Low-Overhead Deadlock Prediction

Yan Cai*
State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
Beijing, China
ycai.mail@gmail.com

Ruijie Meng*
State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, and University of Chinese Academy of Sciences
Beijing, China
mengrj@ios.ac.cn

Jens Palsberg
University of California
Los Angeles, USA
palsberg@cs.ucla.edu

ABSTRACT

Multithreaded programs can have deadlocks, even after deployment, so users may want to run deadlock tools on deployed programs. However, current deadlock predictors such as MagicLock and UnDead have large overheads that make them impractical for end-user deployment and confine their use to development time. Such overhead stems from running an exponential-time algorithm on a large execution trace. In this paper, we present the first low-overhead deadlock predictor, called AirLock, that is fit for both in-house testing and deployed programs. AirLock maintains a small predictive lock reachability graph, searches the graph for cycles, and runs an exponential-time algorithm only for each cycle. This approach lets AirLock find the same deadlocks as MagicLock and UnDead but with much less overhead because the number of cycles is small in practice. Our experiments with real-world benchmarks show that the average time overhead of AirLock is 3.5%, which is three orders of magnitude less than that of MagicLock and UnDead. AirLock's low overhead makes it suitable for use with fuzz testers like AFL and on-the-fly after deployment.

CCS CONCEPTS

• **Software and its engineering → Deadlocks**.

KEYWORDS

Deadlock detection, multithreaded programs, lock reachability graph

ACM Reference Format:
Yan Cai, Ruijie Meng, and Jens Palsberg. 2020. Low-Overhead Deadlock Prediction. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3377811.3380367

1 INTRODUCTION

Multithreaded programs are error-prone due to unexpected thread interleavings that can cause various concurrency bugs. One such kind of bug is deadlocks that can happen because of incorrect synchronizations among multiple threads. A deadlock occurrence prevents an execution from making further progress. A deadlock occurs when a set of threads hold a set of locks and they mutually wait for other locks held by the same set of threads [1, 3]. In this paper, we focus on resource deadlocks only [6]; another kind of deadlock is known as communication deadlocks [28].

Like other kinds of concurrency bugs, deadlocks are difficult to detect due to non-determinism of multithreaded executions. In particular, even if a program has a reachable deadlock, the deadlock may occur in just a small number of executions. However, unlike other kinds of concurrency bugs, if a deadlock occurs, it can be easily detected [2] at run time.

As deadlocks are caused by non-determinism of thread interleavings, in-house testing is unlikely to detect all of them[1]. Even after a multithreaded program is released, deadlocks can still occur. Hence, it is still critical to detect deadlocks in released software, e.g., at end-users. In such scenarios, low overhead on-the-fly detectors should be the first choice.

Unfortunately, to the best of our knowledge, existing deadlock detection techniques are not suitable for on-the-fly detection. They incur a large time overhead (that can be 100−1000x), which prevents them from being applied by end-users as the maximum acceptable time overhead there is usually less than 5% [4, 32, 36, 40].

In detail, a predictive tool analyzes an execution trace and predicts whether deadlocks may occur in alternative executions [1, 3, 6, 7, 29, 38]. As outlined in Figure 1, these approaches map an execution trace into a large data structure and apply an exponential-time algorithm[2] to it to detect cycles as deadlocks. The earliest work is the GoodLock algorithm [3] that maps an execution trace into a lock order graph where (1) locks are nodes, (2) lock orders are edges, and (3) edge weights are thread identifiers and other execution information. Next, GoodLock searches for cycles in the graph as potential deadlocks.

GoodLock adopts the Depth-First Search (DFS) algorithm on the lock order graph. During DFS, edge weights are frequently checked against those of all edges in the current path to see if they satisfy the deadlock definition. Its searching cost increases exponentially with the increasing number of lock acquisitions. There have been several works to improve the practical efficiency of GoodLock, such as MulticoreSDK [38] and IGoodLock [29]. The two latest works are MagicLock [6, 7] and UnDead [54]. MagicLock introduces several

*Co-first author.

[1] In theory, although model checking [13, 21] based techniques (and other synchronization coverage ones [23, 45, 53]) can explore all thread interleavings, they usually scale poorly to such programs as MySQL that have millions of lines of codes [12].
[2] Given a graph $G = (V, E)$, it requires 2^E operations to find all cycles in G in the worst case (corresponding to 2^E cycles) [27].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICSE '20, May 23–29, 2020, Seoul, Republic of Korea
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380367

Core of the Previous Algorithms

1. Map an execution trace to a large data structure;
2. Run an **exponential-time** algorithm on the data structure to find cycles as deadlocks.

Core of the AirLock

1. Map an execution trace to a set of small data structures and a lock reachability graph;
2. Run a **polynomial-time** algorithm on the reachability graph to report:
   - either "an existence of some cycles" (a small percentage),
   - or "an absence of any cycle" (a large percentage);
3. For the existence case, run an **exponential-time** algorithm on **a small subset** of the reachability graph to find cycles and construct deadlocks for each cycle.

Figure 1: The core algorithms of previous detectors and AirLock.

strategies to prune locks and edges that cannot participate in any cycles as well as identifying equivalent edges. UnDead [54] is a simplified version of IGoodLock with some optimizations adopted from MagicLock (i.e., discarding equivalent edges [6] that compromise the ability to detect concrete deadlocks). A difference among them is that UnDead tries to keep traces in memory so that it can begin detection once an execution terminates, whereas the previous ones keep traces in external storage. Nevertheless, these works target offline deadlock detection. They become inefficient when used for on-the-fly deadlock detection, as we show experimentally in Section 5.

In this paper, we present a new deadlock detection algorithm, called AirLock. It does on-the-fly (predictive) deadlock detection with low overhead, and it can be applied to both in-house testing by developers (e.g., in fuzzing testing) and deployed products.

The idea of AirLock is to use an on-the-fly algorithm instead of the offline algorithms in previous works. The core algorithm of AirLock is shown in Figure 1. It collects an execution trace into a set of small data structures and a predictive lock reachability graph that reflects the relationship of every two locks (without any execution information). Next, it runs a polynomial-time algorithm on the reachability graph to conclude whether the trace has some cycles. If so, it runs an exponential-time algorithm on a small subset of its lock reachability graph to find all cycles and constructs all deadlocks.

AirLock is based on the following observations about large benchmarks: (1) most pairs of locks are acquired in consistent orders and they do not form any cycle and (2) only a few pairs of locks are acquired in reversed orders which may cause deadlocks. Hence, it is unnecessary to directly apply a heavy algorithm to a large trace as done by existing works [6, 29, 54]. Instead, the strategy of AirLock is to identify the existence of cycles and then to detect them. Additionally, AirLock reduces the reachability graph on-the-fly without missing any cycles. This design makes AirLock a low-overhead deadlock predictor.

We have implemented AirLock on top of the Pin framework [37] and evaluated it on a set of seven real-world benchmarks. These benchmarks contain six unique deadlocks that are helpful for evaluating the effectiveness of AirLock. We configured the benchmarks with inputs that make them run for 10 to 60 seconds to evaluate efficiency. We also compared AirLock with both MagicLock and UnDead. The experimental results show that all three tools reported the same predictive deadlocks. However, on efficiency, AirLock only incurred an average of 3.5% time overhead; whereas, MagicLock and UnDead incurred an average of 31x and 371x time overhead, respectively. To further evaluate the efficiency of AirLock, we configured it to detect cycles at different frequencies (i.e., every i seconds, $1 \le i \le 10$) on two large-scale benchmarks (MySQL and Firefox). The results show that, at any of these frequencies, AirLock incurred less than 6% time overhead. The results demonstrate that AirLock is an efficient on-the-fly predictive deadlock detector, applicable for both in-housing testing and deployed products at end-users.

The low overhead of AirLock also makes it work well with fuzz testers like AFL.

In summary, the contributions of this paper include:

- a novel on-the-fly predictive deadlock detection approach, based on the insight that most lock acquisition orders do not form any deadlock, which can be concluded by a polynomial-time algorithm;
- the tool AirLock that implements the above insight for on-the-fly predictive deadlock detection;
- an evaluation of AirLock that shows that AirLock only incurred on average 3.5% time overhead on large-scale programs. Even under frequent deadlock detection (per 1 or more seconds), it only incurred less than 6% time overhead (less than 5% time overhead when the detection period of $\ge$ 5 seconds).

2 MOTIVATING EXAMPLE

In this section, we begin with introducing some notations that we will use throughout the paper, and then we walk through an example of how AirLock predicts deadlocks.

2.1 Preliminaries

A multithreaded program has a set of threads and a set of locks. During execution, a lock l can be acquired and released by at most one thread t at a time, denoted as *acq(l)* and *rel(l)*, respectively. A thread can acquire additional locks before it releases any acquired locks; and the set of all these locks held by a thread t is called a *lockset*, denoted as *LS(t)*. A lock l can be destroyed, denoted as *des(l)*; and the destroyed lock l cannot be acquired and released again.

To be consistent with previous works, we also adopt the concept of the lock dependency in our analysis. A **lock dependency** [6, 29] is a triple $d = \langle t, l, LS(t) \rangle$, indicating that, during an execution, a thread t acquires a lock l when it holds a set of locks in *LS(t)*. A **trace** is a set of dependencies. Notice that our notion of trace is a set rather than a list; this is because a set is sufficient for our purposes.

