TO}}$, and the implied definitions of locks held and critical sections handle this (see Section 2). In fact, this trace has a predictable deadlock (witnessed by the correct reordering ρ_2^{CR} in Example 3). This is captured by DCP as follows. The only $\prec_{\mathsf{DCP}}$ orders in this trace are $e_1 \prec^{\rho_2}_{\mathsf{DCP}} e_6$ and $e_1 \prec^{\rho_2}_{\mathsf{DCP}} e_{10}$ (rule (a)), and those due to composition with $\leq^{\rho_2}_{\mathsf{CHB}}$ (rule (c)): $e_1 \prec^{\rho_2}_{\mathsf{DCP}} e_7$, $e_1 \prec^{\rho_2}_{\mathsf{DCP}} e_8$, $e_1 \prec^{\rho_2}_{\mathsf{DCP}} e_{11}$ and $e_1 \prec^{\rho_2}_{\mathsf{DCP}} e_{12}$. Hence, we have $e_4 \,||^{\rho_2}_{\mathsf{DCP}}\, e_{11}$ and according to Theorem 1, ρ_2 has a predictable deadlock.

Let us now illustrate some important aspects of Theorem 1.

Example 10. Consider the trace ρ_8 in Figure 7a. This trace has a deadlock pattern but no predictable deadlock. Any reordering of this trace, that respects intra-thread order and executes both e_1 and e_6 (without executing the matching releases e_5 and e_{10}) cannot proceed in the thread t_1 beyond e_1. This means that the write event e_3 on x will not be executed and thus the last-write event for the read event e_7 will not be e_3. DCP, on the other hand, correctly orders $e_3 \prec^{\rho_8}_{\mathsf{DCP}} e_7$ (rule (a)). This gives $e_2 \leq^{\rho_8}_{\mathsf{DCP}} e_8$ (due to rule (b)). This means Theorem 1 does not report a predictable deadlock in ρ_8.

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

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

Figure 7. (a) Trace ρ_8 on the left; and (b) Trace ρ_9 on the right. Trace ρ_8 does not have a predictable deadlock while trace ρ_9 has a predictable deadlock.

Now consider trace ρ_9, which is a close variant of ρ_8. Here, while DCP orders $e_1 \leq^{\rho_9}_{\mathsf{DCP}} e_7$ and $e_1 \leq^{\rho_9}_{\mathsf{DCP}} e_8$, it does not order the acquire events of the inner critical sections $\mathsf{CS}_{\rho_9}(e_3)$ and $\mathsf{CS}_{\rho_9}(e_7)$ on locks m and ℓ respectively. As a result, Theorem 1 declares that ρ_9 has a predictable deadlock. In fact, the correct reordering $e_1 e_2 e_6 e_7$ of ρ_9 witnesses the deadlock.

4 Algorithm for Deadlock Prediction

In this section we describe an algorithm for predicting deadlocks using our partial order $\leq_{\mathsf{DCP}}$ and Theorem 1. Our algorithm (a) identifies deadlock patterns of the form $\langle e_1, e'_1, e_2, e'_2 \rangle$ in the execution, and, (b) checks if the inner acquire events e'_1 and e'_2 are unordered by $\leq_{\mathsf{DCP}}$. The algorithm uses *vector clocks* to check if two acquire events are unordered by $\leq_{\mathsf{DCP}}$. This vector-clock algorithm is similar in spirit to the vector clock algorithm for detecting data races in executions using the WCP [21] partial order. Our algorithm runs in a streaming online fashion, processing each event as it is observed in the trace, and performing necessary vector clock updates and additional book-keeping to track deadlock patterns. In the following, we give a brief overview of the algorithm, with some details defered to Appendix B. Our notations for vector clocks and associated operations are derived from [21, 30] and can also be found in Appendix B.

The intuition behind the vector clock algorithm is to assign a timestamp D_e to every event e in the trace, such that the partial order imposed by the assigned timestamps is isomorphic to the $\leq_{\mathsf{DCP}}$ partial order. This is formalized in Theorem 2.

The algorithm maintains several vector clocks in its state, updating different vector clocks at each event e, depending

Algorithm 1: *Computing the* $\leq_{\mathsf{DCP}}$ *timestamps for different events*

```
procedure Initialization
    for $t \in$ Threads do
        $\mathbb{P}_t := \bot$;
        $\mathbb{H}_t := \bot[1/t]$;
        $\mathbb{T}_t := \bot[1/t]$;
        $\mathbb{D}_t := \bot[1/t]$;
        for $\ell \in$ Locks do
            $Acq_{t,\ell} := \varnothing$;
            $Rel_{t,\ell} := \varnothing$;
        for $x \in$ Vars do
            $\mathbb{H}^{\mathsf{r}}_{t,x} := \bot$;
            $\mathbb{H}^{\mathsf{w}}_{t,x} := \bot$;
            $\mathbb{T}^{\mathsf{r}}_{t,x} := \bot$;
            $\mathbb{T}^{\mathsf{w}}_{t,x} := \bot$
    for $\ell \in$ Locks do
        $\mathbb{P}_\ell := \bot$;
        $\mathbb{H}_\ell := \bot$;

procedure acquire($t$, $\ell$)
    $\mathbb{H}_t := \mathbb{H}_t \sqcup \mathbb{H}_\ell$;
    $\mathbb{P}_t := \mathbb{P}_t \sqcup \mathbb{P}_\ell$;
    $\mathbb{D}_t := \mathbb{D}_t \sqcup \mathbb{P}_\ell$;
    for $t' \in$ Threads do
        $Acq_{t',\ell} \cdot$ Enqueue($\langle \mathbb{T}_t, \mathbb{D}_t \rangle$)

procedure release($t$, $\ell$)
    while $Acq_{t,\ell} \cdot$ nonempty() do
        $\langle T', D' \rangle := Acq_{t,\ell} \cdot$ Front()
        if $D' \not\sqsubseteq \mathbb{P}_t$ then
            break;
        $H' := Rel_{t,\ell} \cdot$ Front()
        if $T' \not\sqsubseteq \mathbb{T}_t$ then
            $\mathbb{P}_t := \mathbb{P}_t \sqcup H'$;
            $\mathbb{D}_t := \mathbb{D}_t \sqcup H'$;
        $Acq_{t,\ell} \cdot$ Dequeue();
        $Rel_{t,\ell} \cdot$ Dequeue();
    $\mathbb{H}_\ell := \mathbb{H}_t$; $\mathbb{P}_\ell := \mathbb{P}_t$;
    for $t' \in$ Threads do
        $Rel_{t',\ell} \cdot$ Enqueue($\mathbb{H}_t$)

procedure read($t$, $x$)
    for $t' \in$ Threads do
        if $\mathbb{T}^{\mathsf{w}}_{t',x} \not\sqsubseteq \mathbb{T}_t$ then
            $\mathbb{P}_t := \mathbb{P}_t \sqcup \mathbb{H}^{\mathsf{w}}_{t',x}$;
            $\mathbb{H}_t := \mathbb{H}_t \sqcup \mathbb{H}^{\mathsf{w}}_{t',x}$;
            $\mathbb{D}_t := \mathbb{D}_t \sqcup \mathbb{H}^{\mathsf{w}}_{t',x}$;
    $\mathbb{H}^{\mathsf{r}}_{t,x} := \mathbb{H}_t$; $\mathbb{T}^{\mathsf{r}}_{t,x} := \mathbb{T}_t$;

procedure write($t$, $x$)
    for $t' \in$ Threads do
        if $\mathbb{T}^{\mathsf{w}}_{t',x} \not\sqsubseteq \mathbb{T}_t$ then
            $\mathbb{P}_t := \mathbb{P}_t \sqcup \mathbb{H}^{\mathsf{w}}_{t',x}$;
            $\mathbb{H}_t := \mathbb{H}_t \sqcup \mathbb{H}^{\mathsf{w}}_{t',x}$;
            $\mathbb{D}_t := \mathbb{D}_t \sqcup \mathbb{H}^{\mathsf{w}}_{t',x}$;
        if $\mathbb{T}^{\mathsf{r}}_{t',x} \not\sqsubseteq \mathbb{T}_t$ then
            $\mathbb{P}_t := \mathbb{P}_t \sqcup \mathbb{H}^{\mathsf{r}}_{t',x}$;
            $\mathbb{H}_t := \mathbb{H}_t \sqcup \mathbb{H}^{\mathsf{r}}_{t',x}$;
            $\mathbb{D}_t := \mathbb{D}_t \sqcup \mathbb{H}^{\mathsf{r}}_{t',x}$;
    $\mathbb{H}^{\mathsf{w}}_x := \mathbb{H}_t$; $\mathbb{T}^{\mathsf{w}}_x := \mathbb{T}_t$;

procedure fork($t$, $u$)
    $\mathbb{H}_u := \mathbb{H}_t[1/u]$;
    $\mathbb{T}_u := \mathbb{T}_t[1/u]$;
    $\mathbb{D}_u := \mathbb{D}_t[1/u]$;
    $\mathbb{P}_u := \mathbb{P}_t$;

procedure join($t$, $u$)
    $\mathbb{H}_t := \mathbb{H}_t \sqcup \mathbb{H}_u$;
    $\mathbb{T}_t := \mathbb{T}_t \sqcup \mathbb{T}_u$;
    $\mathbb{P}_t := \mathbb{P}_t \sqcup \mathbb{P}_u$;
```

upon the operation op(e). Algorithm 1 describes these updates. We briefly describe the different components of the algorithm below.

Vector Clocks and FIFO Queues. The algorithm maintains vector clocks $\mathbb{P}_t, \mathbb{T}_t, \mathbb{H}_t$ and $\mathbb{D}_t$ for each thread t, vector clocks $\mathbb{P}_\ell$ and $\mathbb{H}_\ell$ for each lock ℓ, vector clocks $\mathbb{T}^{\mathsf{r}}_{t,x}$, $\mathbb{T}^{\mathsf{w}}_{t,x}$, $\mathbb{H}^{\mathsf{r}}_{t,x}$ and $\mathbb{H}^{\mathsf{w}}_{t,x}$ for each pair (t, x) of thread t and memory location x. Additionally, for every thread t and lock ℓ, the algorithm maintains FIFO queues $Rel_{t,\ell}$ and $Acq_{t,\ell}$ that respectively store vector times and pairs of vector times.

Broadly, the algorithm simultaneously maintains the orders $\leq_{\mathsf{TO}}$, $\leq_{\mathsf{CHB}}$, $\prec_{\mathsf{DCP}}$ and $\leq_{\mathsf{DCP}}$ and uses different clocks and queues for this purpose. The clocks $\mathbb{T}_t$ correspond to the partial order $\leq_{\mathsf{TO}}$. Let us denote by T_e the value of the clock $\mathbb{T}_{\mathsf{thr}(e)}$ right after processing the event e according to Algorithm 1. We say that T_e is the $\leq_{\mathsf{TO}}$ timestamp of e—for events $f \leq_{\mathsf{tr}} f'$, $T_f \sqsubseteq T_{f'}$ iff $f \leq_{\mathsf{TO}} f'$. Similarly, the clocks $\mathbb{D}_t$ and $\mathbb{H}_t$ respectively correspond to the partial orders $\leq_{\mathsf{DCP}}$ and $\leq_{\mathsf{CHB}}$. The value of the clock $\mathbb{P}_{\mathsf{thr}(e)}$ after processing an event e, denoted P_e, can be used to identify $\prec_{\mathsf{DCP}}$ predecessors of e. That is, for events f and f' (with $f \leq_{\mathsf{tr}} f'$), $f \prec_{\mathsf{DCP}} f'$ iff $D_f \sqsubseteq P_{f'}$. The clocks $\mathbb{T}^{\mathsf{r}}_{t,x}$ and $\mathbb{T}^{\mathsf{w}}_{t,x}$ store the $\leq_{\mathsf{TO}}$ timestamps $T_{e^{\mathsf{r}}_{t,x}}$ and $T_{e^{\mathsf{w}}_{t,x}}$ of the last events $e^{\mathsf{r}}_{t,x}$ and $e^{\mathsf{w}}_{t,x}$ that respectively read and write to x in thread t in the trace seen so far. Similarly, the clocks $\mathbb{H}^{\mathsf{r}}_{t,x}$ and $\mathbb{H}^{\mathsf{w}}_{t,x}$ store the $\leq_{\mathsf{CHB}}$ timestamps of the last events of the form $\langle t, \mathsf{r}(x) \rangle$ and $\langle t, \mathsf{w}(x) \rangle$ in the trace so far. The clocks $\mathbb{P}_\ell$ and $\mathbb{H}_\ell$ store the $\prec_{\mathsf{DCP}}$ and $\leq_{\mathsf{CHB}}$ timestamps of the last release event on lock ℓ.

The FIFO queue $Acq_{t,\ell}$ maintains pairs $\langle T, D \rangle$ of $\leq_{\mathsf{TO}}$ and $\leq_{\mathsf{DCP}}$ timestamps of some of the acquire events for lock ℓ and the queue $Rel_{t,\ell}$ stores the $\leq_{\mathsf{CHB}}$ timestamps of the corresponding release events. These queues ensure that the corresponding timestamps correctly mimic the $\prec_{\mathsf{DCP}}$ order, so as to incorporate rule (b) of $\prec_{\mathsf{DCP}}$ (Definition 2). According to this rule, if we observe a release event $e = \langle t, \mathtt{rel}(\ell) \rangle$, then for all earlier release events $e' \in \mathsf{Releases}_\sigma(\ell)$ $\not\leq^\sigma_{\mathsf{TO}}$ e, we must have $e' \prec^\sigma_{\mathsf{DCP}} e$ whenever $\mathsf{match}_\sigma(e') \prec^\sigma_{\mathsf{DCP}} e$. In terms of timestamps, this means that if $D_{\mathsf{match}_\sigma(e')} \sqsubseteq P_e$, then both the $\prec_{\mathsf{DCP}}$ and $\leq_{\mathsf{DCP}}$ timestamps of e need to be updated to additionally ensure $D_{e'} \sqsubseteq P_e$. The queue $Acq_{t,\ell}$ stores the necessary information about all earlier ℓ acquire events for the check "$D_{\mathsf{match}_\sigma(e')} \sqsubseteq P_{e'}$" above, and the corresponding release timestamps are stored in $Rel_{t,\ell}$ and can be used to correctly update $\leq_{\mathsf{DCP}}$ and $\prec_{\mathsf{DCP}}$ timestamps for e. The choice of the queue data structure here is motivated by the following observation. Once we update $\mathbb{P}_t$ to ensure $D_{e'} \sqsubseteq P_e$, we no longer need the information about e' because the timestamp P_f of each later event f performed by $\mathsf{thr}(e)$ satisfies $D_{e'} \sqsubseteq P_e \sqsubseteq P_f$. Thus, both e' and $\mathsf{match}_\sigma(e')$ can safely be discarded from the set of events we wish to track. In fact, since rule (c) also ensures that for every ℓ-release event $e'' \leq^\sigma_{\mathsf{tr}} e'$, we have $C_{e'} \sqsubseteq P_e \implies C_{e''} \sqsubseteq P_e$, we can also discard all release events on lock ℓ whose timestamps were pushed before that of e' in $Rel_{t,\ell}$.

