PICKLOCK: A Deadlock Prediction Approach under Nested Locking

Francesco Sorrentino

University of Illinois at Urbana-Champaign
sorrent1@illinois.edu

Abstract. We study the problem of determining whether from a run of a concurrent program, we can predict alternate deadlocking executions of it. We show that if a concurrent program adopts *nested locking*, the problem of predicting deadlocks is efficiently solvable without exploring all interleavings.
In this work we present a fundamentally new predictive approach to detect deadlocks in concurrent programs, not based on cycle detection in lock-graphs. The idea is to monitor an arbitrary run of a concurrent program, use it to predict alternate runs that could be deadlocking, and reschedule them accurately. We implement our prediction algorithm in a tool called PICKLOCK, which is a modular extension of the PENELOPE framework [33].
We show experimentally that PICKLOCK scales well and is effective in predicting deadlocks. In particular, we evaluate it over 13 benchmark concurrent programs and find about 11 deadlocks by using only a single test run as the prediction seed for each benchmark.

1 Introduction

A common cause for unreactiveness in concurrent programs is deadlocked configurations. Deadlocks in a shared-memory concurrent program are unintended conditions that can be mainly classified into two types: *resource-deadlocks* and *communication deadlocks*. A set of threads is resource-deadlocked if each thread deadlocked is waiting for a resource, like a lock, held by another thread in the set, which forms a cycle of lock requests. In communication deadlocks some threads wait for messages that do not get sent because the sender threads are blocked or they have already sent the messages before receiving threads start to wait. In [23] the authors illustrate that it could be really hard to precisely detect all kinds of deadlocks by the same techniques. In this study we focused only on resource deadlocks, from now on referred to as deadlocks.
Deadlocks are very common in concurrent programs— Lu et al. [27] showed that a relevant number of errors (about 30%) found in a characteristic study of concurrency bugs on a collection of software systems can be attributed to deadlocks. Moreover, avoiding other concurrency problems like races and atomicity-violations often involves introducing new synchronizations, which in turn can introduce new deadlocks [27, 28].
Deadlocks often occur under subtle interleaving patterns that the programmer has not taken into consideration. There are too many interleavings to test for, even for a single concurrent program on a single input, making concurrency testing a difficult problem. With the rise of multicore hardware platforms, finding solutions to this problem is very important, as testing is still the most effective way of finding bugs today. Current testing technologies such as stress testing are inadequate in exposing such subtle interleavings.
In this study, we focus on *prediction techniques for discovering deadlocks*— we observe an

arbitrary execution of a concurrent program and from it predict alternate interleavings that can deadlock. We show that if a concurrent program adopts *nested locking* policies (i.e., locks are released by threads in the reverse order in which they were acquired), the problem of predicting potential deadlocks involving any number of threads is efficiently solvable without exploring all interleavings. Nested locking is guaranteed on *Java 1.7* (in general with Java using *synchronized* blocks or methods) and *C#*.

Potential deadlocks can be detected using dynamic analysis [10,14,15,24,28], model checking [18,22], runtime monitoring [37], static analysis [19,29,32,38] or a combination thereof. Analysis based on lock order graphs [1] or a combination of them with happen-before relation has been already explored [15].

In general, static detection techniques and model checking for deadlock detection can analyze the whole program, but lacks precision and usually result in a large number of false positives. Precision is gained using dynamic techniques, but at the cost of incompleteness. Dynamic confirmation techniques work well in confirming if a potential deadlock is a real one. Recent deadlock detection techniques [24, 28] use lock-set based strategies to predict potential deadlocks. Unfortunately, the re-execution phase they provide is weak, largely because such re-execution phases are based on time triggered approaches.

The solution we propose for predicting potential deadlocks and for confirming them is not based on the simple cycle lock requests detection like most of the deadlocks detection work available in the literature [1, 13, 14, 19, 25, 35, 36, 38]. It instead involves taking a concurrent program and a test harness, executing the program under test to get an arbitrarily interleaved execution, and then *predicting* alternate executions leading to deadlocks. Finally, in order to check if a real deadlock has been found, the program being tested is re-executed precisely under these predicted deadlocking schedules.

The main contribution of this paper is the prediction algorithm, which reasons at an abstract level in order to efficiently and accurately predict deadlocking schedules. The algorithm is based on *lock-sets* and *acquisition histories* (the latter are a kind of hierarchical lock-set information), which only ensure that the predicted run respects *lock* acquisitions and releases in the run. In other words, the predicted runs are certainly not guaranteed to be feasible in the original program— if the original program had threads that communicated through shared variables in a way that orchestrated the control flow, the predicted runs may simply not be feasible. However, in the absence of such communication, the predicted runs do respect the locking semantics and hence assure feasibility at that level of abstraction. To realize a more precise prediction we could have used a more sophisticated mechanism, such as the one that uses constraint solvers we proposed in [20]. However, we decided to pursue scalability and use a more lightweight solution here. The crucial observation is that acquisition histories give not only enough traction to detect alternate deadlocking interleavings, but also provide an effective mechanism to *re-schedule* the precise interleaving under which deadlock will occur; the latter helps our re-execution engine to run the predicted schedule and confirm the deadlock, which entirely eliminates all false positives.

We have implemented this methodology in a tool, PICKLOCK, that monitors and reschedules interleavings for Java programs. The infrastructure of the tool is partially built on the PENELOPE framework, presented in [33]. We have applied PICKLOCK to a suite of multi-threaded Java programs and showed that it is efficient and effective in predicting deadlocking schedules. The methodology we present is language-independent and can be applied to other contexts as long as nested locking policies are used.

2 Related Work

Prior work on deadlock detection in concurrent programs have exploited different techniques: dynamic (including postmortem) analysis, model checking and static analysis.
Static approaches attempt to detect possible deadlocks directly on the source code and do not require the execution of the application being tested [12, 19, 29, 32, 38]. Even if this approach exhaustively explores all potential deadlocks, it suffers from high false positives, aggravating the user. For example, Williams et al. in [38] report that on 100,000 potential deadlocks only 7 were real deadlocks. In order to reduce the number of false positives numerous directions have been explored. Williams et al. [38] have used heuristics to try to remove some of the false positives but these have the potential of removing some real deadlocks. von Praun [35] uses a context-sensitive lock-set and a lock graph in his approach. To reduce false positives they suppress certain deadlocks based on lock alias set information, again potentially removing real deadlocks. `RacerX` [19] is a static data race and deadlock detection tool for concurrent programs written in C. Additional programmer's annotations are used to inject the programmer's intent and consequently suppress false positives and improve the `RacerX`'s accuracy. More recently, Naik et al. [29] combine a suite of static analysis techniques to cut the false positive rates. Unfortunately, scalability and problems related to conditional statements still remain a drawback of static analysis.
Several researchers have explored a model-checking approach to detect deadlocks in concurrent programs using model checker such as `SPIN` and `Java Pathfinder` [18, 22, 34]. Joshi et al. [23] monitor the annotated conditional variables as well as lock synchronization and threading operations in a program to generate a trace program containing not only thread and lock operations but also the value of conditionals. Then they apply `Java Pathfinder` to check all abstracted and inferred execution paths of the trace program to detect deadlocks. However, the technique proposed requires manual effort to design and add annotations, which can be error-prone, and suffers from the scalability issue to handle largescale programs. Dynamic analysis techniques have been extensively explored as well [10, 14, 15]. With this approach the execution of one or more runs of a concurrent program are monitored to determine if a deadlock may occur in another execution of the program. Eraser [30] is a dynamic analysis program originally designed to detect possible data-races and later modified to detect possible deadlock conditions. Eraser is neither a sound nor complete algorithm. Bensalem et al. [14, 15] find potential deadlocks in Java programs by finding cycles in the lock graph generated during program execution. All cycles in the lock graph are considered to be potential deadlocks, generating a considerable high number of false positives. They use the happen-before relation to improve the precision of cycle detection and use a guided scheduler to confirm a deadlock. Farchi et al. [10] proposed an approach where they generate a lock graph across multiple runs of the code. Deadlocks prediction is done searching cycles in this graph; unfortunately, this approach may also produce false alarms. `MulticoreSDK` [28] and `DeadlockFuzzer` [24] use lock-set based strategies to predict potential deadlocks. Once a potential deadlock has been found, deadlock confirmation, avoidance, or healing strategies can be applied. Neither approaches are capable of completing large executions, moreover the rescheduling phase is not robust enough to guarantee that the right time to trigger a deadlock is used.
`MagicFuzzer` [16] is a dynamic resource deadlock detection technique based on locks dependencies. It locates potential deadlock cycle from an execution, it iteratively prunes lock dependencies that have no incoming or outgoing relations with other lock dependencies. Similarly to our approach it has a trace recording phase, a potential deadlock detection phase and a deadlock confirmation phase — which avoids false positives. However, like