Two (ordered) locks have the following reachability relationship: **direct reachability** ($\rightarrow$) and **indirect reachability** ($\dashrightarrow$). A lock l_1 *directly reaches* a lock l_2 if there is a lock dependency $\langle t, l_2, LS(t) \rangle$ such that $l_1 \in LS(t)$; and we say that there is a ***direct edge*** from lock l_1 to lock l_2, denoted as $l_1 \rightarrow l_2$. A lock l_1 *indirectly reaches* a lock l_k if there are a sequence of locks $l_2, ..., l_{k-1}$ such that $l_i \rightarrow$

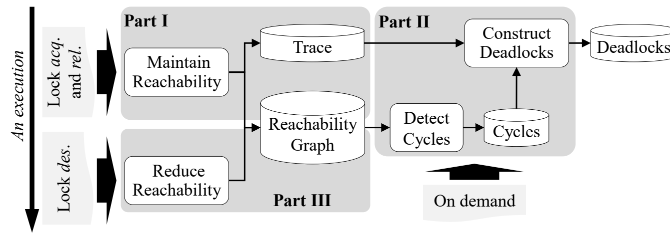


Figure 2: An overview of AirLock.

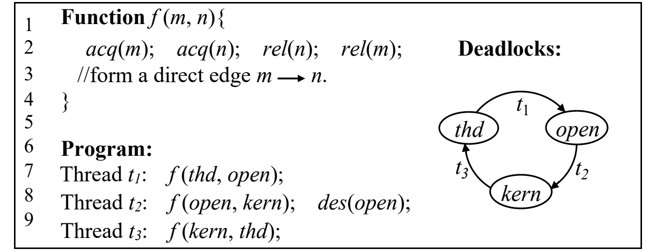


Figure 3: A motivating program adapted from MySQL (Bug ID: 62614).

l_{i+1}, for $1 \le i \le k-1$; and we say there is an ***indirect edge*** from lock l_1 to lock l_k, denoted as $l_1 \dashrightarrow l_k$. For example, as shown in Figure 3, when thread t_1 calls function $f(thd, open)$, it forms a direct edge $thd \to open$. Similarly, thread t_2 forms a direct edge $open \to kern$. Both direct edges $\{thd \to open, open \to kern\}$ together form an indirect edge $thd \dashrightarrow kern$. From above definition, we see that, (1) there can be both direct and indirect edges between two locks, (2) a dependency can produce multiple edges and an edge can be represented as a lock dependency [29], and (3) the same edge can be produced by different lock dependencies.

We introduce two **edge sets** Fr and To to describe edges. Given a lock l, $Fr(l)$ denotes a set of locks that lock l directly or indirectly reaches, and $To(l)$ denotes a set of locks that can directly or indirectly reach lock l. Obviously, given two locks l_1 and l_2, if $l_2 \in Fr(l_1)$, then we have $l_1 \in To(l_2)$.

Now, we define **a predictive (lock) reachability graph** over a set of locks V to be $R = \langle V, E\rangle$ where $E = \{\langle l_1, l_2\rangle \in V \times V | l_2 \in Fr(l_1)\}$. Obviously, R is the transitive closure of the directed graph $G_o = \langle V, E_o\rangle$ where E_o is the set of all direct edges.

In the above definition of indirect edge, if the lock l_k also directly reaches the lock l_1, we say that the sequence of edges $\{l_1 \to l_2, ..., l_{k-1} \to l_k, l_k \to l_1\}$ forms a **direct cycle**. An **indirect cycle** is defined similarly except that, at least one edge is an indirect edge. A **simple cycle** is defined to be a direct or an indirect cycle of two locks.

Note, in the definition of cycles, we only consider the reachability among locks, but exclude aspects such as their forming threads and locksets. This is different from the cycles defined in previous works like [6, 54]. To restrict our approach to report exactly the same (predictive) deadlocks as previous works, we follow the same definition of **deadlocks** [6, 29]: a sequence of m lock dependencies $\langle d_0, ..., d_{m-1}\rangle$ (where $d_i = \langle t_i, l_i, LS(t_i)\rangle$, for $0 \le i \le m-1$) forms a deadlock if:

- for $0 \le i \le m-1$, $l_i \in LS(t_{(i+1 \pmod m)})$, and
- for $0 \le i < j \le m-1$, $LS(t_i) \cap LS(t_j) = \emptyset$.

The definition requires that each thread of the set should hold a set of locks and mutually wait for another lock held by a different thread, and at the same time, no two threads hold the same lock. Given a direct cycle, its corresponding set of deadlocks can be constructed by first replacing each edge with every lock dependency that produces the edge and then check the set of lock dependencies against the deadlock definition.

2.2 Overview and Illustration

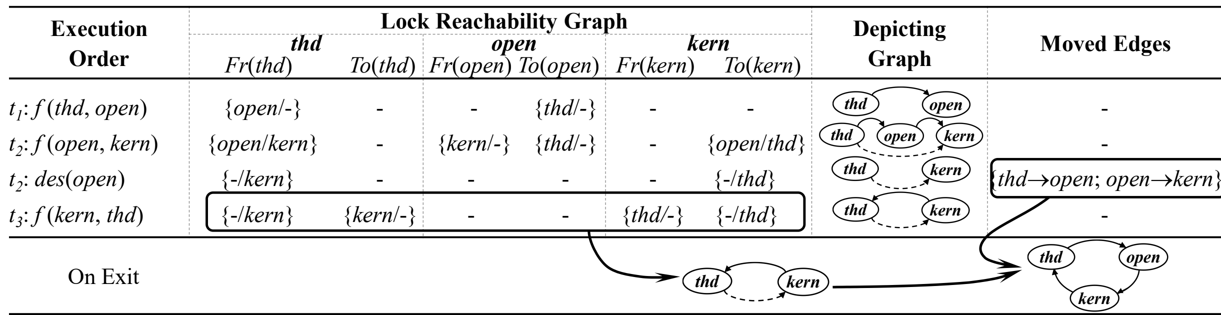
Figure 2 shows an overview of AirLock, consisting of three parallel on-the-fly parts to detect deadlocks. Part I builds a reachability graph on lock acquisitions, and it also records a trace. Part II detects cycles from the lock reachability graph and constructs deadlocks from the trace. And Part III reduces the lock reachability graph when a lock is destroyed, and detects cycles involving the destroyed locks.

Figure 3 shows a motivating program adapted from a deadlock in MySQL2 (Bug ID: 62614) which is one of our benchmark programs. It has three threads (t_1, t_2, t_3) and three locks ($thd, open, kern$). Each thread calls function $f()$ to acquire two locks, and thread t_2 further calls function $des()$ to destroy lock *open*. Note, originally in MySQL2, the lock *open* is not destroyed at the end of thread t_2 but at the program exit point; we made the change to illustrate the correctness of AirLock's reduction. Suppose that the three threads are executed in the order shown in the first column of Figure 4. The corresponding execution trace is a set of three dependencies: $\langle t_1, open, \{thd\}\rangle$, $\langle t_2, kern, \{open\}\rangle$, $\langle t_3, thd, \{kern\}\rangle$. Obviously, this example has a deadlock where thread t_1 holds lock thd and waits for lock $open$, thread t_2 holds lock $open$ and waits for lock $kern$, and thread t_3 holds lock $kern$ and waits for lock thd, and no two threads hold the same lock .

Figure 4 illustrates the on-the-fly deadlock detection process of AirLock. The second major column shows the maintained lock reachability graph (i.e., the edge sets Fr and To) which is also depicted. Note, in each edge set, the elements before and after a slash "/" are locks involved in direct and indirect edges, respectively. For example, the value "$\{open/kern\}$" under $Fr(thd)$ indicates two edges: one direct edge $thd \to open$ and one indirect edge $thd \dashrightarrow kern$. The last column shows edges **Moved** from memory to disk (as we will explain below). Assume that AirLock is configured to detect deadlocks at the program exit point.

Initially, after thread t_1 executes, AirLock produces one direct edge $thd \to open$ which is reflected under both sub-columns $Fr(thd)$ and $To(open)$. Next, after thread t_2 calls function $f(open, kern)$, a new direct edge $open \to kern$ is formed and is reflected under both sub-columns $Fr(open)$ and $To(kern)$. Considering this edge and the previous edge (i.e., $thd \to open$), they produce an indirect edge $thd \dashrightarrow kern$. So, AirLock updates the reachability graph to reflect this indirect edge.

