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 substraces, 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
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 concurrent. Partial order based analysis algorithms are typically “streaming” (that is, events of the traces 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 empirically 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 $\text{acq}(\ell)/\text{rel}(\ell)$ to denote the acquire/release of lock $\ell$. In trace $\rho_1$ (of Figure 1), the program does not deadlock. But we could reorder the events of $\rho_1$ to obtain a deadlock—execute $e_1$ followed by $e_5$. The goal of a partial order based deadlock detection approach would be to 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 $\rho_1$ and can reason about alternate traces where critical sections on the same lock maybe reordered.

The inspiration for our partial order based deadlock detection algorithm 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 $\sigma$ has a predictable deadlock, if there are a pair of threads $t_1$ and $t_2$, and events $e = \text{acq}(\ell)$ and $f = \text{acq}(m)$ 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 $\ell$ 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 $\sigma$, we show that one can construct another trace $\sigma'$ such that the soundness of DCP for deadlock detection in $\sigma$ follows from the soundness of WCP for race detection in $\sigma'$.

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 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×, with a median speedup of more than 6×.

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 \( \sigma \) is a pair \( e = (t, op) \)

1Formally, 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.


with re-entrant locking. For a release event \( e \), the matching acquire event for \( e \) in trace \( \sigma \) will be denoted by \( \text{match}_\sigma(e) \). Similarly, we will denote by \( \text{match}_\sigma(v) \) to be the matching release event (if any) corresponding to the acquire event \( v \), and set it to \( \bot \) if it does not exist. Further, each thread is assumed to be forked and joined at most once.

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

Example 1. Let us illustrate the definitions introduced so far on the trace \( \rho_2 \) shown in Figure 2. Recall that we denote the \( i \)th event in \( \rho_2 \) by \( e_i \). Here, \( \text{Threads}_{\rho_2} = \{ t_1, t_2, t_3 \} \), \( \text{Vars}_{\rho_2} = \{ x \} \) and \( \text{Locks}_{\rho_2} = \{ \ell, m \} \). Further, \( \text{Events}_{\rho_2} = \{ e_1 \}_{1 \leq i \leq 12} \), \( \text{Reads}_{\rho_2} = \{ e_6, e_{10} \} \), \( \text{Writes}_{\rho_2} = \{ e_4, e_8, e_{12} \} \). Finally, match\(_{\rho_2}(e_1) = e_2 \) and match\(_{\rho_2}(e_2) = \bot \). Let us now illustrate the different orders using \( \sigma \). Here, \( e_i \leq^\sigma_{TO} e_j \) iff \( i \leq j \). \( e_1 \leq^\sigma_{PO} e_3 \) because both \( e_1 \) and \( e_3 \) are performed by \( t_1 \). Further, \( e_1 \leq^\sigma_{TO} e_5 \) because of the event \( e_3 \) which forks \( t_2 \), and, \( e_1 \perp_{TO} e_{12} \). The sequence \( \rho'_2 = e_6 e_1 e_2 e_5 e_4 \) is a valid trace that preserves \( \leq^\sigma_{TO} \) but does not preserve the total order \( \leq^\sigma_{TR} \).

On the other hand, the trace \( \rho''_2 = e_1 e_2 e_4 \) does not preserve \( \leq^\sigma_{PO} \) because \( e_4 \in \text{Events}_{\rho''_2} \), \( e_1 \leq^\sigma_{PO} e_4 \) but \( e_3 \notin \text{Events}_{\rho''_2} \).

<table>
<thead>
<tr>
<th>( w(x) )</th>
<th>( \text{acq}(\ell) )</th>
<th>( \text{fork}(t_2) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{acq}(m) )</td>
<td>( \text{re1}(m) )</td>
<td>( r(x) )</td>
</tr>
<tr>
<td>( \text{join}(t_2) )</td>
<td>( \text{rel}(\ell) )</td>
<td></td>
</tr>
<tr>
<td>( \text{acq}(m) )</td>
<td>( r(x) )</td>
<td>( \text{acq}(\ell) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{rel}(\ell) )</td>
</tr>
</tbody>
</table>

Figure 2. Trace \( \rho_2 \) for illustration.
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 $\sigma$ is well nested if for any acquire $e$ in $\text{Acquires}_\sigma$ such that $\exists e' \in \text{Acquires}_\sigma \text{ with } e \leq_{\sigma} e'$ the following two conditions hold: (a) either $f \leq_{\sigma} \text{match}_\sigma(e)$ or $\exists e'' \in \text{Acquires}_\sigma$ such that $e'' \leq_{\sigma} e$ and $f = \text{match}_\sigma(e'')$, (b) if $f \in \text{Acquires}_\sigma$ with $f \leq_{\sigma} \text{match}_\sigma(e)$, then $\exists e' \in \text{Acquires}_\sigma$ such that $e' \leq_{\sigma} e$ and $\text{match}_\sigma(f') \leq_{\sigma} \text{match}_\sigma(e')$.

Conflicting Events. Two events $e_1, e_2 \in \text{Events}_\sigma$ are said to be conflicting if $e_1, e_2 \in \text{Acquires}_\sigma(x)$ for some $x \in \text{Vars}_\sigma$, $w(x) \in \{\text{op}(e_1), \text{op}(e_2)\}$, and $e_1 \leq_{\sigma} e_2$. We use $\exists e_2$ to denote that $e_1$ and $e_2$ are conflicting events.

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

Example 2. In the trace $\rho_2$ from Figure 2, the pair $(e_1, e_6)$ is not a conflicting pair of events since $e_1 \leq_{\rho_2} e_6$. However, $e_1$ conflicts with $e_{10}$ in $\rho_2$ (i.e., $e_1 \neq e_{10}$). The critical sections in $\rho_2$ are $\text{CS}_{\rho_2}(e_2) = \text{CS}_{\rho_2}(e_6) = \{e_2, e_3, e_4, e_5, e_6, e_7, e_8\}$, $\text{CS}_{\rho_2}(e_1) = \text{CS}_{\rho_2}(e_9) = \{e_4, e_5\}$, $\text{CS}(e_9) = \{e_9, e_{10}, e_{11}, e_{12}\}$ and $\text{CS}_{\rho_2}(e_{11}) = \text{CS}_{\rho_2}(e_{12}) = \{e_{11}, e_{12}\}$. In this trace, the set of locks held at $e_1$ is $\emptyset$. On the other hand, $\text{locksHeld}_{\rho_2}(e_2) = \text{locksHeld}_{\rho_2}(e_6) = \{\ell \mid f \in \text{Events}_\sigma\} \cap \text{locksHeld}(f) = \emptyset$. Finally, $\text{next}_{\rho_2}(e_1) = \{e_2\}$, $\text{next}_{\rho_2}(e_9) = \{e_4, e_5\}$, $\text{next}_{\rho_2}(e_{10}) = \{e_7\}$ and $\text{next}_{\rho_2}(e_{12}) = \emptyset$.

Correct Reordering. The goal of this paper is to perform predictive deadlock detection, i.e., given a trace $\sigma$ we want to check if either $\sigma$ or another valid rescheduling of the events of $\sigma$ exhibit a deadlock. Correct reorderings of a trace $\sigma$, identify the space of “valid reschedules” that are semantically sound; any program producing the trace $\sigma$ could produce any of the correct reorderings under a different thread schedule. For a read event $e \in \text{Reads}_\sigma(x)$ on some $x \in \text{Vars}_\sigma$, we denote by $\text{lw}_\sigma(e)$ the last write event $e' \in \text{Writes}_\sigma(x)$ before $e$ in the sequence $\sigma$ if it exists; and $\text{lw}_\sigma(e) = \bot$ otherwise. A trace $\sigma'$ is a correct reordering of a trace $\sigma$ if (a) $\exists \text{Events}_\sigma' \subseteq \text{Events}_\sigma$, (b) $\sigma'$ respects $\leq_{\sigma}$, and (c) for every $e \in \text{Reads}_\sigma', \text{lw}_\sigma(e') = \text{lw}_\sigma(e)$. To ensure that programs producing $\sigma$ can produce each of its correct reorderings, we require correct reorderings of $\sigma$ preserve intra-thread order, fork and join dependencies. Further, in a correct reordering every read event sees the same value as in $\sigma$, 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 $\sigma$ is said to exhibit a predictable deadlock of size $k$ if there is a correct reordering $\sigma'$ of $\sigma$, $k$ distinct events $e_0, e_1, \ldots, e_{k-1} \in \text{Events}_\sigma$, and $k$ distinct locks $\ell_0, \ell_1, \ldots, \ell_{k-1} \in \text{Locks}_\sigma$ such that for every $0 \leq i \leq k-1$, we have

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

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

Deadlock Pattern. A deadlock pattern of size $k$ in a trace $\sigma$ is a tuple of $2k$ events $(e_0, f_0, e_1, f_1, \ldots, e_{k-1}, f_{k-1})$ such that there are distinct locks $\ell_0, \ell_1, \ldots, \ell_{k-1} \in \text{Locks}_\sigma$ for which $\exists e_i \in \text{Events}_\sigma(e_i), f_i \in \text{CS}_\sigma(e_i), \text{op}(f_i) = \text{acq}(\ell_{i+1} \leq_{\sigma} k)$, and $(\text{locksHeld}_\sigma(f_j) \setminus (\ell_{i+1} \leq_{\sigma} k) \cap (\text{locksHeld}_\sigma(f_j) \setminus (\ell_{i+1} \leq_{\sigma} k)) = \emptyset$ for every $0 \leq i, j \leq k-1$. Observe that if a trace $\sigma$ 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 $\ell_i$ to $\ell_j$ if $\ell_j$ is acquired in a critical section of lock $\ell_i$. 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 $(e_0, f_0, \ldots, e_{k-1}, f_{k-1})$ if, for some $i, \text{thr}(e_i) \neq \text{thr}(f_j)$ (event though $f_j$ belongs to $\text{CS}_\sigma(e_i)$ 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.
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 \( \leq_{\text{DCP}} \) in Section 3.1, and state its soundness guarantees (Section 3.2). We illustrate the subtle decisions made in defining \( \leq_{\text{DCP}} \) and demonstrate its effectiveness in reasoning about deadlocks through examples in Section 3.3. \( \leq_{\text{DCP}} \), 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 \( \leq_{\text{CHB}} \), inspired by the happens-before partial order [24].

