Dynamic Testing for Deadlocks via Constraints

Yan Cai, *Member, IEEE*, and Qiong Lu

Abstract—Existing deadlock detectors are either not scalable or may report false positives when suggesting cycles as potential deadlocks. Additionally, they may not effectively trigger deadlocks and handle false positives. We propose a technique called *ConLock*$^+$, which firstly analyzes each cycle and its corresponding execution to identify a set of scheduling constraints that are necessary conditions to trigger the corresponding deadlock. The *ConLock*$^+$ technique then performs a second run to enforce the set of constraints, which will trigger a deadlock if the cycle is a real one. Or if not, *ConLock*$^+$ reports a steering failure for that cycle and also identifies other similar cycles which would also produce steering failures. For each confirmed deadlock, *ConLock*$^+$ performs a static analysis to identify conflicting memory access that would also contribute to the occurrence of the deadlock. This analysis is helpful to enable developers to understand and fix deadlocks. *ConLock*$^+$ has been validated on a suite of real-world programs with 16 real deadlocks. The results show that across all 811 cycles, *ConLock*$^+$ confirmed all of the 16 deadlocks with a probability of ≥80%. For the remaining cycles, *ConLock*$^+$ reported steering failures and also identified that five deadlocks also involved conflicting memory accesses.

Index Terms—Deadlock Triggering, Scheduling, Should-happen-before relation, Constraint, Reliability, Verification.

1 Introduction

Synchronization primitives are widely used in multi-threaded programs [36] to coordinate their threads. If the synchronization order is not properly designed, it may produce concurrency bugs such as data races [12], [36], [55], atomicity violations [35], and deadlocks [6], [8], [28]. A *deadlock* [28] occurs when two or more threads wait to acquire some resource that is held by another thread related to one of the competing threads, resulting in a circular waiting cycle between this set of threads.

Once a deadlock occurs, it is not difficult to detect and reproduce it [54]. However, deadlocks only occur under certain program interleavings, which makes online detection ineffective. Hence, both static [47], [51] and dynamic [6], [13], [38] deadlock prediction techniques are useful to detect deadlocks. Static techniques analyze program code to infer cycles from lock acquisitions by different threads. This analysis may report a large number of false positives [51] (i.e., deadlocks that cannot actually occur in practical executions). Dynamic techniques [8], [9] may also report false positives, but generally less frequently than their static counterparts.

- *Y. Cai is with State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China. E-mail: ycai.mail@gmail.com.*
- *Q. Lu is with Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences, Beijing, China. E-mail: luqiong13@otcaix.iscas.ac.cn.*

Manuscript received (insert date of submission if desired). Please note that all acknowledgments should be placed at the end of the paper, before the bibliography.

The *happened-before relation* [33], [45] or the *segmentation graph* [8] on the corresponding execution trace can be helpful to eliminate some of the false positives; however, true positives may also be eliminated [28], which makes this type of elimination risky.

The most recently developed techniques (including *DeadlockFuzzer* [28], *MagicScheduler* [15], and *ASN* [14]) actively *confirm* each predicted deadlock (i.e. each cycle) by scheduling a particular program execution to achieve the confirmation. *PCT* [10] randomly executes a program (by following pre-generated schedules based on an approximation of the program [10]) and has a probabilistic guarantee of finding any concurrency bugs. A minor adaptation of *PCT* can also be used as an alternative to deadlock confirmation. However, our experiment in Section 5 will show that these methods cannot effectively confirm deadlocks. Additionally, they do not include a strategy for handling false positives, although they may have a high probability of triggering deadlocks [14], [15], [28]. Hence, a program has to be executed at least once to confirm each predicted cycle.

In our discussion, we refer to an execution that is performed to suggest cycles as a *predictive run*. Similarly, we refer to an execution that is performed to confirm whether a suggested cycle is a real deadlock as a *confirmation run*. We also assume that cycles have been suggested by a predictive technique from a predictive run [8], [13].

In this paper, we propose a novel dynamic technique called *ConLock*$^+$, which is a constraint-based approach that manipulates program scheduling to confirm deadlocks and handle false positives. Given a predictive run and a set of cycles, *ConLock*$^+$ firstly analyzes the predictive run and generates a set of constraints for each cycle. Each constraint specifies the order of a pair of events (i.e., the lock acquisitions and releases). This order should be followed when the two events occur in a confirmation run. *ConLock*$^+$ then manipulates a confirmation run and attempts to enforce these constraints for each cycle *c*, which will trigger a deadlock if cycle *c* is actually a real one; or if not, will report a steering failure, which indicates that the current run can no longer meaningfully confirm cycle *c*. If a steering failure is reported, *ConLock*$^+$ then infers other similar cycles and removes a set of cycles (known as the akin cycles of cycle *c*) that cannot be triggered due to the same steering failure. If a deadlock is triggered, *ConLock*$^+$ performs a further static analysis to find conflicting memory accesses that may also contribute to the occurrence of the deadlock.