Next, after thread t_2 destroys lock *open*, AirLock first detects cycles involving lock *open* on the current reachability graph but no one is detected. It then deletes the lock from the reachbility graph. However, now two particular kinds of locks exist (i.e., a lock *thd*



Execution Order	Lock Reachability Graph: *thd* Fr(*thd*)	*thd* To(*thd*)	*open* Fr(*open*)	*open* To(*open*)	*kern* Fr(*kern*)	*kern* To(*kern*)	Depicting Graph	Moved Edges
t_1: *f*(*thd*, *open*)	{*open*/-}	-	-	{*thd*/-}	-	-		-
t_2: *f*(*open*, *kern*)	{*open*/*kern*}	-	{*kern*/-}	{*thd*/-}	-	{*open*/*thd*}		-
t_2: *des*(*open*)	{-/*kern*}	-	-	-	-	{-/*thd*}		{*thd*→*open*; *open*→*kern*}
t_3: *f*(*kern*, *thd*)	{-/*kern*}	{*kern*/-}	-	-	{*thd*/-}	{-/*thd*}		-
On Exit								

Figure 4: An illustration of AirLock on the motivating program.

that reaches lock *open* and a lock *kern* that is reachable from lock *open*); AirLock moves these direct edges into an additional storage (e.g., an external storage) as depicted, indicating that these edges may participate in a cycle that is produced later.

Finally, after thread t_3 executes $f(kern, thd)$, a new direct edge $kern \rightarrow thd$ is produced and the reachability graph is updated accordingly. However, no new indirect edge is produced because there are no other locks except the two locks themselves that reach lock $kern$ or can be reached from locks thd.

At the execution exit point, AirLock detects cycles on the reachability graph, resulting in an indirect cycle $\{kern \rightarrow thd, thd \dashrightarrow kern\}$. Based on this indirect cycle, AirLock performs a DFS search of the edges that are only from and to lock *thd* and lock *kern*, including the corresponding moved edges (in the column **Moved Edges**). This results in a direct cycle of three locks $\{kern \rightarrow thd, thd \rightarrow open, open \rightarrow kern\}$. Based on the trace of the execution, a predictive deadlock (with thread IDs and locksets) of the cycle is reported.

3 OUR APPROACH: AIRLOCK

This section presents the three parts of AirLock namely lock reachability graph maintenance, cycle detection and deadlocks construction, and reachability graph reduction.

3.1 Part I: Maintain Reachability Graph

Part I of AirLock tracks lock acquisitions and releases to maintain a lock reachability graph (i.e., to build *Fr* and *To*). The key here is to ensure that the reachability graph precisely reflects all direct edges and indirect edges. It requires to not only record the direct edges produced on lock acquisitions but also compute all indirect edges due to the insertion of the direct edges. We show Part I in Algorithm 1. Given a lock acquisition, AirLock **records dependencies** in a trace and **records direct edges** produced by the acquisition. Due to the insertion of the new direct edges, AirLock **propagates reachability** of any affected locks.

Note, the edge sets *Fr* and *To* consist of both direct edges and indirect edges. To distinguish them, we introduce two functions $ColorDir(m)$ and $ColorInd(m)$ to mark a lock m when it is added into $Fr(l)$ (or $To(l)$), indicating that the edge from l to m (or from m to l) is a direct edge or an indirect edge.

Record Dependencies. As outlined in core algorithm in Figure 1, AirLock arranges a trace into a set of small data structures. In detail, it indexes a sequence of dependencies by an edge that can be produced by any indexed dependency. Hence, in Algorithm 1, on a lock acquisition $\text{OnACQ}(t, l)$, AirLock records the dependency into the map $Tr(m \rightarrow l)$ (line 7). Our trace is different from previous works that arrange dependencies of a trace to one set [29] or multiple thread-specific sets [6, 54].

Record Direct Edges. Each lock acquisition $\text{OnACQ}(t, l)$ produces a set of direct edges $m \rightarrow l$ for $m \in LS(t)$. All these direct edges are added into the two edge sets $Fr(m)$ and $To(l)$ (line 9 and 10). And, the lockset of thread t is updated to include lock l (which is excluded on the paired release $\text{OnREL}(t, l)$).

Propagate Reachability. When a new direct edge $m \rightarrow l$ is inserted to *Fr* and *To*, the reachability of locks from l and to m has to be updated as illustrated in Figure 5. Considering that our reachability graph is a transitive closure of a graph consisting of all direct edges, the existing algorithms [26, 33, 43] for maintaining

Algorithm 1: Maintain a Lock Reachability Graph

```
1  LS maps a thread t to its lockset.
2  Tr maps an edge m → l to a sequence of dependencies.
3  Fr and To are the two edge sets.
4  ColorDir and ColorInd: two functions that mark each lock
     with different colors.
5  Function OnACQ(t, l)
6      foreach m ∈ LS(t) do
7          Tr(m → l) ← Tr(m → l) ∪ {t, LS(t)\{m}}  ▹ Dependency
8          if ColorDir(l) ∉ Fr(m) then
9              Fr(m) ← Fr(m) ∪ {ColorDir(l)}   ▹ Direct edges
10             To(l) ← To(l) ∪ {ColorDir(m)}   ▹ Direct edges
11             call PropagateReach(m, l).  ▹ Propagate reachability
12     LS(t) ← LS(t) ∪ {l}
13 Function OnREL(t, l)
14     LS(t) ← LS(t)\{l}
15 Function PropagateReach(m, l)
16         ▹ Update reachability graph dued to the direct edge m → l.
17     foreach m′ ∈ To(m) do
18         Fr(m′) ← Fr(m′) ∪ TransInd(Fr(l)) ∪ {ColorInd(l)}
19     foreach l′ ∈ Fr(l) do
20         To(l′) ← To(l′) ∪ TransInd(To(m)) ∪ {ColorInd(m)}
21     Fr(m) ← Fr(m) ∪ TransInd(Fr(l))
22     To(l) ← To(l) ∪ TransInd(To(m))
```

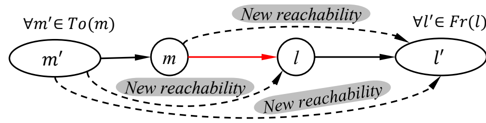


Figure 5: Update reachability given a new direct edge $m \to l$.

the transitive closure of a dynamic graph can be adapted. This is implemented in PropagateReach(m, l) in Algorithm 1, where we introduce a function $TransInd(Set)$ to copy all of the edges in Set as indirect edges.

3.2 Part II: Detect Cycles and Construct Deadlocks

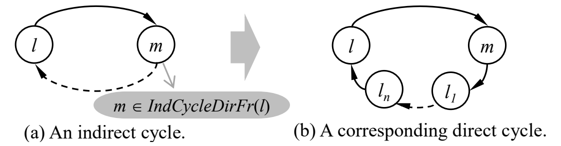
Part II detects cycles from the lock reachability graph and constructs corresponding deadlocks. Instead of directly applying an exponential-time searching algorithm on a large trace (as adapted in previous works), AirLock splits the cycle detection in two phases, as outlined in Figure 1 (step 2 and step 3). First, it only iterates on the reachability graph once, which is enough to **identify all simple cycles** including both direct cycles and indirect cycles. This

Algorithm 2: Detect Cycles

1 C: a set to keep all direct cycles.
2 $IndCycleLocks$: a set to keep locks in indirect simple cycles.
3 $IndCycleDirFr$: a set to map a lock l to its directly reachable locks, such that any lock in the mapped set indirectly reaches lock l.
4 **Function** DetectCycles()
5   **foreach** *lock* l **do**
6   **foreach** $m \in Fr(l)$ **do**
7   **if** $ColorDir(m) \in Fr(l) \land ColorDir(l) \in Fr(m)$ **then** ▹ A direct simple cycle: $l \to m, m \to l$
8         $C \leftarrow C \cup \{\langle l \to m, m \to l \rangle\}$.
9   **if** $ColorDir(m) \in Fr(l) \land ColorInd(l) \in Fr(m)$ **then** ▹ An indirect simple cycle: $l \to m, m \dashrightarrow l$
10         $IndCycleDirFr(l) \leftarrow IndCycleDirFr(l) \cup \{m\}$
11         $IndCycleLocks \leftarrow IndCycleLocks \cup \{l\}$.
12   $Visited(l) \leftarrow False$, for each lock $l \in IndCycleLocks$
13   $S \leftarrow \emptyset$ ▹ A stack structure for DFS
14   **foreach** *lock* $l \in IndCycleLocks$ **do**
15     $Visited[l] \leftarrow True$; call DFS(l); $Visited[l] \leftarrow False$.
16 **Function** DFS(l)
17   **if** $S[0] = l$ **then** ▹ A direct cycle of three or more locks
18     $C \leftarrow C \cup \{\langle S[0] \to S[1], ..., S[k-1] \to S[0] \rangle\}, k = |S|$
19   return.
20   Push l into S.
21   **for** $m \in IndCycleDirFr(l) \cup ExternalFr(l)$ **do**
22   ▹ Only traverse a direct edge: $l \to m$.
23   **if** $Visited[m] = False$ **then**
24       $Visited[m] \leftarrow True$; call DFS($m$);
25       $Visited[m] \leftarrow False$.
26   Pop l from S.

concludes whether the trace contains cycles. For any indirect cycle, AirLock **detects the corresponding direct cycles** via a DFS algorithm. It then **constructs deadlocks** for every direct cycle. We present its detection algorithm (Algorithm 2) and then discuss the benefit of such a design.