**Definition 1 (Conflict-HB).** For a trace \( \sigma \), \( \leq_{\text{CHB}}^{\sigma} \) (read ‘conflict HB’) is the smallest partial order on Events\( \sigma \) such that for any two events \( e_1, e_2 \) if either (a) \( e_1 \leq_{\text{TO}} e_2 \), or (b) \( e_1 \in \text{Releases}_\sigma(\ell) \) and \( e_2 \in \text{Acquires}_\sigma(\ell) \) for some lock \( \ell \in \text{Locks}_\sigma \), or (c) \( e_1 \succ e_2 \), then \( e_1 \leq_{\text{CHB}}^{\sigma} e_2 \).

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 \( \{(a, c) \mid \exists b \cdot (a, b) \in R_1 \land (b, c) \in R_2\} \).

**Definition 2 (Deadlock Causal Precedence).** For a trace \( \sigma \), \( \leq_{\text{DCP}}^{\sigma} \) is the smallest relation such that the following hold.

(a) For \( e_1 \leq_{\text{tr}} \sigma \leq_{\text{tr}} \sigma e_2 \) such that \( e_1 \neq e_2 \), we have \( e_1 \leq_{\text{DCP}}^{\sigma} \sigma e_2 \).

(b) Let \( C_1 \) and \( C_2 \) be two critical sections on some lock \( \ell \in \text{Locks}_\sigma \), such that \( \text{rel}(C_1) \leq_{\text{tr}} \sigma \text{acq}(C_2) \) and \( \text{rel}(C_1) \leq_{\text{TO}} \sigma \text{acq}(C_2) \). Further, let \( e_1 \in C_1 \) and \( e_2 \in C_2 \) be events such that \( e_1 \leq_{\text{DCP}}^{\sigma} \sigma e_2 \). Then, \( \text{rel}(C_1) \leq_{\text{DCP}}^{\sigma} \text{rel}(C_2) \).

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

3.2 Using \( \leq_{\text{DCP}} \) 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 \( \leq_{\text{DCP}}^{\sigma} \) to determine the absence of such a causal dependence and Theorem 1 establishes the soundness of \( \leq_{\text{DCP}}^{\sigma} \) for the purpose of predicting deadlocks.

**Theorem 1.** Let \( \sigma \) be a trace with a deadlock pattern \( (e, f, e', f') \) of size 2 such that \( f \not\in_{\text{DCP}}^{\sigma} f' \). Then, \( \sigma \) 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 \( \leq_{\text{DCP}}^{\sigma} \) in Definition 2. The trace \( p_3 \) has a deadlock pattern of size 2 (with events \( e_5, e_6, e_7 \) and \( e_8 \) participating in the pattern). Since \( e_5 \) and \( e_6 \) are ordered, any correct reordering of \( p_3 \) is a prefix of \( p_3 \). Hence, \( p_3 \) does not have a predictable deadlock. \( \leq_{\text{DCP}}^{p_3} \) ensures soundness by ordering \( e_5 \prec_{\text{DCP}}^{p_3} e_6 \) (rule (a)) and this implies that \( e_2 \leq_{\text{DCP}}^{p_3} e_8 \) because of composition with \( \leq_{\text{CHB}}^{p_3} \). As a result, Theorem 1 does not report any predictable deadlock in \( p_3 \).
Based on Example 4 above, one might naively 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 $\rho_4$ in Figure 3b. In this trace, the tuple $⟨e_5, e_4, e_{11}, e_{12}⟩$ constitute a deadlock pattern, and Goodlock-based algorithms report this pattern as a potential deadlock. Further, naively 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 $e_1, e_2, e_9, e_5, e_6, e_8,$ and $e_{10}$. Further, in order to preserve the last write of $e_9$, it must also ensure $e_2 \lesssim_{DCP} e_9$. Finally, to preserve lock semantics, $e_7$ must be executed in $ρ'_4$ before $e_5$. Hence, any such correct reordering $ρ'_4$ of $ρ_4$ will completely execute the nested critical section $e_1, e_2, e_9, e_5, e_6,$ 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 $≺_{DCP}$ in Definition 2, in fact, orders $e_7 \lesssim_{DCP} e_9$, $e_5$ because of the rule (a) ordering $e_2 \lesssim_{DCP} e_9$. As a result we have $e_4 \lesssim_{DCP} e_{12}$. 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 $≺_{DCP}$. 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 $≺_{DCP}$, the releases $r_1 = rel(C_1)$ and $r_2 = rel(C_2)$ are not ordered by $≺_{DCP}$ if $r_1 \leq_{TO} r_2$ (but of course are ordered by $≺_{DCP} \cup \leq_{TO}$). 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 $⟨e_1, e_2, e_{17}, e_{18}⟩$. In this trace, we have $e_8 \lesssim_{DCP} e_9$ and $e_{14} \lesssim_{DCP} e_{15}$, giving us $e_8 \lesssim_{DCP} e_{15}$ after composing with $e_9 \lesssim_{CHB} e_{14}$. This means that the two critical sections $C_1 = CS_{ρ_5}(e_7)$ and $C_2 = CS_{ρ_5}(e_{13})$ (on the same lock $k$) contain events $e_9$ and $e_{15}$ respectively ordered by $≺_{DCP}$. In the absence of the additional condition ($rel(C_1) \lesssim_{TO} rel(C_2)$) in rule (b), we would have ordered $e_{12} \lesssim_{DCP} e_{16}$. This in turn, composes with the CHB edge $e_2 \lesssim_{CHB} e_4 \lesssim_{CHB} e_{10} \lesssim_{CHB} e_{12} \lesssim_{DCP} e_{16} \lesssim_{CHB} e_{18}$ (due to rule (c)) giving us $e_2 \lesssim_{DCP} e_{18}$. However, the additional condition ($rel(C_1) \lesssim_{TO} rel(C_2)$) ensures that $e_2 \lesssim_{DCP} e_{18}$. Indeed, the trace $ρ_5^{CR} = e_7 e_8 e_9 e_{11} e_{12} e_2 e_4 e_{10} e_{15} e_6 e_8 e_{17}$, is a correct reordering of $ρ_5$ and witnesses a deadlock. Hence, Theorem 1 correctly identifies a predictable deadlock in $ρ_5$.

![Figure 4](image-url) // Figure 4. Trace $ρ_5$.

Observe that $\leq_{TO}$ 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 $≤_{TO}$ to be only program order and instead incorporate fork and join dependencies in the partial order $≤_{DCP}$ as base rules (like rule (a)). In the following, we will demonstrate how our choice of defining $≤_{TO}$ allows us to detect deadlocks that would have been missed otherwise, while remaining sound.

![Figure 5](image-url) // Figure 5. Trace $ρ_6$.

**Example 7.** Consider the trace $ρ_6$ in Figure 5, with a deadlock pattern $⟨e_2, e_3, e_7, e_8⟩$. 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 $≺_{DCP}$ but are ordered by $≺_{DCP} \cup \leq_{TO}$. Therefore, Theorem 1 reports no predictable deadlock.
Example 8. The trace \( \rho_7 \) in Figure 6 has a deadlock pattern of size 2, namely, \((e_1, e_2, e_5, e_6)\). Further, the correct reordering \( \rho_7^{CR} = e_1 e_5 e_6 \) of \( \rho_7 \) also witnesses the deadlock. If \( \prec_{DCP} \) included fork/join dependencies, then we would have had \( e_5 \prec_{DCP} e_6, e_2 \prec_{DCP} e_5 \) and \( e_6 \prec_{DCP} e_5 \) due to rules (c) and (a), respectively. Thus giving \( e_2 \prec_{DCP} e_6 \). That is, including the fork dependency in \( \prec_{DCP} \) forces the two acquires \( e_2 \) and \( e_6 \) to get ordered, thereby missing the predictable deadlock witnessed in \( \rho_7^{CR} \). This justifies our choice of not including such dependencies in \( \prec_{DCP} \). There are no conflicting events in \( \rho_2 \) and thus no two events are ordered by \( \prec_{DCP} \). This in turn means that \( e_2 \) and \( e_6 \) remain unordered according to \( \prec_{DCP} \), and Theorem 1 correctly declares a predictable deadlock in \( \rho_7 \).

Example 9. Let us again consider the trace \( \rho_2 \) from Figure 2. This trace has a deadlock pattern \((e_2, e_4, e_9, e_{11})\). First observe that while \( e_2 \leq_{TO} e_4 \) and \( \text{thr}(e_2) \neq \text{thr}(e_4) \). 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 \( \leq_{TO} \) 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 \( \rho_2^{CR} \) in Example 3). This is captured by DCP as follows. The only \( \prec_{DCP} \) orders in this trace are \( e_1 \prec_{DCP} e_6 \) and \( e_1 \prec_{DCP} e_{10} \) (rule (a)), and those due to composition with \( \prec_{CHB} \) (rule (c)): \( e_1 \prec_{DCP} e_7 \) and \( e_7 \prec_{DCP} e_8 \). Hence, we have \( e_1 \parallel_{DCP} e_1 \) and according to Theorem 1, \( \rho_2 \) has a predictable deadlock.

Let us now illustrate some important aspects of Theorem 1.