ConLock$^+$ is an extension to our preliminary technique *ConLock* that was previously reported [16]. There are three main contributions of *ConLock*$^+$: (1) *ConLock* relies on a set of constraints to schedule a confirmation run. Since the set of generated constraints may be large, *ConLock* equivalently removes some redundant constraints based on two properties [16]. However, even after this equivalent reduction, *ConLock* still has to rely on a heuristic to start its active scheduling until the program execution reaches a set of scheduling points [16]. These points are helpful to enable some constraints to be skipped, such as those which have their first events happening before the scheduled points of the same threads, and this results in the enforced set of constraints not being equivalent to the originally-generated set. In this extension, we use three properties instead of two to further remove redundant constraints without using any scheduling points. As a result, *ConLock*$^+$ schedules a confirmation run that aims to enforce a set of constraints that is equivalent to the originally-generated set. Hence, *ConLock*$^+$ produces a higher confirmation probability on cycles that are real deadlocks. (2) We propose the concept of *akin cycles*, with which *ConLock*$^+$ only needs to check a small set of cycles, significantly reducing both the number of confirmation runs and the total time to check all cycles. (3) For each triggered cycle, *ConLock*$^+$ performs a static analysis to infer possible memory access conflicts. These analyses can help developers to understand the cause of deadlocks and to fix deadlocks.

We have implemented a prototype of *ConLock*$^+$ and validated it on a suite of real-world programs, including a total of 16 real deadlocks. We have compared *ConLock*$^+$ with *MagicScheduler* [15], *DeadlockFuzzer* [28], *PCT* [10], and *ASN* [14] on both real deadlocks and false positives. In these experiments, *ConLock*$^+$ achieved a consistently higher probability (80%–100%) of triggering all 16 real deadlocks; in contrast, the other techniques failed to trigger between 6 to 11 real deadlocks and/or were not effective enough. The other techniques did not handle false positives, while *ConLock*$^+$ correctly reported steering failures on false positives. Compared to *ConLock* [16], *ConLock*$^+$ triggered real deadlocks with higher probabilities, with the exception of two deadlocks. Additionally, *ConLock*$^+$ significantly reduced the number of confirmation runs and the total time to check all cycles. Finally, *ConLock*$^+$ identified all five unique sets of conflicting memory accesses. These accesses also contribute to the occurrence of deadlocks since they cause conflicts (i.e., a read and a write pair) to a shared conditional, resulting in inconsistent lock acquisition orders between different threads.

The remainder of the paper is organized as follows: Section 2 revisits the preliminary background to this work. Section 3 gives an example which has motivated our work. Section 4 presents the *ConLock*$^+$ framework. Section 5 describes a validation experiment, and reports the experimental results. Section 6 reviews related work. Section 7 concludes this paper.

2 Preliminaries

2.1 Dynamic Events and Execution Traces

During execution of a multithreaded program, there are two types of critical operations {*acq*, *rel*} that operate on locks, where *acq* represents a *lock acquisition* and *rel* represents a *lock release*. Other synchronization primitives, such as barriers or notify/wait operations, are handled using these two operations [12], [22].

An ***event*** $e = \langle t, acq/rel, m@s, ls\rangle$ can be used to denote that a thread *t* performs an operation *acq* or *rel* on a lock *m* at a certain program execution point *s* (i.e., a ***Site*** [17], [28]), while that thread *t* is also holding a set of locks *ls* (i.e., a ***Lockset***) at the same time, each of which is also associated with a site where thread *t* is acquiring a corresponding lock.

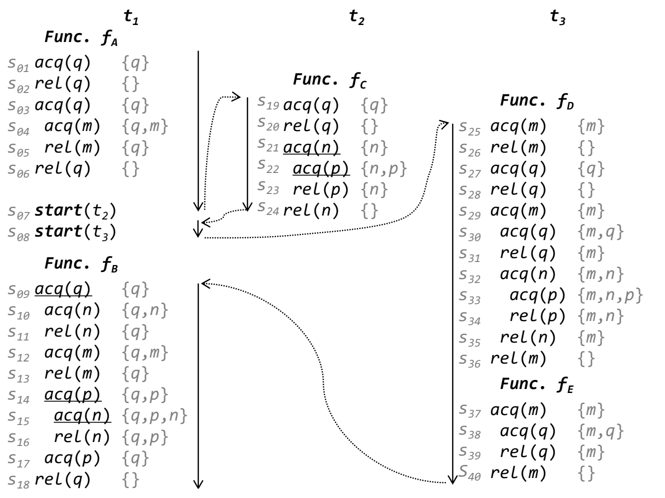
An execution ***trace*** σ of a program *p* is a sequence of events, and σ_t is the projection of trace σ on thread *t*.

In this paper, we focus on dynamic events produced during concrete executions, since static lock acquisition statements may be executed multiple times; and multiple executions of the same lock acquisition statement are usually independently involved in a deadlock (see the example in Fig. 5 of [12]). Therefore, we use the concept of *site* [17] to distinguish between different executions of the same static statement, where the site is computed based on the abstraction algorithm [17] for each event. In this paper, we simply adopt a code line number to denote the site of the event for that line, since each static event is only executed once (i.e., one code line number corresponds to one event). Furthermore, false positives in this paper also refer to dynamic scenarios that cannot form deadlock occurrences under the same inputs as the predictive runs.

2.2 Cycle

A ***cycle*** [15], [28] is a sequence of *k* mutually dependent events. That is, if two events $e_i = \langle t_i, acq, m_i@s_i, ls_i\rangle$ and $e_{i+1} = \langle t_{i+1}, acq, m_{i+1}@s_{i+1}, ls_{i+1}\rangle$ are two consecutive events of a cycle, then event e_i is dependent on event e_{i+1} (i.e., $m_i \in ls_{i+1}$). This dependency indicates that, during an execution, thread t_i may try to acquire a lock m_i which is already being held by thread t_{i+1}. If all of these dependencies occur during an execution then this causes a deadlock, as each thread is waiting for another thread from the same cycle to release a lock, and hence none of the threads can progress any further.