Updates. After processing an event $e = \langle t, op\rangle$, the local components of the clocks $\mathbb{D}_t, \mathbb{T}_t$ and $\mathbb{H}_t$ are incremented after performing each event (i.e., we assign "$\mathbb{T}_t := \mathbb{T}[\mathbb{T}_t(t) + 1/t]$", "$\mathbb{D}_t := \mathbb{D}[\mathbb{D}_t(t) + 1/t]$", and "$\mathbb{H}_t := \mathbb{H}[\mathbb{H}_t(t) + 1/t]$"), to ensure consistency of timestamps. As an example, let e, e', f be events such that $e' \in \mathsf{next}_\sigma(e)$ and $\mathsf{thr}(e) = \mathsf{thr}(e')$, $e \leq^\sigma_{\mathsf{DCP}} f$ but $e' \not\leq^\sigma_{\mathsf{DCP}} f$, then this clock-increment ensures that the timestamps also obey $D_e \sqsubseteq D_f$ but $D_{e'} \not\sqsubseteq D_f$. Since these assignments are common to all events, we do not explicitly include them in Algorithm 1.

At an acquire event $e = \langle t, \mathsf{acq}(\ell)\rangle$, we update $\mathbb{H}_t$ with the $\leq_{\mathsf{CHB}}$ timestamp of the latest release on ℓ (stored in $\mathbb{H}_\ell$). Similarly, to account for rule (c) of $\prec_{\mathsf{DCP}}$, we update $\mathbb{P}_t$ using the clock $\mathbb{P}_\ell$ storing the $\prec_{\mathsf{DCP}}$ timestamp of the latest ℓ-release. The reasoning behind most other updates in the algorithm follows similarly by analyzing the different rules of $\prec_{\mathsf{DCP}}$ and $\leq_{\mathsf{CHB}}$. The check $\mathbb{T}^{\mathsf{w}}_{t',x}$ on line 36 performed at an event $e = \langle t, \mathsf{r}(x)\rangle$ ensures that we only consider earlier write events e' on x that conflict with e (and thus $e' \not\leq^\sigma_{\mathsf{TO}} e$). Similar reasoning applies to the checks on lines 27, 42 and 46.

Let us now state the key observation about the timestamps that the algorithm computes.

Theorem 2. *Let σ be a trace with $e, e' \in \mathsf{Events}_\sigma$ such that $e \leq^\sigma_{\mathsf{tr}} e'$. Let D_e and $D_{e'}$ be the $\leq^\sigma_{\mathsf{DCP}}$ timestamps of respectively e and e' assigned by Algorithm 1. We have, $e \leq^\sigma_{\mathsf{DCP}} e'$ iff $D_e \sqsubseteq D_{e'}$.*

Let us now state the running time and space usage of Algorithm 1. Let n be the number of events and let T, L, V be the number of threads, locks and memory locations in the trace. The following theorem states the asymptotic time and space complexity for Algorithm 1, assuming constant time for arithmetic operations.

Theorem 3. *Algorithm 1 uses $O(nT^2)$ time and $O(T(L + TV) + nT)$ space.*

4.1 Lower Bounds

Algorithm 1 runs in linear time and uses linear space (Theorem 3). This is optimal — any linear time algorithm computing $\leq_{\mathsf{DCP}}$ must use linear space. The proof of this result is very similar to the proof that establishes the linear space lower bound to compute WCP [22]; we therefore skip this proof.

While the observations in the previous paragraph establish the optimality of our algorithm as a $\leq_{\mathsf{DCP}}$-based deadlock prediction algorithm, it doesn't say anything about the hardness of the deadlock prediction problem itself. We now establish lower bounds for the deadlock prediction problem. Recall that the deadlock prediction problem, DeadlockPred, is the following problem: Given a trace σ, determine if σ has a predictable deadlock. We can prove the following lower bound for *any* algorithm for this problem.

Theorem 4. *Let A be an algorithm for* DeadlockPred *running in time $T(n)$ and using space $S(n)$. Then $T(n)S(n) = \Omega(n^2)$.*

The proof of Theorem 4 is postponed to Appendix C. An immediate corollary of Theorem 4 is that the space requirements of any linear time algorithm for DeadlockPred is at least linear.

Corollary 5. *If A is a linear time algorithm for* DeadlockPred *using space $S(n)$ then $S(n) = \Omega(n)$.*

5 Incorporating Control and Data Flow

So far in this article, the notion of correct reordering has been a bit conservative. In particular, each correct reordering σ' of σ ensured that every read e in σ' observes the same value as in σ (by forcing $\mathsf{lw}_\sigma(e) = \mathsf{lw}_{\sigma'}(e)$). Such a restriction ensured that the values of all the expressions in the branch statements (conditionals) executed in the program are preserved and thus the same control flow is executed in each of the correct reorderings. However, not every value read may influence the decision at a branch. It may thus be possible to infer reorderings that do not preserve the last writes corresponding to some of the read events, thereby allowing us to be less conservative. Thus, if we additionally track *branch* events in our traces and additionally identify such read events that do not affect *any* of these branch events, we can potentially identify more deadlocks, in previous unexplored reorderings that no more restrain the last write before such read events.

In this section, we explore how such fine grained information in traces can be exploited for enhancing the power of of our deadlock prediction algorithm. The following example provides a good illustration.

Example 11. Consider programs P_1 (Figure 8a) and P_2 (Figure 8c). Let us consider the executions of these programs obtained when the scheduler first schedules thread T1 followed by thread T2. If, like before, we do not track any branch events, both the executions thus obtained are the trace ρ_3 shown in Figure 3a. In this case, all correct reorderings of ρ_3 are prefixes of ρ_3, and thus ρ_3 has no predictable deadlock. However, if we decide to trace *branch* events, P_1 generates trace ρ_{10} Figure 8b, and P_2 generates ρ_3 from Figure 3a; notice that the only difference between their executions is the presence of the branch event in ρ_{10}. In this case, the read event e_6 in ρ_{10} affects the branch event e_7, and thus the only

```
public class TestBranch extends Thread {
  public static Object L1 = new Object();
  public static Object L2 = new Object();
  public static int x;
  public static void main (String [] args)
    throws Exception{
    x := 0
    new T1().start();
    new T2().start();
  }
  static class T1 extends Thread {
    public void run () {
      synchronized (L1) {synchronized (L2) {}}
      x := 42;
    }
  }
  static class T2 extends Thread {
    public void run () {
      if(x == 42){
        synchronized (L2) {synchronized (L1) {}}
      }
    }
  }
}
```

	t_1	t_2
1	acq(ℓ)	
2	acq(m)	
3	rel(m)	
4	rel(ℓ)	
5	w(x)	
6		r(x)
7		branch
8		acq(m)
9		acq(ℓ)
10		rel(ℓ)
11		rel(m)

```
public class TestNoBranch extends Thread {
  public static Object L1 = new Object();
  public static Object L2 = new Object();
  public static int x;
  public static void main (String [] args)
    throws Exception{
    x := 0
    new T1().start();
    new T2().start();
  }
  static class T1 extends Thread {
    public void run () {
      synchronized (L1) {synchronized (L2) {}}
      x := 42;
    }
  }
  static class T2 extends Thread {
    public void run () {
      System.out.println("x = " + x);
      synchronized (L2) {synchronized (L1) {}}
    }
  }
}
```

Figure 8. Programs (a) P_1 on the left and (c) P_2 on the right. In the middle (b) shows a trace ρ_{10} generated by P_1. A trace of program P_2 is shown in Figure 3a. Program P_1 does not have a predictable deadlock, while P_2 does.

correct reorderings that we can infer are prefixes of ρ_{10}. However, since there are no branch events in ρ_3 (even though we decided to track all branch events), the value that the read event e_6 reads does not affect the executability of any event and thus we can infer the trace $\rho_3^{\mathsf{CR}} = e_1 e_6 e_7$ as a correct reordering of ρ_3 (even though e_6 no longer sees the same last write), and thus we can conclude that ρ_3 has a predictable deadlock (in the case when we decide to track branch events).

Let us now formalize some ideas presented above. The first step towards this is to formulate a more general notion of correct reordering that will be parametric on a *data flow predicate* DF [2]. For an event e of σ, $\mathsf{DF}_\sigma(e) \subseteq \mathsf{Reads}_\sigma$ is the set of read events that must see the same value in any correct reordering, for event e to happen. Armed with such a predicate, correct reordering can be relaxed as follows.

Definition 3 (Correct Reordering with Data Flow)**.** Let σ be a trace and DF_σ be a data flow predicate on the events of σ. A trace σ' is a correct reordering of σ modulo DF_σ if

1. $\mathsf{Events}_{\sigma'} \subseteq \mathsf{Events}_\sigma$,
2. σ' respects $\leq^\sigma_{\mathsf{TO}}$, and
3. for every $e \in \sigma'$ and for every $e' \in \mathsf{DF}_\sigma(e)$, we must have that (a) $e' \in \sigma'$, and (b) $\mathsf{lw}_{\sigma'}(e') = \mathsf{lw}_\sigma(e')$.

Observe that the following simple observation about Definition 3 holds.

Proposition 6. *Suppose* DF^1_σ *and* DF^2_σ *are two data flow predicates such that for every event* e, $\mathsf{DF}^1_\sigma(e) \subseteq \mathsf{DF}^2_\sigma(e)$. *Then if* σ' *is a correct reordering of* σ *with respect to* DF^2_σ *then* σ' *is also a correct reordering of* σ *with respect to* DF^1_σ.

The definition of correct reordering presented in Section 2, is equivalent to Definition 3 for a specific conservative interpretation of the data flow predicate. In Java, branches and writes depend on the values of local registers, which in turn depend on local reads. Thus, we could conservatively assume that a write or a branch event e, depends on all read events performed by the same thread before e. These local reads in turn may depend on the values written last (probably by other threads), and this dependency must be propagated. This leads us to a definition of *relevant reads* for an event e, which we denote by $\mathsf{RelRds}_\sigma(e)$. Before defining this set, let $\mathsf{PriorRds}_\sigma(e) = \{f \in \mathsf{Reads}_\sigma \mid f \leq^\sigma_{\mathsf{tr}} e \text{ and } \mathsf{thr}(f) = \mathsf{thr}(e)\}$. Now, $\mathsf{RelRds}_\sigma(e)$ is the smallest set such that (a) $\mathsf{PriorRds}_\sigma(e) \subseteq \mathsf{RelRds}_\sigma(e)$, and (b) for all $f \in \mathsf{RelRds}_\sigma(e)$, $\mathsf{PriorRds}_\sigma(\mathsf{lw}_\sigma(f)) \subseteq \mathsf{RelRds}_\sigma(e)$. Consider the data flow predicate given by $\mathsf{DF}^\top_\sigma(e) = \mathsf{RelRds}_\sigma(e)$ for all events e. It is easy to see that correct reorderings with respect to $\mathsf{DF}^\top_\sigma$ is the same as the definition of correct reordering given in Section 2.

We can relax the data flow predicate, and thereby the definition of correct reordering, while at the same time ensuring that our predictions are sound. This requires that a trace additionally have *branch* events. Thus, a trace may contain events of the form $\langle t, \mathsf{branch}\rangle$, which indicates that thread t performed a branch-event. The set of branch events in σ will be denoted as $\mathsf{Branches}_\sigma$. The main reason to ensure that reads see the same value is to ensure that branch events are evaluated in the same manner. Therefore, we could consider a data flow predicate that only demands that the relevant reads of branch events be preserved.

$$\mathsf{DF}^{\mathsf{br}}_\sigma(e) = \begin{cases} \mathsf{RelRds}_\sigma(e) & \text{if } e \in \mathsf{Branches}_\sigma \\ \varnothing & \text{otherwise} \end{cases}$$

Notice that thanks to Proposition 6, we can conclude that if σ' is a correct reordering with respect to $\mathsf{DF}^\top_\sigma$ then σ' is also a correct reordering with respect to $\mathsf{DF}^{\mathsf{br}}_\sigma$.

Example 12. Let us reconsider the programs P_1 and P_2, the corresponding traces ρ_3 and ρ_{10} and their reorderings modulo the two data flow predicates $\mathsf{DF}^\top$ and $\mathsf{DF}^{\mathsf{br}}$. The only correct reorderings of ρ_{10} modulo $\mathsf{DF}^{\mathsf{br}}$ are prefixes of ρ_{10}.

[2]Strictly, speaking this will not be a predicate i.e., boolean valued. Nevertheless, we will call it so.

For ρ_3, the only correct reorderings of ρ_3 modulo $\mathsf{DF}^\top$ are prefixes of ρ_3. On the other hand, the trace $e_1 e_6 e_7$ is also correct reordering of ρ_3 modulo $\mathsf{DF}^{\mathsf{br}}$.

Central to exploiting the new fine grained correct reordering definition, is to realize that the data flow predicate $\mathsf{DF}^{\mathsf{br}}$ can be used to identify data access events that will not influence any branch event in σ, and therefore can be ignored. The set of important data access events include all the reads that are relevant to any branch-event *and* all the write events on variables x that have a relevant read event. The reason we include *all* write events, as opposed to just those that actually influence a branch, is because the positioning of all these write events is important — we need to ensure that none of these write events (even those that are never read) ever interfere between a read-event and its last write event. Taking RelRds_σ to be $\cup_{e \in \mathsf{Branches}_\sigma} \mathsf{RelRds}_\sigma(e)$, the set of all *relevant access events*, RelAcc_σ, is given by

$$\mathsf{RelAcc}_\sigma = \mathsf{RelRds}_\sigma \cup \bigcup_{\substack{x \in \mathsf{Vars}_\sigma \\ \exists e \in \mathsf{Reads}_\sigma(x) \cap \mathsf{RelRds}_\sigma}} \mathsf{Writes}_\sigma(x).$$

Any data access event that that is not relevant is *irrelevant*. Thus, $\mathsf{IrrAcc}_\sigma = \mathsf{Accesses}_\sigma \setminus \mathsf{RelAcc}_\sigma$.

We need one more definition before describing how to predict deadlocks modulo $\mathsf{DF}^{\mathsf{br}}$. For a set of events $E \subseteq \mathsf{Events}_\sigma$, $\mathsf{filter}(\sigma, E)$ is the sequence obtained by ignoring the events in E, i.e., it is a projection of σ on the set $\mathsf{Events}_\sigma \setminus E$. Notice, that $\mathsf{filter}(\sigma, \mathsf{IrrAcc}_\sigma)$ is a trace (that is, satisfies lock semantics) because none of the acquires and releases are dropped.

We now state main result that we will exploit. Its proof can be found in Appendix D.

Lemma 7. *Let σ be a trace, and let ρ be a correct reordering of* $\mathsf{filter}(\sigma, \mathsf{IrrAcc}_\sigma)$ *with respect to* $\mathsf{DF}^\top$. *Then there is a trace τ such that $\rho = \mathsf{filter}(\tau, \mathsf{IrrAcc}_\sigma)$ and τ is a correct reordering of σ with respect to* $\mathsf{DF}^{\mathsf{br}}$.