Example 10. Consider the trace \( \rho_8 \) 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 \( e_2 \) and \( e_4 \) (without executing the matching releases \( e_3 \) and \( e_{10} \)) cannot proceed in the thread \( t_1 \) beyond \( e_1 \). This means that the write event \( e_5 \) on \( x \) will not be executed and thus the last-write event for the read event \( e_7 \) will not be \( e_5 \). DCP, on the other hand, correctly orders \( e_3 \prec_{DCP} e_7 \) (rule (a)). This gives \( e_2 \prec_{DCP} e_8 \) (due to rule (b)). This means Theorem 1 does not report a predictable deadlock in \( \rho_8 \).

Now consider trace \( \rho_3 \), which is a close variant of \( \rho_8 \). Here, while DCP orders \( e_1 \prec_{DCP} e_7 \) and \( e_1 \prec_{DCP} e_8 \), it does not order the acquire events of the inner critical sections \( CS_{\rho_3}(e_3) \) and \( CS_{\rho_3}(e_7) \) on locks \( m \) and \( f \) respectively. As a result, Theorem 1 declares that \( \rho_3 \) has a predictable deadlock. In fact, the correct reordering \( e_1 e_2 e_3 e_7 \) of \( \rho_3 \) witnesses the deadlock.

4 Algorithm for Deadlock Prediction

In this section we describe an algorithm for predicting deadlocks using our partial order \( \leq_{DCP} \) and Theorem 1. Our algorithm (a) identifies deadlock patterns of the form \((e_1, e’_1, e_2, e’_2)\) in the execution, and (b) checks if the inner acquire events \( e’_1 \) and \( e’_2 \) are unordered by \( \leq_{DCP} \). The algorithm uses vector clocks to check if two acquire events are unordered by \( \leq_{DCP} \). 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 deferred 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 \( \leq_{DCP} \) 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
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 \( P_t, T_t, H_t, \) and \( D_t \) for each thread \( t \), vector clocks \( P_\ell, H_\ell \) for each lock \( \ell \), vector clocks \( T_{\ell,x}, T_{t,x}, H_{\ell,x}, H_{t,x} \) and \( T^w_{\ell,x}, H^w_{\ell,x} \) for each pair \((t,x)\) of thread \( t \) and memory location \( x \). Additionally, for every thread \( t \) and lock \( \ell \), the algorithm maintains FIFO queues \( Rel_t, t \) and \( Acq_t, t \) that respectively store vector times and pairs of vector times.

Broadly, the algorithm simultaneously maintains the orders \( \leq_{T_O}, \leq_{C_H}, \leq_{D_C}, \) and \( \leq_{D_P} \) and uses different clocks and queues for this purpose. The clocks \( T_t \) correspond to the partial order \( \leq_{T_O} \). Let us denote by \( T_r \) the value of the clock \( T_t \) right after processing the event \( e \) according to Algorithm 1. We say that \( T_t \) is the \( \leq_{T_O} \) timestamp of \( e \) for events \( f \leq_{T_O} f' \) if \( T_t \preceq_{T_O} T_{f'} \). Similarly, the clocks \( D_\ell \) and \( H_\ell \) respectively correspond to the partial orders \( \leq_{D_C} \) and \( \leq_{C_H} \). The value of the clock \( P_{\text{thr}(e)} \) after processing an event \( e \), denoted \( P_e \), can be used to identify \( \leq_{D_P} \) predecessors of \( e \). That is, for events \( f \) and \( f' \) (with \( f \leq_{D_P} f' \)), \( f \preceq_{D_P} f' \) if \( D_f \preceq_{D_P} P_{f'} \). The clocks \( T^w_{t,x} \) and \( T^w_{\ell,x} \) store the \( \leq_{T_O} \) timestamps \( T^w_{el,x} \) and \( T^w_{el,x} \) of the last events \( e^w_{l,x} \) and \( e^w_{l,x} \) that respectively

\[
\text{Algorithm 1: Computing the } \leq_{D_C} \text{ timestamps for different events}
\]

1. **procedure** Initialization
   2. for \( t \in \text{Threads} \) do
   3. \( P_t := \bot; \)
   4. \( H_t := H[1/t]; \)
   5. \( T_t := T[1/t]; \)
   6. \( D_t := D[1/t]; \)
   7. for \( \ell \in \text{Locks} \) do
   8. \( Acq_{\ell,t} := \emptyset; \)
   9. \( Rel_{\ell,t} := \emptyset; \)
   10. for \( x \in \text{Vars} \) do
   11. \( H^w_{\ell,x} := \bot; \)
   12. \( H^w_{t,x} := \bot; \)
   13. \( T^w_{\ell,x} := \bot; \)
   14. \( T^w_{t,x} := \bot; \)
   15. for \( t \in \text{Locks} \) do
   16. \( P_t := \bot; \)
   17. \( H_t := \bot; \)
   18. \( P_t := P_t \cup P_{\ell}; \)
   19. \( D_\ell := D_\ell \cup P_{\ell}; \)
   20. for \( \ell \in \text{Threads} \) do
   21. \( Acq_{\ell,t} := \emptyset; \)

22. **procedure** release\((t, \ell)\)
   23. while \( Acq_{\ell,t} \cdot \text{nonempty} \) do
   24. end
   25. \( H^w_{\ell,x} := H^w_{t,x}; \)
   26. \( H^w_{\ell,x} := H^w_{t,x}; \)
   27. \( T^w_{\ell,x} := T^w_{t,x}; \)

28. **procedure** read\((t, x)\)
   29. if \( T^w_{\ell,x} \preceq_{T_O} \ell \) then
   30. end
   31. \( H^w_{\ell,x} := H^w_{t,x}; \)
   32. \( H^w_{\ell,x} := H^w_{t,x}; \)
   33. \( T^w_{\ell,x} := T^w_{t,x}; \)

34. **procedure** acquire\((t, \ell)\)
   35. \( H_t := H_t \cup H_{\ell}; \)
   36. \( P_t := P_t \cup P_{\ell}; \)
   37. \( D_\ell := D_\ell \cup P_{\ell}; \)
   38. for \( \ell \in \text{Threads} \) do
   39. \( H^w_{\ell,x} := H^w_{t,x}; \)
   40. \( H^w_{\ell,x} := H^w_{t,x}; \)

41. **procedure** write\((t, x)\)
   42. if \( T^w_{\ell,x} \preceq_{T_O} \ell \) then
   43. end
   44. \( H^w_{\ell,x} := H^w_{t,x}; \)
   45. \( H^w_{\ell,x} := H^w_{t,x}; \)
   46. \( T^w_{\ell,x} := T^w_{t,x}; \)

47. **procedure** fork\((t, u)\)
   48. \( H_t := H_t \cup H_{u}; \)
   49. \( T_t := T_t \cup T_{u}; \)
   50. \( P_t := P_t \cup P_{u}; \)

51. **procedure** join\((t, u)\)
   52. \( H_t := H_t \cup H_{u}; \)
   53. \( T_t := T_t \cup T_{u}; \)
   54. \( P_t := P_t \cup P_{u}; \)
no longer need the information about \( e' \) because the timestamps of each later event \( f \) performed by \( \text{thr}(e) \) satisfies \( D_e' \subseteq P_e \subseteq P_f \). Thus, both \( e' \) and \( \text{match}_S(e') \) can safely be discarded from the set of events we wish to track. In fact, since rule (c) also ensures that for every \( \ell \)-release event \( e'' \leq_\tau e' \), we have \( C_{e'} \subseteq P_e \implies C_{e'} \subseteq P_{e''} \), we can also discard all release events on lock \( \ell \) whose timestamps were pushed before that of \( e' \) in \( \text{Rel} _{i, \ell} \).

**Updates.** After processing an event \( e = (t, \text{op}) \), the local components of the clocks \( D_t, T_t \) and \( H_t \) are incremented after performing each event (i.e., we assign \( "T_t \mapsto T_t(t + 1)/t" \), \( "D_t := D[D_t(t) + 1/t]" \), and \( "H_t := H[H_t(t) + 1/t]" \), to ensure consistency of timestamps. As an example, let \( e, e', f \) be events such that \( e' \in \text{next}_S(e) \) and \( \text{thr}(e) = \text{thr}(e') \), \( e \leq_{\text{DCP}} f \) but \( e' \not\leq_{\text{DCP}} f \), then this clock-increment ensures that the timestamps also obey \( D_t \subseteq D_f \) but \( D_{e'} \not\subseteq D_f \). Since these assignments are common to all events, we do not explicitly include them in Algorithm 1.

At an acquire event \( e = (t, \text{acq}(\ell)) \), we update \( H_t \) with the \( \leq_{\text{CHB}} \) timestamp of the latest release on \( \ell \) (stored in \( H_\ell \)). Similarly, to account for rule (c) of \( \leq_{\text{DCP}} \), we update \( P_t \) using the clock \( P_t \) storing the \( \leq_{\text{DCP}} \) timestamp of the latest \( \ell \)-release. The reasoning behind most other updates in the algorithm follows similarly by analyzing the different rules of \( \leq_{\text{DCP}} \) and \( \leq_{\text{CHB}} \). The check \( \text{next}_S(t, x) \) on line 36 performed at an event \( e = (t, r(x)) \) ensures that we only consider earlier writes \( e' \) on \( x \) that conflict with \( e \) (and thus \( e' \not\leq_{\text{TO}} e \)). 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 \( \sigma = (t, \text{acq}(\ell)) \), we update \( H_t \) with the \( \leq_{\text{CHB}} \) timestamp of the latest release on \( \ell \) (stored in \( H_\ell \)). Similarly, to account for rule (c) of \( \leq_{\text{DCP}} \), we update \( P_t \) using the clock \( P_t \) storing the \( \leq_{\text{DCP}} \) timestamp of the latest \( \ell \)-release. The reasoning behind most other updates in the algorithm follows similarly by analyzing the different rules of \( \leq_{\text{DCP}} \) and \( \leq_{\text{CHB}} \). The check \( \text{next}_S(t, x) \) on line 36 performed at an event \( e = (t, r(x)) \) ensures that we only consider earlier writes \( e' \) on \( x \) that conflict with \( e \) (and thus \( e' \not\leq_{\text{TO}} e \)). Similar reasoning applies to the checks on lines 27, 42 and 46.