Therefore, a cycle models a potential deadlock. We refer to site s_i in event e_i of cycle *c* as the ***deadlocking site*** of thread t_i in cycle *c*.



```
t1                          t2                          t3
Func. fA
s01 acq(q)   {q}
s02 rel(q)   {}
s03 acq(q)   {q}
s04  acq(m)  {q,m}
s05  rel(m)  {q}
s06 rel(q)   {}
s07 start(t2)
s08 start(t3)
Func. fB
s09 acq(q)    {q}
s10  acq(n)   {q,n}
s11  rel(n)   {q}
s12  acq(m)   {q,m}
s13  rel(m)   {q}
s14  acq(p)   {q,p}
s15   acq(n)  {q,p,n}
s16   rel(n)  {q,p}
s17  acq(p)   {q}
s18 rel(q)    {}

Func. fC
s19 acq(q)    {q}
s20 rel(q)    {}
s21 acq(n)    {n}
s22  acq(p)   {n,p}
s23  rel(p)   {n}
s24 rel(n)    {}

Func. fD
s25 acq(m)    {m}
s26 rel(m)    {}
s27 acq(q)    {q}
s28 rel(q)    {}
s29 acq(m)    {m}
s30  acq(q)   {m,q}
s31  rel(q)   {m}
s32  acq(n)   {m,n}
s33   acq(p)  {m,n,p}
s34   rel(p)  {m,n}
s35  rel(n)   {m}
s36 rel(m)    {}
Func. fE
s37 acq(m)    {m}
s38  acq(q)   {m,q}
s39  rel(q)   {m}
s40 rel(m)    {}
```

Fig. 1. An example program containing three deadlocks and three false positives (see TABLE 1), where functions f_B and f_C are adapted from `JDBC Connector 5.0` [3] (Bug ID: 2147).

For instance, the underlined five statements in Fig. 1 represent a single deadlock modelled as the cycle $c_4 = \langle\langle t_1, acq, n@s_{15}, \{q@s_{09}, p@s_{14}\}\rangle, \langle t_2, acq, p@s_{22}, \{n@s_{21}\}\rangle\rangle$. The deadlocking sites of threads t_1 and t_2 are s_{15} and s_{22}, respectively.

Given a cycle c, we denote all locks that are waiting to be acquired by any thread in c as **WLock$_c$**. Similarly, we denote the set of all locks *held* by any thread in c as the set **HLock$_c$**. Moreover, the site that acquires lock $m \in$ WLock$_c$ and the site that acquires lock $n \in$ HLock$_c$ are denoted by **WSite$_c$**(m) and **HSite$_c$**(n), respectively.

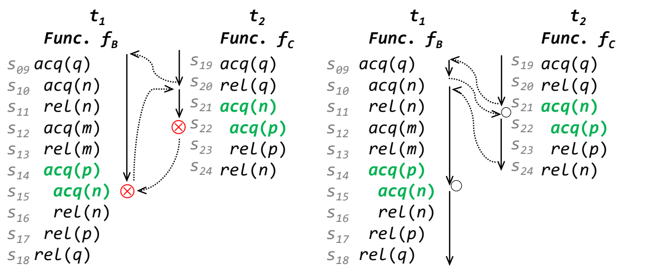
For the above cycle c_4 in Fig. 1, we have: WLock$_{c4}$ = $\{n, p\}$, WSite$_{c4}$ $(n) = s_{15}$, WSite$_{c4}$ $(p) = s_{22}$, HLock$_{c4}$ = $\{n, p, q\}$, HSite$_{c4}$ $(n) = s_{21}$, HSite$_{c4}$ $(p) = s_{14}$, and HSite$_{c4}$ $(q) = s_{09}$.

Note that a cycle may actually be a false positive rather than a real deadlock. It is possible for deadlock detectors to report false positives as they infer a cycle based on events that occur at different times during the same program execution. Therefore, there is no guarantee that these events will occur at the same time in another execution.

Given a cycle $c = \langle e_1, e_2, \dots e_n\rangle$, if all n events $e_1, e_2, \dots e_n$ can be executed by n threads $t_1, t_2, \dots t_n$ at the same time, the cycle c is a real deadlock; otherwise, the cycle c is a false positive. For example, the cycle $c_1 = \langle\langle t_1, acq, m@s_{04}, \{q@s_{03}\}\rangle, \langle t_3, acq, q@s_{30}, \{m@s_{29}\}\rangle\rangle$ in Fig. 1 is a false positive, since the event in thread t_3 only happens after the occurrence of the event in thread t_1 (since thread t_1 starts thread t_3 at line s_{08}).