Detect Simple Cycles. To detect all simple cycles, AirLock only needs to traverse the edge set Fr once, as shown in lines 5–11. Given a lock l, it traverses all locks in $Fr(l)$. For any $m \in Fr(l)$, if the lock l is also in $Fr(m)$, a direct cycle $\{l \to m, m \to l\}$ or a indirect cycle $\{l \to m, m \dashrightarrow l\}$ is detected. For efficiency purpose in the later detection, we introduce two data structures $IndCycleDirFr$ and $IndCycleLocks$ for indirect cycles. The structure $IndCycleDirFr(l)$ (having the same structure as Fr) keeps all identified direct edges in indirect cycles (line 10, the edge $l \to m$). The structure $IndCycleLocks$ keeps all locks in indirect cycles (like the keys of $IndCycleDirFr$).



(a) An indirect cycle. (b) A corresponding direct cycle.

Figure 6: An illustration on indirect Cycles.

Detect Corresponding Direct Cycles. For an indirect cycle, there must exist a direct cycle as illustrated in Figure 6. Given a set of indirect cycles as locks in $IndCycleDirFr$ and a set of all locks in all indirect cycles, AirLock searches a corresponding direct cycle for each indirect cycle based on a DFS algorithm (lines 16–26). Note, in line 21, Algorithm 2 also considers edges ($ExternalFr(l)$) kept in disk (due to the reachability reduction, see Section 3.3).

Construct Deadlocks. As explained in Section 2.1, a direct cycle detected by AirLock is different from the one detected by previous works (e.g. [6, 29]) where AirLock only considers the reachability of locks without any execution information (e.g., thread IDs and lockset). However, AirLock also records all dependencies as a trace (i.e., Tr in Algorithm 1) for constructing all deadlocks, resulting in exactly the same deadlocks as those reported by previous works.

In detail, given a direct cycle, AirLock checks all sequences of dependencies indexed by all edges of this cycle. A direct cycle will correspond to a set of permutations (i.e., a set of dependencies, one from each indexed trace). Each permutation will be checked against the deadlock definition and the satisfied ones are finally reported. This process is efficient by taking the concept of *Equivalent dependencies* [6] such that only permutations consisting of *non-equivalent dependencies* from each indexed sequence of dependencies are checked.

Discussion. Let E be the all edges in Fr. The time complexity of the first phase (i.e., lines 5–11) is roughly a polynomial-time complexity $O(E^2)$ as the algorithm implicitly checks every pair of edges in Fr. The second phase (lines 12–26) is still an exponential-time DFS algorithm. However, from line 21, the DFS algorithm traverses all direct edges in indirect cycles. Considering our insight that most of nested lock acquisitions do not participate in any cycle, there will be a small number of simple cycles. This results in a much smaller searching space than that by previous works [6, 29, 54],

which have to explore much more paths where most of them do not finally form cycles. Hence, Algorithm 2 is very efficient in practice.

3.3 Part III: Reduce Reachability Graph

The first two parts work well in terms of detecting all direct cycles. However, a multithreaded program usually creates numerous locks and edges. During an execution, the cumulative number of locks could be very large, bringing increasing memory and time consumption. One straightforward solution is to remove all destroyed locks and related edges during an execution. However, this brings a challenge on how to guarantee the correctness (i.e., not to miss any cycle) after reduction. It is because a destroyed lock may also participate in a cycle that is formed later. For example, in our example (Figure 4), when the lock *open* is destroyed, we delete all edges involving the lock. Then, no cycle will be reported as the two edges (formed by threads t_1 and t_2) have been deleted.

We propose a reachability reduction algorithm with guarantees on the reduction correctness as shown in Algorithm 3. Basically, when a lock is destroyed, AirLock tries to ensure its reduction correctness. If the correctness cannot be immediately determined, it splits all related edges from the reachability graph (e.g., to keep

Algorithm 3: Reduction and Cycle Detection

```
1  Function OnDestroyLock(l)
2    if Indegree(l) = 0 ∧ Outdegree(l) > 0 then                    ▷ Case 2.1
3      foreach m ∈ Fr(l) do
4        To(m) ← To(m)\{ColorDir(l), ColorInd(l)}
5        DeleteAllEdges(ColorDir(l), ColorDir(m)).
6    if Indegree(l) > 0 ∧ Outdegree(l) = 0 then                    ▷ Case 2.2
7      foreach m ∈ To(l) do
8        Fr(m) ← Fr(m)\{ColorDir(l), ColorInd(l)}
9        DeleteAllEdges(ColorDir(m), ColorDir(l)).
10   if Indegree(l) > 0 ∧ Outdegree(l) > 0 then                    ▷ Case 3
11     foreach m ∈ Fr(l) do
12       To(m) ← To(m)\{ColorDir(l), ColorInd(l)}
13       if ColorDir(m) ∈ Fr(l) ∧ ColorDir(l) ∈ Fr(m)
          then              ▷ A direct simple cycle: {l → m, m → l}
14         C ← C ∪ {⟨l → m, m → l ⟩}.
15       if ColorDir(m) ∈ Fr(l) ∧ ColorInd(l) ∈ Fr(m)
          then              ▷ An indirect simple cycle: {l → m, m ⇢ l}
16         IndCycleDirFr(l) ← IndCycleDirFr(l) ∪ {m}
17         IndCycleLocks ← IndCycleLocks ∪ {l}.
18     foreach m ∈ To(l) do
19       Fr(m) ← Fr(m)\{ColorDir(l), ColorInd(l)}
20       if ColorDir(m) ∈ To(l) ∧ ColorInd(m) ∈ Fr(l)
          then              ▷ An indirect simple cycle: {m → l, l ⇢ m}
21         IndCycleDirFr(m) ← IndCycleDirFr(m) ∪ {l}
22         IndCycleLocks ← IndCycleLocks ∪ {m}.
23                          ▷ Only save direct edges in Fr(l) and To(l).
24     foreach ColorDir(m) ∈ Fr(l) do
25       ExternalFr(l) ← ExternalFr(l) ∪ {ColorDir(m)}
26     foreach ColorDir(m) ∈ To(l) do
27       ExternalFr(m) ← ExternalFr(m) ∪ {ColorDir(l)}
28   Fr(l) := ∅; To(l) := ∅
```

into external disk) for later cycle detection. In such a way, the reachability graph (e.g., in memory) is always for live locks. We present our reduction algorithm and then give an informal analysis on its correctness.

Reduction. On destroying lock l, there are three reduction cases according to whether the indegree or/and the outdegree of this lock is zero [3]: (Case 1) both indegree and outdegree of lock l are zero, (Case 2) only one of them is zero, and (Case 3) both of them are nonzeros. For Case 1, nothing should be taken because this lock does not reach any other locks and vice versa. For Case 2, obviously, this lock is not involved in any cycle; hence, all its information should be removed. Besides, if the indegree of lock l is zero (Case 2.1), for any other lock m (i.e., $m \in Fr(l)$) that is directly or indirectly reachable from lock l, we remove edge $l \rightarrow m$ as well as all dependencies in the trace indexed by this edge (see line 3–6). If the outdegree of lock l is zero (Case 2.2), we perform the similar action.

For Case 3, since there are locks both reachable from and to lock l, lock l may be involved in some cycles. Hence, we first perform a cycle detection. However, we only detect all simple cycles involving lock l (lines 13–17 and 20–22). After that, we move all direct edges (in $Fr(l)$ and $To(l)$) into disk. These edges are known as *External Edges* (i.e., *ExternalFr* in the Algorithm 3) and are searched during detection of all direct cycles in Algorithm 2.

Reduction Correctness. We briefly show the correctness analysis based on the mathematical induction manner: before and after reduction of k locks and all edges involving these locks, AirLock detects the same set of cycles.

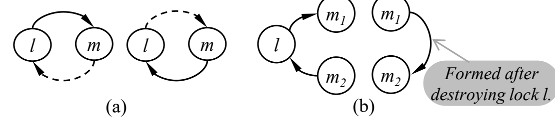


Figure 7: Correctness illustration for Reduction.

Base case (i.e., $k = 1$): before and after reduction of the first lock, say l, the above claim holds. Let's analyze the three reduction cases:

- For Cases 1 and 2, obviously, the lock cannot participate in any cycle. Hence, reducing the lock does not affect any cycles to be detected. Our claim holds.
- Recall that Case 3 is: the lock l has both incoming and outgoing edges. Before reduction, the lock can only participate in two kinds of cycles: cycles already formed and cycles formed later as shown in 7 (a) and (b), respectively. For already formed cycles, Algorithm 3 can detect them before reducing the lock. For any cycles to be formed later, there must exist two lock m_1 and m_2 that form two edges $m_2 \rightarrow l$ and $l \rightarrow m_1$ as shown in Figure 7(b), such that the edge $m_1 \rightarrow m_2$ is formed after destroying lock l. Hence, there must be an indirect edge $m_2 \dashrightarrow m_1$ (see how Algorithm 1 updates reachability). This results in that, after remove lock l and edges $m_2 \rightarrow l$ and $l \rightarrow m_1$, an indirect cycle $\{m_1 \rightarrow m_2, m_2$

[3]Note, we do not explicitly maintain either indegree or outdegree for each lock as we only need to know whether the value is zero or not. This can be easily analyzed from the keys of the edges sets Fr and To in implementation.