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 \( O(T(L + V) + nT) \) 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 \( \leq_{\text{DCP}} \) 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 \( \leq_{\text{DCP}} \)-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 \( \sigma \), determine if \( \sigma \) 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 \( \sigma' \) of \( \sigma \) ensured that every read \( e \) in \( \sigma' \) observes the same value as \( e \) in \( \sigma \) (by forcing \( \text{lw}_e(\sigma) = \text{lw}_e(\sigma') \)). 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 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 \( T_1 \) followed by thread \( T_2 \). If, like before, we do not track any branch events, both the executions thus obtained are the trace \( \rho_3 \) shown in Figure 3a. In this case, all correct reorderings of \( \rho_3 \) are prefixes of \( \rho_3 \), and thus \( \rho_3 \) has no predictable deadlock. However, if we decide to track branch events, \( P_1 \) generates trace \( \rho_{10} \) Figure 8b, and \( P_2 \) generates \( \rho_3 \) from Figure 3a; notice that the only difference between their executions is the presence of the branch event in \( \rho_{10} \). In this case, the read event \( e_6 \) in \( \rho_{10} \) affects the branch event \( e_7 \), and thus the only
Figure 8. Programs (a) $P_1$ on the left and (c) $P_2$ on the right. In the middle (b) shows a trace $\rho_{10}$ 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 $\rho_{10}$. However, since there are no branch events in $\rho_3$ (even though we decided to track all branch events), the value that the read event $e_0$ reads does not affect the executability of any event and thus we can infer the trace $\rho_{10}^{\text{CR}} = e_1 e_0 e_2$ as a correct reordering of $\rho_3$ (even though $e_0$ no longer sees the same last write), and thus we can conclude that $\rho_3$ 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 $\text{DF}$\(^2\). For an event $e$ of $\sigma$, $\text{DF}_\sigma(e) \subseteq \text{Reads}_\sigma$ 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 $\sigma$ be a trace and $\text{DF}_\sigma$ be a data flow predicate on the events of $\sigma$. A trace $\sigma'$ is a correct reordering of $\sigma$ modulo $\text{DF}_\sigma$ if

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

Observe that the following simple observation about Definition 3 holds.

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

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 $\text{RelRds}_\sigma(e)$. Before defining this set, let $\text{PriorRds}_\sigma(e) = \{ f \in \text{Reads}_\sigma \mid f \leq^\sigma e \text{ and } \text{thr}(f) = \text{thr}(e) \}$. Now, $\text{RelRds}_\sigma(e)$ is the smallest set such that (a) $\text{PriorRds}_\sigma(e) \subseteq \text{RelRds}_\sigma(e)$, and (b) for all $f \in \text{RelRds}_\sigma(e)$, $\text{PriorRds}_\sigma(\text{lw}_{\sigma}(f)) \subseteq \text{RelRds}_\sigma(e)$. Consider the data flow predicate given by $\text{DF}_\sigma^T(e) = \text{RelRds}_\sigma(e)$ for all events $e$. It is easy to see that correct reorderings with respect to $\text{DF}_\sigma^T$ 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 $(t, \text{branch})$, which indicates that thread $t$ performed a branch-event. The set of branch events in $\sigma$ will be denoted as $\text{Branches}_\sigma$. 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.

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

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

**Example 12.** Let us reconsider the programs $P_1$ and $P_2$, the corresponding traces $\rho_3$ and $\rho_{10}$ and their reorderings modulo the two data flow predicates $\text{DF}^T_\sigma$ and $\text{DF}^\text{br}_\sigma$. The only correct reorderings of $\rho_{10}$ modulo $\text{DF}^\text{br}_\sigma$ are prefixes of $\rho_{10}$.
For $\rho_3$, the only correct reorderings of $\rho_3$ modulo $DFT$ are prefixes of $\rho_3$. On the other hand, the trace $e_1e_2e_3$ is also correct reordering of $\rho_3$ modulo $DF^{br}$.

Central to exploiting the new fine grained correct reordering definition, is to realize that the data flow predicate $DF^{br}$ can be used to identify data access events that will not influence any branch event in $\sigma$, 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 $RelRds_\sigma$ to be $\bigcup_{e \in Branches_\sigma} RelRds_\sigma(e)$, the set of all relevant access events, $RelAcc_\sigma$, is given by

$$RelAcc_\sigma = RelRds_\sigma \cup \{ x \in Vars_\sigma \mid x \in Reads_\sigma(e) \cap RelRds_\sigma \}$$

Any data access event that is not relevant is irrelevant. Thus, $IrrAcc_\sigma = Accesses_\sigma \setminus RelAcc_\sigma$.

We need one more definition before describing how to predict deadlocks modulo $DF^{br}$. For a set of events $E \subseteq Events_\sigma$, filter($\sigma, E$) is the sequence obtained by ignoring the events in $E$, i.e., it is a projection of $\sigma$ on the set $Events_\sigma \setminus E$. Notice, that filter($\sigma, IrrAcc_\sigma$) 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 $\sigma$ be a trace, and let $\rho$ be a correct reordering of filter($\sigma, IrrAcc_\sigma$) with respect to $DFT$. Then there is a trace $\tau$ such that $\rho = \text{filter}(\tau, IrrAcc_\sigma)$ and $\tau$ is a correct reordering of $\sigma$ with respect to $DF^{br}$.

Lemma 7 suggests the following approach. To predict deadlocks in $\sigma$ for correct reorderings with respect to $DF^{br}$, compute $IrrAcc_\sigma$, filter the trace to obtain trace $\pi$, and analyze $\pi$ using Algorithm 1 in Section 4. If $\pi$ has a predictable deadlock then Lemma 7 guarantees that so does $\sigma$. The main challenge remaining is then to figure out how to compute $IrrAcc_\sigma$. We show that there is a single pass, linear time, vector clock based algorithm that can “compute” the set $IrrAcc_\sigma$. This algorithm does not explicitly compute the set of all events in $IrrAcc_\sigma$ (which would be huge and linear in $\sigma$). 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 $e \in IrrAcc_\sigma$, 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 $DF^{br}$. 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 $\leq_{DCP}$ 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 $\leq_{DCP}$ 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, PDM 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) $\langle e, f, f', e' \rangle$.

### 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 $Acq_t, l$ and $Rel_t, l$ 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).