Lemma 7 suggests the following approach. To predict deadlocks in σ for correct reorderings with respect to $\mathsf{DF}^{\mathsf{br}}$, compute IrrAcc_σ, filter the trace to obtain trace π, and analyze π using Algorithm 1 in Section 4. If π has a predictable deadlock then Lemma 7 guarantees that so does σ. The main challenge remaining is then to figure out how to compute IrrAcc_σ. We show that there a single pass, linear time, vector clock based algorithm that can "compute" the set IrrAcc_σ. This algorithm does not explicitly compute the set of all events in IrrAcc_σ (which would be huge and linear in σ). Instead it computes a vector clock and a set of program variables. Using the vector clock and set of variables, a second pass can determine if $e \in \mathsf{IrrAcc}_\sigma$, and if not, the algorithm will process it as Algorithm 1. Thus, we have a two-pass, linear time algorithm for predicting deadlocks with respect to $\mathsf{DF}^{\mathsf{br}}$. Details of this algorithm are presented in Appendix D.

6 Experimental Evaluation

Here, we discuss the evaluation of our approach on real-world benchmarks, with traces of length as high as 1.45 billion events. A summary of our results is presented in Table 1. Additional details about our evaluation can be found in Appendix E.

Implementation. We have implemented the deadlock prediction algorithm based on the $\leq_{\mathsf{DCP}}$ partial order in our tool DeadTrack , written in Java. DeadTrack implements the vector clock algorithm described in Section 4. We also incorporate control flow information observed in the trace as described in Section 5 for enhancing $\leq_{\mathsf{DCP}}$ based prediction. DeadTrack also implements a Goodlock style lock graph algorithm to identify 2 thread deadlock patterns. We compare against the only existing sound dynamic deadlock prediction tool Dirk [20], and in order to compare against the same trace, we analyze traces generated using Dirk's logging library. A direct comparison against DeadlockFuzzer [19] was not possible as the tool uses deprecated libraries and Java functionality which resulted in it only being able to run on a handful of the benchmarks. Further, the Sherlock tool [12] is not available in the public domain, and could not be compared against.

Benchmarks and Setup. We evaluated our approach on a set of comprehensive benchmarks, also used in prior work [20]. The names of these benchmarks are listed in Column 1 of Table 1. The benchmarks ArrayList and HashMap have been derived from [19]. Avrora, Batik, Tomcat, Eclipse, Jython, Luindex, Lusearch, FOP, PMD and Xalan are derived from the Dacapo benchmark suite [4]. Transfer, Bensalem [3] and Picklock [31] is an illustrative example from [20]. Deadlock and Dining Phil, are derived from the SIR repository and the benchmarks Dbcp1, Dbcp2, Derby2 and Log4J2 are derived from the JaConTeBe suite [26]. The examples LongDeadlock, TrueDeadlock and FalseDeadlock have been custom designed by us (see Appendix E). Our experiments were conducted on a machine with 64 GB of heap space and an 8-core 46-bit Intel Xeon(R) processor @ 2.6GHz. In Columns 3, 5, 7 and 11 of Table ??, we report the number of different pairs of program locations (p, p') corresponding to which there are event pairs f, f' participating in a deadlock pattern (or a readl deadlock for the case of Dirk, DCP and DCPDF) $\langle e, f, e', f' \rangle$.

6.1 Results

Let us now discuss some observations from our evaluation (summarized in Table 1).

Prediction Capability. Note that the SMT solving approach employed in Dirk [20] is maximal—any predictable deadlock will be, in theory, reported. On the other hand, prediction using Theorem 1 not guaranteed to report all deadlocks.

Table 1. Deadlock findings for each benchmark program. Column 1 and 2 describe the benchmark name and the size of the traces generates. Columns 3, 5, 7 and 11 report the number of deadlocks as reported by Goodlock, Dirk, DCP and DCPDF (DCP with data and control flow information). Columns 4, 6, 8 and 12 report the time (in seconds) taken by Goodlock, Dirk, DCP and DCPDF for analyzing the entire trace. Columns 9 and 13 report the speedup of DeadTrack over Dirk. Columns 10 and 14 report the size of the FIFO queues $Acq_{t,\ell}$ and $Rel_{t,\ell}$ used in the vector clock algorithms for DCP and DCPDF, as a fraction of the total size of the trace. We run Dirk with the parameters `--chunk-size 100000 --chunk-offset 5000 --solve-time 60000` (window size $100K$ and solver timeout 60 seconds).

1	2	3	4	5	6	7	8	9	10	11	12	13	14
		Goodlock		Dirk		DCP				DCPDF			
Benchmark	#Events	D.locks	Time (s)	D.locks	Time (s)	D.locks	Time (s)	Speed-up	|Queue|	D.locks	Time (s)	Speed-up	|Queue|
Deadlock	37	1	0.03	1	0.48	0	0.03	15.0×	10.81%	1	0.03	14.55×	10.81%
TrueDeadlock	38	1	0.29	0	0.1	1	0.03	3.45×	10.53%	1	0.26	0.39×	10.53%
PickLock	50	1	0.03	1	0.16	1	0.03	5.0×	12.0%	1	0.04	4.32×	12.0%
Dining	58	1	0.03	1	0.62	1	0.03	20.0×	6.9%	1	0.04	17.71×	6.9%
Bensalem	59	1	0.03	1	0.66	1	0.03	20.0×	16.95%	1	0.24	2.74×	16.95%
Transfer	62	1	0.03	1	0.18	0	0.03	5.45×	6.45%	1	0.04	4.0×	6.45%
FalseDeadlock	651	0	0.06	1	9.57	0	0.07	145.0×	0.92%	0	0.08	121.14×	0.92%
DBCP1	1.78K	1	0.08	1	0.77	1	0.07	11.67×	0.56%	1	0.09	8.46×	0.56%
Derby2	1.93K	1	0.25	1	0.57	1	0.17	3.33×	0.16%	1	0.09	6.63×	0.16%
Log4j2	2.23K	1	0.09	1	1.44	1	0.09	16.18×	0.49%	1	0.11	12.97×	0.49%
DBCP2	2.67K	1	0.12	1	0.8	1	0.09	8.6×	0.52%	1	0.12	6.96×	0.52%
LongDeadlock	6.03K	1	0.1	0	15.98h	1	0.15	386046.98×	0.07%	1	0.2	283354.68×	0.07%
HashMap	45.52K	3	0.31	2	58.12m	3	8.32	418.95×	0.04%	3	8.22	424.1×	0.04%
ArrayList	45.87K	3	0.32	3	54.49m	3	10.54	310.08×	0.03%	3	17.3	188.92×	0.03%
lusearch-fix	304.02M	0	11.86m	0	1.19h	0	11.56m	6.16×	0.0%	0	23.06m	3.09×	0.0%
pmd	6.6M	0	14.07	0	1.56m	0	14.72	6.35×	0.0%	0	24.84	3.76×	0.0%
fop	24.48M	0	46.68	0	5.82m	0	50.01	6.98×	0.0%	0	1.33m	4.37×	0.0%
luindex	26.74M	0	57.83	0	6.3m	0	59.64	6.34×	0.0%	0	1.85m	3.41×	0.0%
tomcat	30.64M	0	1.21m	0	7.27m	0	1.28m	5.66×	0.0%	0	3.15m	2.31×	0.0%
batik	63.63M	0	2.4m	0	14.96m	0	2.4m	6.24×	0.0%	0	4.24m	3.53×	0.0%
eclipse	104.75M	6	4.82m	0	24.91m	0	26.34m	0.95×	0.1%	0	13.84m	1.8×	0.1%
xalan	203.76M	0	8.2m	0	48.99m	0	10.31m	4.75×	0.01%	0	19.22m	2.55×	0.01%
jython	242.99M	0	6.53m	0	59.64m	0	6.6m	9.03×	0.0%	0	11.64m	5.12×	0.0%
avrora	1.45B	0	46.35m	0	5.71h	0	1.0h	5.71×	0.05%	0	1.64h	3.48×	0.05%

However, on the set of benchmarks (derived from [20]), one can see that $\leq_{\mathsf{DCP}}$ based deadlock prediction is indeed powerful and all deadlocks reported by Dirk are also reported by the DCP and DCPDF engines in DeadTrack , and most of the times match the upper bound given by Goodlock (Column 3). The additional deadlock in Transfer missed by DCP, but identified by DCPDF was manually inspected to be schedulable due to reasoning that involves control flow information (see Section 5). The benchmark TrueDeadlock (see Appendix E) has a predictable deadlock, with a deadlock pattern that manifests due to fork and join dependencies. Dirk misses this deadlock pattern and thus the deadlock. Appendix E also discusses how prediction power of Dirk can be affected due to *windowing* using a parametric version of LongDeadlock.

Soundness. We manually inspected the deadlocks pointed out by DeadTrack 's vector clock implementation based in $\leq_{\mathsf{DCP}}$ to experimentally verify its soundness guarantee (Theorem 1). Most deadlocks reported by DeadTrack are also reported by Dirk, which relies on SMT solving for confirming deadlock patterns. The extra deadlock in HashMap was manually verified to be correct. The real predictable deadlock in LongDeadlock (see Appendix E) is missed by Dirk because the solver in Dirk times out when analyzing the trace of length about 6000. We found that the benchmark FalseDeadlock (see Appendix E), in fact, does not have a deadlock pattern, because of a common lock protecting the pattern. Dirk, nevertheless, reports this as a deadlock, violating the soundness guarantee of the underlying approach. Further, the soundness guarantee of $\leq_{\mathsf{DCP}}$ applies to traces that obey well-nesting (see Section 2), while the benchmarks eclipse and avrora do not satisfy this guarantee. However, $\leq_{\mathsf{DCP}}$ does not report any deadlock on these benchmarks and is thus vacuously sound.

Scalability. Our linear time vector clock algorithm is always faster than Dirk, by a good margin (mean speedup of $> 16,000$ and median speed-up of 6.6). Dirk uses an SMT solver as a back-end engine and in order to scale to large traces, Dirk resorts to windowing, i.e., splitting the trace into smaller chunks. On the other hand, our approach does not depend upon windowing. Further, Dirk first identifies deadlock patterns and then in a separate phase checks feasibility constraints for each of the pattern found. It does not report any deadlock patterns in the larger DaCaPo benchmarks. Unlike Dirk, we do not have a pass where we first search for deadlock patterns. This explains the odd-one-out example eclipse where DeadTrack is a bit slower. The DCPDF engine however performs much better in eclipse because a lot of events can be identified as irrelevant accesses (Section 5). Finally, the theoretical linear space bound for $\leq_{\mathsf{DCP}}$ deadlock detection is not a bottleneck and the FIFO data structures used do not grow prohibitively large on the examples with large traces.

7 Conclusions

We present a linear time, partial-order based technique for sound deadlock prediction that has promising performance on benchmark examples. There are several avenues for future work. We currently only characterize predictable deadlocks of size 2, and extending the DCP partial order for detecting many-thread deadlocks is a promising direction. Another interesting direction is to enhance prediction power by identifying weaker partial orders than DCP. Our algorithm can benefit from an inexpensive unsound, yet maximal, analysis to filter out infeasible deadlock patterns.

Acknowledgements

We thank Christian Gram Kalhauge and Jens Palseberg for helpful discussions and support in setting up their tool Dirk.

References

[1] R. Agarwal, L. Wang, and S.D. Stoller. 2005. Detecting potential deadlocks with static analysis and runtime monitoring. In *Proceedings of the Haifa Verification Conference*. 191–207.

[2] S. Bensalem, J.C. Fernandez, K. Havelund, and L. Mounier. 2006. Confirmation of deadlock potentials detected by runtime analysis. In *Proceedings of the Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*. 41–50.

[3] S. Bensalem, A. Griesmayer, A. Legay, T.-H. Nguyen, and D.A. Peled. 2011. Efficient deadlock detection for concurrent systems. In *Proceedings of the IEEE/ACM International Conference on Formal Methods and Models for Codesign*. 119–129.

[4] 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. *SIGPLAN Not.* 41, 10 (Oct. 2006), 169–190.

[5] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. 2002. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. *SIGPLAN Not.* 37, 11 (Nov. 2002), 211–230.

[6] Y. Cai and W.K. Chan. 2012. MagicFuzzer: Scalable deadlock detection for large scale applications. In *Proceedings of the International Conference on Software Engineering*. 606–616.

[7] Yan Cai and Wing-Kwong Chan. 2014. Magiclock: scalable detection of potential deadlocks in large-scale multithreaded programs. *IEEE Transactions on Software Engineering* 40, 3 (2014), 266–281.

[8] Yan Cai, Changjiang Jia, Shangru Wu, Ke Zhai, and Wing Kwong Chan. 2015. ASN: a dynamic barrier-based approach to confirmation of deadlocks from warnings for large-scale multithreaded programs. *IEEE Transactions on Parallel and Distributed Systems* 26, 1 (2015), 13–23.

[9] Yan Cai and Qiong Lu. 2016. Dynamic testing for deadlocks via constraints. *IEEE Transactions on Software Engineering* 42, 9 (2016), 825–842.

[10] Yan Cai, Shangru Wu, and Wing Kwong Chan. 2014. ConLock: A constraint-based approach to dynamic checking on deadlocks in multithreaded programs. In *Proceedings of the 36th international conference on software engineering*. ACM, 491–502.

[11] Dawson Engler and Ken Ashcraft. 2003. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. *SIGOPS Oper. Syst. Rev.* 37, 5 (Oct. 2003), 237–252.

[12] M. Eslamimehr and J. Palsberg. 2014. Sherlock: Scalable deadlock detection for concurrent programs. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 353–365.

[13] C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata. 2002. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 234–245.

[14] A. Garcia and C. Laneve. 2017. Deadlock detection of Java bytecode. In *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation*.

[15] J. Harrow. 2000. Runtime checking of multithreaded applications with Visual Threads. In *Proceedings of the SPIN Symposium on Model Checking of Software*. 331–342.

[16] K. Havelund. 2000. Using runtime analysis to guide model checking of Java programs. In *Proceedings of the SPIN Symposium on Model Checking of Software*. 245–264.

[17] Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. 2014. Maximal Sound Predictive Race Detection with Control Flow Abstraction. *SIGPLAN Not.* 49, 6 (June 2014), 337–348.

[18] P. Joshi, M. Naik, K. Sen, and D. Gay. 2010. An effective dynamic analysis for detecting generalized deadlocks. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 327–336.

[19] Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. 2009. A randomized dynamic program analysis technique for detecting real deadlocks. In *ACM Sigplan Notices*, Vol. 44. ACM, 110–120.

[20] Christian Gram Kalhauge and Jens Palsberg. 2018. Sound deadlock prediction. *PACMPL* 2, OOPSLA (2018), 146:1–146:29.

[21] 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 (PLDI 2017)*. ACM, New York, NY, USA, 157–170. https://doi.org/10.1145/3062341.3062374

[22] Dileep Kini, Umang Mathur, and Mahesh Viswanathan. 2017. Dynamic Race Prediction in Linear Time. *CoRR* abs/1704.02432 (2017). arXiv:1704.02432 http://arxiv.org/abs/1704.02432

[23] E. Kushilevitz and N. Nisan. 2006. *Communication Complexity*. Cambridge University Press.

[24] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978), 558–565.

[25] T. Li, C.S. Ellis, A.R. Lebeck, and D.J. Sorin. 2005. Pulse: A dynamic deadlock detection mechanism using speculative execution. In *Proceedings of the USENIX Annual technical conference*. 31–44.

[26] Ziyi Lin, Darko Marinov, Hao Zhong, Yuting Chen, and Jianjun Zhao. 2015. Jacontebe: A benchmark suite of real-world java concurrency bugs (T). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 178–189.