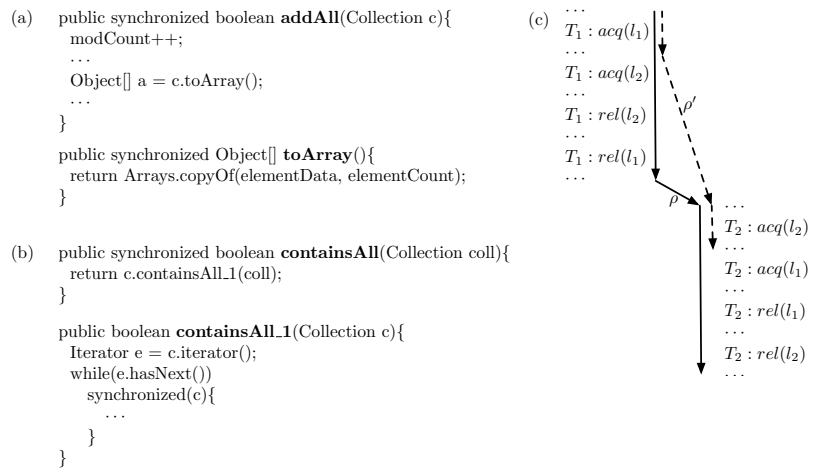

Fig. 1. (a) – Simplified code for $addAll$ and $toArray$ of the $Vector$ library of Java 1.4. (b) – Simplified code for $containsAll$ and $containsAll\_1$ of the $Collections$ library of Java 1.4. (c) – Observed execution ρ of the program under test (dotted arrows indicate the predicted run ρ' generated from ρ).

most of the techniques based on cycle detection, the detection phase is not precise, even assuming that the communication between threads occurs only through locks, overloading the confirmation phase.

`ConLock` [39] is the most recent work that implements predictive detection techniques following our same motivations. `ConLock` analyzes a given cycle and the execution trace that produce the cycle. It generates a set of constraints and a set of nearest scheduling points. Then, it schedules a confirmation run with the aim to not violate a reduced set of constraints from the chosen nearest scheduling points. `ConLock` is able to detect false positives, however the confirmation phase still relies on probability ($> 71\%$).

3 Motivating Example

It is very common to incur a deadlock when programs misuse APIs offered by third-party libraries [25, 38]. Even if the program does not contain logic bugs per se, the interaction of methods defined in synchronized classes may still result in deadlocks. This is a general problem for all synchronized Collection classes in the `JDK`, including the `Vector` library. Since `Vector` is a synchronized class, programmers could easily assume that concurrent accesses to vectors are not a concern. However, potential deadlocks could still be present and hidden from the calling application.

In Figure $1(a)$, we show a simplified version of the methods $addAll$ and $toArray$ as defined in the `JDK` library. $addAll$ appends all of the elements in the specified collection c to the end of $this$ Vector. Internally it calls the method $toArray$, which returns an array containing all the elements in $this$ Vector in the correct order. As the $addAll$ needs to be multi-thread safe, it follows that locks for the vector being added to and the parameter need to be acquired. Specifically, the method acquires the lock associated with the vector being added to first, and then it acquires the lock associated with the parameter. Similarly, in Figure $1(b)$, we report a simplified version of the methods $containsAll$ and $containsAll\_1$ (in AbstractCollection.java) as defined in the `JDK`. $containsAll$ acquires the lock associ-

ated with $this$ vector first, then from inside the method $containsAll\_1$ it acquires the lock associated with the specified collection c.

Let us assume that in the program under test there are two threads that execute concurrently and use two vectors V_1 and V_2. T_1 wants to add all elements of V_2 to V_1, calling the method V_1.addAll(V_2), while T_2 concurrently invokes the method V_2.containsAll(V_1), in order to check if the vector V_2 contains all the elements of V_1.
In Figure $1(c)$, we report a possible observed run ρ (focus on the solid arrows) of the program under test. We are assuming that the code of T_1 is entirely executed followed by the code of T_2 (this execution does not deadlock). Our prediction algorithm observes the synchronization events (such as locks acquire/release) but suppresses the semantics of computations entirely and does not observe them. They have been replaced by "..." symbols in the figure as they play no role in our analysis.
Given the observed run ρ, we ask whether there exists an alternative run ρ' in which a deadlock potentially occurs. Our prediction algorithm will predict a run ρ' in which the acquisition of the lock associated with V_1 (let us say l_1) by T_1 is followed by the acquisition of the lock associated with V_2 (let us say l_2) done by T_2 (illustrated by the dotted arrows in the Figure $1(c)$).
Once a potential deadlocking run is found, in the last phase of PICKLOCK, our re-execution engine will orchestrate the execution of the program under test to follow the predicted run. The program under test will then deadlock inside the `JDK`, producing a concrete deadlocking interleaving.

4 Preliminaries

In this Section we introduce some notations that will be used in the rest of the paper. Then, we elaborate on some observations that motivate our various design choices.

4.1 Prediction model

We assume an infinite set of thread identifiers $\mathcal{T} = \{T_1, T_2, \ldots\}$ and an infinite set of global locks $\mathcal{L} = \{l_1, l_2, \ldots\}$, used in a nested fashion (i.e. threads release locks in the reverse order in which they were acquired).
`PickLock` observes three kinds of actions for a given thread T_i, defined as:

$$\Sigma_{T_i} = \{T_i{:}\,\mathit{acq}(l),\ T_i{:}\,\mathit{rel}(l) |\ l \in \mathcal{L}\} \cup \{T_i{:}\ tc\ T_j \mid T_j \in \mathcal{T}\}$$

Action $T_i{:}\,acq(l)$ represents acquiring the lock l and the action $T_i{:}\,rel(l)$ represents releasing of the lock l, by thread T_i. The action $T_i{:}\ tc\ T_j$ denotes the thread T_i creating the thread T_j. We define $\Sigma = \bigcup_{T_i \in T} \Sigma_{T_i}$ as the set of actions of all threads. A word w in Σ^*, in order to represent a run, must satisfy several syntactic restrictions, represented by the following definitions. ($\sigma|_A$ denotes the word σ projected to the letters in A).