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>#Events</th>
<th>Goodlock D.locks</th>
<th>Goodlock Time (s)</th>
<th>Dirk D.locks</th>
<th>Dirk Time (s)</th>
<th>DCP D.locks</th>
<th>DCP Time (s)</th>
<th>Speed-up</th>
<th>[Queue]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deadlock</td>
<td>37</td>
<td>1</td>
<td>0.03</td>
<td>1</td>
<td>0.48</td>
<td>0</td>
<td>0.03</td>
<td>15.0×</td>
<td>10.81%</td>
</tr>
<tr>
<td>TrueDeadlock</td>
<td>38</td>
<td>1</td>
<td>0.29</td>
<td>0</td>
<td>0.1</td>
<td>1</td>
<td>0.03</td>
<td>3.45×</td>
<td>10.53%</td>
</tr>
<tr>
<td>PickLock</td>
<td>50</td>
<td>1</td>
<td>0.03</td>
<td>1</td>
<td>0.16</td>
<td>1</td>
<td>0.03</td>
<td>5.0×</td>
<td>12.0%</td>
</tr>
<tr>
<td>Dining</td>
<td>58</td>
<td>1</td>
<td>0.03</td>
<td>1</td>
<td>0.62</td>
<td>1</td>
<td>0.03</td>
<td>20.0×</td>
<td>6.9%</td>
</tr>
<tr>
<td>Bensalem</td>
<td>59</td>
<td>1</td>
<td>0.03</td>
<td>1</td>
<td>0.66</td>
<td>1</td>
<td>0.03</td>
<td>20.0×</td>
<td>16.95%</td>
</tr>
<tr>
<td>Transfer</td>
<td>62</td>
<td>1</td>
<td>0.03</td>
<td>1</td>
<td>0.18</td>
<td>0</td>
<td>0.03</td>
<td>5.45×</td>
<td>6.45%</td>
</tr>
<tr>
<td>FalseDeadlock</td>
<td>651</td>
<td>0</td>
<td>0.06</td>
<td>1</td>
<td>9.57</td>
<td>0</td>
<td>0.07</td>
<td>145.0×</td>
<td>0.92%</td>
</tr>
<tr>
<td>DBCP1</td>
<td>1.78K</td>
<td>1</td>
<td>0.08</td>
<td>1</td>
<td>0.77</td>
<td>1</td>
<td>0.07</td>
<td>11.67×</td>
<td>0.56%</td>
</tr>
<tr>
<td>Derby2</td>
<td>1.93K</td>
<td>1</td>
<td>0.25</td>
<td>1</td>
<td>0.57</td>
<td>1</td>
<td>0.17</td>
<td>3.33×</td>
<td>0.16%</td>
</tr>
<tr>
<td>Log4j2</td>
<td>2.23K</td>
<td>1</td>
<td>0.09</td>
<td>1</td>
<td>1.44</td>
<td>1</td>
<td>0.09</td>
<td>16.18×</td>
<td>0.49%</td>
</tr>
<tr>
<td>DBCP2</td>
<td>2.67K</td>
<td>1</td>
<td>0.12</td>
<td>1</td>
<td>0.8</td>
<td>1</td>
<td>0.09</td>
<td>8.6×</td>
<td>0.52%</td>
</tr>
<tr>
<td>LongDeadlock</td>
<td>6.03K</td>
<td>1</td>
<td>0.1</td>
<td>0</td>
<td>15.98×</td>
<td>1</td>
<td>0.15</td>
<td>386046.98×</td>
<td>0.07%</td>
</tr>
<tr>
<td>HashMap</td>
<td>45.52K</td>
<td>3</td>
<td>0.31</td>
<td>2</td>
<td>58.12m</td>
<td>3</td>
<td>8.32</td>
<td>418.95×</td>
<td>0.04%</td>
</tr>
<tr>
<td>ArrayList</td>
<td>45.87K</td>
<td>3</td>
<td>0.32</td>
<td>3</td>
<td>54.49m</td>
<td>3</td>
<td>10.54</td>
<td>310.08×</td>
<td>0.03%</td>
</tr>
<tr>
<td>lusearch-fix</td>
<td>304.02M</td>
<td>0</td>
<td>11.86m</td>
<td>0</td>
<td>1.19h</td>
<td>0</td>
<td>11.56m</td>
<td>6.16×</td>
<td>0.0%</td>
</tr>
<tr>
<td>pm overlay</td>
<td>6.6M</td>
<td>0</td>
<td>14.07</td>
<td>0</td>
<td>1.56m</td>
<td>0</td>
<td>14.72</td>
<td>6.35×</td>
<td>0.0%</td>
</tr>
<tr>
<td>fop</td>
<td>24.48M</td>
<td>0</td>
<td>46.68</td>
<td>0</td>
<td>5.82m</td>
<td>0</td>
<td>50.01</td>
<td>6.98×</td>
<td>0.0%</td>
</tr>
<tr>
<td>luindex</td>
<td>26.74M</td>
<td>0</td>
<td>57.83</td>
<td>0</td>
<td>6.3m</td>
<td>0</td>
<td>59.64</td>
<td>6.34×</td>
<td>0.0%</td>
</tr>
<tr>
<td>tomcat</td>
<td>30.64M</td>
<td>0</td>
<td>1.21m</td>
<td>0</td>
<td>7.27m</td>
<td>0</td>
<td>1.28m</td>
<td>5.66×</td>
<td>0.0%</td>
</tr>
<tr>
<td>batik</td>
<td>63.63M</td>
<td>0</td>
<td>2.4m</td>
<td>0</td>
<td>14.96m</td>
<td>0</td>
<td>2.4m</td>
<td>6.24×</td>
<td>0.0%</td>
</tr>
<tr>
<td>eclipse</td>
<td>104.75M</td>
<td>6</td>
<td>4.82m</td>
<td>0</td>
<td>24.91m</td>
<td>0</td>
<td>26.34m</td>
<td>0.95×</td>
<td>0.1%</td>
</tr>
<tr>
<td>xalan</td>
<td>203.76M</td>
<td>0</td>
<td>8.2m</td>
<td>0</td>
<td>48.99m</td>
<td>0</td>
<td>10.31m</td>
<td>4.75×</td>
<td>0.01%</td>
</tr>
<tr>
<td>jython</td>
<td>242.99M</td>
<td>0</td>
<td>6.53m</td>
<td>0</td>
<td>59.64m</td>
<td>0</td>
<td>6.6m</td>
<td>9.03×</td>
<td>0.0%</td>
</tr>
<tr>
<td>avora</td>
<td>1.45B</td>
<td>0</td>
<td>46.35m</td>
<td>0</td>
<td>5.71h</td>
<td>0</td>
<td>1.08</td>
<td>5.71×</td>
<td>0.05%</td>
</tr>
</tbody>
</table>

However, on the set of benchmarks (derived from [20]), one can see that $\leq_{DCP}$ 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 $\leq_{DCP}$ to experimentally verify its soundness guarantee (Theorem 1). Most deadlocks reported by DeadTrack are also reported by Dirk, which relies on SMT solver 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 $\leq_{DCP}$ applies to traces that obey well-nesting (see Section 2), while the benchmarks eclipse and avora do not satisfy this guarantee. However, $\leq_{DCP}$ 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 $\leq_{DCP}$ 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


A Proof of Theorem 1
We first recall the \( \preceq_{\text{WCP}} \) partial order from [21].

**Definition 4** (Happens Before). For a trace \( \sigma \), \( \preceq_{\text{HB}} \) is the smallest partial order on \( \text{Events}_\sigma \) such that:
- For any two events \( e_1 \preceq e_2 \), and
- For any two events \( e_1 \leq e_2 \) if \( e_1 \preceq e_2 \) and \( e_1 \in \text{E}\), then \( e_1 \leq e_2 \).

**Definition 5** (Weak Causally Precedes). For a trace \( \sigma \), \( \prec_{\text{WCP}} \) is the smallest relation such that the following hold:
- Let \( C_1 \) and \( C_2 \) be two critical sections on some lock \( \ell \) that \( \text{E} \subseteq \text{Locks}_\sigma \) such that \( \text{Erel}(C_1) \preceq_{\text{rel}} \text{Eacq}(C_2) \) and \( \text{Erel}(C_1) \preceq_{\text{rel}} \text{Eacq}(C_2) \).
- Let \( C_1 \) and \( C_2 \) be two critical sections on some lock \( \ell \) such that \( \text{Erel}(C_1) \preceq_{\text{rel}} \text{Eacq}(C_2) \).
- \( \preceq_{\text{WCP}} \) is closed under left and right composition with \( \preceq_{\text{HB}} \).

An ordered pair of events \( (e_1, e_2) \in \preceq_{\text{HB}} \) is a WCP-race if \( e_1 = e_2 \) and \( e_1 \preceq_{\text{WCP}} e_2 \). Further, \( (e_1, e_2) \) is the first WCP-race if for all WCP-races \( (e_1', e_2') \in \sigma \), \( e_2' \preceq_{\text{HB}} e_2 \) and if \( e_2 = e_2' \) then \( e_2' \preceq_{\text{HB}} e_2 \).

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

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

**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 \( \sigma \) with a deadlock pattern \( (e, f, f', f'' \ldots) \) with \( f |_{\text{DCP}} f' \), we produce another trace \( \text{Wap}(\sigma) \) by essentially enclosing all the access events in \( \sigma \) using fresh locks. The transformed trace \( \text{Wap}(\sigma) \) then orders all conflicting events in the original trace \( \sigma \), and thus \( \text{Wap}(\sigma) \) cannot have a predictable data race. Further, the two concurrent acquire events \( f \) and \( f' \) are unordered by \( \text{Wap}(\sigma) \). Let us give the details of this transformation below.

Let \( \sigma \) be a trace. We will assume that \( \text{Threads}_\sigma \) is an ordered set indexed from 1 to \( m \) and every event in \( \sigma \) is associated with a unique identifier. For \( \{t_1, \ldots, t_k\} \subseteq \text{Threads}_\sigma \), \( x \in \text{Vars}_\sigma \) and \( e \in \text{Events}_\sigma \), let us denote, by \( \text{Wap}(t_1, \ldots, t_k(e, x) \), the sequence
\[
\langle t_1, \text{acq}(t_{1, x}), \ldots, t_k, \text{acq}(t_{k, x}) \rangle \cdot e \langle t_k, \text{rel}(t_{k, x}), \ldots \rangle.
\]

We will refer to the newly introduced locks of the form \( t_{i, x} \) as fake locks.

**Definition 6.** For an event \( e = \langle t, o \rangle \) in \( \sigma \) and \( x \in \text{Vars}_\sigma \), let \( \text{Wrap}(e) \) be a homomorphism defined as follows.

\[
\text{Wrap}(e) = \begin{cases} 
\text{Wrap}(e, x) & \text{if } e \in \text{Reads}_\sigma(x) \\
\text{Wrap}(e, x) & \text{if } e \in \text{Write}_\sigma(x) \end{cases}
\]

We can naturally lift \( \text{Wrap} \) to a trace \( \sigma \) by replacing every event \( e \in \sigma \) by \( \text{Wrap}(e) \). We will assume that every event in a trace has an associated unique identifier. For every event \( e' \in \text{Wrap}(t_1, \ldots, t_k(e, x)) \), let \( \text{Src}(e') = e \).

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

Let us first present a small technical lemma that relates correct reorderings of a trace and the corresponding wrapped trace \( \text{Wrap}(\sigma) \).

**Lemma 9.** Let \( \sigma \) be a trace and \( \alpha = \text{Wrap}(\sigma) \). Let \( \text{FakeEvents} = \{ e \in \text{Acq}(t_{1, x}), \text{Releases}_\sigma(t_{1, x}) | t \in \text{Threads}_\sigma, x \in \text{Vars}_\sigma \} \). Let \( \alpha' \) be a correct reordering of \( \alpha \). Then, \( \text{Clear}_E(\alpha') \) is a correct reordering of \( \sigma \).

**Proof.** Let \( \sigma' = \text{Clear}_E(\alpha') \).

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

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

We next establish the relationship between different partial orders on the two traces \( \sigma \) and its transformation \( \text{Wrap}(\sigma) \).

**Lemma 10.** Let \( \sigma \) be a trace and \( e_1, e_2 \in \text{Events}_\text{Wrap}(\sigma) \). If \( e_1 \preceq_{\text{HB}} \text{Wrap}(\sigma) e_2 \), then \( \text{Src}(e_1) \preceq_{\text{HB}} \text{Src}(e_2) \).