[27] S. Lu, S. Park, E. Seo, and Y. Zhou. 2008. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the International Conference on Architechtural support for Programming Languages and Operating Systems*. 329–339.

[28] M. Naik, C.-S. Park, K. Sen, and D. Gay. 2009. Effective Static Deadlock Detection. In *Proceedings of the International Conference on Software Engineering*. 386–396.

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

[30] Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. 2012. Sound Predictive Race Detection in Polynomial Time. *SIGPLAN Not.* 47, 1 (Jan. 2012), 387–400.

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

[32] F. Sorrentino. 2015. PickLock: A deadlock prediction approach under nexted locking. In *Proceedings of the SPIN Symposium on Model Checking Software*. 179–199.

[33] A. Williams, W. Thies, and M. Ernst. 2005. Static deadlock detection for Java libraries. In *Proceedings of the European Conference on Object-Oriented Programming*. 602–629.

[34] J. Zhou, S. Silvestro, H. Liu, Y. Cai, and T. Liu. 2017. UnDead: Detecting and preventing deadlocks in production software. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*.

A Proof of Theorem 1

We first recall the $\leq_{\mathsf{WCP}}$ partial order from [21].

Definition 4 (Happens Before). For a trace σ, $\leq^{\sigma}_{\mathsf{HB}}$ is the smallest partial order on Events_σ such that

(a) for any two events $e_1 \leq^{\sigma}_{\mathsf{tr}} e_2$, if $e_1 <^{\sigma}_{\mathsf{TO}} e_2$, then $e_1 \leq^{\sigma}_{\mathsf{HB}} e_2$, and
(b) for any two events $e_1 \leq^{\sigma}_{\mathsf{tr}} e_2$, if $e_1 \in \mathsf{Releases}_\sigma(\ell)$ and $e_2 \in \mathsf{Acquires}_\sigma(\ell)$ for some lock $\ell \in \mathsf{Locks}_\sigma$, then $e_1 \leq^{\sigma}_{\mathsf{HB}} e_2$.

Definition 5 (Weak Causally Precedes). For a trace σ, $\prec^{\sigma}_{\mathsf{WCP}}$ is the smallest relation such that the following hold.

(a) Let C_1 and C_2 be two critical sections on some lock $\ell \in \mathsf{Locks}_\sigma$ such that $\mathtt{rel}(C_1) \leq^{\sigma}_{\mathsf{tr}} \mathtt{acq}(C_2)$ and there are events $e_1 \in C_1$ and $e_2 \in C_2$ such that $e_1 \asymp e_2$. Then, $\mathtt{rel}(C_1) \prec^{\sigma}_{\mathsf{WCP}} e_2$.
(b) Let C_1 and C_2 be two critical sections on some lock $\ell \in \mathsf{Locks}_\sigma$ such that $\mathtt{rel}(C_1) \leq^{\sigma}_{\mathsf{tr}} \mathtt{acq}(C_2)$ and $\mathtt{rel}(C_1) \not\leq^{\sigma}_{\mathsf{TO}} \mathtt{acq}(C_2)$. Further, let $e_1 \in C_1$ and $e_2 \in C_2$ be events such that $e_1 \prec^{\sigma}_{\mathsf{WCP}} e_2$. Then, $\mathtt{rel}(C_1) \prec^{\sigma}_{\mathsf{WCP}} \mathtt{rel}(C_2)$.
(c) $\prec^{\sigma}_{\mathsf{WCP}}$ is closed under left and right composition with $\leq^{\sigma}_{\mathsf{HB}}$. That is, $\prec^{\sigma}_{\mathsf{WCP}} \circ \leq^{\sigma}_{\mathsf{HB}} \subseteq \prec^{\sigma}_{\mathsf{WCP}}$ and $\leq^{\sigma}_{\mathsf{HB}} \circ \prec^{\sigma}_{\mathsf{WCP}} \subseteq \prec^{\sigma}_{\mathsf{WCP}}$.

Finally, $\leq^{\sigma}_{\mathsf{WCP}} = \prec^{\sigma}_{\mathsf{WCP}} \cup <^{\sigma}_{\mathsf{TO}}$.

An ordered pair of events $(e_1, e_2) \in \leq^{\sigma}_{\mathsf{tr}}$ is a *WCP-race* if $e_1 \asymp e_2$ and $e_1 \not\leq^{\sigma}_{\mathsf{WCP}} e_2$. Further, (e_1, e_2) is the *first* WCP-race if for all WCP-races (e_1', e_2') in σ, $e_2 \leq^{\sigma}_{\mathsf{tr}} e_2'$ and if $e_2 = e_2'$ then $e_1' \leq^{\sigma}_{\mathsf{tr}} e_1$.

Let us now recall the soundness guarantee of WCP from [21].

Theorem 8 (WCP soundness). *WCP is weakly sound, i.e., given any trace σ, if σ exhibits a WCP-race then σ exhibits a predictable deadlock, or there is a correct reordering σ' of σ such that $\sigma' = \sigma'' e_1 e_2$ or $\sigma' = \sigma'' e_2 e_1$, where (e_1, e_2) is the first WCP race in σ.*

Remark 1. The rule (a) presented above in Definition 5 is a slightly weaker than in [21]. Further, the presentation in [21] does not consider `fork` and `join` events. The proof of soundness for WCP, nevertheless, holds for these cases as well.

A.1 Transforming the trace.

The proof of soundness of DCP relies on the soundness guarantee for WCP. In particular, we proceed as follows. Given a trace σ with a deadlock pattern $\langle e, f, e', f' \rangle$ with $f \,||^{\sigma}_{\mathsf{DCP}}\, f'$, we produce another trace $\mathtt{wrap}(\sigma)$ by essentially enclosing all the access events in σ using fresh locks. The transformed trace $\mathtt{wrap}(\sigma)$ then orders all conflicting events in the original trace σ, and thus $\mathtt{wrap}(\sigma)$ cannot have a predictable data race. Further, the two concurrent acquire events f and f' are unordered by WCP in $\mathtt{wrap}(\sigma)$. Let us give the details of this transformation below.

Let σ be a trace. We will assume that $\mathsf{Threads}_\sigma$ is an ordered set indexed from 1 to m and every event in σ is associated with a unique identifier. For $\{t_1, \ldots, t_k\} \subseteq \mathsf{Threads}_\sigma$, $x \in \mathsf{Vars}_\sigma$ and $e \in \mathsf{Events}_\sigma$, let us denote, by $\mathtt{wrap}_{t_1,\ldots,t_k}(e, x)$, the sequence

$$\langle t_1, \mathtt{acq}(\ell_{t_1,x})\rangle, \cdots, \langle t_k, \mathtt{acq}(\ell_{t_k,x})\rangle \cdot e\, \langle t_k, \mathtt{rel}(\ell_{t_k,x})\rangle, \ldots, \langle t_1, \mathtt{rel}(\ell_{t_1,x})\rangle.$$

We will refer to the newly introduced locks of the form $\ell_{t_i,x}$ as *fake* locks.

Definition 6. For an event $e = \langle t, o \rangle$ in trace σ and $x \in \mathsf{Vars}_\sigma$, let $\mathtt{wrap}(e)$ be a homomorphism defined as follows.

$$\mathtt{wrap}(e) = \begin{cases} e & \text{if } e \notin \mathsf{Reads}_\sigma(x) \cup \mathsf{Writes}_\sigma(x) \\ \mathtt{wrap}_t(e, x) & \text{if } e \in \mathsf{Reads}_\sigma(x) \\ \mathtt{wrap}_{t_1,\ldots,t_m}(e, x) & \text{if } e \in \mathsf{Writes}_\sigma(x) \end{cases}$$

We can naturally lift `wrap` to a trace σ by replacing every event e in σ by $\mathtt{wrap}(e)$. We will assume that every event in a trace has an associated unique identifier. For every event e' in $\mathtt{wrap}_{t_1,\ldots,t_k}(e, x)$, we let $\mathtt{src}(e') = e$.

For a trace σ and a set of events $E \subseteq \mathsf{Events}_\sigma$, let $\mathtt{clear}_E(\sigma)$ be the trace that results from removing every element in E from σ.

Let us first present a small technical lemma that relates correct reorderings of a trace σ and the corresponding wrapped trace $\mathtt{wrap}(\sigma)$

Lemma 9. *Let σ be a trace and $\alpha = \mathtt{wrap}(\sigma)$. Let fakeEvents $= \{e \in \mathsf{Acquires}_\alpha(\ell_{t_i,x}) \cup \mathsf{Releases}_\alpha(\ell_{t,x}) \mid t \in \mathsf{Threads}_\sigma, x \in \mathsf{Vars}_\sigma\}$. Let α' be a correct reordering of α. Then, $\mathtt{clear}_{\mathit{fakeEvents}}(\alpha')$ is a correct reordering of σ.*

Proof. Let $\sigma' = \mathtt{clear}_{\mathsf{fakeEvents}}(\alpha')$.

First, let us argue that σ' is a valid trace, that is, σ' does not violate lock semantics. This is because α' is a valid trace and does not violate lock semantics (for a superset of locks).

Let us now argue that for every thread $t \in \mathsf{Threads}_{\sigma'}$, $\sigma'|_t$ is a prefix of $\sigma|_t$. This is because $\alpha'|_t$ is a prefix of $\alpha|_t$ (as α' is a correct reordering of α) and fakeEvents $\cap\, \mathsf{Events}_\sigma = \emptyset$.

Next, we argue that all reads in σ' read the same last write as in σ. This is because (i) there are no read or write events in fakeEvents and (ii) all reads in α' read the same last write in α (as α' is a correct reordering of α). □

We next establish the relationship between different partial orders on the two traces σ and its transformation $\mathtt{wrap}(\sigma)$.

Lemma 10. *Let σ be a trace and $e_1, e_2 \in \mathsf{Events}_{\mathtt{wrap}(\sigma)}$. If $e_1 \leq^{\mathtt{wrap}(\sigma)}_{\mathsf{HB}} e_2$ then $\mathtt{src}(e_1) \leq^{\sigma}_{\mathsf{CHB}} \mathtt{src}(e_2)$.*

Proof. Let $e_1, e_2 \in \mathsf{Events}_{\mathtt{wrap}(\sigma)}$ such that $e_1 \leq^{\mathtt{wrap}(\sigma)}_{\mathsf{HB}} e_2$. Then, there is a path

$$e_1 = f_0 \leq^{\mathtt{wrap}(\sigma)}_{\mathsf{HB}} f_1 \leq^{\mathtt{wrap}(\sigma)}_{\mathsf{HB}} f_2 \cdots \leq^{\mathtt{wrap}(\sigma)}_{\mathsf{HB}} f_n = e_2$$

such that for every $0 \leq i \leq n-1$, either $f_i <^{\mathtt{wrap}(\sigma)}_{\mathsf{TO}} f_{i+1}$ or $f_i = \langle t_i, \mathtt{rel}(\ell)\rangle$ and $f_{i+1} = \langle t_j, \mathtt{acq}(\ell)\rangle$. In the first case, we have $\mathtt{src}(f_i) <^{\sigma}_{\mathsf{TO}} \mathtt{src}(f_{i+1})$ and thus $\mathtt{src}(f_i) \leq^{\sigma}_{\mathsf{CHB}} \mathtt{src}(f_{i+1})$. In the latter case, if ℓ is not a fake lock, then $\mathtt{src}(f_i) = f_i$ and

$\mathsf{src}(f_{i+1}) = f_{i+1}$ and thus $\mathsf{src}(f_i) \leq^{\sigma}_{\mathsf{CHB}} \mathsf{src}(f_{i+1})$. Otherwise, we have that $\mathsf{src}(f_i) \asymp \mathsf{src}(f_{i+1})$ and thus $\mathsf{src}(f_i) \leq^{\sigma}_{\mathsf{CHB}} \mathsf{src}(f_{i+1})$. Then, from the transitivity of $\leq_{\mathsf{CHB}}$, it follows that $\mathsf{src}(e_1) \leq^{\sigma}_{\mathsf{CHB}} \mathsf{src}(e_2)$. □

Lemma 11. *Let σ be a trace and $e_1, e_2 \in \mathsf{Events}_{\mathsf{wrap}(\sigma)}$. If $e_1 \leq^{\mathsf{wrap}(\sigma)}_{\mathsf{WCP}} e_2$ then $\mathsf{src}(e_1) \leq^{\sigma}_{\mathsf{DCP}} \mathsf{src}(e_2)$.*

Proof. If $e_1 <^{\mathsf{wrap}(\sigma)}_{\mathsf{TO}} e_2$, then clearly, $\mathsf{src}(e_1) <^{\sigma}_{\mathsf{TO}} \mathsf{src}(e_2)$ and thus $\mathsf{src}(e_1) \leq^{\sigma}_{\mathsf{DCP}} \mathsf{src}(e_2)$. Otherwise, we have $e_1 <^{\mathsf{wrap}(\sigma)}_{\mathsf{WCP}} e_2$. We will induct on the rank of $\leq^{\mathsf{wrap}(\sigma)}_{\mathsf{WCP}}$ edges.

- **Case $e_1 \in \mathsf{Releases}_{\mathsf{wrap}(\sigma)}(\ell)$, and there is an $e'_1 \in CS(e_1)$ such that $e'_1 \asymp e_2$ and $e_2 \in CS(\ell)$.**
  If e_1 is a fake lock, then $\mathsf{src}(e_1) = e'_1 \asymp e_2 = \mathsf{src}(e_2)$ and thus $\mathsf{src}(e_1) \leq^{\sigma}_{\mathsf{DCP}} \mathsf{src}(e_2)$. Otherwise, $\mathsf{src}(e_1) = e_1 \in \mathsf{Releases}_{\sigma}(\ell)$, and, in the trace σ, we have $e'_1 \in CS(e_1)$ and $e_2 \in CS(\ell)$, and thus $e_1 \leq^{\sigma}_{\mathsf{DCP}} e_2$ by rule (b) of $\leq_{\mathsf{DCP}}$.
- **Case $e_1, e_2 \in \mathsf{Releases}_{\sigma}(\ell)$ and there are events $e'_1, e'_2 \in \mathsf{Events}_{\mathsf{wrap}(\sigma)}$ such that $e'_1 \in CS(e_1)$, $e'_2 \in CS(e_2)$ and the WCP edge $(e'_1, e'_2) \in \leq^{\mathsf{wrap}(\sigma)}_{\mathsf{WCP}}$ has a lower rank.**
  Our inductive hypothesis ensures that $\mathsf{src}(e'_1) \leq^{\sigma}_{\mathsf{DCP}} \mathsf{src}(e'_2)$. Now, if ℓ is a fake lock, then $\mathsf{src}(e_1) = \mathsf{src}(e'_1)$ and $\mathsf{src}(e_2) = \mathsf{src}(e'_2)$ and thus we have $\mathsf{src}(e_1) \leq^{\sigma}_{\mathsf{DCP}} \mathsf{src}(e_2)$. Otherwise, in trace σ, we have $\mathsf{src}(e'_1) \in CS(e_1)$ and $\mathsf{src}(e'_2) \in CS(e_2)$. Thus, by rule (c) of $\leq_{\mathsf{DCP}}$, we have $\mathsf{src}(e_1) \leq^{\sigma}_{\mathsf{DCP}} \mathsf{src}(e_2)$.
- **Case HB-composition**
  - **There is an event $e_3 \in \mathsf{Events}_{\mathsf{wrap}(\sigma)}$ such that $e_1 \leq^{\mathsf{wrap}(\sigma)}_{\mathsf{HB}} e_3$ and the WCP edge $e_3 \leq^{\mathsf{wrap}(\sigma)}_{\mathsf{WCP}} e_2$ has a lower rank.**
    By Lemma 10, we have $\mathsf{src}(e_1) \leq^{\sigma}_{\mathsf{CHB}} \mathsf{src}(e_3)$. Also, by induction hypothesis, we have $\mathsf{src}(e_3) \leq^{\sigma}_{\mathsf{DCP}} \mathsf{src}(e_2)$. Thus, we have $\mathsf{src}(e_1) \leq^{\sigma}_{\mathsf{DCP}} \mathsf{src}(e_2)$ by $\leq^{\sigma}_{\mathsf{CHB}}$ composition.
  - **There is an event $e_3 \in \mathsf{Events}_{\mathsf{wrap}(\sigma)}$ such that $e_1 \leq^{\mathsf{wrap}(\sigma)}_{\mathsf{HB}} e_3$ and the WCP edge $e_3 \leq^{\mathsf{wrap}(\sigma)}_{\mathsf{WCP}} e_2$ has a lower rank.**
    Similar to the previous case.