$\dashrightarrow m_1\}$ can be detected after the edge $m_1 \rightarrow m_2$ is formed. Besides, during reduction, Algorithm 3 also keeps all direct edges from and to lock l in disk (i.e., $ExternalFr$). When detecting all direct cycles of the indirect cycle $\{m_1 \rightarrow m_2, m_2 \dashrightarrow m_1\}$, by searching edges in disk, Algorithm 2 can detect the direct cycle involving lock l. (Note, the edges from m_2 to l, from l to m_1, and from m_1 to m_2 can be either direct or indirect ones, which does not affect the analysis.) Therefore, for Case 3, after reducing lock l and edges involving l, our claim holds.

Now, assume that after reducing the first k locks and all edges from/to them, the same set of cycles can be detected. Let's show that reducing the $(k+1)^{th}$ lock, our claim still holds. Note, the $(k+1)^{th}$ lock can have incoming or/and outgoing edges before reducing some of the k locks. But here we only consider the current state: whether any cycles can be missed by reducing the $(k+1)^{th}$ lock. Obviously, by repeating the steps in the base analysis, no cycle can be missed from the current state by removing one lock and edges involving in this lock. (A more detailed analysis can be done by split the execution into two executions at the point right before reducing the $(k+1)^{th}$ lock and apply base analysis on the second execution.) Thus, we can see that our claim still holds by reducing the $(k+1)^{th}$ locks.

Combining the two analyses above, our claim holds. That is, our reduction guarantees not to missing any cycle while reducing locks and their corresponding edges.

3.4 Frequent Deadlock Detection

Existing predictive deadlock detectors [6, 29, 54] only detect deadlocks after a program exits or is about to exit. They cannot immediately report deadlocks whenever they are formed, especially for long-running programs (e.g., server programs) or non-terminating executions. Unlike these, AirLock (i.e., Part II) can be configured to run whenever there is a detection need (e.g., periodically or on user demand). It can detect deadlocks at runtime. In our experiments (Section 5.4), we show that AirLock is scalable to detect deadlocks per-second during runtime with less than 6% time overhead on two programs running for 36 and 100 seconds, respectively.

4 DISCUSSION OF AIRLOCK

In this section, we briefly discuss AirLock and other similar works in terms of maximality and soundness on reported deadlocks.

Maximality. Given the same trace and the same deadlock definition (i.e., the one in the last paragraph of Section 2.1), AirLock can detect the same set of predictive deadlocks as that by previous works including IGoodLock, MagicLock, and UnDead. This set of deadlocks is maximal with respect to the trace, because all approaches consider all permutations of lock acquisitions and filter out those not satisfying the deadlock definition.

Soundness. All approaches above including AirLock are unsound by reporting false positives. To the best of our knowledge, Dirk [31] is the latest work on sound deadlock prediction. However, like sound data race prediction [24, 25], it needs to track additional events and relies on constraint solvers, bringing heavy performance challenges. These make them unsuitable for efficient on-the-fly deadlock prediction.

5 EXPERIMENTS

In this section, we present a set of experiments to demonstrate the effectiveness and the efficiency of AirLock as an on-the-fly predictive deadlock detection tool, and its scalability on high frequency deadlock detection. We also conducted a self-comparison on the two strategies (i.e., cycle detection and reachability reduction) of AirLock.

5.1 Benchmarks

We collected a set of seven real-world C/C++ benchmarks including HawkNL, SQLite, two different versions of MySQL database servers, two browsers (Firefox and Chromium), and Thunderbird. They contain six unique deadlocks that are similar in number to previous work [6, 54]. All these benchmarks and deadlocks have been extensively studied in previous works [6, 7, 54]. We excluded one benchmark MySQL-6.0.4 in the papers of MagicLock and UnDead, because MySQL-6.0.4 uses customized synchronization primitives, not standard Pthreads.

In Table 1, we show the statistics of these benchmarks, including their names (versions), Bug ID, source lines of code (SLOC), NO. of threads, NO. of locks ((total number of locks)/(max number of live locks)), a summary of inputs or deadlock descriptions, the time cost of their native executions, and the number of (predictive) deadlocks reported by three techniques (all/unique/real). The last column shows the number of locks in each direct cycle.

5.2 Implementation and Experimental Setup

AirLock was implemented on top of the Pin framework [37] for C/C++ programs with Pthread. It works under the Probe mode of Pin which itself incurs almost zero overhead.

We have reviewed a list of tools on deadlock detection to identify potential competitors for comparison. UnDead is the only online predictive detector that we have found. MagicLock, Sherlock [18], and WOLF [47] are listed as state-of-the-art deadlock detection tools [14] published in 2019 where both Sherlock and WOLF focus on soundness. ConLock [10] and ConLock+ [9] focus on triggering real deadlocks reported by MagicLock. Dirk [31] is a heavy-weight detector to detect sound deadlocks. Additionally, PickLock [49] focuses on soundness and was evaluated on a set of small java programs.

Finally, we selected the two representative deadlock detectors, MagicLock and UnDead for comparison purpose. Other predictive tools like IGoodLock [29] and MulticoreSDK [38] have been compared with MagicLock in previous works [6, 54] and UnDead is also based on IGoodLock with additional dependency pruning strategies.

For MagicLock, we used its implementation (provided by the author [6]). The original UnDead release is available online [54]; it contains two parts: deadlock detection and deadlock tolerance. We evaluated its detection part.

We conducted the experiments on a DELL Precision 5520 with a 2.80 GHz i7-7700HQ processor, Ubuntu 16.04 (x64), and GCC 4.8. We ran each benchmark ten times to collect data and compute averages. We set each execution time to be at most 10 hours.

Benchmarks	Bug ID	SLOC (k)	#Thd	#Locks	Summary of Inputs	Native Time	#Cycles (all/unique/real) UnDead	MagicLock	AirLock	#Locks in Cycles
Hawknl (1.6b3)	n/a	9.3	401	603/202	Deadlock in *nlshutdown()* and *nlclose()*	10.1s	400/1/1	400/1/1	400/1/1	2
SQLite (3.25.2)	1672	268.9	17	15/14	Deadlock in *sqlite3UnixEnterMutex()* and *sqlite3UnixLeaveMutex()*	56.3s	7075/5/1	7075/5/1	7075/5/1	2
MySQL1 (5.1.57)	60682	1146.7	314	1763/667	*show innodb status* deadlocks if *LOCK_thd_data* points *LOCK_open*	26.8s	101/2/2	101/2/2	101/2/2	2
MySQL2 (5.5.17)	62614	1282.7	50	287/121	*PUGE BINARY LOG* acquires two locks in the wrong order	20.1s	1100/2/2	1100/2/2	1100/2/2	2,3
FireFox (64.0)	n/a	9735.4	115	41694/3787	Open 30 web pages	60.9s	0/0/0	0/0/0	0/0/0	0
Chromium (71.0)	n/a	28146.2	37	261756/13055	Open 30 web pages	59.1s	0/0/0	0/0/0	0/0/0	0
Thunderbird (60.2.1)	n/a	9822.9	76	17107/2525	Fetch 1000 e-mails from a Gmail.com account	42.8s	0/0/0	0/0/0	0/0/0	0

Table 1: Basic statistics of the benchmarks.

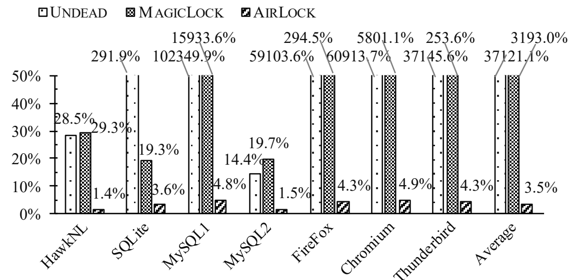


Figure 8: Time overhead.

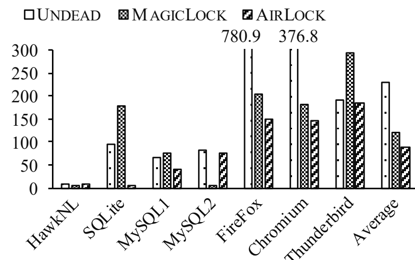


Figure 9: Memory consumption (MegaBytes).

5.3 Effectiveness and Efficiency

5.3.1 Overall Analysis. From the penultimate column of Table 1, we see that the three techniques detected the same number of deadlocks.

Figures 8 and 9 show the time overhead and memory consumption, respectively for each benchmark, as well as their averages. Note, MagicLock is an offline technique and we collected its costs on time and memory from its offline detection phase.

On time overhead, it is obvious that AirLock significantly outperformed both MagicLock and UnDead. It only incurred 1.4% to 4.9% overhead. However, UnDead incurred 14.4% to 28.5% overhead on two benchmarks; on remaining benchmarks, it incurred 2.9x to 1023x overhead. Actually, UnDead did not finish on Firefox and Chromium after running for 10 hours (i.e., our time limit). MagicLock incurred 19.3% to 29.3% overhead on three benchmarks. However, on remaining benchmarks, it incurred 2.5x to 159x overhead. On average, AirLock only incurred 3.5% overhead but UnDead and MagicLock incurred 371x and 31x overhead, respectively. [4]

The above results confirm that: analyzing and reducing the lock reachability graph before detecting cycles and deadlocks leads to efficient deadlock analysis. Besides, the optimization of MagicLock may be effective on some benchmarks but may also be ineffective on other benchmarks (e.g., MySQL2). However, considering the total detection time of MagicLock, it is acceptable as an offline deadlock detector for development. UnDead is based on the IGoodLock algorithm which has been shown to be inefficient [6].