**Proof.** Let \( e_1, e_2 \in \text{Events}_\text{Wrap}(\sigma) \) such that \( e_1 \leq_{\text{HB}} \text{Wrap}(\sigma) e_2 \). Then, there is a path
\[
e_1 = f_0 \leq_{\text{HB}} \text{Wrap}(\sigma) f_1 \leq_{\text{HB}} \text{Wrap}(\sigma) f_2 \leq_{\text{HB}} \text{Wrap}(\sigma) f_n = e_2
\]
such that for every \( 0 \leq i \leq n - 1 \), either \( f_i \preceq_{\text{HB}} f_{i+1} \) or \( f_i = (t_i, \text{rel}(t_i)) \) and \( f_{i+1} = (t_{i+1}, \text{acq}(t_{i+1})) \). In the first case, we have \( \text{Src}(f_i) \preceq_{\text{HB}} \text{Src}(f_{i+1}) \) and thus \( \text{Src}(f_i) \leq_{\text{HB}} \text{Src}(f_{i+1}) \). In the latter case, if \( f_i \) is not a fake lock, then \( \text{Src}(f_i) = f_i \) and...
src(f_{i+1}) = f_{i+1} and thus src(f_i) \leq_{\text{CHB}} src(f_{i+1}). Otherwise, we have that src(f_i) \neq src(f_{i+1}) and thus src(f_i) \leq_{\text{CHB}} src(f_{i+1}). Then, from the transitivity of \leq_{\text{CHB}}, it follows that src(e_1) \leq_{\text{CHB}} src(e_2).

\begin{lemma}
Let \sigma be a trace and e_1, e_2 \in \text{Events}_{\text{wrap}(\sigma)}. If e_1 \sim_{\text{wrap}(\sigma)} e_2 then src(e_1) \leq_{\text{DCP}} src(e_2).
\end{lemma}

\begin{proof}
If e_1 \not<_{\text{wrap}(\sigma)} e_2, then clearly, src(e_1) \not<_{\TO} src(e_2) and thus src(e_1) \leq_{\text{DCP}} src(e_2). Otherwise, we have e_1 \not\sim_{\text{WCP}} e_2. We will induct on the rank of \leq_{\text{WCP}} edges.

\begin{itemize}
\item Case e_1 \in \text{Releases}_{\text{wrap}(\sigma)}(t), and there is an e'_1 \in CS(e_1) such that e'_1 \not< e_2 and e_2 \in CS(t).
\end{itemize}

If e_1 is a fake lock, then src(e_1) = e'_1 \times e_2 = src(e_2) and thus src(e_1) \leq_{\text{DCP}} src(e_2). Otherwise, src(e_1) = e_1 \in \text{Releases}_{\sigma}(t), and, in the trace \sigma, we have e'_1 \in CS(e_1) and e_2 \in CS(t), and thus e_1 \leq_{\text{DCP}} e_2 by rule (b) of \leq_{\text{DCP}}.

\begin{itemize}
\item Case e_1, e_2 \in \text{Releases}_{\sigma}(t) and there are events e'_1, e'_2 \in \text{Events}_{\text{wrap}(\sigma)} such that e'_1 \in CS(e_1), e'_2 \in CS(e_2), and the WCP edge (e'_1, e'_2) \not\sim_{\text{wrap}(\sigma)} has a lower rank.
\end{itemize}

Our inductive hypothesis ensures that src(e'_1) \leq_{\text{DCP}} src(e'_2). Now, if t is a fake lock, then src(e_1) = src(e'_1) and src(e_2) = src(e'_2) and thus we have src(e_1) \leq_{\text{DCP}} src(e_2). Otherwise, in trace \sigma, we have src(e'_1) \in CS(e_1) and src(e'_2) \in CS(e_2). Thus, by rule (c) of \leq_{\text{DCP}}, we have src(e_1) \leq_{\text{DCP}} src(e_2).

\begin{itemize}
\item Case HB-composition
\end{itemize}

- There is an event e_3 \in \text{Events}_{\text{wrap}(\sigma)} such that e_1 \leq_{\text{HB}} e_3 and the WCP edge e_3 \sim_{\text{wrap}(\sigma)} e_2 has a lower rank.

By Lemma 10, we have src(e_1) \leq_{\text{CHB}} src(e_3). Also, by induction hypothesis, we have src(e_3) \leq_{\text{DCP}} src(e_2). Thus, we have src(e_1) \leq_{\text{DCP}} src(e_2) by \leq_{\text{CHB}} composition.

- There is an event e_3 \in \text{Events}_{\text{wrap}(\sigma)} such that e_1 \leq_{\text{HB}} e_3 and the WCP edge e_3 \sim_{\text{WCP}} e_2 has a lower rank.

Similar to the previous case.

\end{proof}

\subsection{A.2 Adding a fake race in \text{wrap}(\sigma)}

In the trace \text{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 \text{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 \text{wrap}(\sigma) are in WCP race.

So let us first define some useful notation here.

\textbf{Definition 7.} Let \sigma be a trace and let e_1, e_2 \in \text{Events}_\sigma be events such that \sigma ||_{\text{WCP}} e_2 and \text{locksHeld}_\sigma(e_1) \cap \text{locksHeld}_\sigma(e_2) = \emptyset. Then, e'_1 ||_{\text{WCP}} e'_2, where d is a fresh variable for \sigma, \rho = \text{addFresh}(\sigma, e_1, e_2, d) and the events e'_1 = (t_1, w(d)), e'_2 = (t_2, w(d)) \in \text{Events}_\sigma are the newly added write events.
Proof. Without loss of generality, assume $e_1 \preceq_{\sigma}^p e_2$ and thus $e'_1 \preceq_{\sigma}^p e'_2$. Let us on the contrary assume that $e'_1 \preceq_{\text{WCP}}^p e'_2$.

If $e'_1 \prec_{\text{TO}} e'_2$, then $e_1 \prec_{\text{TO}} e_2$, which contradicts $e_1 \parallel_{\text{WCP}} e_2$. Otherwise, $e'_1 \preceq_{\text{WCP}} e'_2$. Thus, there is a path
$$e'_1 = e_0, e_1, \ldots, e_n = e'_2,$$

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

Now consider the HB path
$$e'_1 = g_0 \preceq_{\sigma}^p g_1 \cdots \preceq_{\sigma}^p g_k = f_1$$

where for each $0 \leq i \leq k$, we have either $f_i \prec_{\text{TO}} f_{i+1}$ or $f_i, f_{i+1}$ are $rel(l)-acq(l)$ 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 $e_1 \prec_{\text{TO}} g_1$ because $e_1$ is the event right after $e'_1$ in $thr(e'_1)$. And thus, we have $e_1 \preceq_{\text{HB}} f_1$ and thus $e_1 \preceq_{\text{WCP}} e'_2$.

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 $locks Held_\rho(e'_1) \cap locks Held_\rho(e'_2) = \emptyset$. Thus, $(f_{n-1}, f_n)$ is an $\preceq_{\text{HB}}$ edge.

So we have, $e_1 \preceq_{\text{WCP}} f_{n-1}$ and thus, $e_1 \preceq_{\text{WCP}} f_{n-1}$ because of Claim 12.

This clearly gives $e_1 \prec_{\text{WCP}} e_2$ which is a contradiction. □

Corollary 14. Let $\sigma$ be a trace and let $e_1, e_2 \in Events_\sigma$ be events such that $locks Held_\sigma(e_1) \cap locks Held_\sigma(e_2) = \emptyset$. Let $d$ be a fresh variable for $\sigma$, and let us denote by $\rho = addFresh(\sigma, e_1, e_2, d)$ the trace obtained by adding events $e'_1 = \langle thr(e_1), w(d) \rangle$ and $e'_2 = \langle thr(e_2), w(d) \rangle$ right above $e_1$ and $e_2$ respectively. If $e_1 \parallel_{\text{WCP}} e_2$ then $(e'_1, e'_2)$ is a WCP race in $\rho$.

We are now ready to prove the main theorem.

Theorem 15. Let $\sigma$ be a trace and let $e_1, e_2 = \langle t, \text{acq}(\ell_1) \rangle, e_2 = \langle t', \text{acq}(\ell_2) \rangle \in Events_\sigma$ such that $\ell_1 \neq \ell_2$. If $\ell_2 \in locks Held_\sigma(e_1)$, $\ell_1 \in locks Held_\sigma(e_2)$, $locks Held_\sigma(e_1) \cap locks Held_\sigma(e_2) = \emptyset$ and $e_1 \parallel_{\text{WCP}} e_2$, then $\sigma$ has a predictable deadlock.

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

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

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

1. there is a correct reordering $\rho'$ of $\rho$ that has a deadlock
2. there is a correct reordering $\rho'$ of $\rho$ of the form $\rho'' e_1' e_2' \rho'' e_1' e_2'$.

In both the above cases, we will show that the original trace $\sigma$ has a predictable deadlock.

1. Case Predictive deadlock.

Here we have a correct reordering $\rho'$ of $\rho$ that exhibits a deadlock. First note that $\rho'' = \text{clear}(e_1', e_2')$ is a correct reordering of $\rho''$ and thus of $\rho$. Therefore, by Lemma 9, the trace $\sigma' = \text{clear}(\text{fakeEvents}(\rho''))$ is a correct reordering of $\sigma$, where $\text{fakeEvents} = \{ e \mid \text{Acquires}_\rho(\ell_{i,x}) \cup \text{Releases}_\rho(\ell_{i,x}) \mid \ell \in Threads_\sigma, x \in Vars_\sigma \}$.

Let us argue that $\sigma'$ has a deadlock. Let the witness to the deadlock in $\rho''$ be events $f_1, f_2 \ldots f_k$ such that $f_1 \prec_{\text{TO}} f_2 \ldots f_k$ and (i) $f_i = \langle t_i, \text{acq}(\ell_i) \rangle$, where $\ell_i = \langle t_{i-1}, \text{acq}(\ell_{i+1}) \rangle$ 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, $f_i \notin \text{fakeEvents}$ 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, $f_i \in \text{Events}_\rho$ for every $1 \leq i \leq k$. Also, match($f_i) \notin \sigma'$ because $match(f_i) \notin \rho'$. Finally, next$_\sigma(t_i) = next_{\rho''}(t_i) = \langle t_j, \text{acq}(\ell_{i+1}) \rangle$. Thus, $\sigma'$ exhibits the same deadlock.