□

A.2 Adding a fake race in $\mathsf{wrap}(\sigma)$

In the trace $\mathsf{wrap}(\sigma)$, there are no races. This is because, all conflicting events are enclosed within critical sections of some common lock. We will be using the soundness theorem of WCP to infer the presence of deadlocks. For this, we will introduce additional write events right before the two acquire events (in $\mathsf{wrap}(\sigma)$) that describe the deadlock. We will then show that because the two acquire events are unordered by DCP and the threads executing them do not hold a common lock while executing them, then the two write events introduced in $\mathsf{wrap}(\sigma)$ are in WCP race.

So let us first define some useful notation here.

Definition 7. Let σ be a trace and let $e_1, e_2 \in \mathsf{Events}_{\sigma}$. Let d be a fresh variable (that is, d has no access event in σ). The trace $\mathsf{addFresh}(\sigma, e_1, e_2, d)$ is constructed by adding events $e'_1 = \langle t_1, \mathsf{w}(d)\rangle$ and $e'_2 = \langle t_2, \mathsf{w}(d)\rangle$ just before e_1 and e_2 respectively, where $t_i = \mathsf{thr}(e_i)$, $i \in \{1, 2\}$. More formally,

$$\mathsf{addFresh}(\sigma, e_1, e_2, d) = \begin{cases} \varepsilon & \text{if } \sigma = \varepsilon \\ \mathsf{addFresh}(\sigma', e_1, e_2, d) \cdot \langle t_i, \mathsf{w}(d)\rangle \cdot e_i & \text{if } \sigma = \sigma' \cdot e_i,\ i \in \{1,2\} \\ \mathsf{addFresh}(\sigma', e_1, e_2, d) \cdot e & \text{if } \sigma = \sigma' \cdot e,\ e \notin \{e_1, e_2\} \end{cases}$$

Claim 12. *Let σ be a trace and let $e_1, e_2 \in \mathsf{Events}_{\sigma}$ such that $\mathsf{locksHeld}_{\sigma}(e_1) \cap \mathsf{locksHeld}_{\sigma}(e_2) = \varnothing$. Let d be a fresh variable for σ, let $\rho = \mathsf{addFresh}(\sigma, e_1, e_2, d)$ be defined as above, and let e'_1, e'_2 be the $\mathsf{w}(d)$ events newly introduced. Let $f_1, f_2 \in \mathsf{Events}_{\rho} \setminus \{e'_1, e'_2\}$, be two distinct events trace ordered between e'_1 and e'_2 (exclusive). We have,*

1. *If $f_1 \leq^{\rho}_{\mathsf{HB}} f_2$, then $f_1 \leq^{\sigma}_{\mathsf{HB}} f_2$*
2. *If $f_1 <^{\rho}_{\mathsf{WCP}} f_2$, then $f_1 <^{\sigma}_{\mathsf{WCP}} f_2$*

Proof. Let f_1, f_2 be the two events as described above.

1. Let $f_1 \leq^{\rho}_{\mathsf{HB}} f_2$. Then there is an HB path, consisting of $<^{\rho}_{\mathsf{TO}}$ and `rel-acq` edges, going strictly down the trace order. Each of these edges is also present in σ and thus $f_1 \leq^{\sigma}_{\mathsf{HB}} f_2$.
2. Let $f_1 <^{\rho}_{\mathsf{WCP}} f_2$. We can induct on the rank of this edge
   - **Case rule-a edge**. Then f_1 is a $\mathtt{rel}(\ell)$ event and there is an event f'_1 such that $f'_1 \asymp f_2$. Then, clearly $f'_1 \in \mathsf{Events}_{\sigma}$ and thus we have $f_1 <^{\sigma}_{\mathsf{WCP}} f_2$.
   - **Case rule-b edge**. Then, f_1 and f_2 are both $\mathtt{rel}(\ell)$ events (for some ℓ) and there are events $f'_1 \in CS(f_1)$ and $f'_2 \in CS(f_2)$. Neither of f'_1 and f'_2 are one of e'_1 or e'_2 because $\mathsf{locksHeld}_{\rho}(e'_1) = \mathsf{locksHeld}_{\sigma}(e_1)$ and $\mathsf{locksHeld}_{\rho}(e'_2) = \mathsf{locksHeld}_{\sigma}(e_2)$ and thus we have $\mathsf{locksHeld}_{\sigma}(e'_1) \cap \mathsf{locksHeld}_{\sigma}(e'_2) = \varnothing$. Thus, we also have $f_1 <^{\sigma}_{\mathsf{WCP}} f_2$
   - **Case rule-c edge**. Then, we consider two cases
     - there is an $f_3 \notin \{f_1, f_2\}$ such that either $f_1 \leq^{\rho}_{\mathsf{HB}} f_3 <^{\rho}_{\mathsf{WCP}} f_2$. Clearly, $f_1 \leq^{\rho}_{\mathsf{tr}} f_3 \leq^{\rho}_{\mathsf{tr}} f_2$ and is thus also present in the trace σ. By induction, we have $f_3 <^{\sigma}_{\mathsf{WCP}} f_2$. Also, we saw above that $f_1 \leq^{\sigma}_{\mathsf{HB}} f_3$. This gives us the desired result.
     - there is an $f_3 \notin \{f_1, f_2\}$ such that either $f_1 <^{\rho}_{\mathsf{WCP}} f_3 \leq^{\rho}_{\mathsf{HB}} f_2$. This case is similar to the previous one and thus skipped.

□

Lemma 13. *Let σ be a trace and let $e_1, e_2 \in \mathsf{Events}_{\sigma}$ be events such that $e_1 \,||^{\sigma}_{\mathsf{WCP}}\, e_2$ and $\mathsf{locksHeld}_{\sigma}(e_1) \cap \mathsf{locksHeld}_{\sigma}(e_2) = \varnothing$. Then, $e'_1 \,||^{\rho}_{\mathsf{WCP}}\, e'_2$, where d is a fresh variable for σ, $\rho = \mathsf{addFresh}(\sigma, e_1, e_2, d)$ and the events $e'_1 = \langle t_1, \mathsf{w}(d)\rangle$, $e'_2 = \langle t_2, \mathsf{w}(d)\rangle \in \mathsf{Events}_{\rho}$ are the newly added write events.*

Proof. Without loss of generality, assume $e_1 \leq^{\sigma}_{\text{tr}} e_2$ and thus $e'_1 \leq^{\rho}_{\text{tr}} e'_2$. Let us on the contrary assume that $e'_1 \leq^{\rho}_{\text{WCP}} e'_2$.

If $e'_1 <^{\rho}_{\text{TO}} e'_2$, then $e_1 <^{\sigma}_{\text{TO}} e_2$, which contradicts $e_1 \,||^{\sigma}_{\text{WCP}}\, e_2$. Otherwise, $e'_1 \prec^{\rho}_{\text{WCP}} e'_2$. Thus, there is a path

$$e'_1 = f_0, f_1, \ldots f_n = e'_2$$

such that (i) for every $0 \leq i < n$, we have either $f_i \leq^{\rho}_{\text{HB}} f_{i+1}$ or $f_i \prec^{\rho}_{\text{WCP}} f_{i+1}$ is a rule-(a) or rule-(b) WCP edge, and (ii) there is a $0 \leq j < n$ such that $f_j \prec^{\rho}_{\text{WCP}} f_{j+1}$. Consider the first edge (f_0, f_1). This cannot be a rule-(a) or rule-(b) WCP edge. This is because every such edge originates from a $\texttt{rel}(\cdot)$ event but e'_1 is a $\mathsf{w}(d)$ event. Thus, $e'_1 \leq^{\rho}_{\text{HB}} f_1$.

Now consider the HB path

$$e'_1 = g_0 \leq^{\rho}_{\text{HB}} g_1 \cdots \leq^{\rho}_{\text{HB}} g_k = f_1$$

where for each $0 \leq i \leq k$, we have either $f_i <^{\rho}_{\text{TO}} f_{i+1}$ or f_i, f_{i+1} are $\texttt{rel}(\ell)$-$\texttt{acq}(\ell)$ events for some lock ℓ. In particular, the first edge (g_0, g_1) can only be a thread-order edge because e'_1 is a write event. In this case, we clearly have $e_1 <^{\rho}_{\text{TO}} g_1$ because e_1 is the event right after e'_1 in $\mathsf{thr}(e'_1)$. And thus, we have $e_1 \leq^{\rho}_{\text{HB}} f_1$ and thus $e_1 \prec^{\rho}_{\text{WCP}} e'_2$.

Also, notice that the edge (f_{n-1}, f_n) cannot be a WCP rule (b) edge because its target is not a release event. Also, it cannot be a rule (a) edge because the only event that conflicts with $f_n = e'_2$ is e'_1 but we have that $\mathsf{locksHeld}_{\rho}(e'_1) \cap \mathsf{locksHeld}_{\rho}(e'_2) = \mathsf{locksHeld}_{\rho}(e_1) \cap \mathsf{locksHeld}_{\rho}(e_2) = \varnothing$. Thus, (f_{n-1}, f_n) is an $\leq^{\rho}_{\text{HB}}$ edge.

So we have, $e_1 \prec^{\rho}_{\text{WCP}} f_{n-1}$ and thus, $e_1 \prec^{\sigma}_{\text{WCP}} f_{n-1}$ because of Claim 12.

This clearly gives $e_1 \prec^{\sigma}_{\text{WCP}} e_2$ which is a contradiction. □

Corollary 14. *Let σ be a trace and let $e_1, e_2 \in \mathsf{Events}_{\sigma}$ be events such that $\mathsf{locksHeld}_{\sigma}(e_1) \cap \mathsf{locksHeld}_{\sigma}(e_2) = \varnothing$. Let d be a fresh variable for σ, and let us denote by $\rho = \mathsf{addFresh}(\sigma, e_1, e_2, d)$ the trace obtained by adding events $e'_1 = \langle \mathsf{thr}(e_1), \mathsf{w}(d)\rangle$ and $e'_2 = \langle \mathsf{thr}(e_2), \mathsf{w}(d)\rangle$ right above e_1 and e_2 respectively. If $e_1 \,||^{\sigma}_{\text{WCP}}\, e_2$ then (e'_1, e'_2) is a WCP race in ρ.*

We are now ready to prove the main theorem.

Theorem 15. *Let σ be a trace and $e_1 = \langle t, \mathsf{acq}(\ell_1)\rangle$, $e_2 = \langle t', \mathsf{acq}(\ell_2)\rangle \in \mathsf{Events}_{\sigma}$ such that $\ell_1 \neq \ell_2$. If $\ell_2 \in \mathsf{locksHeld}_{\sigma}(e_1)$, $\ell_1 \in \mathsf{locksHeld}_{\sigma}(e_2)$, $\mathsf{locksHeld}_{\sigma}(e_1) \cap \mathsf{locksHeld}_{\sigma}(e_2) = \varnothing$ and $e_1 \,||^{\sigma}_{\text{DCP}}\, e_2$, then σ has a predictable deadlock.*

Proof. Using Lemma 11, we have that $e_1 = \texttt{src}(e_1) \,||^{\sigma'}_{\text{WCP}}\, \texttt{src}(e_2) = e_2$. Let d be a fresh variable for $\texttt{wrap}(\sigma)$ (and thus fresh for σ also), and let $\rho = \texttt{addFresh}(\texttt{wrap}(\sigma), e_1, e_2, d)$ be the trace obtained by adding events $e'_1 = \langle \mathsf{thr}(e_1), \mathsf{w}(d)\rangle$ and $e'_2 = \langle \mathsf{thr}(e_2), \mathsf{w}(d)\rangle$ right above events e_1 and e_2 respectively in the trace $\texttt{wrap}(\sigma)$.

From Corollary 14, we have that (e'_1, e'_2) is a WCP race in ρ. Also, for every other conflicting pair of events $(f_1, f_2) \in \leq^{\rho}_{\text{tr}}$, both f_1 and f_2 are protected by a common lock, and thus, $f_1 \leq^{\rho}_{\text{WCP}} f_2$. Hence, (e'_1, e'_2) is the first (in fact, the only) WCP race for the trace ρ.

Then, by the soundness theorem of WCP (Theorem 8), we have one of the two cases :

1. there is a correct reordering ρ' of ρ that has a deadlock
2. there is a correct reordering ρ' of ρ of the form $\rho'' e'_1 e'_2$ or $\rho'' e'_2 e'_1$.

In both the above cases, we will show that the original trace σ has a predictive deadlock.