3 Motivating Example

Fig. 1 shows a motivating example program. The program contains three threads t_1, t_2, and t_3 that acquire and release four locks q, m, n, and p within five functions f_A to f_E. Thread t_1 firstly acquires two locks q and m within function f_A and then starts two threads t_2 and t_3. Next, it executes several lock acquisitions within function f_B. Threads t_2 and t_3 also execute several lock acquisitions and releases within function f_C and within functions f_D and f_E, respectively.



```
t1                 t2                  t1                 t2
Func. fB           Func. fC            Func. fB           Func. fC
s09 acq(q)         s19 acq(q)          s09 acq(q)         s19 acq(q)
s10  acq(n)        s20 rel(q)          s10  acq(n)        s20 rel(q)
s11  rel(n)        s21 acq(n)          s11  rel(n)        s21 acq(n)
s12  acq(m)        s22  acq(p)         s12  acq(m)        s22  acq(p)
s13  rel(m)        s23  rel(p)         s13  rel(m)        s23  rel(p)
s14  acq(p)        s24 rel(n)          s14  acq(p)        s24 rel(n)
s15   acq(n)                           s15   acq(n)
s16   rel(n)                           s16   rel(n)
s17  rel(p)                            s17  rel(p)
s18 rel(q)                             s18 rel(q)
```

(a) Execution 1
(a scenario with deadlock)

(b) Execution 2
(a scenario without deadlock)

Fig. 2. Two execution scenarios of deadlock c_4.

Fig. 1 also shows the execution trace σ, indicated by the arrows. During this trace, no deadlock occurs. However, six cycles are reported when this trace is analyzed using existing dynamic *predictive* techniques (e.g., [15], [28], [38]), which are listed in TABLE 1. Out of these six cycles, only three cycles c_3, c_4, and c_5 are real deadlocks, while the others are false positives. However, without actually triggering these cycles, it is not known whether they are real deadlocks or false positives.

Therefore, it is desirable that active deadlock detection techniques can both (1) detect real deadlocks and (2) handle false positives. In the remainder of this section, we will illustrate the limitations of existing techniques in relation to both of these aspects.

3.1 Detecting Real Deadlocks

Randomized Testing schedules random program executions repeatedly to discover the schedules that may cause deadlocks to occur. This type of techniques require a large number of executions to trigger deadlock occurrences. For example, *PCT* [10] is a representative randomized testing technique. It probabilistically guarantees to detect a concurrency bug with a probability of $1/(n \times k^{d-1})$ if the bug has a depth d involving n threads that totally execute k steps. For cycle c_4, the guaranteed probability is $1/(2 \times 40^{3-1}) \approx 3.125 \times 10^{-4}$. For large-scale programs (e.g., MySQL that can produce more than 4×10^6 events, see Section 5), such a guarantee may not be high enough.

Randomized Active Schedulers introduce scheduling manipulations based on random testing, with the aim of exploring certain parts of executions. However, the existing techniques (e.g., *DeadlockFuzzer* [28] and *MagicScheduler* [15]) have a low probability of triggering a given cycle. In the remainder of this subsection, we will illustrate this problem for the deadlock c_4. For ease of explanation, we will only consider two functions f_B and f_C within threads t_1 and t_2, respectively, based on two executions, as shown in Fig. 2.

DeadlockFuzzer (*DF* for short) and *MagicScheduler* (*MS* for short) both use a heuristic to schedule each individual thread randomly to trigger an occurrence of a given cycle and *they suspend a thread if the thread holds a set of locks and tries to acquire another lock at the deadlocking site of this thread* [15], [28]. Let us consider a deadlock c_4. *DF* and *MS* aim to suspend thread t_1 immediately before it executes the event $acq(n)$ at site s_{15}, and to suspend thread t_2 immediately before it executes the event $acq(p)$ at site s_{22}. This

TABLE 1
SIX CYCLES IN THE EXAMPLE PROGRAM

Cycle	False positive?	Can be handled by				
		PCT	*HB*	*DF/MS*	*ASN*	*ConLock*$^+$
$c_1 = \langle\langle t_1, acq, m@s_{04}, \{q@s_{03}\}\rangle, \langle t_3, acq, q@s_{30}, \{m@s_{29}\}\rangle\rangle$	Yes	✗	✓	✗	✗	✓
$c_2 = \langle\langle t_1, acq, m@s_{04}, \{q@s_{03}\}\rangle, \langle t_3, acq, q@s_{38}, \{m@s_{37}\}\rangle\rangle$	Yes	✗	✓	✗	✗	✓
$c_3 = \langle\langle t_1, acq, m@s_{12}, \{q@s_{09}\}\rangle, \langle t_3, acq, q@s_{30}, \{m@s_{29}\}\rangle\rangle$	No	?	✗	?	✓ *	✓
$c_4 = \langle\langle t_1, acq, n@s_{15}, \{q@s_{09}, p@s_{14}\}\rangle, \langle t_2, acq, p@s_{22}, \{n@s_{21}\}\rangle\rangle$	No	?	✗	?	? *	✓
$c_5 = \langle\langle t_1, acq, m@s_{12}, \{q@s_{09}\}\rangle, \langle t_3, acq, q@s_{38}, \{m@s_{37}\}\rangle\rangle$	No	?	✗	?	✓ *	✓
$c_6 = \langle\langle t_1, acq, n@s_{15}, \{q@s_{09}, p@s_{14}\}\rangle, \langle t_3, acq, p@s_{33}, \{m@s_{29}, n@s_{32}\}\rangle\rangle$	Yes	✗	✗	✗	✗	✓

*The symbols "✓","✗", and "**?**" indicate whether a technique can, or cannot, handle a cycle, or depends on randomized scheduling, respectively.*
For ASN on the real deadlocks (c3, c4, and c5), the analysis further considers the happened-before relation (i.e., it ignores events at s_{01}–s_{06}) and the results are marked with a star ""; otherwise, ASN fails to handle these three cycles.*

type of schedule steering has also been adopted to trigger occurrences of known data races [27] or suspicious data races [32] via crowdsourcing validation [32], [52], and targets at two simultaneous memory accesses with or without lock acquisitions.