2. Case Predictive race.

In this case, there is a correct reordering $\rho' = \rho'' e_1' e_2'$ of $\rho$, $(i, j \in \{1, 2\}, i \neq j)$ of $\rho$. In particular, $\rho''$ is also a correct reordering of $\rho$. Let $\sigma' = \text{clear}(\text{fakeEvents}(\rho''))$.

By Lemma 9, is a correct reordering of $\sigma$. Now, let $l_i$ denote the last event in $\ell_i thr(e_i)$, where $i \in \{1, 2\}$. The next $t_1$ event after $l_1$ in trace $\sigma$ is $e_1$. Similarly, the next $t_2$ event after $l_2$ in trace $\sigma$ is $e_2$. Hence, $l_1$ is also the next $t_1$ event in $\sigma'$. Also, $l_2 \in \text{locks Held}_\sigma(l_1)$ and $l_1 \in \text{locks Held}_\sigma(l_2)$.

Thus, $\sigma'$ 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 $V : \text{Threads}_\sigma \rightarrow \mathbb{N}$ mapping each thread of a trace $\sigma$ to a non-negative integer. For a vector time $V$, $V[n/t] = \lambda u \cdot u = t$ then $n$ else $V(t)$. We denote $\bot = \lambda \cdot 0$. 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
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 \( \preceq_{DCP} \). First, the WCP algorithm maintains, for every pair \((t, x)\) of lock \( t \) and memory location \( x \), vector clocks of the form \( L_{t,x}^r \) and \( L_{t,x}^w \). These are used for the rule (a) of WCP that orders critical sections when they contain conflicting events. For \( \preceq_{DCP} \), 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 \( \preceq_{DCP} \) is closed under composition with \( \preceq_{CHB} \), unlike WCP which is closed under composition with HB (happens-before). \( \preceq_{CHB} \) in addition to ordering HB-ordered events, also orders conflicting events. Algorithm 1, hence, uses additional clocks \( H_{t,x}^r \) and \( H_{t,x}^w \) to correctly maintain \( \preceq_{CHB} \) and \( \preceq_{DCP} \). 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) \( \preceq_{DCP} \). To maintain this distinction, our FIFO queues \( Acq_{t,\ell} \) also enqueue the \( \preceq_{TO} \) timestamp \( T_t \) (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 \( \preceq_{WCP} \), one only needs to check if their WCP \( \preceq_{WCP} \) timestamps \( C_e \) and \( C_{e'} \) satisfy \( C_e \subseteq C_{e'} \). On the other hand, for \( \preceq_{DCP} \) ordered, one needs to carefully—\( e \preceq_{DCP} e' \) iff \( C_e \subseteq C_{e'} \).

**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 \( D_x = T_x \cup P_x \) for every event \( e \). This means that the vector clock \( D_x \) need not be maintained explicitly, and can be generated from the clocks \( T_x \) and \( P_x \). Second, we incorporate \( D \) into the style optimization—it is enough to increment the local clocks \( T_t(t) \) and \( H_t(t) \) after a 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 \((t, T_{\ell,t}(t))\) at an acquire event \( e = (t, a, r(t)) \) instead of full vector times \((T_{\ell,t}, D_{\ell,t})\) 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 \( D'(t') \geq P_{r(t')} \) and \( T'(t') > T_{r(t')} \), where \( t' \) is the thread of the acquire event for which Algorithm 1 enqueued the pair \((T', D')\) of vector times. We similarly also observe that instead of maintaining separate vector clocks \( T_{\ell,t}^m \) 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 \( x \) (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 \((e, f, e', f')\) of size 2 with \( f \parallel_{DCP} f' \). To find such patterns, for every lock \( t \), we keep track of a set \( \text{History}(t) \) of pairs \((\text{locksHeld}_t(e), D_e)\), where \( e \in \text{Acquires}_t(f) \). 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 \( f = (t_f, acq(\ell_f)) \), we check if there is a lock \( t \in \text{locksHeld}_t(f) \) and a tuple \((L, D) \in \text{History}(t)\) such that \( t_f \in L \neq \emptyset \neq D, \) and \( L \setminus \{\ell\} \neq \text{locksHeld}_t(f) \setminus \{\ell_f\} = \emptyset \). 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 DeadlockPred using space \( S(n) \). Then \( S(n) = \Omega(n) \).

**Proof of Lemma 16.** Consider the language \( L_n = \{uv | u, v \in \{0,1\}^n \text{ and } u = v\} \). 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 \( u_2u_2 \) 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 DeadlockPred, for every \( n \) — given a string \( w \), we will construct a trace \( \sigma_n \) with constantly many (i.e., independent of \( n \)) locks, threads, and variables such that \( w \in L_n \) if and only if \( \sigma_n \notin \text{DeadlockPred} \); 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 \( w = b_1b_2\cdots b_nb_1c_1c_2\cdots c_n \) that is a candidate input to \( L_n \).

It will be convenient to introduce some notation to denote events that will constitute the trace \( \sigma_n \). The bits \( b_i \) and
The variable $\{p, q, s, u, v\}$ 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.

\[
[p]_w^1 = \langle 1, w(p) \rangle \\
[q]_w^2 = \langle 2, w(q) \rangle \\
[s]_w^3 = \langle 3, w(s) \rangle \\
[u]_w^3 = \langle 3, w(u) \rangle \\
[v]_w^4 = \langle 4, w(v) \rangle
\]

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 $\sigma$. There will be the acquire and release events on locks $\ell$ and $m$ that will form a deadlock pattern. The acquire events on these locks will be named as follows.

\[
[a]_{acq}^1 = \langle 1, acq(\ell) \rangle \\
a_{acq}^2 = \langle 2, acq(\ell) \rangle \\
[a]_{acq}^3 = \langle 3, acq(\ell) \rangle \\
[a]_{acq}^4 = \langle 4, acq(\ell) \rangle
\]

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

\[
f = \langle 1, acq(n) \rangle \\
g = \langle 5, acq(n) \rangle
\]

As before the releases corresponding to $f$ and $g$ will be denoted using the $\text{match}_{\sigma_n}(\cdot)$ predicate. For $n = 5$, the trace $\sigma_5$ is shown in Figure 9; in this trace we have explicitly written all events except $[b_1]_w^2$, $[c_1]_w^3$ and $[c_1]_w^4$, instead of using the notation introduced above. For a
by identifying the last write events of different read events.

\[
\begin{align*}
\text{lw}_{c_n}(p_i^j) &= [p_i^j]_w^r \\
\text{lw}_{c_n}(u_i^j) &= [u_i^j]_w^r \\
\text{lw}_{c_n}(v_i^j) &= [v_i^j]_w^r
\end{align*}
\]

In addition, the last write event for every \([s_j]^3\)-event is the \([s_j]^3\)-event immediately preceding it. Next, notice that the use of \([x_i, x_j]\) to encode \(c_i\) when \(i\) is odd, and \([y_i, y_j]\) when \(i\) is even, means that \(\text{lw}_{c_n}([c_i]^3) = [c_i]^3_w\). Next, observe that the presence of \([v_i]^3_w, \ [s_i]^3_w, \text{ and } [q_i]^3_w \) as the last events of threads 4, 3, and 2, respectively, means that any correct reordering of \(\sigma_n\) that contains the event \([v_i]^3\) also contains all the events of threads 2, 3, and 4.

We will now argue that if \(w \in L_n\), then \(b_i = c_i\) for every \(i\) — then the deadlock pattern formed by events \(([a_i]^3, [a_i]^3_m, [a_i]^3_s, [a_i]^3_r)\) is not schedulable. The reasoning is based on observing that certain events must be ordered in every correct reordering of \(\sigma_n\). The first observation is that \([c_i]^3\) must be before \([a_i]^3\) because of the events \(([v_i]^3_w, [s_i]^3_w)\). Next, the events \(([q_i]^3_w, [q_i]^3_v)\) ensure that \([b_i]^3_w\) must be before \([c_i]^3_w\). Since \(b_i = c_i\) and \(\text{lw}_{c_n}([c_i]^3) = [c_i]^3_w\), we must have \([b_i]^3_w\) before \([c_i]^3_w\). The last pair of \(([s_i]^3_w, [s_i]^3_r)\) events ensure that now we must have \([b_i-1]^3_w\) before \([c_{i-1}]^3_w\). Again, \(b_i-1 = c_{i-1}\) coupled with last write properties, mean that we can conclude that \([b_{i-1}]^3_w\) must be before \([c_{i-1}]^3_w\). This reasoning can be repeatedly applied to conclude that for every \(i\), \([b_i]^3_w\) must be before \([c_i]^3_w\). In particular, that means that \([b_i]^3_w\) is before \([a_i]^3_m\). Now the pair \(([u_i]^3_w, [a_i]^3_m)\) ensure that \([c_i]^3_w\) is before match_{\sigma_n}(g). These observation together with the fact that \(f\) is before \([b_i]^3_w\) (because of the pair \(([p_i]^3_w, [p_i]^3_s)\)) imply that \(f\) is before match_{\sigma_n}(g) in every correct reordering. Since correct reorderings preserve lock semantics, we can strengthen this observation to conclude that match_{\sigma_n}(f) must be before \(g\). Finally, as \([a_i]^3_m\) is inside CS_{\sigma_n}(f), we must have \([a_i]^3_m\) before \(g\) and therefore, before \([a_i]^3_r\). This means that the deadlock pattern is not schedulable.