On memory consumption, Figure 9 shows that there is no large difference. [5] On average memory, MagicLock and AirLock used almost the same amount and UnDead took about twice.

	HawkNL	SQLite	MySQL1	MySQL2	FireFox	Chromium	Thunderbird
$Moved$	0	0	0.34	0	3.2	4.3	0.28
$\Delta Mem.$	0	0	-1.4	0	-23.4	-50.1	-2.6
$Traces$	0.01	0.11	1.2	0.69	22.5	1.2	0.79

Table 2: External storage consumption and reduced memory consumption by AirLock (in MegaBytes).

AirLock adopts the strategy that may move edges from memory to external disk when a lock is destroyed, besides tracking an execution trace for deadlock construction. Therefore, we also collected its external storage consumption for moved edges (*Moved*), its reduction to memory consumption ($\Delta Mem.$), and the sizes of traces ($Traces$), as shown in Table 2.

From the table, AirLock consumed 0 to 4.3 MB external storage and reduced 0 to 50.1 MB memory consumption. It seems that moving edges into external storage can have little effect in terms of memory consumption. We will present further analysis in the next subsection. From the last row, we see that AirLock kept an acceptable size of execution traces for each execution.

5.3.2 Detailed Comparisons. Besides the overall comparison, we introduced one more comparison point: the trend of runtime data, including the number of edge sets for AirLock and the numbers of dependencies for UnDead and MagicLock. We collected such data every two seconds during execution. For MagicLock, we saved a copy of its trace every two seconds and calculated the data.

[4] Variable system load can affect execution time, yet any system load affects both AirLock and the execution itself. We speculate that this may bring proportional increases, resulting in an overhead percentage similar to what we reported. In effort to deal with variations, we repeated our experiment 10 times and computed the average.

[5] From Figure 9, AirLock consumed more memory than that by MagicLock. AirLock keeps a trace in memory and has a steady memory consumption, which varies across programs. In Figure 10 we see that the trend for the number of edges is increasing for MagicLock but steady for AirLock. So, we estimate that for longer execution time, AirLock will consume less memory than MagicLock.

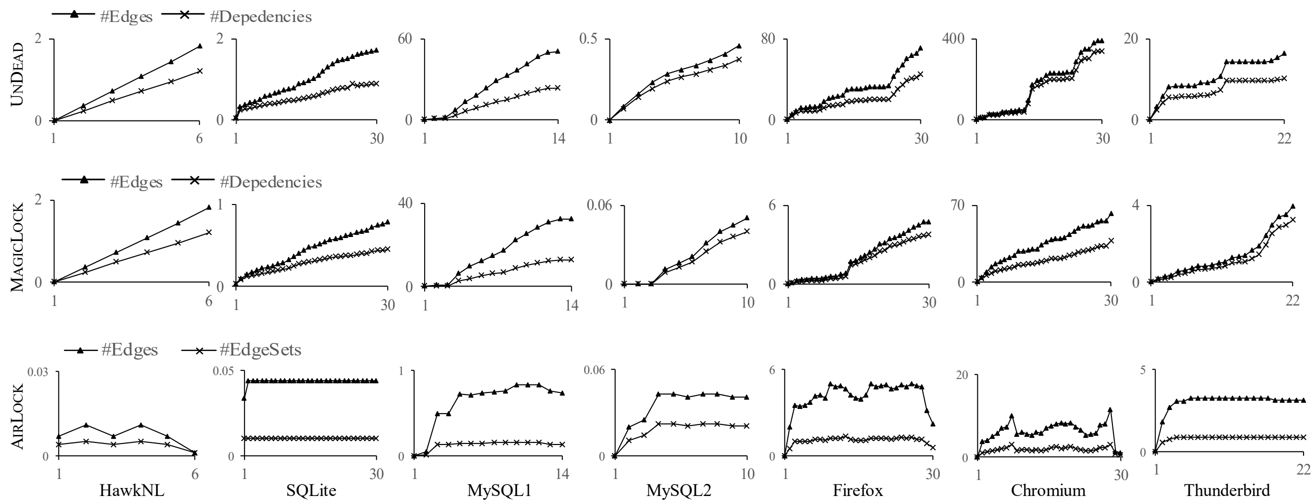


Figure 10: Trends on memory consumption in terms of the number of edges, dependencies and edge sets (where the x-axis shows the execution time ($\times 2$ seconds) and the y-axis shows the numbers ($\times 1,000$)).

To compare the three tools directly, we further converted the two kinds of data to the number of edges as follows: we transformed each edge set or each dependency into multiple direct edges according to the size of each edge set or the size of the lockset in each dependency. Note, AirLock only keeps edge sets for live locks; for any destroyed lock, its edge sets are deleted or moved out of memory (see Algorithm 3); hence, the number of edge sets also reflects the number of live locks. In each subfigure of Figure 10 , the x-axis shows the execution time (e.g., i stands for $i \times 2$ seconds) and the y-axis shows the number ($\times 1,000$) of periodically collected data.

Figure 10 clearly shows that AirLock maintained a stable number of edges and a stable number of edge sets after executing for several seconds; however, both UnDead and MagicLock incurred obvious increasing numbers of edges and dependencies. This confirms the insight behind the design of AirLock: most locks are dynamically created and destroyed, and the total number of live locks (i.e., the number of edge sets) keeps stable during executions.

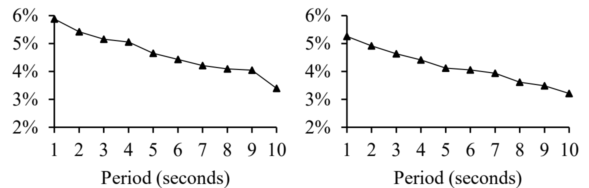
Summary. From the overall results in Table 1 and in Figures 8 and 9, as well as the detailed data in Figure 10, AirLock is applicable to on-the-fly deadlock detection while MagicLock is acceptable for offline deadlock detection for development.

5.4 Scalability under High Frequency Detection

AirLock is designed as an on-the-fly deadlock detector whose detection can be frequently conducted. To evaluate this feature, we conducted another experiment. We selected two benchmarks MySQL1 and Firefox where we are able to increase the sizes of inputs such that they can run for a longer time. In detail, for MySQL1, we configured a stress testing tool Sysbench [6] to send 10^8 SQL queries to it; for Firefox, we configured it to open 60 pages (note, these workloads are different from those used to measure native time in Table 1). Under the two configurations, MySQL1 was able

	Native	UnDead	MagicLock	AirLock
MySQL1	36.8s	>10 Hours	39.0s (106.1%)	1.1s (3.1%)
Firefox	100.4s	>10 Hours	324.7s (323.4%)	3.0s (3.1%)

(a) Overall results (time and overhead) of three techniques



(b) Overhead of AirLock on Firefox with periodical detections (c) Overhead of AirLock on MySQL1 with periodical detections

Figure 11: Scalability of AirLock

to run for >36 seconds and Firefox was able to run for >100 seconds. During their executions, we configured AirLock to detect deadlocks periodically. We set 10 different periods from 1 second to 10 seconds. That is, for every period of i seconds, AirLock detects deadlocks once. For each configuration, we collected the overall time overhead. For comparison, we also run AirLock (with the default configuration, i.e., one deadlock detection at the execution exit point), and UnDead and MagicLock under the same inputs on two benchmarks.

Figure 11 shows the overall results. In Figure 11(a), we show the detailed data on two benchmarks, including their native execution time, the time cost of each technique as well as the corresponding overhead. In Figures 11(b) and (c), we show the time overhead of AirLock with different deadlock detection periods (from 1 second to 10 seconds).

Figure 11(a) shows the similar result on time overhead as the previous one in Figure 8. That is, when both benchmarks run for about 36.8 or 100.4 seconds, both UnDead and MagicLock incurred

[6] https://github.com/akopytov/sysbench

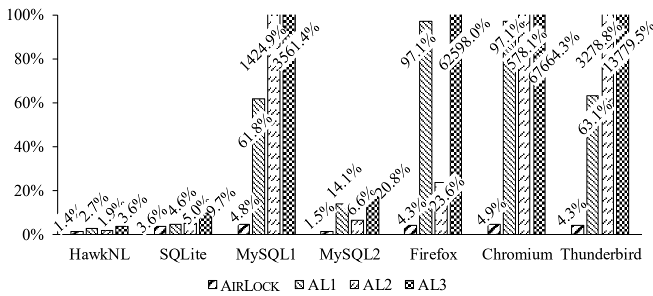


Figure 12: Evaluation on efficiency of AirLock's strategies (the y-axis is the time overhead of AirLock and its three variants AL1, AL2, and AL3).

larger overhead than AirLock. In details, UnDead failed to finish detection in 10 hours; MagicLock incurred >1x to >3x time overhead. AirLock incurred at most 3.1% time overhead.