Definition 1 (Lock-validity). *A run $\rho \in \Sigma^*$ is* lock-valid *if it respects the semantics of the locking mechanism. Formally, let $\Sigma_l = \{T_i{:}\,\mathit{acq}(l), T_i{:}\,\mathit{rel}(l) \mid T_i \in \mathcal{T}\}$ denote the set of locking actions on lock l. Then ρ is lock-valid if for every $l \in \mathcal{L}$, $\rho|_{\Sigma_l}$ is a prefix of*

$$\left[\bigcup_{T_i \in \mathcal{T}} (T_i{:}\,acq(l)\ T_i{:}\,rel(l))\right]^*$$

Definition 2 (Creation-validity). *A run $\rho \in \Sigma^*$ over a set of threads $\mathcal{T}$ is* creation-valid *if every thread is created at most once and its events happen after this creation, i.e., for every $T_i \in \mathcal{T}$, there is at most one occurrence of the form $T_j{:}\ tc\ T_i$ in w, and, if there is such an occurrence, then all occurrences of letters of Σ_{T_i} happen after this occurrence.*

Let ρ be a global execution, and $e = (T, i)$ be an event in $\{\rho_T\}_{T \in \mathcal{T}}$. Then we say that the j'th action ($1 \leq j \leq |\rho|$) in ρ *is the event* e (or, $Event(\rho[j]) = e = (T, i)$), if $\rho[j] = T{:}a$ (for some action a) and $\rho_T[1, i] = \rho[1, j]|_T$. In other words, the event $e = (T, i)$ appears at the position j in ρ in the particular interleaving of the threads that constitutes ρ. Reversely, for any event e in $\{\rho_T\}_{T \in \mathcal{T}}$, let $Occur(e, \rho)$ denote the (unique) j ($1 \leq j \leq |\rho|$) such that the j'th action in ρ is the event e, i.e $Event(\rho[j]) = e$. Therefore, we have $Event(\rho[Occur(e, \rho)]) = e$, and $Occur(Event(\rho[j])) = j$. Finally, let $Tid(e, \rho)$ denote the thread $T \in \mathcal{T}$ executing the event e in ρ.

While the run ρ defines a total order on the set of events in it $(E, \leq)$, there is an induced total order between the events of each thread. We formally define this as $\sqsubseteq_i$ for each thread T_i, as follows: for any $e_m, e_n \in E$, if e_m and e_n belong to thread T_i and $m \leq n$ then $e_m \sqsubseteq_i e_n$. The partial order that is the union of all the program orders is $\sqsubseteq = \cup_{T_i \in \mathcal{T}} \sqsubseteq_i$.
Given an execution ρ over a set of locks $\mathcal{L}$ and threads $\mathcal{T}$, we would like to *infer* alternative executions ρ' from ρ that deadlock. Our prediction model respect *lock-validity*, *creation-validity* and the program-order of the original run.

Definition 3 (Prediction model [33]). *Let ρ be a run over a set of threads $\mathcal{T}$ and locks $\mathcal{L}$. A run ρ' is* inferred *from ρ if (i) for each $T_i \in \mathcal{T}$, $\rho'|_{T_i}$ is a prefix of $\rho|_{T_i}$, (ii) ρ' is lock-valid, (iii) creation-valid. We will refer to the set of executions inferred from ρ with $Infer(\rho)$.*

Notice that our prediction model is an *abstraction* of the problem of finding alternate executions that are deadlocking in the concrete program. Not all the executions in $Infer(\rho)$ may be valid/feasible in the original program (this could happen if the threads communicate using other mechanisms). In this sense we talk about *potential deadlocks*. A more precise prediction model can be obtained adding to Def. 3 the *data-validity* constraints (for more details we refer the reader to [20, 31]). However, our rescheduling phase takes care of this problem, getting rid of the false positives.

4.2 Lock-sets and acquisition-histories

Let ρ_T indicate the local execution of T. Consider ρ_T (for any T), the *lock-set held after* ρ_T is the set of all locks T holds: $LockSet(\rho_T) = \{l \in \mathcal{L} \mid \exists k.\rho_T[k] = T{:}acq(l),\ there\ is\ no\ j, k < j \leq i\ s.t.\ \rho_T[j] = T{:}rel(l)\}$. The *acquisition history* [26] of the execution of a thread has more nuanced information, and will play a crucial role in both detecting deadlocks and finding re-executions of the program that manifest the deadlocks. The acquisition history of ρ_T records, for each lock l held by T at the end of ρ_T, the set of locks that T acquired (and possibly released) after the last acquisition of the lock l.

Formally, the acquisition history of ρ_T, $AH(\rho_T) : LockSet(\rho_T) \to 2^{\mathcal{L}}$, where $AH_l(\rho_T)$ is the set of all locks $l' \in \mathcal{L}$ such that $\exists k.\rho_T[k] = T{:}acq(l)$ and there is no $j > k$ such that $\rho_T[j] = T{:}rel(l)$ and $\exists h > k.\rho_T[h] = T{:}acq(l')$.
Two acquisition histories $AH_l(\rho_{T_1})$ and $AH_{l'}(\rho_{T_2})$ are said to be *not compatible*, denoted as $AH_l(\rho_{T_1}) \not\sim_c AH_{l'}(\rho_{T_2})$, if there exist two locks l and l' such that $l' \in AH_l(\rho_{T_1})$ and $l \in AH_{l'}(\rho_{T_2})$. They are otherwise said to be *compatible*.

4.3 Relation between co-reachability and deadlock

A result by Kahlon et al. [26] argues that global reachability of two threads communicating via nested locks is effectively and compositionally solvable by extracting locking information from the two threads in terms of acquisition histories. In particular, it states that there

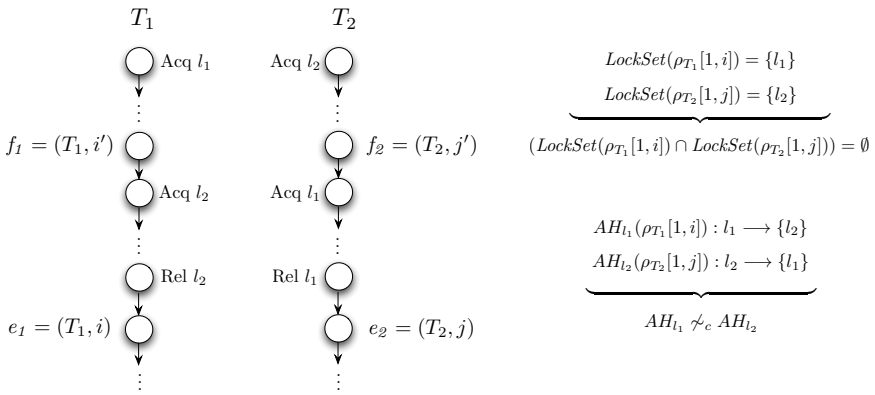

Fig. 2. Lock-sets and acquisition histories associated with a deadlocking configuration of threads T_1 and T_2.

is an execution that ends with event e_1 in one thread and event e_2 in the other thread, if, and only if, the acquisition histories at e_1 and e_2 are compatible and the lock-sets held are disjoint.

Lemma 1 (Kahlon et al. [26]). *Let ρ be an execution of a concurrent program P, let $\{\rho_T\}_{T\in\mathcal{T}}$ be its set of local executions, and let T_1 and T_2 be two different threads. Let $e_1 = (T_1, i)$ be an event of thread T_1 and $e_2 = (T_2, j)$ be an event of thread T_2 of these local executions.*
Then the event e_1 and e_2 of T_1 and T_2 respectively are co-reachable in P
if, and only if,
$LockSet(\rho_{T_1}[1,i]) \cap LockSet(\rho_{T_2}[1,j]) = \emptyset$, and the acquisition history of $\rho_{T_1}[1,i]$ and the acquisition history of $\rho_{T_2}[1,j]$ are compatible.

This pairwise reachability result is the base of our approach and the following Theorem is a direct consequence of it.

Theorem 1. *There is a potential deadlocking run $\rho' \in Infer(\rho)$ involving two threads*
if, and only if,
$\exists\ e_1 = (T_1, i)$, $e_2 = (T_2, j)$ s.t. $LockSet(\rho_{T_1}[1,i]) \cap LockSet(\rho_{T_2}, [1,j]) = \emptyset$ and the acquisition histories of $\rho_{T_1}[1,i]$ and $\rho_{T_2}[1,j]$ are not compatible.

Proof. One side of the implication follows from the Lemma 1. If there is a deadlock, involving T_1 and T_2, it means that we reached an event $f_1 = (T_1, i')$ and an event $f_2 = (T_2, j')$ in the local executions in which the two threads are not allowed to make further operations (Figure 2). From the Lemma 1, it follows that lock-sets at f_1 and f_2 are disjoint and the acquisition histories of $\rho_{T_1}[1, i']$ and $\rho_{T_2}[1, j']$ are compatible. Moreover, because the threads T_1 and T_2 are blocked, the operation that they are trying to do is an acquire of some lock (the release is not a blocking operation).
In particular, T_1 and T_2 are trying to acquire different locks, because if they were trying to acquire the same lock at least one of the threads would have been able to move. The fact that T_1 (resp. T_2) can not make an acquire implies that it is requiring a lock, l_2 (resp. l_1) owned by T_2 (resp. T_1). It follows that at $\rho_{T_1}[1, i' + 1]$ and $\rho_{T_2}[1, j' + 1]$, the lock-sets are not disjoint.