1. **Case Predictive deadlock**.
   Here we have a correct reordering ρ' of ρ that exhibits a deadlock. First note that $\rho'' = \texttt{clear}_{\{e'_1, e'_2\}}(\rho')$ is a correct reordering of ρ' and thus of ρ. Therefore, by Lemma 9, the trace $\sigma' = \texttt{clear}_{\mathsf{fakeEvents}}(\rho'')$ is a correct reordering of σ, where $\mathsf{fakeEvents} = \{e \in \mathsf{Acquires}_{\rho''}(\ell_{t_i,x}) \cup \mathsf{Releases}_{\rho''}(\ell_{t,x}) \mid t \in \mathsf{Threads}_{\sigma}, x \in \mathsf{Vars}_{\sigma}\}$.
   Let us argue that σ' has a deadlock. Let the witness to the deadlock in ρ' be events $f_1, f_2 \ldots f_k$ such that $f_i = \langle t_i, \mathsf{acq}(\ell_i)\rangle$ with $\bigwedge_{1 \leq i \neq j \leq k} t_i \neq t_j$ and $\bigwedge_{1 \leq i \neq j \leq k} \ell_i \neq \ell_j$ $\mathsf{match}_(e_i) \notin \rho'$ and $\mathsf{next}_{\rho'}(t_i) = \langle t_i, \mathsf{acq}(\ell_{(i+1)\%k})\rangle$ for all $1 \leq i \leq k$.
   First, see that $f_i \in \rho''$ because the dropped events e'_1 and e'_2 are write events. Also, $f_i \notin \mathsf{fakeEvents}$ as the fake locks cannot be in a deadlock because they are always the innermost locks and the order of acquisition b/w two fake locks never gets inverted. Thus, $f_i \in \mathsf{Events}_{\sigma'}$ for every $1 \leq i \leq k$. Also, $\mathsf{match}_(f_i) \notin \sigma'$ because $\mathsf{match}_(f_i) \notin \rho'$. Finally, $\mathsf{next}_{\sigma'}(t_i) = \mathsf{next}_{\rho''}(t_i) = \langle t_i, \mathsf{acq}(\ell_{(i+1)\%k})\rangle$. Thus, σ' exhibits the same deadlock.
2. **Case Predictive race**.
   In this case, there is a correct reordering $\rho' = \rho'' e'_i e'_j$ $(i, j \in \{1, 2\},\ i \neq j)$ of ρ. In particular, ρ'' is also a correct reordering of ρ. Let $\sigma' = \texttt{clear}_{\mathsf{fakeEvents}}(\rho'')$. By Lemma 9, is a correct reordering of σ
   Now, let l_i denote the last event in $t_i\mathsf{thr}(e_i)$, where $i \in \{1, 2\}$. The next t_1 event after l_1 in trace σ is e_1. Similarly, the next t_2 event after l_2 in trace σ is e_2. Hence, l_i is also the next t_i event in σ'. Also, $\ell_2 \in \mathsf{locksHeld}_{\sigma'}(l_1)$ and $\ell_1 \in \mathsf{locksHeld}_{\sigma'}(l_2)$.
   Thus, σ' exhibits a deadlock.

□

B Additional Discussion about the Vector Clock Algorithm

In this section, we give more details about the algorithm.

Vector clocks and timestamps. We briefly recall vector times, clocks and associated notations following the vocabulary in [21, 30]. A *vector time* or *timestamp* is a map $V : \mathsf{Threads}_{\sigma} \to \mathbb{N}$ mapping each thread of a trace σ to a non-negative integer. For a vector time V, $V[n/t] = \lambda u \cdot$ if $u = t$ then n else $V(t)$. We denote $\bot = \lambda t \cdot 0$. For two vector times V_1 and V_2, $V_1 \sqcup V_2 = \lambda t \cdot \max(V_1(t), V_2(t))$, and

$V_1 \sqsubseteq V_2 \equiv \forall t, V_1(t) \leq V_2(t)$. A *vector clock* is a place holder for vector times, or, in other words, a variable that takes on values from the space of vector times. All operations on vector times can, thus, directly be lifted to vector clocks.

Differences from WCP algorithm. Let us point out the differences in the vector clock algorithm for recognizing the WCP [21] partial order and our partial order $\leq_{\mathsf{DCP}}$. First, the WCP algorithm maintains, for every pair (ℓ, x) of lock ℓ and memory location x, vector clocks of the form $\mathbb{L}^{\mathsf{r}}_{\ell,x}$ and $\mathbb{L}^{\mathsf{w}}_{\ell,x}$. These are used for the rule (a) of WCP that orders critical sections when they contain conflicting events. For $\leq_{\mathsf{DCP}}$, this rule is encompassed by rule (b) in Definition 2, and thus, there is no need to maintain these additional vector clocks. Another important difference arises due to the fact that $\leq_{\mathsf{DCP}}$ is closed under composition with $\leq_{\mathsf{CHB}}$, unlike WCP which is closed under composition with HB (happens-before). $\leq_{\mathsf{CHB}}$, in addition to ordering HB-ordered events, also orders conflicting events. Algorithm 1, hence, uses additional clocks $\mathbb{H}^{\mathsf{r}}_{t,x}$ and $\mathbb{H}^{\mathsf{w}}_{t,x}$ to correctly maintain $\leq_{\mathsf{CHB}}$, $\prec_{\mathsf{DCP}}$ and $\leq_{\mathsf{DCP}}$. Next, the presentation in [21] (and also in earlier partial orders like CP [30]) does not include fork and join events, as against our presentation. Finally, rule (b) in WCP [21] orders releases even if they belong to the same thread, unlike the rule (b) $\prec_{\mathsf{DCP}}$. To maintain this distinction, our FIFO queues $Acq_{t,\ell}$ also enqueue the $\leq_{\mathsf{TO}}$ timestamp $\mathbb{T}_t$ (line 21). This distinction also shows up in the comparison (line 24) at a release event—in order to check if events e and e' are ordered by the irreflexive version of WCP ($\prec_{\mathsf{WCP}}$), one only needs to check if their WCP ($\leq_{\mathsf{WCP}}$) timestamps C_e and $C_{e'}$ satisfy $C_e \sqsubseteq C_{e'}$. On the other hand, for $\prec_{\mathsf{DCP}}$ ordered, one needs to careful—$e \prec_{\mathsf{DCP}} e'$ iff $C_e \sqsubseteq P_{e'}$.

Optimizations. We incorporate several optimizations over the basic vector clock algorithm. The optimized algorithm is shown in Algorithm 2. First, let us observe the relationship $D_e = T_e \sqcup P_e$ for every event e. This means that the vector clock $\mathbb{D}_t$ need not be maintained explicitly, and can be generated from the clocks $\mathbb{T}_t$ and $\mathbb{P}_t$. Second, we incorporate Djit$^+$ style optimization—it is enough to increment the local clocks $\mathbb{T}_t(t)$ and $\mathbb{H}_t(t)$ after a release, read, write or fork event and the increments performed after an acquire or a join event are unnecessary and can be skipped. Next, we observe that it is enough to keep track of pairs $\langle t_e, \mathbb{T}_{t_e}(t_e)\rangle$ at an acquire event $e = \langle t_e, \mathsf{acq}(\ell_e)\rangle$ instead of full vector times $\langle T_{t_e}, D_{t_e}\rangle$ in the FIFO data structure $Acq_{t,\ell}$. This is because the checks on line 24 and 27 in Algorithm 1 are equivalent to checking if "$D'(t') > \mathbb{P}_t(t')$" and $T'(t') > \mathbb{T}_t(t')$, where t' is the thread of the acquire event for which Algorithm 1 enqueued the pair $\langle T', D'\rangle$ of vector times. We similarly also observe that instead of maintaining separate vector clocks $\mathbb{T}^a_{t,x}$ for each pair of thread and variable, it is sufficient to maintain the threads that performed the last read or write on x. We store the thread that last wrote to x (LWT_x) and also the set of threads that read from x since the last write to x ($\mathsf{rThreads}_x$). This simplifies the checks on line 36, 42 and 45 in Algorithm 1.

Detecting unordered deadlock patterns. Recall that Theorem 1 requires us to identify a deadlock pattern $\langle e, f, e', f'\rangle$ of size 2 with $f \,||_{\mathsf{DCP}}\, f'$. To find such patterns, for every lock ℓ, we keep track of a set $\mathsf{History}(\ell)$ of pairs $\langle \mathsf{locksHeld}_\sigma(e), D_e\rangle$, where $e \in \mathsf{Acquires}_\sigma(\ell)$. This construction is similar to traditional Goodlock style lock graph construction, but also keeps track of fork and join dependencies—when a thread t forks a thread u, every event e_u of u also keeps track of all the set of locks held by t at the time of the fork. Finally, at an acquire event $f = \langle t_f, \mathsf{acq}(\ell_f)\rangle$, we check if there is a lock $\ell \in \mathsf{locksHeld}_\sigma(f) \setminus \{\ell_f\}$ and a tuple $\langle L, D\rangle \in \mathsf{History}(\ell)$ such that $\ell_f \in L$, $D \not\sqsubseteq D_f$, and $L \setminus \{\ell\} \cap \mathsf{locksHeld}_\sigma(\ell_f) \setminus \{\ell_f\} = \varnothing$. If so, we declare that the trace has a deadlock.

C Proof of Theorem 4

Before proving Theorem 4 we will prove a simpler observation. The proof of this simpler observation contains all the ideas we need to prove Theorem 4, and therefore, makes the proof easier to understand. To state the simpler observation, let us define the notion of a *one pass* algorithm. A one pass algorithm is one that reads each symbol of the input at most once. Formally, one can think of a one pass algorithm as a Turing machine with a read-only input tape and a read-write worktape, where the input head is constrained to move right in each step. The space requirements of such an algorithm (as always) is measured in terms of the number of cells used on the work-tape.

Lemma 16. *Let A be a one pass algorithm for* $\mathsf{DeadlockPred}$ *using space $S(n)$. Then $S(n) = \Omega(n)$.*

Proof of Lemma 16. Consider the language $L_n = \{uv \mid u, v \in \{0,1\}^n \text{ and } u = v\}$. Observe that any (finite) automaton recognizing L_n must have at least 2^n states. This is because for any automaton M with $< 2^n$ states, there must be strings $u_1 \neq u_2 \in \{0,1\}^n$ such that M reaches the same state on both u_1 and u_2. Hence M either accepts both u_1u_1 and u_2u_1 or rejects both of them, which means that M cannot be recognizing L_n. Therefore, any one pass algorithm for L_n uses space $\Omega(n)$.

Our proof of Lemma 16 will essentially "reduce" L_n to $\mathsf{DeadlockPred}$, for every n — given a string w, we will construct a trace σ_n with constantly many (i.e., independent of n) threads, locks, and variables such that $w \in L_n$ if and only if $\sigma_n \notin \mathsf{DeadlockPred}$; moreover $|\sigma| = O(n)$. Specifically, the trace we construct will have 5 threads, three locks $\{\ell, m, n\}$, and 9 variables $\{p, q, s, u, v, x_0, x_1, y_0, y_1\}$. To describe the reduction, let fix a string $w = b_1b_2\cdots b_nc_1c_2\cdots c_n$ that is a candidate input to L_n.

It will be convenient to introduce some notation to denote events that will constitute the trace σ_n. The bits b_i and

Algorithm 2: *Optimized vector clock algorithm for* $\leq_{\mathsf{DCP}}$

```
procedure Initialization
    for $t \in$ Threads do
        $\mathbb{P}_t := \bot$;
        $\mathbb{H}_t := \bot[1/t]$;
        $\mathbb{T}_t = \bot[1/t]$
        for $\ell$ do
            $Acq_{t,\ell} := \varnothing$;
            $Rel_{t,\ell} := \varnothing$
    for $\ell \in$ Locks do
        $\mathbb{P}_\ell := \bot$;
        $\mathbb{H}_\ell := \bot$;
    for $x \in$ Vars do
        rThreads$_x = \varnothing$;
        LWT$_x$ = NIL
        $\mathbb{H}^{\mathsf{w}}_x := \bot$;
        for $t \in$ Threads do $\mathbb{H}^{\mathsf{r}}_{t,x} := \bot$;

procedure acquire($t$, $\ell$)
    $\mathbb{H}_t := \mathbb{H}_t \sqcup \mathbb{H}_\ell$;
    $\mathbb{P}_t := \mathbb{P}_t \sqcup \mathbb{P}_\ell$;
    for $t' \neq t$ do
        $Acq_{t',\ell}$ · Enqueue($\langle t, \mathbb{T}_t(t) \rangle$)

procedure release($t$, $\ell$)
    while $Acq_{t,\ell}$.nonempty() do
        $\langle t', c' \rangle := Acq_{t,\ell}$ · Front()
        $H' := Rel_{t,\ell}$ · Front()
        if $c' > \mathbb{P}_t(t')$ then
            break;
        if $c' > \mathbb{T}(t')$ then
            $\mathbb{P}_t := \mathbb{P}_t \sqcup H'$;
        $Acq_{t,\ell}$ · Dequeue();
        $Rel_{t,\ell}$.Dequeue();
    $\mathbb{H}_\ell := \mathbb{H}_t$; $\mathbb{P}_\ell := \mathbb{P}_t$;
    for $t' \neq t$ do
        $Rel_{t',\ell}$ · Enqueue($\mathbb{H}_t$)
    $\mathbb{T}_t(t)$++; $\mathbb{H}_t(t)$++;

procedure read($t$, $x$)
    if $\mathbb{H}^{\mathsf{w}}_x(LWT_x) > \mathbb{T}_t(LWT_x)$ then
        $\mathbb{P}_t := \mathbb{P}_t \sqcup \mathbb{H}^{\mathsf{w}}_x$;
        $\mathbb{H}_t := \mathbb{H}_t \sqcup \mathbb{H}^{\mathsf{w}}_x$;
    $\mathbb{H}^{\mathsf{r}}_{t,x} := \mathbb{H}_t$;
    rThreads$_x$ := rThreads$_x \cup \{t\}$;
    $\mathbb{T}_t(t)$++; $\mathbb{H}_t(t)$++;

procedure write($t$, $x$)
    for $t' \in rThreads_x$ do
        if $\mathbb{H}^{\mathsf{r}}_{t',x}(t') > \mathbb{T}_t(t')$ then
            $\mathbb{P}_t := \mathbb{P}_t \sqcup \mathbb{H}^{\mathsf{r}}_{t',x}$;
            $\mathbb{H}_t := \mathbb{H}_t \sqcup \mathbb{H}^{\mathsf{r}}_{t',x}$;
    if $\mathbb{H}^{\mathsf{w}}_x(LWT_x) > \mathbb{T}_t(LWT_x)$ then
        $\mathbb{P}_t := \mathbb{P}_t \sqcup \mathbb{H}^{\mathsf{w}}_x$;
        $\mathbb{H}_t := \mathbb{H}_t \sqcup \mathbb{H}^{\mathsf{w}}_x$;
    $\mathbb{H}^{\mathsf{w}}_x := \mathbb{H}_t$;
    rThreads$_x = \varnothing$;
    LWT$_x = t$;
    $\mathbb{T}_t(t)$++; $\mathbb{H}_t(t)$++;

procedure fork($t_p$, $t_c$)
    $\mathbb{H}_{t_c} := \mathbb{H}_{t_p}[1/t_c]$;
    $\mathbb{T}_{t_c} := \mathbb{T}_{t_p}[1/t_c]$;
    $\mathbb{P}_{t_c} := \mathbb{P}_{t_p}$;
    $\mathbb{T}_{t_p}(t_p)$++; $\mathbb{H}_{t_p}(t_p)$++;

procedure join($t_p$, $t_c$)
    $\mathbb{H}_{t_p} := \mathbb{H}_{t_p} \sqcup \mathbb{H}_{t_c}$;
    $\mathbb{T}_{t_p} := \mathbb{T}_{t_p} \sqcup \mathbb{T}_{t_c}$;
    $\mathbb{P}_{t_p} := \mathbb{P}_{t_p} \sqcup \mathbb{P}_{t_c}$;
```

c_i of w will be encoded as reads and writes on variables $\{x_0, x_1, y_0, y_1\}$. The variable chosen to encode this bit will depend on the value of the bit b_i/c_i and whether the index i is odd or even; if i is odd then we use $\{x_0, x_1\}$ and if i is even then we use $\{y_0, y_1\}$. The bit b_i will be encoded as a single event (denoted) $[b_i]^2_{\mathsf{w}}$, while for c_i we will have two events $[c_i]^3_{\mathsf{w}}$ and $[c_i]^4_{\mathsf{r}}$.