For AirLock, as shown in Figure 11(b) and (c), even with a detection period of 1 second, it incurred less than 6% time overhead. With the increasing detection period from 1 second to 10 seconds, its time overhead decreased gradually. When the detection period is $\geq$ 5 seconds, AirLock incurred less than 5.0% time overhead on both benchmarks. In practice, such detection period is already highly frequent for long-running programs.

Summary. From the above analysis, AirLock can scale up to intensive deadlock detection on-the-fly. This makes it applicable to be integrated with multithreaded programs to provide anytime deadlock detection service.

5.5 Self-Comparison of AirLock

AirLock consists of three parallel parts where Parts II and III implement efficient cycle detection including two strategies: detecting simple cycles (lines 5–11 in Algorithm 2) first before a DFS search and reduce the reachability graph (in Algorithm 3). We built three tools AL1, AL2, and AL3, and conducted an additional experiment to answer what extent the two strategies accelerate cycle detection. AL1 is based on AirLock by disabling its detection of simple cycles (i.e., to directly use DFS for cycle detection among all edges); AL2 is based on AirLock by disabling the reachability reduction; and AL3 is based on AirLock by disabling both. All three tools were configured to detect deadlocks once at the execution exit point. The result is shown in Figure 12 where we also show the overhead of AirLock from Figure 8 for comparison purpose.

From the figure, we see that on three benchmarks (HawKNL, SQLite, and MySQL2), AirLock compromised its overhead by less than 20%. However, on the remaining four benchmarks, the overhead is compromised significantly by 0.6x to 676x. Such an overhead is even much larger than that by MagicLock and UnDead.

Besides, we also collected the data on the ratio of inconsistent lock acquisition orders out of all. It shows that, only 0.4% (0.11% on average) of lock acquisitions exhibit inconsistent lock orders. The exception is MySQL2, for which the percentage is 4%.

This above result further confirms the efficiency of the strategies in AirLock.

6 RELATED WORKS

6.1 Deadlock Detection

Besides dynamic approaches, deadlocks can also be detected by **static approaches** [16, 44, 51]. Like dynamic approaches, static ones can analyze program code to construct lock order graphs. They are scalable to the whole program and do not suffer heavy overhead. However, they usually report many false positives [51] due to imprecise static analysis techniques as well as lack of runtime information such as happens-before relation [35]. There are some sound static deadlock analyses, which are usually restricted to certain languages, for example, for C# programs [48], for C programs [34], or for barrier synchronizations [15]. They may rely on other techniques to guarantee soundness (e.g., pointer analysis [22, 52]).

Predictive deadlock detection usually produces **false positives**. There are two kinds of approaches to isolate real deadlocks. One is to trigger real deadlocks out of all detected ones. DeadlockFuzzer [29] adopts a straightforward scheduling (i.e., pause a thread right before it acquires its second lock and waits for all other threads to go into the same state) to trigger deadlock occurrences. It is known that such a scheduling can produce thrashing as pausing a thread may prevent other threads from making progress, resulting in a low probability to trigger deadlock occurrences. There are already a sequence of works trying to improve the probability [8–10, 46] by identifying a set of constraints and satisfy these constraints. Our AirLock focuses on predictive deadlock detection, it can be integrated with these tools to isolate real deadlocks.

Another kind of works aims to directly detect **real deadlocks** (i.e., without producing false positives during detection). The most recent one is the Dirk [31] where the similar idea is also adopted in RVPredict [25] on sound data race detection. This approach, unlike many previous deadlock detection tools, further monitors memory accesses and then extracts various constraints (from memory accesses and from synchronizations). By solving these constraints, it detects deadlocks that are deemed to really occur if all constraints are satisfied. AirLock focuses on on-the-fly predictive deadlock detection; it is challenging to analyze memory accesses on-the-fly without incurring heavy time overhead.

AirLock is an on-the-fly predictive deadlock detector. There are works on detecting deadlocks with a subsequent **deadlock prevention/healing** [30, 50, 54]. Gadara [50] statically inserts code to prevent deadlocks. Dimmunix [30] tries to bring deadlock immunity to a software product. It detects deadlock occurrences and prevents their second occurrences in later executions. UnDead [54] (as discussed in this paper) further tries to detect predictive deadlocks and to fix both real deadlocks and predictive deadlocks. Both Dimmunix and UnDead may report false positives; UnDead may further introduce other concurrency bugs due to its incomplete fixing strategy [54]. There are also some works targeting on preventing deadlocks in certain types of applications, e.g., database applications [20].

Deadlocks occur under certain thread interleaving and certain concurrent test cases. There are works that schedule threads [46]

and generate additional concurrent test cases [18, 42]. AirLock can be integrated into them to detect deadlocks efficiently.

6.2 Lightweight Online Data Race Detection

Predictive deadlock detection was not suitable for on-the-fly detection before AirLock. However, there are several works aiming at on-the-fly data race detection. Data race occurrences involve memory accesses and their detection usually incurs heavy overhead (e.g., up to 100x on C/C++ programs [41]). FastTrack [19] introduces the Epoch concept, together with optimization, it reduces overhead to 8x for Java programs. However, this overhead level is still high.

There are different kinds of works on reducing overhead for data race detection. The first kind is based on the **crowd-sourced** approach. RaceMob [32] adopts static analysis to firstly identify all potential data races (even with false positives). It distributes a small set of potential data races to each end-user, aiming at confirming the reality of them. As it requires that only one potential data race can be confirmed in each execution, its overhead is extremely low.

The second kind is based on **sampling**. By sampling a small set of memory accesses per execution, one can reduce overhead per execution. LiteRace [39] is based on a cold-region hypothesis: data races are more likely to exist in cold region. It adaptively samples cold regions (functions) only, removing overhead on monitoring hot regions. Pacer [5] is based on a short-distance hypothesis. It periodically samples an execution and fully tracks memory accesses in sampled periods. During non-sampled periods, it only checks for data race occurrences without updating tracking data. Pacer incurs an overhead proportional to a sampling rate.

The third kind is based on **hardware**. DataCollider [17] takes full advantage of data breakpoints. It samples a first memory access and then traps a second one by setting a data breakpoint on the same memory address. CRSampler [11] proposes clock race which can be sampled via data breakpoints (like DataCollider) but does not need to pause any thread (unlike DataCollider) to trap a second memory access.

Compared with these sampling approaches, AirLock fully tracks executions and does not miss any deadlock while incurring low overhead.

7 CONCLUSION

This paper presents a novel low-overhead on-the-fly predictive deadlock detection approach AirLock. The main novelty is that, AirLock maintains the lock reachability graph involving locks only and efficiently detects cycles on it. For each detected cycle, it then constructs a predictive deadlock. The experiments on seven real-world programs demonstrated that AirLock was significantly more efficient than existing works by incurring about 3.5% time overhead on average, making it suitable for on-the-fly deadlock detection, even under high frequency (e.g., per five seconds) deadlock detection.

ACKNOWLEDGMENTS

We sincerely thank the anonymous reviewers for helpful suggestions and insights for improving the paper. This work is supported in part by the National Key Research and Development Program of China (No. 2018YFB1403400), National Natural Science Foundation of China (NSFC) (Grant No. 61932012), the Key Research Program of Frontier Sciences, CAS (Grant No. ZDBS-LY-7006 and QYZDJ-SSW-JSC036), the Youth Innovation Promotion Association of the Chinese Academy of Sciences (YICAS) (Grant No. 2017151), and the Young Elite Scientists Sponsorship Program by CAST (Grant No. 2017QNRC001).

REFERENCES

[1] R. Agarwal, S. Bensalem, E. Farchi, K. Havelund, Y. Nir-Buchbinder, S. D. Stoller, S. Ur, and L. Wang. 2010. Detection of Deadlock Potentials in Multithreaded Programs. *IBM J. Res. Dev.* 54, 5 (Sept. 2010), 520–534. https://doi.org/10.1147/JRD.2010.2060276

[2] Rahul Agarwal, Liqiang Wang, and Scott D. Stoller. 2006. Detecting Potential Deadlocks with Static Analysis and Run-time Monitoring. In *Proceedings of the First Haifa International Conference on Hardware and Software Verification and Testing (HVC'05)*. Springer-Verlag, Berlin, Heidelberg, 191–207. https://doi.org/10.1007/11678779_14

[3] Saddek Bensalem and Klaus Havelund. 2006. Dynamic Deadlock Analysis of Multi-threaded Programs. In *Proceedings of the First Haifa International Conference on Hardware and Software Verification and Testing (HVC'05)*. Springer-Verlag, Berlin, Heidelberg, 208–223. https://doi.org/10.1007/11678779_15

[4] Swarnendu Biswas, Man Cao, Minjia Zhang, Michael D. Bond, and Benjamin P. Wood. 2017. Lightweight Data Race Detection for Production Runs. In *Proceedings of the 26th International Conference on Compiler Construction (CC 2017)*. ACM, New York, NY, USA, 11–21. https://doi.org/10.1145/3033019.3033020

[5] Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. 2010. PACER: Proportional Detection of Data Races. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. ACM, New York, NY, USA, 255–268. https://doi.org/10.1145/1806596.1806626

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

[7] Yan Cai and W. K. Chan. 2012. MagicFuzzer: Scalable Deadlock Detection for Large-scale Applications. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 606–616. http://dl.acm.org/citation.cfm?id=2337223.2337294

[8] Y. Cai, C. Jia, S. Wu, K. Zhai, and W. K. Chan. 2015. ASN: A Dynamic Barrier-Based Approach to Confirmation of Deadlocks from Warnings for Large-Scale Multithreaded Programs. *IEEE Transactions on Parallel and Distributed Systems* 26, 1 (Jan 2015), 13–23. https://doi.org/10.1109/TPDS.2014.2307864

[9] Y. Cai and Q. Lu. 2016. Dynamic Testing for Deadlocks via Constraints. *IEEE Transactions on Software Engineering* 42, 9 (Sep. 2016), 825–842. https://doi.org/10.1109/TSE.2016.2537335

[10] Yan Cai, Shangru Wu, and W. K. Chan. 2014. ConLock: A Constraint-based Approach to Dynamic Checking on Deadlocks in Multithreaded Programs. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 491–502. https://doi.org/10.1145/2568225.2568312

[11] Yan Cai, Jian Zhang, Lingwei Cao, and Jian Liu. 2016. A Deployable Sampling Strategy for Data Race Detection. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 810–821. https://doi.org/10.1145/2950290.2950310

[12] Vitaly Chipounov, Vlad Georgescu, Cristian Zamfir, and George Candea. 2009. Selective symbolic execution. In *Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep)*.