From the nested nature of the locking policies, it follows that there exists a point in ρ_{T_1} in which l_2 is released, that has the same lock-set of f_1, let us say e_1. Similarly, there exists a point in ρ_{T_2} in which l_1 is released, that has the same lock-set of f_2, let us say e_2. It follows that at e_1 and e_2 the lock-sets are disjoint (they were disjoint also in f_1 and f_2). Because T_1 held l_1 while acquired l_2 and T_2 held l_2 while acquired l_1 then l_1 and l_2 are such that the acquisition histories of $\rho_{T_1}[1,i]$ and $\rho_{T_2}[1,j]$ are not compatible, that completes this side of the proof.
It remains to prove that when $\exists e_1, e_2$ satisfying the hypothesis then there exist an event, f_1, executed by T_1 with $Occur(f_1,\rho) < Occur(e_1,\rho)$, and an event, f_2, executed by T_2 with $Occur(f_2,\rho) < Occur(e_2,\rho)$, such that T_1 and T_2 are deadlocked.
Let us pick the e_1 e_2 such that $(Occur(e_1,\rho) + Occur(e_2,\rho))$ is minimal, moreover for the sake of exposition we can assume that there is a unique pair of locks (l_1, l_2) such that the acquisition histories of $\rho_{T_1}[1,i]$ and $\rho_{T_2}[1,j]$ are not compatible. We need to prove that f_1 and f_2 respectively in T_1 and T_2 are co-reachable and deadlocking.
From the assumption it follows that in ρ_{T_1} and in ρ_{T_2}, before that the events e_1 and e_2 are respectively executed, the locks l_1 and l_2 are acquired in reverse order by T_1 and T_2. We can assume that the execution orders are those depicted in Figure 2.
We pick as f_1 the event right before the acquisition of l_2 in ρ_{T_1} and the event right before the acquisition of l_1 in ρ_{T_2} as f_2.
f_1 and f_2 are deadlocking by definition. In order to prove that they are co-reachable, from Lemma 1, we need to prove:

1. $LockSet(\rho_{T_1}[1,i']) \cap LockSet(\rho_{T_2}[1,j']) = \emptyset$.
2. acquisition histories at $\rho_{T_1}[1,i']$ and $\rho_{T_2}[1,j']$ are compatible.

Due to the nested nature of the locking policies, the event f_1 occurs after the acquisition by T_1 (resp. T_2) of l_1 (resp. l_2), belonging to $LockSet(\rho_{T_1}[1,i])$ (resp. $LockSet(\rho_{T_2}[1,j])$). It follows $LockSet(\rho_{T_1}[1,i]) \subseteq LockSet(\rho_{T_1}[1,i'])$ and $LockSet(\rho_{T_2}[1,j]) \subseteq LockSet(\rho_{T_2}[1,j'])$.
By contradiction, let us assume that $LockSet(\rho_{T_1}[1,i']) \cap LockSet(\rho_{T_2}[1,j']) \neq \emptyset$. That is, it exists a lock l_3 such that $l_3 \in LockSet(\rho_{T_1}[1,i']) \cap LockSet(\rho_{T_2}[1,j'])$. But from the inclusions stated above it follows that $l_3 \in LockSet(\rho_{T_1}[1,i]) \cap LockSet(\rho_{T_2}[1,j])$ that contradicts the hypothesis.
We can conclude that the lock-sets are disjoint when T_1 is at $\rho_{T_1}[1,i']$ and T_2 is at $\rho_{T_2}[1,j']$.
It remains to prove the point 2. Let us assume by contradiction that exist l_1 and l_2 such that the acquisition histories at $\rho_{T_1}[1,i']$ and $\rho_{T_2}[1,j']$ are not compatible. We found two events satisfying the hypothesis, moreover $Occur(f_1,\rho) < Occur(e_1,\rho)$ and $Occur(f_2,\rho) < Occur(e_2,\rho)$ it follows that $(Occur(e_1,\rho) + Occur(e_2,\rho))$ was not minimal, contradicting the assumption. □

4.4 The importance of acquisition histories

Potential deadlocks could be detected using a multitude of approaches. The question we want to address in this Section is: *"Why use acquisition histories?"*. The ideal prediction algorithm would predict potential deadlocks that are feasible at least with respect to the synchronization mechanisms.
The majority of the approaches that have been proposed are based on cycle detection in *lock order graphs* [11, 12, 15]. In a lock order graph, a node represents a lock. A directed edge from node l_1 to node l_2 labeled T represents that, during the execution, the thread T acquires the lock l_2 while holding the lock l_1.

For the rest of this Section we consider deadlocks involving only two threads for the sake of exposition. Let us consider a program in which two threads T_1 and T_2 run concurrently and they use four locks l_1, l_2, l_3 and l_4. Given an execution ρ of such program, in Figure 3(a) we report the local executions ρ_{T_1} and ρ_{T_2} and the lock order graph associated with it (b). The classic technique based on cycle detection [11, 12, 15, 16, 24, 28], first constructs a lock order graph. Then it detects whether there are any cycles on the graph (in Figure 3(b) dotted lines are detected cycles). Finally, it tries to trigger the potential deadlock using active scheduling strategies. Another simple algorithm to detect potential deadlocks keeps the set of locks (for each event) held by the thread when an event is executed (similar to what we do), and also keeps the information about *next lock required*. A potential deadlock is found when given two events e_1 and e_2 (executed by two distinct threads), they have disjoint lock-sets and the next lock required at e_1 (resp. e_2) is in the lock-set associated with e_2 (resp. e_1).

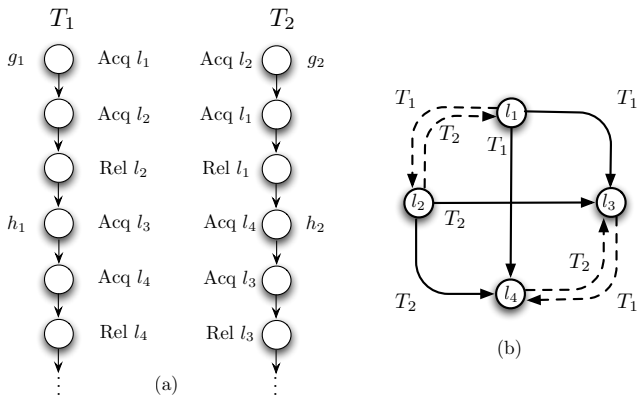

Fig. 3. (a) – A problematic scenario for the *next lock required* and the *cycle detection* approaches. (b) – Lock order graph associated with the execution on the left (dotted lines are detected cycles).

Consider the local executions in Figure 3(a). The next lock required approach will report a potential deadlock at (g_1, g_2). Moreover, it will report a potential deadlock at (h_1, h_2). Notice that these two potential deadlocks are also reported by the lock order graph-based approach. Unfortunately, (h_1, h_2) is not a reachable configuration. MagicFuzzer [16] is the most recent technique based on lock order graph. With respect to its competitors, this approach is more efficient because the size of the graph built is one order of magnitude smaller. It iteratively prunes lock dependencies that each have no incoming or outgoing edge. However, even this approach will wrongly report the two potential deadlocks.

The algorithm we propose is precise and lightweight. Precise because it will report a potential deadlock configuration only when it is feasible (at least respect to the synchronization mechanisms), potentially saving significant aggravation to the user if a manual inspection of the potential deadlocks is required (or if a re-execution phase is provided). It is lightweight because it just tracks lock-sets and some small additional data, acquisition histories. Deadlock prediction algorithms would keep track of at least lock-sets in order to avoid false positive due to *gate* locks. One can develop an algorithm that executes the run till the last lock-free point, and then tries to create the deadlock. However, we still need a mechanism to generate the run with the deadlock. This is what acquisition histories helps us do.