However, as a deadlock can involve at least four lock acquisitions, the above steering may fail. For example, it could be challenging to directly apply the above strategy to trigger cycle c_4. If the random execution is *Execution* 1, the deadlock could be triggered. However, if the random execution is *Execution* 2, the deadlock will not be triggered, since thread t_2 reaches site s_{22} first, and is then suspended by *DF* and *MS*. When thread t_1 executes next, after acquiring lock q at site s_{09}, it cannot acquire a further lock n at site s_{10} as it is held by thread t_2, therefore, both schedulers have to resume thread t_2. After thread t_2 releases lock n at site s_{24}, there is no longer any possibility that the deadlock will be triggered, even if thread t_1 reaches site s_{15}.

The above type of problem is known as *thrashing* [28]. Thrashing can be formally defined as occurring if, during confirmation of a cycle $c = \langle e_1, e_2, \ldots e_n\rangle$ by an active scheduler X, thread t_i is suspended by X immediately before executing event $e_i \in c$ before another thread t_j executes the event $e_j \in c$, and thread t_j is blocked from acquiring a lock m that is held by the suspended thread t_i (i.e., $m \in ls(t_i)$). Thus, thread t_j will not reach its deadlocking site to execute event e_j unless thread t_i releases lock m.

For deadlock c_4, thrashing may also occur during other executions as well as *Execution* 2. For example, if thread t_1 is the first to reach its deadlocking site s_{15} before thread t_2 reaches site s_{19}, thread t_2 can then not reach its deadlocking site s_{22}, as it cannot acquire lock q at site s_{19} since it is already being held by thread t_1 at site s_{09}.

ASN [14] was previously proposed by us to prevent the occurrence of thrashing. This technique suspends threads in a given cycle two (or three) more times than *DF* and *MS*, to improve the probability of triggering a deadlock. However, it heavily relies on randomized scheduling before suspending any thread. Although *ASN* guarantees that some types of deadlocks will be triggered, it is limited due to its three basic assumptions (see Section 4.4 in [14]). Additionally, when there are two or more deadlocks, thrashing may also still occur [14]. For the three real deadlocks i.e., c_3, c_4, and c_5 from the example program, *ASN* may also encounter thrashing and hence produces a low probability.

This can be illustrated by reference to cycle c_4 involving threads t_1 and t_2. *ASN* firstly identifies three barriers (considering its optimization): for thread t_1, they are sites s_{09}, s_{14}, and s_{15}; and for thread t_2, they are sites s_{19}, s_{21} and s_{22}. During scheduling, *ASN* firstly waits for both threads to reach their first barriers (i.e., s_{09} for t_1 and s_{19} for t_2) and then suspends both threads, which is feasible. Next, *ASN* tries to suspend the two threads at their second barriers (i.e., s_{14} for t_1 and s_{21} for t_2). However, if thread t_1 is the first to acquire lock q at its first barrier (i.e., site s_{09}), thread t_2 is then blocked at site s_{19} and hence cannot reach its second barrier. To address this thrashing, *ASN* has to continue thread t_1 to release lock q at site s_{18}. As a result, cycle c_4 cannot be triggered because thread t_1 has passed its deadlocking site s_{15}. On the other hand, if thread t_2 could acquire lock q at site s_{19}, the cycle c_4 could be triggered. For this reason, the ability of *ASN* is also compromised by thrashing, similar to *DF* and *MS*.

Therefore, existing techniques are ineffective at triggering real deadlocks. In TABLE 1, we compare the ability of techniques discussed above and our *ConLock*$^+$ technique (which will be explained in Section 4) to trigger all real deadlocks (i.e., c_3, c_4, and c_5) on the example program. The first column shows the six cycles predicted from the execution shown in Fig. 1. The second column indicates whether each cycle is a false positive or not. The last main column ("`Can be handled by`") shows whether each technique can correctly trigger/identify each real deadlock (indicated by a tick "✓"), or correctly identifies each false positive (indicated by a cross "✗", to be discussed in the next subsection), or whether it only triggers/identifies a real deadlock depending on particular scheduling (indicated by a "**?**"). The technique *HB* in the last major column is discussed in the next subsection.

3.2 Handling False Positives

Due to a low probability of triggering or detecting real deadlocks, existing techniques cannot identify false positives. For example, *PCT* is only designed to detect real concurrency bugs (including deadlocks); other active schedulers *DF*, *MS*, and *ASN* cannot trigger all real deadlocks due to thrashing, not mentioning handling false positives. Although *ASN* has a higher probability of triggering a real deadlock than *DF* and *MS*, it cannot handle the scenario where two or more deadlocks occur. Our

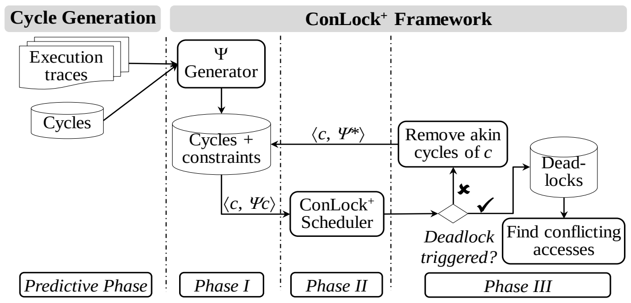


Fig. 3. An overview of the *ConLock*+ framework.