Let us now show that if \(w \notin L_n\) then the deadlock pattern \(([a_i]^3, [a_i]^3_m, [a_i]^3_s, [a_i]^3_r)\) is schedulable. If \(w \notin L_n\), then for some \(i\), \(b_i \neq c_i\). Consider the following trace \(\sigma\)

\[
\begin{align*}
[c_i]^3_w & \cdot [u_i]^3_w \cdot g \cdot [u_i]^3 \cdot \text{match}_{\sigma_n}(g) \cdot [a_i]^3_m \\
[c_2]^3_w & \cdot [s_i]^3_w \cdot [s_i]^3 \cdot [c_i]^3_r \cdot f \cdot [p_i]^3_w \cdot [p_i]^3_s \\
[b_i]^3_w & \cdot [c_i]^3_w \cdot [s_i]^3_w \cdot [s_i]^3 \cdot [c_i]^3_r \\
[b_2]^3_w & \cdot [c_2]^3_w \cdot [s_2]^3_w \cdot [s_2]^3 \cdot [c_2]^3_r \cdot \cdots \\
[b_1]^3_w & \cdot [c_1]^3_w \cdot [s_1]^3_w \cdot [s_1]^3 \cdot [c_1]^3_r \cdot [b_i-1]^3_w \cdot [b_i-1]^3_s \\
[b_{i-1}]^3_w & \cdot [c_{i-1}]^3_w \cdot [s_{i-1}]^3_w \cdot [s_{i-1}]^3 \cdot [c_{i-1}]^3_r \cdot \cdots \\
[b_j]^3_w & \cdot [c_j]^3_w \cdot [s_j]^3_w \cdot [s_j]^3 \cdot [c_j]^3_r \cdot [b_j-1]^3_w \cdot [b_j-1]^3_s \\
[b_n]^3_w & \cdot [q_n]^3_w \cdot [c_n]^3_w \cdot [s_n]^3_w \cdot [s_n]^3 \cdot [c_n]^3_r \cdot [q_n]^3_s \cdot [v_n]^3_w \cdot [v_n]^3_r \\
\end{align*}
\]

Observe that \(\sigma\) is correct reordering of \(\sigma_n\), and at this point threads 1 and 5 are deadlocked.

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

---

**Figure 9.** Trace \(\sigma_5\) used in the linear space lower bound for streaming algorithms.
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 $O(n)$ 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

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

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 $\sigma_n$ with $\#^n$ “junk” events between match$_\sigma_n(f)$ and $[c_1]^2_\sigma$. This establishes Theorem 4.

## D Algorithm for Incorporating Data Flow

We first prove Lemma 7.

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

To prove that $\tau$ is a correct reordering of $\sigma$ with respect to $\text{DF^br}$, we just need to ensure that certain read events are present and last write of these events is preserved. $\text{DF^br}_\sigma$ is only non-empty for branch-events. Consider a branch event $b \in \text{Events}_\sigma$ and a read-event $e \in \text{RelRds}_b(b)$; without loss of generality, let us assume that $e$ is a $r(x)$-event. Observe that since $b \not\in \text{IrrAcc}_\sigma$ and filter$(\tau, \text{IrrAcc}_\tau) = \rho$, we have that $b \in \text{Events}_\rho$. Thus, $e \in \text{Events}_\rho$. Again since $e \not\in \text{IrrAcc}_\rho$, we have $e \in \text{Events}_\rho$. Next, since $e$ is a relevant read, we have $\text{Writes}_\rho(x) = \text{Writes}_\sigma(x)$. Therefore, we have $\text{lw}_\sigma(e) = \text{lw}_\sigma(e)$. Since $\rho$ is a correct reordering of $\pi$ with respect to $\text{DF}^T$, we have $\text{lw}_\rho(x) = \text{lw}_\pi(x)$. Finally, we again have $\text{Writes}_\tau(x) = \text{Writes}_\rho(x)$, which means that $\text{lw}_\sigma(e) = \text{lw}_\rho(e)$. Putting all these observations together, we have $\text{lwe}_\sigma(e) = \text{lwe}_\rho(e)$. Thus, $\tau$ is a correct reordering of $\sigma$ with respect to $\text{DF^br}_\sigma$. $\square$

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

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

Observe that the following proposition follows trivially from the definitions.

**Proposition 17.** If $e_1 \in \text{Reads}_\sigma$ and $e_1 \preceq_{\text{DF}} e_2$ then $e_1 \in \text{RelRds}_\sigma(e_2)$.

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 $e$, $C^e_\sigma$ will denote the vector timestamp of $e$ that is consistent with $\preceq_{\text{DF}}$. For a thread $t$ and variable $x$, let first$(t, x)$ denote the first event of the form $(t, r(x))$; this maybe undefined, if thread $t$ does not have any $r(x)$-events. As per this notation, $C^\text{DF}_{\text{first}(t, x)}$ is the vector timestamp of first $r(x)$-event in $t$. Next, define $B^\text{DF} = \bigcup_{e \in \text{Branches}_\rho} C^e_\sigma$; observe that for a read event $e$, $C^e_\sigma \subseteq B^\text{DF}$ if and only there is some branch-event $b$ such that $e \preceq_{\text{DF}} b$. Finally, define

$$\text{RelVar}_\sigma(x) \iff \exists r \in \text{ThreadVar}_\sigma. C^r_{\text{first}(t, x)} \subseteq B^\text{DF}.$$ 

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

$$\text{RelAcc}_\sigma = \{e \in \text{Reads}_\sigma | C^e_\sigma \subseteq B^\text{DF}\} \cup \{e \in \text{Writes}_\sigma(x) | x \in \text{RelVar}_\sigma(x)\}. \quad (1)$$

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

1. If $e$ is a data access event that does not belong to $\text{RelVar}_\sigma$ 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.
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){
                }
            }
        }
    }

    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.
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.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
```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. Columns 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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Total</th>
<th>Acquires</th>
<th>Releases</th>
<th>Reads</th>
<th>Writes</th>
<th>Forks</th>
<th>Joins</th>
<th>Branches</th>
<th>Acquires</th>
<th>Releases</th>
<th>Reads</th>
<th>Writes</th>
<th>Forks</th>
<th>Joins</th>
<th>Branches</th>
<th>Goodlock2</th>
<th>DCP</th>
<th>DCPDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deadlock</td>
<td>37</td>
<td>4</td>
<td>4</td>
<td>8</td>
<td>6</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>TrueDeadlock</td>
<td>38</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>PickLock</td>
<td>50</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Dining</td>
<td>58</td>
<td>4</td>
<td>4</td>
<td>10</td>
<td>8</td>
<td>2</td>
<td>0</td>
<td>17</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Bensalem</td>
<td>59</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Transfer</td>
<td>62</td>
<td>4</td>
<td>4</td>
<td>13</td>
<td>12</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>FalseDeadlock</td>
<td>651</td>
<td>6</td>
<td>6</td>
<td>206</td>
<td>205</td>
<td>3</td>
<td>3</td>
<td>202</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>DBCP1</td>
<td>1.78K</td>
<td>19</td>
<td>17</td>
<td>557</td>
<td>559</td>
<td>2</td>
<td>0</td>
<td>279</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Derby2</td>
<td>1.93K</td>
<td>7</td>
<td>4</td>
<td>694</td>
<td>692</td>
<td>2</td>
<td>0</td>
<td>253</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Log4j2</td>
<td>2.23K</td>
<td>20</td>
<td>16</td>
<td>757</td>
<td>392</td>
<td>3</td>
<td>0</td>
<td>507</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>DBCP2</td>
<td>2.67K</td>
<td>22</td>
<td>20</td>
<td>906</td>
<td>485</td>
<td>2</td>
<td>0</td>
<td>652</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LongDeadlock</td>
<td>6.03K</td>
<td>4</td>
<td>4</td>
<td>2.0K</td>
<td>2.0K</td>
<td>2</td>
<td>0</td>
<td>2.0K</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>HashMap</td>
<td>45.52K</td>
<td>1.09K</td>
<td>1.09K</td>
<td>13.66K</td>
<td>9.2K</td>
<td>338</td>
<td>0</td>
<td>7.93K</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>lusearch-fix</td>
<td>304.02M</td>
<td>206.4K</td>
<td>206.4K</td>
<td>151.53M</td>
<td>40.98M</td>
<td>8</td>
<td>0</td>
<td>70.84M</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pmd</td>
<td>6.6M</td>
<td>54</td>
<td>54</td>
<td>3.27M</td>
<td>600.21K</td>
<td>0</td>
<td>0</td>
<td>1.84M</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>fop</td>
<td>24.48M</td>
<td>1.22K</td>
<td>1.22K</td>
<td>9.72M</td>
<td>1.9M</td>
<td>0</td>
<td>0</td>
<td>7.69M</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>lunitest</td>
<td>26.74M</td>
<td>338</td>
<td>338</td>
<td>13.97M</td>
<td>2.69M</td>
<td>0</td>
<td>0</td>
<td>7.54M</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>tomcat</td>
<td>30.64M</td>
<td>3.53K</td>
<td>3.52K</td>
<td>16.78M</td>
<td>2.11M</td>
<td>41</td>
<td>0</td>
<td>9.49M</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>batik</td>
<td>63.63M</td>
<td>22.47K</td>
<td>22.47K</td>
<td>27.15M</td>
<td>11.62M</td>
<td>1</td>
<td>0</td>
<td>18.97M</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>eclipse</td>
<td>104.75M</td>
<td>281.55K</td>
<td>281.55K</td>
<td>42.96M</td>
<td>6.65M</td>
<td>14</td>
<td>3</td>
<td>42.51M</td>
<td>226</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>xalan</td>
<td>203.76M</td>
<td>447.16K</td>
<td>447.16K</td>
<td>104.36M</td>
<td>15.03M</td>
<td>10</td>
<td>10</td>
<td>55.35M</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>jython</td>
<td>242.99M</td>
<td>4.1M</td>
<td>4.1M</td>
<td>72.3M</td>
<td>14.0M</td>
<td>1</td>
<td>0</td>
<td>65.18M</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>avrora</td>
<td>1.45B</td>
<td>1.07M</td>
<td>1.07M</td>
<td>677.5M</td>
<td>207.07M</td>
<td>3</td>
<td>3</td>
<td>252.0M</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Figure 13. Performance of DeadTrack and Dirk on traces generated by LongDeadlock by varying number of iterations.