4.5 Concise deadlock prediction

A set of threads is deadlocked if each thread in the set requests a lock, held by another thread in the set, forming a cycle of lock requests. A cycle with n components is a sequence, but

there are n total permutations of the components to represent the same cycle. Detecting one permutation suffices to represent the cycle. In a cycle, each thread can occur only once [16]. We can then use a thread-driven approach to consider only one permutation in place of the whole set of permutations representing the same cycle.
Additionally, the algorithm we propose predicts deadlocks which have the minimal number of threads involved (as contrasted with algorithms based on cycle detection). Our algorithm is run repeatedly, it starts looking for deadlocks involving only two threads. If no deadlocks are found the algorithm proceeds looking for deadlocks involving three threads and so on until a potential deadlock is found or all the possible combinations of threads have been considered. If the potential deadlock predicted is real, this feature will be very helpful for the developer through the debugging process. More details will be discussed in the next Section.

5 Prediction Algorithms

Given an execution ρ with nested locking, we would like to *infer* other executions ρ', containing a deadlock, from ρ. Theorem 1 allows us to engineer an efficient algorithm to predict deadlocking executions. Our algorithm is based on the pairwise reachability, which is solvable compositionally by computing lock-sets and acquisition histories for each thread. In this Section we will consider first the prediction of deadlocks involving only two threads and then the more general case involving any number of threads.

5.1 Deadlocks prediction: 2-threads

The aim of the algorithm is to find two deadlocking events, executed by two distinct threads, given an observed execution ρ. Notice that we are looking for potential deadlocks at this point. These deadlocks may not be feasible in the original program (this could happen if the threads communicate using other mechanisms; for example, if a thread writes a particular value to a global variable which another thread uses to choose an execution path).
The algorithm is divided into three phases. In the first phase, it gathers the lock-sets and acquisition histories by examining the events of each thread *individually*. In the second phase, it tests the compatibility of the lock-sets and acquisition histories of every pair of witnesses f_1 and f_2 in different threads, collected in the first phase. In the third phase from such f_1 and f_2 it rolls back to two co-reachable events, e_1 in T_1 and e_2 in T_2, such that the two threads are deadlocked.
Phase I. In the first phase, the algorithm gathers witnesses for each thread T. The algorithm gathers the witnesses by processing the local executions ρ_T in a single pass. It continuously updates the lock-set and acquisition history, adding events to the set of witnesses, making sure that there are no events with the same lock-set and acquisition history.The set of witnesses, indicated with AH^1_ρ, is a set of 3-tuples $\{((T,i), \mathit{LockSet}(\rho_T[1,i]), AH(\rho_T[1,i])) \mid T \in \mathcal{T}, 1 \leq i \leq |\rho_T|\}$. We use corresponding projection functions $ev(x)$, $ls(x)$ and $ah(x)$ to extract the components from $x \in AH^1_\rho$.
Note that phase I considers every event at most once, in one pass, in a streaming fashion, and hence runs in time linear in the length of the execution.
Phase II. In the second phase, the algorithm checks whether there are pairs of not compatible witnesses collected in the first phase. More precisely, it checks whether, for any pair of threads T_1 and T_2, there is an event f_1 executed by T_1 and an event f_2 executed by T_2

```
1 for each x, y ∈ AH^1_ρ do
2    if Tid(ev(x)) > Tid(ev(y)) then
3       if ls(x) ∩ ls(y) = ∅  ∧  ah(x) ⊀_c ah(y) then
4          pass the pair (x, y) to the third phase;
5 end
```

Fig. 4. Phase II: Prediction Algorithm.

in AH^1_ρ that have disjoint lock-sets and not compatible acquisition histories. The existence of any such pair of events would mean (by Theorem 1) that there is a potential deadlock configuration involving the threads T_1 and T_2.

The algorithm runs the procedure in Figure 4 for finding deadlocks. Notice that the condition $>$ (in place of $\neq$) on line 2 avoids reporting redundant deadlocks, as mentioned in Section 4.5. This reduction is sound because the lock-sets intersection and the acquisition history's compatibility are commutative operations.

Phase III. In the third phase, the algorithm retrieves two deadlocking events f_1, f_2 from a pair of events (e_1, e_2) generated in the second phase. Acquisition histories and the lock-sets of e_1 and e_2 are used to backtrack until an appropriate deadlocking configuration (f_1, f_2) is found. The algorithm stops the backtracking process when the two events f_1 and f_2 have disjoint lock-set and compatible acquisition histories.

In particular, given a pair of events (e_1, e_2) indicating the presence of a deadlock (i.e. e_1 and e_2 have disjoint lock-set and not compatible acquisition histories) we want to retrieve a pair of deadlocking events (f_1, f_2). Let us call a *c*ut-point the pair of events in the execution (f_1, f_2) such that there is an alternate schedule that can reach exactly up to f_1 and f_2 simultaneously. Any schedule that reach exactly up to f_1 and f_2 simultaneously gives a deadlock.

```
   Parameters e1 = (T1, i), e2 = (T2, j)
1  for each l1, l2 s.t. l2 ∈ AH1(l1) and l1 ∈ AH2(l2) with AH1 ∈ AH(ρ_T1[1, i]) and AH2 ∈ AH(ρ_T2[1, j]) do
2     f1 = (T1, i') in ρ_T1 with i' < i and the action performed is T1 : acquire(l2)
3     f2 = (T2, j') in ρ_T2 with j' < j and the action performed is T2 : acquire(l1)
4     if LockSet(ρ_T1[1, i']) ∩ LockSet(ρ_T2[1, j']) ≠ ∅  ∨  AH(ρ_T1[1, i']) ⊀_c AH(ρ_T2[1, j']) then
5        return null;
6     else LockSet(ρ_T1[1, i']) ∩ LockSet(ρ_T2[1, j']) = ∅  ∧  AH(ρ_T1[1, i']) ∼_c AH(ρ_T2[1, j']) then
7        return the cut-point (f1, f2);
8  end
```

Fig. 5. Phase III.

The algorithm to retrieve the cut-point runs the procedure in Figure 5. Essentially T_1 and T_2 are backtracked at the events were the problematic locks were acquired (lines 2-3). Let us assume that T_1 holds l_1 when acquires l_2 at $f_1 = (T_1, i')$. Similarly, T_2 holds l_2 when acquires l_1 at $f_2 = (T_2, j')$.
If $LockSet(\rho_{T_1}[1, i']) \cap LockSet(\rho_{T_2}[1, j']) = \emptyset \ \wedge \ AH(\rho_{T_1}[1, i']) \sim_c AH(\rho_{T_2}[1, j'])$, we have found two co-reachable deadlocking point (line 6) and we return the pair (f_1, f_2) (line 7).

If the f_1 and f_2 are not co-reachable (line 4) then we have two possible cases: (1) $LockSet(\rho_{T_1}[1, i']) \cap LockSet(\rho_{T_2}[1, j']) \neq \emptyset$ or (2) $LockSet(\rho_{T_1}[1, i']) \cap LockSet(\rho_{T_2}[1, j']) = \emptyset$ and $AH(\rho_{T_1}[1, i']) \not\sim_c AH(\rho_{T_2}[1, j'])$. In both cases there is another deadlock involved and consequently in AH^1_ρ there are two events x' and y' related to it, and that will be considered in Phase II. Then we can interrupt the retrieving process (line 5).