$$[b_i]^2_{\mathsf{w}} = \begin{cases} \langle 2, \mathsf{w}(x_{b_i}) \rangle & \text{if } i \text{ is odd} \\ \langle 2, \mathsf{w}(y_{b_i}) \rangle & \text{otherwise} \end{cases}$$
$$[c_i]^3_{\mathsf{w}} = \begin{cases} \langle 3, \mathsf{w}(x_{c_i}) \rangle & \text{if } i \text{ is odd} \\ \langle 3, \mathsf{w}(y_{c_i}) \rangle & \text{otherwise} \end{cases}$$
$$[c_i]^4_{\mathsf{r}} = \begin{cases} \langle 4, \mathsf{r}(x_{c_i}) \rangle & \text{if } i \text{ is odd} \\ \langle 4, \mathsf{r}(y_{c_i}) \rangle & \text{otherwise} \end{cases}$$

The variables $\{p, q, s, u, v\}$ will each be shared between 2 threads with one thread the exclusive writer and the other the exclusive reader. The types of each variable are as follows: p is written by 1 and read by 2; q is written by 2 and read by 4; s is written by 3 and read by 4; u is written by 3 and read by 5; and finally v is written by 4 and read by 5. We will denote access events on this variables as follows.

$$\begin{array}{ll} [p]^1_{\mathsf{w}} = \langle 1, \mathsf{w}(p) \rangle & [p]^2_{\mathsf{r}} = \langle 2, \mathsf{r}(p) \rangle \\ [q]^2_{\mathsf{w}} = \langle 2, \mathsf{w}(q) \rangle & [q]^4_{\mathsf{r}} = \langle 4, \mathsf{r}(q) \rangle \\ [s]^3_{\mathsf{w}} = \langle 3, \mathsf{w}(s) \rangle & [s]^4_{\mathsf{r}} = \langle 4, \mathsf{r}(s) \rangle \\ [u]^3_{\mathsf{w}} = \langle 3, \mathsf{w}(u) \rangle & [u]^5_{\mathsf{r}} = \langle 5, \mathsf{r}(u) \rangle \\ [v]^4_{\mathsf{w}} = \langle 4, \mathsf{w}(v) \rangle & [v]^5_{\mathsf{r}} = \langle 5, \mathsf{r}(v) \rangle \end{array}$$

While there will be many access events on variable s, there will be exactly one read and write event on the other variables in the trace σ. There will be the acquire and release events on locks ℓ and m that will form a deadlock pattern. The acquire events on these locks will be named as follows.

$$\begin{array}{ll} [a]^1_\ell = \langle 1, \mathsf{acq}(\ell) \rangle & [a]^5_\ell = \langle 5, \mathsf{acq}(\ell) \rangle \\ [a]^1_m = \langle 1, \mathsf{acq}(m) \rangle & [a]^5_m = \langle 5, \mathsf{acq}(m) \rangle \end{array}$$

Their matching release events will just be denoted using the $\mathsf{match}_{\sigma_n}(\cdot)$ predicate. Finally, there will be two acquire events on lock n which will be denoted as f and g.

$$f = \langle 1, \mathsf{acq}(n) \rangle \quad g = \langle 5, \mathsf{acq}(n) \rangle$$

As before the releases corresponding to f and g will be denoted using the $\mathsf{match}_{\sigma_n}(\cdot)$ predicate.

For $n = 5$, the trace σ_5 is shown in Figure 9; in this trace we have explicitly written all events except $[b_i]^2_{\mathsf{w}}$, $[c_i]^3_{\mathsf{w}}$ and $[c_i]^4_{\mathsf{r}}$, instead of using the notation introduced above. For a

	t_1	t_2	t_3	t_4	t_5
1	$\mathsf{acq}(n)$				
2	$\mathsf{w}(p)$				
3		$\mathsf{r}(p)$			
4		$[b_1]^2_{\mathsf{w}}$			
5		$[b_2]^2_{\mathsf{w}}$			
6		$[b_3]^2_{\mathsf{w}}$			
7		$[b_4]^2_{\mathsf{w}}$			
8		$[b_5]^2_{\mathsf{w}}$			
9		$\mathsf{w}(q)$			
10	$\mathsf{acq}(\ell)$				
11	$\mathsf{acq}(m)$				
12	$\mathsf{rel}(m)$				
13	$\mathsf{rel}(\ell)$				
14	$\mathsf{rel}(n)$				
15			$[c_1]^3_{\mathsf{w}}$		
16			$\mathsf{w}(u)$		
17			$[c_2]^3_{\mathsf{w}}$		
18			$\mathsf{w}(s)$		
19				$\mathsf{r}(s)$	
20				$[c_1]^4_{\mathsf{r}}$	
21			$[c_3]^3_{\mathsf{w}}$		
22			$\mathsf{w}(s)$		
23				$\mathsf{r}(s)$	
24				$[c_2]^4_{\mathsf{r}}$	
25			$[c_4]^3_{\mathsf{w}}$		
26			$\mathsf{w}(s)$		
27				$\mathsf{r}(s)$	
28				$[c_3]^4_{\mathsf{r}}$	
29			$[c_5]^3_{\mathsf{w}}$		
30			$\mathsf{w}(s)$		
31				$\mathsf{r}(s)$	
32				$[c_4]^4_{\mathsf{r}}$	
33				$\mathsf{r}(q)$	
34				$[c_5]^4_{\mathsf{r}}$	
35				$\mathsf{w}(v)$	
36					$\mathsf{acq}(n)$
37					$\mathsf{r}(u)$
38					$\mathsf{rel}(n)$
39					$\mathsf{acq}(m)$
40					$\mathsf{r}(v)$
41					$\mathsf{acq}(\ell)$
42					$\mathsf{rel}(\ell)$
43					$\mathsf{rel}(m)$

Figure 9. Trace σ_5 used in the linear space lower bound for streaming algorithms.

general n, the trace σ_n is the following; here we will use the notation for events introduced in the previous paragraph.

$$\begin{array}{l}
f \cdot [p]^1_{\mathsf{w}} \cdot [p]^2_{\mathsf{r}} \cdot [b_1]^2_{\mathsf{w}} \cdots [b_n]^2_{\mathsf{w}} \cdot [q]^2_{\mathsf{w}} \cdot \\
[a]^1_{\ell} \cdot [a]^1_m \cdot \mathsf{match}_{\sigma_n}([a]^1_m) \cdot \mathsf{match}_{\sigma_n}([a]^1_{\ell}) \cdot \mathsf{match}_{\sigma_n}(f) \cdot \\
[c_1]^3_{\mathsf{w}} \cdot [u]^3_{\mathsf{w}} \cdot [c_2]^3_{\mathsf{w}} \cdot [s]^3_{\mathsf{w}} \cdot [s]^4_{\mathsf{r}} \cdot [c_1]^4_{\mathsf{r}} \cdot \\
[c_3]^3_{\mathsf{w}} \cdot [s]^3_{\mathsf{w}} \cdot [s]^4_{\mathsf{r}} \cdot [c_2]^4_{\mathsf{r}} \cdots [c_i]^3_{\mathsf{w}} \cdot [s]^3_{\mathsf{w}} \cdot [s]^4_{\mathsf{r}} \cdot [c_{i-1}]^4_{\mathsf{r}} \cdots \\
[c_n]^3_{\mathsf{w}} \cdot [s]^3_{\mathsf{w}} \cdot [s]^4_{\mathsf{r}} \cdot [c_{n-1}]^4_{\mathsf{r}} \cdot [q]^4_{\mathsf{r}} \cdot [c_n]^4_{\mathsf{r}} \cdot [v]^4_{\mathsf{w}} \cdot \\
g \cdot [u]^5_{\mathsf{r}} \cdot \mathsf{match}_{\sigma_n}(g) \cdot \\
[a]^5_m \cdot [v]^5_{\mathsf{r}} \cdot [a]^5_{\ell} \cdot \mathsf{match}_{\sigma_n}([a]^5_{\ell}) \cdot \mathsf{match}_{\sigma_n}([a]^5_m)
\end{array}$$

Before sketching the correctness of the reduction, let us make some simple observations about the trace σ_n. We begin by identifying the last write events of different read events.

$$\begin{array}{ll}
\mathsf{lw}_{\sigma_n}([p]^2_{\mathsf{r}}) = [p]^1_{\mathsf{w}} & \mathsf{lw}_{\sigma_n}([q]^4_{\mathsf{r}}) = [q]^2_{\mathsf{w}} \\
\mathsf{lw}_{\sigma_n}([u]^5_{\mathsf{r}}) = [u]^3_{\mathsf{w}} & \mathsf{lw}_{\sigma_n}([v]^5_{\mathsf{r}}) = [v]^4_{\mathsf{w}}
\end{array}$$

In addition, the last write event for every $[s]^4_{\mathsf{r}}$-event is the $[s]^3_{\mathsf{w}}$-event immediately preceding it. Next, notice that the use of $\{x_0, x_1\}$ to encode c_i when i is odd, and $\{y_0, y_1\}$ when i is even, means that $\mathsf{lw}_{\sigma_n}([c_i]^4_{\mathsf{r}}) = [c_i]^3_{\mathsf{w}}$. Next, observe that the presence of $[v]^4_{\mathsf{w}}$, $[s]^3_{\mathsf{w}}$, and $[q]^2_{\mathsf{w}}$ as the last events of threads 4, 3, and 2, respectively, means that any correct reordering of σ_n that contains the event $[v]^5_{\mathsf{r}}$ also contains all the events of threads 2, 3, and 4.

We will now argue that if $w \in L_n$ — i.e., $b_i = c_i$ for every i — then the deadlock pattern formed by events $\langle [a]^1_{\ell}, [a]^1_m, [a]^5_m, [a]^5_{\ell} \rangle$ is *not* schedulable. The reasoning is based on observing that certain events must be ordered in *every* correct reordering of σ_n. The first observation is that $[c_n]^4_{\mathsf{r}}$ must be before $[a]^5_{\ell}$ because of the events $([v]^4_{\mathsf{w}}, [v]^5_{\mathsf{r}})$. Next, the events $([q]^2_{\mathsf{w}}, [q]^4_{\mathsf{r}})$ ensure that $[b_n]^2_{\mathsf{w}}$ must be before $[c_n]^4_{\mathsf{r}}$. Since $b_n = c_n$ and $\mathsf{lw}_{\sigma_n}([c_n]^4_{\mathsf{r}}) = [c_n]^3_{\mathsf{w}}$, we must have $[b_n]^2_{\mathsf{w}}$ before $[c_n]^3_{\mathsf{w}}$. The last pair of $([s]^3_{\mathsf{w}}, [s]^4_{\mathsf{r}})$ events ensure that now we must have $[b_{n-1}]^2_{\mathsf{w}}$ before $[c_{n-1}]^4_{\mathsf{r}}$. Again, $b_{n-1} = c_{n-1}$ coupled with last write properties, mean that we can conclude that $[b_{n-1}]^2_{\mathsf{w}}$ must be before $[c_{n-1}]^3_{\mathsf{w}}$. This reasoning can be repeatedly applied to conclude that for every i, $[b_i]^2_{\mathsf{w}}$ must be before $[c_i]^3_{\mathsf{w}}$. In particular, that means that $[b_1]^2_{\mathsf{w}}$ is before $[c_1]^3_{\mathsf{w}}$. Now the pair $([u]^3_{\mathsf{w}}, [u]^5_{\mathsf{r}})$ ensure that $[c_1]^3_{\mathsf{w}}$ is before $\mathsf{match}_{\sigma_n}(g)$. These observation together with the fact that f is before $[b_1]^2_{\mathsf{w}}$ (because of the pair $([p]^1_{\mathsf{w}}, [p]^2_{\mathsf{r}})$) imply that f is before $\mathsf{match}_{\sigma_n}(g)$ in every correct reordering. Since correct reorderings preserve lock semantics, we can strengthen this observation to conclude that $\mathsf{match}_{\sigma_n}(f)$ must be before g. Finally, as $[a]^1_m$ is inside $\mathrm{CS}_{\sigma_n}(f)$, we must have $[a]^1_m$ before g and therefore, before $[a]^5_m$. This means that the deadlock pattern is not schedulable.

Let us now show that if $w \notin L_n$ then the deadlock pattern $\langle [a]^1_{\ell}, [a]^1_m, [a]^5_m, [a]^5_{\ell} \rangle$ is schedulable. If $w \notin L_n$, then for some i, $b_i \neq c_i$. Consider the following trace σ

$$\begin{array}{l}
[c_1]^3_{\mathsf{w}} \cdot [u]^3_{\mathsf{w}} \cdot g \cdot [u]^5_{\mathsf{r}} \cdot \mathsf{match}_{\sigma_n}(g) \cdot [a]^5_m \cdot \\
[c_2]^3_{\mathsf{w}} \cdot [s]^3_{\mathsf{w}} \cdot [s]^4_{\mathsf{r}} \cdot [c_1]^4_{\mathsf{r}} \cdot f \cdot [p]^1_{\mathsf{w}} \cdot [p]^2_{\mathsf{r}} \cdot \\
[b_1]^2_{\mathsf{w}} \cdot [c_3]^3_{\mathsf{w}} \cdot [s]^3_{\mathsf{w}} \cdot [s]^4_{\mathsf{r}} \cdot [c_2]^4_{\mathsf{r}} \cdot \\
[b_2]^2_{\mathsf{w}} \cdot [c_4]^3_{\mathsf{w}} \cdot [s]^3_{\mathsf{w}} \cdot [s]^4_{\mathsf{r}} \cdot [c_3]^4_{\mathsf{r}} \cdots \\
[b_{i-2}]^2_{\mathsf{w}} \cdot [c_i]^3_{\mathsf{w}} \cdot [s]^3_{\mathsf{w}} \cdot [s]^4_{\mathsf{r}} \cdot [c_{i-1}]^4_{\mathsf{r}} \cdot [b_{i-1}]^2_{\mathsf{w}} \cdot [b_i]^2_{\mathsf{w}} \cdot \\
[b_{i+1}]^2_{\mathsf{w}} \cdot [c_{i+1}]^3_{\mathsf{w}} \cdot [s]^3_{\mathsf{w}} \cdot [s]^4_{\mathsf{r}} \cdot [c_i]^4_{\mathsf{r}} \cdots \\
[b_j]^2_{\mathsf{w}} \cdot [c_j]^3_{\mathsf{w}} \cdot [s]^3_{\mathsf{w}} \cdot [s]^4_{\mathsf{r}} \cdot [c_{j-1}]^4_{\mathsf{r}} \cdots \\
[b_n]^2_{\mathsf{w}} \cdot [q]^2_{\mathsf{w}} \cdot [c_n]^3_{\mathsf{w}} \cdot [s]^3_{\mathsf{w}} \cdot [s]^4_{\mathsf{r}} \cdot [c_{n-1}]^4_{\mathsf{r}} \cdot [q]^4_{\mathsf{r}} \cdot [c_n]^4_{\mathsf{r}} \cdot \\
[v]^4_{\mathsf{w}} \cdot [v]^5_{\mathsf{r}}
\end{array}$$

Observe that σ is correct reordering of σ_n, and at this point threads 1 and 5 are deadlocked. $\square$

Let us now return to the proof of Theorem 4. Recall that in communication complexity [23], Alice and Bob have input strings u and v, respectively, and their goal is to compute some predicate on their joint inputs (u, v). In doing so, we only measure the number of bits Alice and Bob communicate to each other. Now, consider the problem where Alice and Bob want to determine if their inputs u and v are equal. It is well know [23] any deterministic protocol solving this problem must communicate $\Omega(n)$ bits, n is the length of strings u and v; in other words, there is no better algorithm than the naïve one where Alice (or Bob) sends her entire input to Bob (Alice). Now consider the problem defined by the language

$$K_n = \{u\#^n v \mid u, v \in \{0,1\}^n \text{ and } u = v\}.$$

Consider any Turing machine M that solves L_n; here, we consider the usual Turing machine model where M can go back and forth on its input. If M's running time is $T(n)$ then it makes at most $\frac{T(n)}{n}$ sojourns across $\#^n$, since it takes n steps to cross the substring $\#^n$. If M's space usage is $S(n)$ then each trip across $\#^n$ "communicates" $S(n)$-bits from u to v (or vice versa). Thus, the total number of bits communicated during M's computation on $u\#^n v$ is at most $\frac{T(n)S(n)}{n}$, which from the communication complexity lower bound we know to be at least n. Putting these observations together, we can argue that any Turing machine solving K_n has the property that $T(n)S(n) = \Omega(n^2)$. Now we can reduce membership in K_n to deadlock prediction as in Lemma 16, except that the trace we construct will pad σ_n with $\#^n$ "junk" events between $\mathsf{match}_{\sigma_n}(f)$ and $[c_1]^3_{\mathsf{w}}$. This establishes Theorem 4.