[13] Edmund M Clarke Jr, Orna Grumberg, Daniel Kroening, Doron Peled, and Helmut Veith. 2018. *Model checking*. MIT press.

[14] Tiago Cogumbreiro, Raymond Hu, Francisco Martins, and Nobuko Yoshida. 2018. Dynamic Deadlock Verification for General Barrier Synchronisation. *ACM Trans. Program. Lang. Syst.* 41, 1, Article Article 1 (Dec. 2018), 38 pages. https://doi.org/10.1145/3229060

[15] Tiago Cogumbreiro, Raymond Hu, Francisco Martins, and Nobuko Yoshida. 2018. Dynamic Deadlock Verification for General Barrier Synchronisation. *ACM Trans. Program. Lang. Syst.* 41, 1, Article 1 (Dec. 2018), 38 pages. https://doi.org/10.1145/3229060

[16] Jyotirmoy Deshmukh, E. Allen Emerson, and Sriram Sankaranarayanan. 2009. Symbolic Deadlock Analysis in Concurrent Libraries and Their Clients. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE '09)*. IEEE Computer Society, Washington, DC, USA, 480–491. https://doi.org/10.1109/ASE.2009.14

[17] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. 2010. Effective Data-race Detection for the Kernel. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, Berkeley, CA, USA, 151–162. http://dl.acm.org/citation.cfm?id=1924943.1924954

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

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

[20] Mark Grechanik, B. M. Mainul Hossain, Ugo Buy, and Haisheng Wang. 2013. Preventing Database Deadlocks in Applications. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, New York, NY, USA, 356–366. https://doi.org/10.1145/2491411.2491412

[21] Klaus Havelund and Thomas Pressburger. 2000. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer* 2, 4 (2000), 366–381.

[22] Michael Hind. 2001. Pointer Analysis: Haven'T We Solved This Problem Yet?. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '01)*. ACM, New York, NY, USA, 54–61. https://doi.org/10.1145/379605.379665

[23] Shin Hong, Jaemin Ahn, Sangmin Park, Moonzoo Kim, and Mary Jean Harrold. 2012. Testing Concurrent Programs to Achieve High Synchronization Coverage. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA 2012)*. ACM, New York, NY, USA, 210–220. https://doi.org/10.1145/2338965.2336779

[24] Jeff Huang, Qingzhou Luo, and Grigore Rosu. 2015. GPredict: Generic predictive concurrency analysis. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 847–857.

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

[26] Yannis E Ioannidis, Raghu Ramakrishnan, et al. 1988. Efficient Transitive Closure Algorithms.. In *VLDB*, Vol. 88. 382–394.

[27] Donald B Johnson. 1975. Finding all the elementary circuits of a directed graph. *SIAM J. Comput.* 4, 1 (1975), 77–84.

[28] Pallavi Joshi, Mayur Naik, Koushik Sen, and David Gay. 2010. An Effective Dynamic Analysis for Detecting Generalized Deadlocks. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '10)*. ACM, New York, NY, USA, 327–336. https://doi.org/10.1145/1882291.1882339

[29] Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. 2009. A Randomized Dynamic Program Analysis Technique for Detecting Real Deadlocks. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 110–120. https://doi.org/10.1145/1542476.1542489

[30] Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, and George Candea. 2008. Deadlock Immunity: Enabling Systems to Defend Against Deadlocks. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, USA, 295–308. http://dl.acm.org/citation.cfm?id=1855741.1855762

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

[32] Baris Kasikci, Cristian Zamfir, and George Candea. 2013. RaceMob: Crowdsourced Data Race Detection. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 406–422. https://doi.org/10.1145/2517349.2522736

[33] Valerie King and Garry Sagert. 2002. A fully dynamic algorithm for maintaining the transitive closure. *J. Comput. System Sci.* 65, 1 (2002), 150–167.

[34] Daniel Kroening, Daniel Poetzl, Peter Schrammel, and Björn Wachter. 2016. Sound Static Deadlock Analysis for C/Pthreads. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, New York, NY, USA, 379–390. https://doi.org/10.1145/2970276.2970309

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

[36] Brandon Lucia and Luis Ceze. 2013. Cooperative Empirical Failure Avoidance for Multithreaded Programs. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, New York, NY, USA, 39–50. https://doi.org/10.1145/2451116.2451121

[37] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 190–200. https://doi.org/10.1145/1065010.1065034

[38] Zhi Da Luo, Raja Das, and Yao Qi. 2011. Multicore SDK: A Practical and Efficient Deadlock Detector for Real-World Applications. In *Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation (ICST '11)*. IEEE Computer Society, Washington, DC, USA, 309–318. https://doi.org/10.1109/ICST.2011.22

[39] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. 2009. LiteRace: Effective Sampling for Lightweight Data-race Detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 134–143. https://doi.org/10.1145/1542476.1542491

[40] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs. , 14 pages. http://dl.acm.org/citation.cfm?id=1855741.1855760

[41] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, Vol. 42. ACM, 89–100.

[42] Michael Pradel and Thomas R. Gross. 2012. Fully Automatic and Precise Detection of Thread Safety Violations. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 521–530. https://doi.org/10.1145/2254064.2254126

[43] Paul Purdom. 1970. A transitive closure algorithm. *BIT Numerical Mathematics* 10, 1 (1970), 76–94.

[44] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. 2012. Scalable and Precise Dynamic Datarace Detection for Structured Parallelism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 531–542. https://doi.org/10.1145/2254064.2254127

[45] Neha Rungta, Eric G. Mercer, and Willem Visser. 2009. Efficient Testing of Concurrent Programs with Abstraction-Guided Symbolic Execution. In *Proceedings of the 16th International SPIN Workshop on Model Checking Software*. Springer-Verlag, Berlin, Heidelberg, 174–191. https://doi.org/10.1007/978-3-642-02652-2_16

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

[47] Malavika Samak and Murali Krishna Ramanathan. 2014. Trace Driven Dynamic Deadlock Detection and Reproduction. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP âĂŹ14)*. Association for Computing Machinery, New York, NY, USA, 29âĂŞ42. https://doi.org/10.1145/2555243.2555262

[48] Anirudh Santhiar and Aditya Kanade. 2017. Static Deadlock Detection for Asynchronous C# Programs. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 292–305. https://doi.org/10.1145/3062341.3062361

[49] Francesco Sorrentino. 2015. PickLock: A Deadlock Prediction Approach under Nested Locking. In *Proceedings of the 22nd International Symposium on Model Checking Software - Volume 9232 (SPIN 2015)*. Springer-Verlag, Berlin, Heidelberg, 179âĂŞ199. https://doi.org/10.1007/978-3-319-23404-5_13

[50] Yin Wang, Terence Kelly, Manjunath Kudlur, Stéphane Lafortune, and Scott Mahlke. 2008. Gadara: Dynamic Deadlock Avoidance for Multithreaded Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, USA, 281–294. http://dl.acm.org/citation.cfm?id=1855741.1855761

[51] Amy Williams, William Thies, and Michael D. Ernst. 2005. Static Deadlock Detection for Java Libraries. In *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP'05)*. Springer-Verlag, Berlin, Heidelberg, 602–629. https://doi.org/10.1007/11531142_26

[52] Robert P. Wilson and Monica S. Lam. 1995. Efficient Context-sensitive Pointer Analysis for C Programs. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI '95)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/207110.207111

[53] Jie Yu, Satish Narayanasamy, Cristiano Pereira, and Gilles Pokam. 2012. Maple: A Coverage-driven Testing Tool for Multithreaded Programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12)*. ACM, New York, NY, USA, 485–502. https://doi.org/10.1145/2384616.2384651

[54] Jinpeng Zhou, Sam Silvestro, Hongyu Liu, Yan Cai, and Tongping Liu. 2017. UNDEAD: Detecting and Preventing Deadlocks in Production Software. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. IEEE Press, Piscataway, NJ, USA, 729–740. http://dl.acm.org/citation.cfm?id=3155562.3155654