5.2 Deadlocks prediction: n-threads

Unfortunately, the lock-set/acquisition history analysis we considered is unable to catch potential deadlocks involving more than two threads. We illustrate the n-threads deadlock prediction using the classical *Dining Philosophers Problem*. Four philosophers, identifiable by a unique id $i \in \{1, 2, 3, 4\}$, sit at a table around a bowl of spaghetti. A fork is placed between each pair of adjacent philosophers. A philosopher i can only eat spaghetti when she has both left and right forks (resp. Fi and $F((i+1) \mod 4)$). When the philosophers get hungry is not deterministic. Each fork can be held by only one philosopher and so a philosopher can use the fork only if it's not being used by another philosopher. When a philosopher i is hungry she tries to acquire her left fork first $F(i)$, and once she obtained that she tries to acquire the right fork $F((i+1) \mod 4)$. After she finishes eating, she needs to put down both the forks so they become available to others. In particular, she puts the right fork down first and then the left fork. The code for the program is adapted from [17] and it is reported in Figure 6.

```
class Philo {
  public static void main(String[] args) {
    Fork F1 = new Fork();
    Fork F2 = new Fork();
    Fork F3 = new Fork();
    Fork F4 = new Fork();
    new Philosopher(1, F1, F2).start();
    new Philosopher(2, F2, F3).start();
    new Philosopher(3, F3, F4).start();
    new Philosopher(4, F4, F1).start();
  }
}

class Philosopher extends Thread {
  int id;
  Fork F1, F2;
  public Philosopher(int i, Fork f1, Fork f2) {
    this.F1 = f1;
    this.F2 = f2;
    this.id = i;
  }
  public void Dine() {
    System.out.println(id);
  }
  public void run() {
    synchronized (F1) {
      synchronized (F2) {
        Dine();
      }
    }
  }
}

class Fork { public int num; }
```

Fig. 6. Dining Philosophers adapted from [17].

Let us consider Figure 7(a). All the acquisition histories associated with the events of T_1, T_2, T_3 and T_4 are pairwise compatible, and the lock-sets are pairwise disjoint. It follows that no potential deadlocks are found in this case. The problem is that the elements in AH^1_ρ are built from single thread information, therefore in order to increase the power of the lock-set/acquisition history analysis, we need a set whose elements synthesize the information of multiple threads. We present in the following the algorithm for the detection of potential deadlocks involving multiple threads.

Let us consider two elements $x, y \in AH^1_\rho$ such that $T_1 = Tid(ev(x))$ and $T_2 = Tid(ev(y))$ are distinct, the intersection of the lock-sets $ls(x)$ and $ls(y)$ is empty and the acquisition histories in $ah(x)$ and $ah(y)$ are compatible.

Definition 4. *We say that x and y can be composed when the following conditions hold:*

- $\exists l, l' \subseteq \mathcal{L}$ *s.t.* $l \in ls(x)$ *and* $l' \in ls(y)$
- l' *is in some acquisition history defined in* $ah(x)$

Before going through the details of the composition, we need to slightly modify the definition of the witnesses set as defined in the previous Section in order to gather composed elements. In particular, the first component of the set AH^1_ρ was the event (T_j, i), where T_j was the thread executing the event. In place of the single thread T_j executing the event now

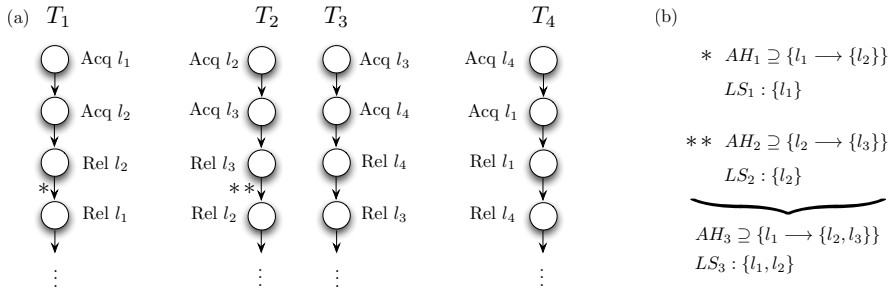

Fig. 7. (a) Local executions ρ_T for the four threads involved in the Dining Philosophers program of Figure 6. (b) Composition of acquisition histories and locks-sets for threads T_1 (at *) and T_2 (at **).

we define a subset of $\mathcal{T}$ containing the thread executing the event. Notice that Theorem 1 still holds if we adapt the requirement of events executed by different threads to require that $Tid(ev(x)) \cap Tid(ev(y)) = \emptyset$.
When two elements x and $y \in AH^1_\rho$ can be composed the resulting element is $z = ((Tid(ev(x)) \cup Tid(ev(y)), (Occur(ev(x), \rho) + Occur(ev(y), \rho))), ls(x) \cup ls(y), merge(ah(x), ah(y)))$.
Figure 7(b) shows how the *merge* procedure, i.e. the composition of acquisition histories, is realized. The intuition behind this is that once we find $x, y \in AH^1_\rho$, with $Tid(ev(x)) = T_1$ and $Tid(ev(y)) = T_2$ respectively, that can be combined; we can assume that all the events executed by T_1 (up to $Occur(ev(x), \rho)$ and T_2 (up to $Occur(ev(y), \rho)$ are executed by a unique thread T'. Then, we collect events, lock-set and acquisition history of such *super thread* T'.
The new set AH^2_ρ is defined as the union of AH^1_ρ and all the elements obtainable by the composition of pairs of elements in AH^1_ρ.

Theorem 2. *There is a potential deadlocking run $\rho' \in Infer(\rho)$ involving 3 (or 4) threads if, and only if,*
$\exists x, y \in AH^2_\rho$ such that $Tid(ev(x)) \cap Tid(ev(y)) = \emptyset$, the LockSets $ls(x)$ and $ls(y)$ are disjoint and the acquisition histories in $ah(x)$ and $ah(y)$ are not compatible.

In order to report only one permutation in place of the whole set of permutations representing the same cycle, we add some restrictions to the composition procedure. In particular, $\forall i \in Tid(ev(x))$ and $\forall j \in Tid(ev(y))$ we require $i < j$.
The result can be generalized for potential deadlocking run involving $n > 1$ threads. In particular, in order to detect potential deadlocks involving n threads where $2^{m-1} \leq n \leq 2^m$ for some integer $m > 1$, requires inductively constructing and analyzing the set AH^m_ρ. A deadlock involving n threads is found building $O(log(n))$ sets.

Dining Philosophers In order to explain our algorithm for the prediction of deadlocks involving more than 2 threads we use the dining philosophers problem introduced in the previous Section. At the first round the algorithm generates the set of witnesses AH^1_ρ. No deadlocks are found in this round because $\forall x, y \in AH^1_\rho$ *LockSets* $ls(x)$ and $ls(y)$ are disjoint and the acquisition histories in $ah(x)$ and $ah(y)$ are compatible.
The algorithm then proceeds with the second round, generating AH^2_ρ. AH^2_ρ, contains all the elements of AH^1_ρ and the elements obtained from the composition of T_1 and T_2, T_2 and

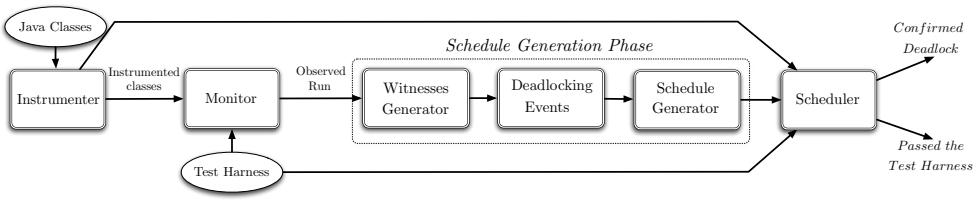

Fig. 8. PICKLOCK's structure.

T_3 and T_3 and T_4. Notice that the elements obtainable from the composition of T_4 and T_1 are discarded by the restriction we introduced at the end of Section 5.2. Moreover, no other compositions exist.

Among the elements of AH^2_ρ there are $x = (\{\{T_1, T_2\}, i), \{l_1, l_2\}, \{l_1 \rightarrow \{l_2, l_3\}, l_2 \rightarrow \{l_3\}\})$ and $y = (\{\{T_3, T_4\}, j), \{l_3, l_4\}, \{l_3 \rightarrow \{l_4, l_1\}, l_4 \rightarrow \{l_1\}\})$. x and y satisfy the Theorem 2, so a potential deadlock is found.