Algorithm 1: *ConLock*+ Framework

```
    Input: $p$ – a program
    Input: $\sigma$ – an execution trace of the program $p$
    Input: $\epsilon$ – a set of cycles from $\sigma$
01 $\Psi := \varnothing$      //a set of pairs of a cycle $c$ and its constraint set $\Psi c$
02 for each cycle $c \in \epsilon$ do
03     $\Psi c := \Psi$-Generator $(\sigma, c)$                //Algorithm 2
04     $\Psi := \Psi \cup \{\langle c, \Psi c\rangle\}$
05 end for
06 for each cycle $c \in \epsilon$ do
07     call ConLock+_scheduler($p$, $c$, $\Psi c$)            //Algorithm 3
08     if a steering failure is reported then
09         $\epsilon' := $ Find_akinCycles($\epsilon$, $c$, $\Psi$)       //Algorithm 4
10         $\epsilon := \epsilon \setminus \epsilon'$
11     else if a real deadlock is triggered then
12         call Find_deadlockSensitiveAccesses($p$, $c$) //Algorithm 5
13     end if
14 end for
```

experiment shows that many real-world programs (e.g., `MySQL`) may contain multiple deadlocks.

The happened-before relationship [33] may be used to handle false positives by identifying cycles where the lock acquisitions cannot occur concurrently [9], [45]. For example, in cycle c_1, the fork-join relationship between the two threads prevents the two lock acquisitions on lock m (at sites s_{04} and s_{30}) from executing concurrently. Therefore, cycle c_1 can be marked as a false positive. Although the use of the happened-before relationship can filter out some false positives, it may also filter out real deadlocks [28]. Additionally, not all false positives can be filtered out by the happened-before relationship, e.g., c_6. In TABLE 1, we also compare the use of the happened-before relationship with other techniques in column "`HB`".

4 CONLOCK+

4.1 Overview

There are three phases within *ConLock*+ that check all of the given cycles within an execution trace. Fig. 3 depicts the workflow of *ConLock*+ and Algorithm 1 shows its algorithm framework.

Phase I: For each given cycle c, *ConLock*+ generates a set of constraints Ψc according to Ψ-Generator (lines 02–05). Each constraint specifies the order that two events from different threads should follow in a confirmation run.

Phase II: For each cycle c, *ConLock*+ actively schedules a confirmation run which attempts to enforce all constraints of this cycle (i.e., Ψc, line 07). There are two possible outcomes: (1) a deadlock occurs, indicating that cycle c is a real deadlock; or (2) a steering failure occurs, indicating that cycle c is a false positive (with respect to the current confirmation run, see Section 2.1). A steering failure occurs when *ConLock*+ cannot schedule an execution by enforcing all of the constraints (i.e., further scheduling would violate at least one order specified by a constraint in Ψc). These two outcomes significantly differentiate *ConLock*+ from existing techniques. However, thrashing may also occur using *ConLock*+; in these cases, a second confirmation run on the same cycle is scheduled.

Phase III: If a steering failure occurs in Phase II, *ConLock*+ removes all akin cycles of cycle c (lines 08–10). An akin cycle c' of cycle c indicates that if a steering failure occurs during confirming cycle c, it may also occur when cycle c' is confirmed (see Section 4.4). Therefore, confirmation of all akin cycles can be skipped, which reduces the number of confirmation runs and the total time required to check all cycles. If a real deadlock occurs in Phase II, *ConLock*+ performs a static analysis to infer whether any conflicting accesses (i.e., deadlock sensitive accesses) contribute to the deadlock occurrence (line 12), aside from the lock acquisitions.

ConLock+ repeatedly performs the last two phases until all cycles have been processed.

4.2 Phase I: Generation of Constraint Set

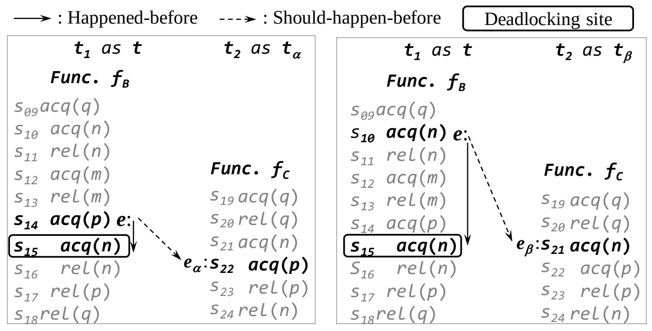
If a cycle indicates a real deadlock, a schedule must exist that will suspend each thread in the cycle at its deadlocking site. This understanding has been used by existing active schedulers [15], [28] to extract information from a predictive run that guides the manipulation of a confirmation run. However, an active scheduler should also avoid the occurrence of thrashing as much as possible. Hence, we include the following constraint: prior to executing a lock acquisition at the deadlocking site, each thread should not be artificially blocked by any other thread from the same cycle as much as possible. That is, for an active scheduler, a thread must only be suspended for some necessary reasons that may trigger the corresponding deadlock occurrence.

We then formulate a novel relationship entitled the ***should-happen-before*** relationship to (1) effectively prevent the occurrence of thrashing and (2) precisely suspend each thread involved in a cycle at its deadlocking site. We note that the *should-happen-before relationship* is the relationship between two events from the execution trace of a predictive run (where the run itself has no deadlock occurrence). It denotes that the two events *should happen* in a specified order in a confirmation run.