D Algorithm for Incorporating Data Flow

We first prove Lemma 7.

Proof of Lemma 7. Let $\pi = \mathsf{filter}(\sigma, \mathsf{IrrAcc}_\sigma)$. We are given that ρ is a correct reordering of π with respect to $\mathsf{DF}^\top$. Let τ be any sequence that respects $\leq^\sigma_{\mathsf{TO}}$ and $\mathsf{filter}(\tau, \mathsf{IrrAcc}_\sigma) = \rho$. Clearly, such a τ should exist because we can simply close the set Events_ρ with respect to $\leq^\sigma_{\mathsf{TO}}$ and schedule all extra events as late as possible, as long as they are consistent with $\leq^\sigma_{\mathsf{TO}}$. Observe that since all acquires and releases are preseved by filtering, since ρ is well formed, so is τ. Thus, τ is a trace.

To prove that τ is a correct reordering of σ with respect to $\mathsf{DF}^{\mathsf{br}}$, we just need to ensure that certain read events are present and last write of these events is preserved. $\mathsf{DF}^{\mathsf{br}}_\sigma$ is only non-empty for branch-events. Consider a branch event $b \in \mathsf{Events}_\tau$ and a read-event $e \in \mathsf{RelRds}_\sigma(b)$; without loss of generality, let us assume that e is a $\mathsf{r}(x)$-event. Observe that since $b \notin \mathsf{IrrAcc}_\sigma$ and $\mathsf{filter}(\tau, \mathsf{IrrAcc}_\sigma) = \rho$, we have that $b \in \mathsf{Events}_\rho$. Thus, $e \in \mathsf{Events}_\rho$. Again since $e \notin \mathsf{IrrAcc}_\sigma$, we have $e \in \mathsf{Events}_\tau$. Next, since e is a relevant read, we have $\mathsf{Writes}_\sigma(x) = \mathsf{Writes}_\pi(x)$. Therefore, we have $\mathsf{lw}_\sigma(e) = \mathsf{lw}_\pi(e)$. Since ρ is a correct reordering of π with respect to $\mathsf{DF}^\top$, we have $\mathsf{lw}_\rho(e) = \mathsf{lw}_\pi(e)$. Finally, we again have $\mathsf{Writes}_\tau(x) = \mathsf{Writes}_\rho(x)$, which means that $\mathsf{lw}_\tau(e) = \mathsf{lw}_\rho(e)$. Putting all these observations together, we have $\mathsf{lw}_\tau(e) = \mathsf{lw}_\sigma(e)$. Thus, τ is a correct reordering of σ with respect to $\mathsf{DF}^{\mathsf{br}}_\sigma$. □

We describe our algorithm for predicting deadlocks in trace σ for correct reorderings with respect to $\mathsf{DF}^{\mathsf{br}}$. We begin by defining a simple partial order on events of σ that captures data flow dependencies.

Definition 8. $\leq^\sigma_{\mathsf{DF}} \subseteq \mathsf{Events}_\sigma \times \mathsf{Events}_\sigma$ is a smallest partial order such that for any e_1, e_2 with $e_1 \leq^\sigma_{\mathsf{tr}} e_2$, (a) if $\mathsf{thr}(e_1) = \mathsf{thr}(e_2)$ then $e_1 \leq^\sigma_{\mathsf{DF}} e_2$, and (b) if $e_1 = \mathsf{lw}_\sigma(e_2)$ then $e_1 \leq^\sigma_{\mathsf{DF}} e_2$.

Observe that the following proposition follows trivially from the definitions.

Proposition 17. *If* $e_1 \in \mathsf{Reads}_\sigma$ *and* $e_1 \leq^\sigma_{\mathsf{DF}} e_2$ *then* $e_1 \in \mathsf{RelRds}_\sigma(e_2)$.

The partial order can be easily computed by a simple, one pass vector clock algorithm that increments its local clock after every write-event, and updates its entire vector clock at reads using the vector clock of the last write on the same variable. For an event e, C^{DF}_e will denote the vector timestamp of e that is consistent with $\leq^\sigma_{\mathsf{DF}}$. For a thread t and variable x, let $\mathsf{first}(t, x)$ denote the first event of the form $\langle t, \mathsf{r}(x)\rangle$; this maybe undefined, if thread t does not have any $\mathsf{r}(x)$-events. As per this notation, $C^{\mathsf{DF}}_{\mathsf{first}(t,x)}$ is the vector timestamp of first $\mathsf{r}(x)$-event in t. Next, define $\mathbb{B}^{\mathsf{DF}} = \bigsqcup_{e \in \mathsf{Branches}_\sigma} C^{\mathsf{DF}}_e$; observe that for a read event e, $C^{\mathsf{DF}}_e \sqsubseteq \mathbb{B}^{\mathsf{DF}}$ if and only there is some branch-event b such that $e \leq^\sigma_{\mathsf{DF}} b$. Finally, define

$$\mathsf{RelVar}_\sigma(x) \text{ iff } \exists t \in \mathsf{Threads}_\sigma.\ C^{\mathsf{DF}}_{\mathsf{first}(t,x)} \sqsubseteq \mathbb{B}^{\mathsf{DF}}.$$

Observe that using this notation we can define the set of relevant accesses as

$$\begin{aligned}\mathsf{RelAcc}_\sigma = \quad & \{e \in \mathsf{Reads}_\sigma \mid C^{\mathsf{DF}}_e \sqsubseteq \mathbb{B}^{\mathsf{DF}}\} \cup \\ & \{e \in \mathsf{Writes}_\sigma(x) \mid x \in \mathsf{RelVar}_\sigma(x)\}.\end{aligned} \tag{1}$$

We can now describe our two-phase algorithm. In Phase 1, we (a) run the vector clock algorithm for $\leq_{\mathsf{DF}}$ while maintaining $\mathbb{B}^{\mathsf{DF}}$ and $C^{\mathsf{DF}}_{\mathsf{first}(t,x)}$ for each (t, x), and (b) at the end return $\mathbb{B}^{\mathsf{DF}}$ and $\mathsf{RelVar}_\sigma(x)$ for each x. In Phase 2, for each event e in trace σ, we do the following.

1. If e is a data access event that does not belong to RelAcc_σ as defined by Equation 1, we skip e.
2. Otherwise, we process e as per algorithm in Section 4.

Correctness follows from the argument in Section 5.

E Additional Discussion in Evaluation

E.1 Custom Benchmarks

We list here the three custom examples used in our evaluation.

The next interesting benchmark is TrueDeadlock.java shown in Figure 10. Here, thread $T1$ acquires lock $L1$ and then forks thread $T2$. At the same time $T3$ acquires lock $L2$. Now none of these threads can proceed—$T3$ waits for lock $L1$, $L1$ is acquired by $T1$, which is waiting for $T2$ to end and $T2$ is waiting for lock $L2$. This deadlock is missed by Dirk because Dirk cannot identify a deadlock pattern here.

```java
public class TrueDeadlock {

  public static Object L1 = new Object();
  public static Object L2 = new Object();

  public static void main (String [] args) {
    T1 t1 = new T1();
    T3 t3 = new T3();
    t1.start();
    t3.start();
  }

  static class T1 extends Thread {
    public void run () {
      synchronized (L1) {
        T2 t2 = new T2();
        t2.start();
        try{
          t2.join();
        }
        catch(Exception ex){
        }
      }
    }
  }
```

```java
  static class T2 extends Thread {
    public void run () {
      synchronized (L2) {
      }
    }
  }

  static class T3 extends Thread {
    public void run () {
      synchronized (L2) {
        synchronized (L1) {
        }
      }
    }
  }
}
```

Figure 10. TrueDeadlock.java

Finally Figure 11 describes a Java program FalseDeadlock.java. This program does not have a deadlock because of a common lock $L1$ acquired in a different thread. Dirk identifies this as a deadlock pattern. To our surprise, Dirk also tags this program a deadlock.

The program LongDeadlock.java (Figure 12) is a simple program with a single deadlock pattern which is predictable.

The parameter ITERS in this program can be varied, thereby giving traces of different sizes. We generated traces by varying ITERS in the set $\{10^n\}_{n=1}^{9}$. The time taken to analyze these traces has been depicted in Figure 13. Dirk does not report any deadlock for $n > 2$ and also times out (with a limit of 24 hours) for ITERS = 10000.

E.2 Additional Data

In Table 2, we report additional information about the traces generated. The Columns 10-12 denote the number of deadlock patterns in the trace. For Transfer and ArrayList, the data flow incorporation did indeed point out more deadlocks.

```java
public class FalseDeadlock {

  public static Object L1 = new Object();
  public static Object L2 = new Object();
  public static Object L3 = new Object();

  public static long x, y;
  public static final long ITERS = 1000;

  public static void main (String [] args) {
    x = 0;
    y = 0;
    T1 t1 = new T1();
    T3 t3 = new T3();
    t1.start();
    t3.start();
    try{
          t1.join();
        }
        catch(Exception ex){

        }
        try{
          t3.join();
        }
        catch(Exception ex){

        }
  }

  static class T1 extends Thread {
    public void run () {
      synchronized (L1) {
        T2 t2 = new T2();
        t2.start();
        try{
          t2.join();
        }
        catch(Exception ex){

        }
      }
    }
  }

  static class T2 extends Thread {
    public void run () {
        synchronized (L2) {
          synchronized (L3) {
            for(int i = 0; i < ITERS; i++){
              x = x + 1;
            }
          }
        }
      }
    }

  static class T3 extends Thread {
    public void run () {
      synchronized (L1) {
          synchronized (L3) {
          synchronized (L2) {
            for(int i = 0; i < ITERS; i++){
              y = y + 1;
            }
          }
          }
        }
      }
  }
}
```

Figure 11. FalseDeadlock.java

```
public class LongDeadlock {

  public static Object L1 = new Object();
  public static Object L2 = new Object();
  public static long x, y;
  public static final long ITERS = 1000;

  public static void main (String [] args) {
    x = 0;
    y = 0;
    new T1().start();
    new T2().start();
  }

  static class T1 extends Thread {
    public void run () {
      synchronized (L1) {
        synchronized (L2) {
          for(int i = 0; i < ITERS; i++){
            x = x + 1;
          }
        }
      }
    }
  }

  static class T2 extends Thread {
    public void run () {
      synchronized (L2) {
        synchronized (L1) {
          for(int i = 0; i < ITERS; i++){
            y = y + 1;
          }
        }
      }
    }
  }
}
```

Figure 12. LongDeadlock.java

Table 2. Columns 2-9 denote the different kinds of events in the traces logged. Columnd 10-12 denote the different number of deadlock patterns (counting every pattern as many times it repeats in the trace) in the trace, identifying by Goodlock, DCP and DCPDF

1	2	3	4	5	6	7	8	9	10	11	12
	Events								Dynamic Patterns		
Benchmark	Total	Acquires	Releases	Reads	Writes	Forks	Joins	Branches	Goodlock2	DCP	DCPDF
Deadlock	37	4	4	8	6	2	0	0	1	0	1
TrueDeadlock	38	4	4	4	2	3	3	0	1	1	1
PickLock	50	8	8	8	4	2	2	0	2	2	2
Dining	58	4	4	10	8	2	0	17	1	1	1
Bensalem	59	10	10	10	3	3	1	0	1	1	1
Transfer	62	4	4	13	12	2	2	6	1	0	1
FalseDeadlock	651	6	6	206	205	3	3	202	0	0	0
DBCP1	1.78K	19	17	557	559	2	0	279	1	1	1
Derby2	1.93K	7	4	694	692	2	0	253	1	1	1
Log4j2	2.23K	20	16	757	392	3	0	507	1	1	1
DBCP2	2.67K	22	20	906	485	2	0	652	1	1	1
LongDeadlock	6.03K	4	4	2.0K	2.0K	2	0	2.0K	1	1	1
HashMap	45.52K	1.09K	1.09K	13.66K	9.2K	338	0	7.93K	9	9	9
ArrayList	45.87K	1.44K	1.44K	15.13K	7.78K	450	0	6.28K	25	15	25
lusearch-fix	304.02M	206.4K	206.4K	151.53M	40.98M	8	0	70.84M	0	0	0
pmd	6.6M	54	54	3.27M	600.21K	0	0	1.84M	0	0	0
fop	24.48M	1.22K	1.22K	9.72M	1.9M	0	0	7.69M	0	0	0
luindex	26.74M	338	338	13.97M	2.69M	0	0	7.54M	0	0	0
tomcat	30.64M	3.53K	3.52K	16.78M	2.11M	41	0	9.49M	0	0	0
batik	63.63M	22.47K	22.47K	27.15M	11.62M	1	0	18.97M	0	0	0
eclipse	104.75M	281.55K	281.55K	42.96M	6.65M	14	3	42.51M	226	0	0
xalan	203.76M	447.16K	447.16K	104.36M	15.03M	10	10	55.35M	0	0	0
jython	242.99M	4.1M	4.1M	72.3M	14.0M	1	0	65.18M	0	0	0
avrora	1.45B	1.07M	1.07M	677.5M	207.07M	3	3	252.0M	0	0	0

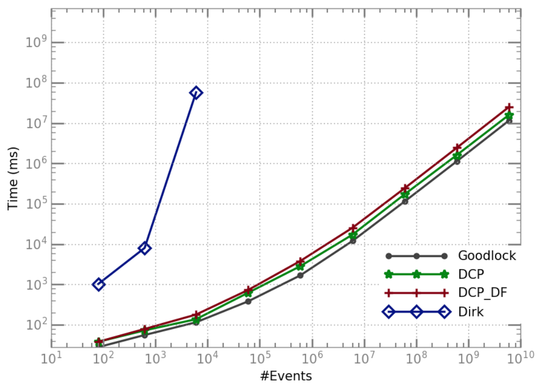

Figure 13. Performance of DeadTrack and Dirk on traces generated by LongDeadlock by varying number of iterations.