At the moment PICKLOCK does not implement the phases II and III for the case when $n > 2$. PICKLOCK is fully implemented for the two threads deadlock case, and all experiments in Table 1 were conducted using this full implementation of the deadlock prediction algorithms involving exactly 2 threads. We did not implement the $n > 2$ case as we have not found practical examples where this was necessary. We did apply prediction algorithms for deadlocks involving $n > 2$ threads (i.e. only phase I), but did not detect any potential deadlocks in these benchmarks.

6 Implementation

In this Section, we present an overview of the structure of PICKLOCK (depicted in Figure 8). Many of the components are developed upon our previous work, PENELOPE [33]. Here, we will briefly review the monitoring and rescheduling phases as they largely derive from [33]. We then focus on the schedule generation phase that provides the main core of this work and includes the algorithms presented in the Section 5.

Monitoring: We implemented our monitoring instrumenter using the Bytecode Engineering Library (BCEL) [3]. Every class file in bytecode is (automatically) transformed so a *call* to a global monitor is made after each *relevant* action is performed. These relevant actions include thread creations, entry/exit to synchronized blocks, and methods. Thread creation events are used to capture the hard causality constraint of thread creation. Specifically, if a predicted run is not feasible with respect to the thread creation it will be filtered out from the set of predicted runs.

Rescheduling predicted schedules: The scheduler is implemented using BCEL [3]; we instrument the scheduling algorithm into the Java classes using bytecode transformations so that the events that were monitored interact with the scheduler. At each point according to the predicted run, the scheduler orchestrates the appropriate thread to perform an exact sequence of events. Each thread stops at the first relevant event and waits for a signal from the scheduler to proceed. Only then does the thread execute the number of events it was asked to execute. The thread then notifies the scheduler, releases the processor, and waits for further instructions.

Once the execution reaches the potential deadlocking points, the scheduler releases all threads to execute as they please. There is a timeout mechanism after which the status of

the threads is checked and, if a real deadlock is found, PICKLOCK will report it to the user.
Schedule Generation: After finding a pair of deadlocking events (e_1, e_2) (Section 5), PICKLOCK generates alternate schedules that reach these events concurrently and hopefully expose deadlocks. Schedules that reach the deadlocking events concurrently are theoretically possible (indeed the Theorem 1 relies on the existence of such a schedule [21,26]). Our scheduling algorithm guarantees that the predicted deadlock is reproduced if in the original program the threads do not use mechanisms other than locks to communicate. However, note that the rescheduling phase controls only the interleaving of the program's statements, by instrumenting a controller of threads and rewriting the program. External non-determinism caused by the OS, *events*, etc. is not controlled, and it is not controlled by this current PICKLOCK release. Our algorithm synthesizes a schedule that also (heuristically) ensures maximum conformance to the original observed execution. Building schedules that adhere as much as possible to the causal order of the original observed execution is crucial to building feasible schedules— the program under test may have many causal orderings such as communications that need to be respected to ensure feasibility. Experiments in this paper and earlier papers on PENELOPE [20, 21, 33] show that most predicted runs are indeed feasible and validate our choices and algorithms.
In order to achieve such results we use a pruning technique for the runs that removes a large prefix of them while maintaining the property that any run predicted from the suffix will still be feasible. Consider an execution ρ and a pair of deadlocking events $\alpha = (e_1, e_2)$. The idea behind pruning is to first construct the causal partial order of events of ρ and then remove two sets of events from it. The first set consists of events that are *causally after* e_1 and e_2 . The second set is a causally prefix-closed set of events (a configuration) that are *causally before* e_1 and e_2, and in which all the locks are free at the end of execution of this configuration. The intuition behind this is that such a configuration can be replayed in the newly predicted execution precisely as it occurred in the original run, and then stitched to a run predicted from the suffix, since the suffix will start executing in a state in which no locks are held.
Let e_1' and e_2' be the *last* events in T_1 and T_2, respectively, that are before e_1 and e_2, in the local executions, with lock-sets empty. The crux of the scheduling phase is then to schedule from e_1' and e_2' through e_1 and e_2.
The algorithm, borrowed from [33], works by building a graph of *causal edges* between events. For every lock l in the lock-set of e_2, if l occurs in the acquisition history of e_1 with respect to some lock l', then we know that after the last acquisition of l' by T_1 there was an acquisition (followed by a release) of the lock l. Hence we know that we must schedule the last release of lock l' in T_1 (say event e_1'') *before* the last acquisition of l in T_2 (say e_2''). We capture this by adding a causal edge from e_1'' to e_2''. Similarly, we examine the lock-set of e_1 and the acquisition history of e_2 and throw in causal edges. It turns out that since the acquisition histories are compatible, this graph will by *acyclic*, and hence there is a schedule that respects these orderings. The algorithm simply topologically sorts this graph to obtain a schedule (in the implementation, the topological sorting gives preference to the ordering in the original execution— if e_1'' and e_2'' have no causal ordering, we pick e_1'' first if e_2'' occurred before e_1'' in the original schedule).

7 Evaluation

We ran PICKLOCK on a benchmark suite of 13 concurrent programs against several test cases and input parameters. Experiments were performed on an Apple MacBook with 2.4

Application (LOC)	Input	Base Time	Number of Threads	Number of Syncs Events	Number of Observed Events	Time to Monitor	Number of Predicted Runs	Number of Schedulable Predictions	Total Time	Deadlocks Found
Elevator (566)	Data	7.3s	3	6.6K	14K	7.4s	0	-	7.6s	**0**
	Data2	7.3s	5	22K	30K	7.4s	0	-	7.5s	**0**
	Data3	19.2s	5	138K	150K	19.4s	0	-	19.9s	**0**
RayTracer (1.5K)	A-10	5.0s	10	20	648	5.0s	0	-	5.2s	**0**
	A-20	3.6s	20	40	1.7K	4.4s	0	-	4.1s	**0**
	B-10	42.4s	10	20	648	42.5s	0	-	43.1s	**0**
DBCP 1.2.1 (168K)	DBCP-44	1.6s	4	216	1.5K	1.8s	2	1	4.1s	**1**
Vector (1.3K)	VT1	<1s	4	108	525	<1s	1	1	1.0s	**1**
	VT2	<1s	4	12	63	<1s	1	1	1.1s	**1**
Stack (1.4K)	ST1	<1s	4	112	527	<1s	1	1	1.1s	**1**
	ST2	<1s	4	14	69	<1s	1	1	1.3s	**1**
StringBuffer (1.4K)	SBT1	<1s	3	18	82	<1s	1	1	1.2s	**1**
	SBT2	<1s	4	16	75	<1s	1	1	1.1s	**1**
ArrayList (1.6K)	AL	$< 1s$	4	60	783	$< 1s$	1	1	1.2s	**1**
PrintWriter (1.2K)	PW	$< 1s$	4	14	80	$< 1s$	1	1	1.2s	**1**
HashMap (1.3K)	HM1	$< 1s$	4	28	138	$< 1s$	1	1	1.3s	**1**
Java Logging (43K)	$id\_6487638$	<1s	5	228	1.4K	< 1s	2	2	2.3s	**1**
Apache FtpServer (22K)	LGN	60s	4	20	582	1m1s	0	-	1m2s	**0**
Hedc (30K)	Std	1.71s	7	198	774	1.74s	0	-	1.73s	**0**
Weblech v.0.0.3 (35K)	Std	4.91s	3	114	1.6K	4.92s	0	-	4.95s	**0**

Table 1. Experimental results for deadlocks prediction using PICKLOCK.

Ghz Intel Core i5 processors and 4GB of memory, running OS X 10.7.3 and Sun's Java HotSpot 32-bit Client VM 1.5.0.