4.2.1 *An Intuition of the Should-Happen-Before Relationship*

Suppose that we try to trigger cycle $c_4 = \langle\langle t_1, acq, n@s_{15}, \{q@s_{09}, p@s_{14}\}\rangle, \langle t_2, acq, p@s_{22}, \{n@s_{21}\}\rangle\rangle$ in the example execution. Let us consider the two threads and their events (prior to the two deadlocking sites s_{15} and s_{22}) shown in Fig. 2.

To precisely trigger a deadlock, thread t_2 must be suspended at site s_{22}. However, as discussed in Section 3.1, simply suspending thread t_2 at site s_{22} may prevent thread t_1 from executing the event at site s_{10} (i.e., $acq(n)$), since



(a) *Rule 1* on the lock p: $s_{14} \rightsquigarrow s_{22}$ (b) *Rule 2* on the lock n: $s_{10} \rightsquigarrow s_{21}$

Fig. 4. Examples of *Rule* 1 and *Rule* 2 on *Execution* 2.

lock n has already been acquired by thread t_2 at site s_{21} (i.e., a thrashing occurs). Therefore, before suspending thread t_2 at its deadlocking site s_{22}, thread t_1 should firstly execute all of its acquisitions and releases on lock n.

In other words, if it is expected that a lock (e.g., lock n) will be held by a thread (e.g., thread t_2 at site s_{21}) and another thread within the cycle will be waiting for this lock (e.g., thread t_1 at site s_{15}), any acquisitions and releases (e.g., at site s_{10} and s_{11}) on this lock should happen before the event where the lock is held (e.g., at site s_{21}).

However, if thread t_1 is simply suspended at its deadlocking site s_{15} to trigger cycle c_4, thread t_2 may also be prevented from executing $acq(q)$ at site s_{19} as lock q is already held by thread t_1 at site s_{09} (i.e., thrashing occurs). This case is different from the case above (i.e., on lock n) as when deadlock c_4 occurs, lock p is held by thread t_1 only and thread t_2 is not waiting for it. Therefore, application of the intuition discussed above to events on lock p will not resolve the thrashing in this case.

Therefore, for any held lock (e.g., lock p), regardless of whether or not there is another thread waiting for it, any acquisitions and releases should happen before the event where it is held (e.g., at site s_{09}).

In the next subsection, we will formally introduce the two rules for the *should-happen-before relationship*.

4.2.2 Should-Happen-Before Relationship

We firstly revisit the *happened-before relationship* [33] (HBR for short, denoted as $\rightarrowtail$) that describes the relationship between two events over a given trace of the predictive run. The HBR [33] is defined by the following rules:

a) *Program order*: if two events e_1 and e_2 are executed by the same thread, and event e_1 appears before event e_2 in the trace, then $e_1 \rightarrowtail e_2$.

b) *Lock acquire and release*: if two events e_r and e_a are a lock release event and a lock acquisition event, respectively, and the event e_r appears prior to the event e_a in the trace within the same lock, then $e_r \rightarrowtail e_a$.

c) *Transitivity*: if $e_1 \rightarrowtail e_2$ and $e_2 \rightarrowtail e_3$, then $e_1 \rightarrowtail e_3$.

To simplify our subsequent presentation, we refer to an event e_i which occurs at the deadlocking site of thread t_i as $\boldsymbol{\varepsilon(c, t_i)}$. We also use the site of an event e to denote e when describing the two relationships. For example, for cycle c_4, $\varepsilon(c_4, t_1) = s_{15}$ and $\varepsilon(c_4, t_2) = s_{22}$.

Given an execution trace σ, a cycle c on σ, and three threads t, t_α, and t_β that are involved in cycle c, where $t \neq t_\alpha$ and $t \neq t_\beta$, the ***should-happen-before relationship*** (SHBR for short, denoted as $\rightsquigarrow$) is defined by the following rules:

***Rule* 1**: Suppose that e and e_α are two events that are performed by two different threads t and t_α, respectively, and they both operate on the same lock m. If the three conditions (1) $m \in \mathtt{WLock}_c$, (2) $e \rightarrowtail \varepsilon(c, t)$, and (3) $e_\alpha = \varepsilon(c, t_\alpha)$ are satisfied, then $e \rightsquigarrow e_\alpha$.

***Rule* 2**: Suppose that e and e_β are two events performed by two threads t and t_β, respectively, and they both operate on the same lock n. If the three conditions (1) $n \in \mathtt{HLock}_c$, (2) $e \rightarrowtail \varepsilon(c, t)$, and (3) $e_\beta = \langle t_\beta, acq, n@\mathtt{HSite}_c(n), ls_\beta\rangle$ for some ls_β are satisfied, then $e \rightsquigarrow e_\beta$. (Note that $e_\beta \neq \varepsilon(c, t_\beta)$, instead $e_\beta \rightarrowtail \varepsilon(c, t_\beta)$.)

In detail, *Rule* 1 ***prevents*** predictable thrashing from occurring on locks in the set $\mathtt{WLock}_c$. Fig. 4(a) uses *Execution* 2 (see Fig. 2) to illustrate this rule via lock p and cycle c_4. In Fig. 4(a), lock p is in $\mathtt{WLock}_{c4}$, the site s_{22} is the deadlocking site for thread t_2 (i.e., t_α in *Rule* 1) that operates on this lock p; and the deadlocking site for thread t_1 (i.e., thread t in *Rule* 1) is the site s_{15}. *Rule* 1 specifies that any lock acquisition or release event on this lock p performed by thread t_1 (e.g., the event e at site s_{14}) that happened before the event $\varepsilon(c_4, t_1)$ at site s_{15} *should happen before* the event (i.e., e_α) performed by thread t_2 at its deadlocking site s_{22}. Thus, according to *Rule* 1, we get $s_{14} \rightsquigarrow s_{22}$.