The benchmarks are all concurrent Java programs that use `synchronized` blocks and methods as means of synchronization. They include `Elevator` from ETH [36], `RayTracer` from the Java Grande MT benchmarks [8], `Vector`, `Stack`, `StringBuffer`, `ArrayList`, `PrintWriter` and `Hashmap` from Java Collections Framework, `Logging` from Java Library, `DBCP` from the Apache Commons Project [2], `Apache FtpServer` from [7], `Hedc` from [5] and `Weblech` from [9].

The concurrent program `Elevator` simulates multiple lifts in a building; `RayTracer` renders a frame of an arrangement of spheres from a given view point; `DBCP` is the Database Connection Pool in the Apache Commons suite; `Apache FtpServer` is a FTP server by Apache; and `Vector`, `Stack`, `StringBuffer`, `Logging`, `ArrayList`, `PrintWriter` and `Hashmap` are Java libraries; `Hedc` is a Web crawler application and `Weblech` is a websites download tool.

Our tool was also applied to other programs. For example, Colt [6] and Pool from the Apache Common Project [4] in three different releases 1.2, 1.3, 1.5. However, no deadlocks were found and to the best of our knowledge no resource deadlocks have been reported for these programs. Because no additional insights were given by these programs we did not report them in the Table 1.

Table 1 includes information about conditions under which the tests were performed, such as input files and parameters. In `Elevator`, an input file is included in the bench-

mark which provides the inputs and specifies the number of threads. In `Vector`, we wrote test cases with two threads and two small vectors in which each thread executes exactly one method from class `Vector`. Test cases for `Stack`, `StringBuffer`, `Logging`, `ArrayList`, `PrintWriter`, `HashMap`, `DBCP` were designed in a similar manner. The test cases for Java Collections in particular are artificial; these are not real bugs in the JDK Collections but could become errors if the clients use their API erroneously. We used such test cases to validate our tool considering that they were also used in previous works [24, 25, 29, 38]. For `FtpServer`, we wrote a test case with a client and a server where the client logs in and requests a connection. There is no test harness for `Hedc` and we simply ran the program. For `Weblech`, we simply downloaded a website using the standard settings. The data for `RayTracer` is incorporated in the benchmark, which comes in sizes A and B, while the user specifies the number of threads.

Table 1 provides information about the benchmarks for *deadlocks* as well as information about all three phases: monitoring, run prediction, and scheduling. For the monitoring phase, the number of threads, the number of synchronization events, and the total number of observed events are reported, as well as the monitoring time. For the prediction phase, we report the number of potential deadlocks found. In the scheduling part, we report the total number of schedulable predictions. Finally, we present the total time for the test and the number of deadlocks found.

PICKLOCK found all previously known deadlocks in the benchmarks analyzed. More details can be found at http://web.engr.illinois.edu/ sorrent1/papers/SPIN15.html
We ran the programs under the test harness several times, and found none of the reported bugs in any of these benchmarks by merely running tests randomly. PICKLOCK predicts about 11 deadlocking executions, a successful attempt at finding deadlocks on these benchmarks. The runtime overhead for the testing is minimal, around 10% of the base run time. This is in contrast to similar tools (e.g. JADE [29]) analyzing the same benchmarks.
PICKLOCK does not produce false positives. If a deadlock is reported by PICKLOCK, it is a real deadlock.

References

1. `http://jlint/sourceforge.net/`.
2. `http://commons.apache.org/dbcp`.
3. `http://jakarta.apache.org/bcel`.
4. `http://commons.apache.org`.
5. `http://www.hedc.ethz.ch`.
6. `http://acs.lbl.gov/ hoschek/colt`.
7. `http://mina.apache.org/ftpserver`.
8. `http://www.javagrande.org/`.
9. `http://weblech.sourceforge.net`.
10. R. Agarwal, S. Bensalem, E. Farchi, K. Havelund, Y. Nir-Buchbinder, S. D. Stoller, S. Ur, and L. Wang. Detection of deadlock potentials in multi-threaded programs. *IBM Journal of Research and Development*, 54(5), 2010.
11. R. Agarwal and S. D. Stoller. Run-time detection of potential deadlocks for programs with locks, semaphores, and condition variables. In *PADTAD*, pages 51–60, 2006.
12. R. Agarwal, L. Wang, and S. D. Stoller. Detecting potential deadlocks with static analysis and runtime monitoring. In *PADTAD*, pages 191–207, 2005.
13. C. Artho and A. Biere. Applying static analysis to large-scale, multi-threaded java programs. In *Australian Software Engineering Conference*, pages 68–75, 2001.

14. S. Bensalem, J. C. Fernandez, K. Havelund, and L. Mounier. Confirmation of deadlock potentials detected by runtime analysis. In *PADTAD*, pages 41–50, 2006.
15. S. Bensalem and K. Havelund. Dynamic deadlock analysis of multi-threaded programs. In *Haifa Verification Conference*, pages 208–223, 2005.
16. Y. Cai and W. K. Chan. MagicFuzzer: Scalable deadlock detection for large-scale applications. In *ICSE*, pages 606–616, 2012.
17. F. Chen, A. Farzan, J. Meseguer, and G. Rosu. Formal analysis of java programs in javafan. In *CAV*, pages 501–505, 2004.
18. C. Demartini, R. Iosif, and R. Sisto. A deadlock detection tool for concurrent java programs. *Softw., Pract. Exper.*, 29(7):577–603, 1999.
19. D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. *SIGOPS Oper. Syst. Rev.*, 37:237–252, 2003.
20. A. Farzan, P. Madhusudan, N. Razavi, and F. Sorrentino. Predicting null-pointer dereferences in concurrent programs. In *SIGSOFT FSE*, pages 47–58, 2012.
21. A. Farzan, P. Madhusudan, and F. Sorrentino. Meta-analysis for atomicity violations under nested locking. In *CAV*, pages 248–262, 2009.
22. K. Havelund and T. Pressburger. Model checking java programs using java pathfinder. *STTT*, 2(4):366–381, 2000.
23. P. Joshi, M. Naik, K. Sen, and D. Gay. An effective dynamic analysis for detecting generalized deadlocks. In *SIGSOFT FSE*, pages 327–336, 2010.
24. P. Joshi, C. S. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *PLDI*, pages 110–120, 2009.
25. H. Jula, D. M. Tralamazza, C. Zamfir, and G. Candea. Deadlock Immunity: enabling systems to defend against deadlocks. In *OSDI*, pages 295–308, 2008.
26. V. Kahlon, F. Ivancic, and A. Gupta. Reasoning about threads communicating via locks. In *CAV*, pages 505–518, 2005.
27. S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ASPLOS*, pages 329–339, 2008.
28. Z. D. Luo, R. Das, and Y. Qi. Multicore SDK: A practical and efficient deadlock detector for real-world applications. In *ICST*, pages 309–318, 2011.
29. M. Naik, C. S. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *ICSE*, pages 386–396, 2009.
30. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4), 1997.
31. T.F. Şerbănuţă, F. Chen, and G. Roşu. Maximal causal models for sequentially consistent systems. Technical report, University of Illinois at Urbana-Champaign, October 2011.
32. V. K. Shanbhag. Deadlock-detection in java-library using static-analysis. In *APSEC*, pages 361–368, 2008.
33. F. Sorrentino, A. Farzan, and P. Madhusudan. PENELOPE: weaving threads to expose atomicity violations. In *SIGSOFT FSE*, pages 37–46, 2010.
34. W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003.
35. C. von Praun. Detecting synchronization defects in multi-threaded object-oriented programs. In *Ph.D. Thesis*, Swiss Federal Institute of Technology, Zurich, 2004.
36. C. von Praun and T. R. Gross. Object race detection. In *OOPSLA*, pages 70–82, 2001.
37. Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. A. Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *OSDI*, pages 281–294, 2008.
38. A. Williams, W. Thies, and M. D. Ernst. Static deadlock detection for java libraries. In *ECOOP*, pages 602–629, 2005.
39. Y,Cai, S. Wu, and W. K. Chan. ConLock: a constraint-based approach to dynamic checking on deadlocks in multithreaded programs. In *ICSE*, pages 491–502, 2014.