Rule 2 further prevents predictable thrashing on locks in the set $\mathtt{HLock}_c$. Fig. 4(b) uses *Execution* 2 to illustrate this rule via lock n. In Fig. 4(b), lock n is in $\mathtt{HLock}_{c4}$, and thread t_2 (i.e., thread t_β in *Rule* 2) holds a lockset $\{n@s_{21}\}$ when thread t_2 is about to acquire lock p at its deadlocking site s_{22}. We also recall that the deadlocking site for thread t_1 (i.e., thread t in *Rule* 2) is the site s_{15}. *Rule* 2 specifies that any lock acquisition or release event on lock n, performed by thread t_1 and happened before the event occurring at its deadlocking site s_{15}, *should happen before* the lock acquisition event on n at site s_{21} (i.e., the event e_β). Thus, by *Rule* 2, we obtain $s_{10} \rightsquigarrow s_{21}$. Lock n has also been released by thread t_1 at site s_{11}. Therefore, by the same method we also obtain $s_{11} \rightsquigarrow s_{21}$.

Generation of all SHBRs on c_4: *Rule* 1 and *Rule* 2 are now applied to identify a complete set of SHBRs with respect to cycle c_4. We recall that *Execution* 2 in Fig. 2(b) operates on three locks $\{n, q, p\}$ from cycle c_4. Cycle c_4 has two deadlocking sites: the site s_{15} for thread t_1 and the site s_{22} for thread t_2. Additionally, $\mathtt{WLock}_{c4}$ is $\{n, p\}$ and $\mathtt{HLock}_{c4}$ is $\{n, q, p\}$.

For lock n: *Rule* 2 has been applied on this lock in order to identify $s_{10} \rightsquigarrow s_{21}$, and $s_{11} \rightsquigarrow s_{21}$ in the above illustration of *Rule* 2. Thread t_1 performs an event on lock n at its deadlocking site s_{15} (i.e., $\varepsilon(c_4, t_1)$). For thread t_2, there is only one event $e = \langle t_2, acq, n@s_{21}, \{\}\rangle$ operating on lock n and $e \rightarrowtail \varepsilon(c_4, t_2)$. By *Rule* 1, we get $s_{21} \rightsquigarrow s_{15}$.

For lock p: *Rule* 1 has been applied on this lock to identify $s_{14} \rightsquigarrow s_{22}$. As thread t_2 is not operating any event on lock p that was happened before the event $\varepsilon(c_4, t_2)$ at site s_{22}, *Rule* 2 produces no further SHBR for lock p.

For lock q: *Rule* 1 gives no SHBR on this lock because lock q is not in $\mathtt{WLock}_{c4}$. For cycle c_4, lock q is within the

Algorithm 2: Ψ-Generator

```
    Input: $\sigma$ – an execution trace
    Input: $c$ – a cycle from the trace $\sigma$
    Output: $\Psi c$ – a constraint set with respect to the cycle $c$
01  $\Psi c := \emptyset$, WLOCK$_c := \emptyset$, HLOCK$_c := \emptyset$
02  for each event $\langle t, req, m@s, ls\rangle$ in $c$ do
03  |  WLOCK$_c :=$ WLOCK$_c \cup \{m\}$
04  |  for each $n@s_n \in ls$, HLOCK$_c :=$ HLOCK$_c \cup \{n\}$
05  end for
06  for each $t \in c$ do
07  |  let $i := p$ such that $\boldsymbol{\sigma}_t[p] = \varepsilon(c, t)$
08  |  while $i$ -- > 0 do
09  |  |  let $\boldsymbol{\sigma}_t[i]$ be $e = \langle t, op, l@s, ls\rangle$ //$e \rightarrowtail \varepsilon(c, t)$, $op \in \{acq, rel\}$
10  |  |  if $l \in$WLock then //By Rule 1
11  |  |  |  for each $e_\alpha = \langle t_\alpha, acq, m@s_\alpha, ls_\alpha\rangle = \varepsilon(c, t_\alpha) \wedge t_\alpha \neq t$ do
12  |  |  |  |  if $l = m$ then, $\Psi c := \Psi c \cup \{e \rightsquigarrow e_\alpha\}$ // $m \in$ WLOCK$_c$
13  |  |  |  end for
14  |  |  end if
15  |  |  if $l \in$ HLOCK$_c$ then //By Rule 2
16  |  |  |  let $e_\beta := \langle t_\beta, acq, l@s_\beta, ls_\beta\rangle$, where $t_\beta \in Threads(c) \wedge t_\beta \neq t$
17  |  |  |  let $\varepsilon(c, t_\beta) = \langle t_\beta, acq, m@s', ls'\rangle$ //the deadlocking site of $t_\beta$
18  |  |  |  if $l@s_\beta \in ls'$ then, $\Psi c := \Psi c \cup \{e \rightsquigarrow e_\beta\}$ // $s_\beta$= HSITE$_c(l)$
19  |  |  end if
20  |  end while
21  end for
```

lockset of an event for thread t_1. By *Rule* 2, any event on lock q that happened before $\varepsilon(c_4, t_2)$ should happen before the lock acquisition event on q performed by thread t_1 at site s_{09}. Since for thread t_2, $s_{19} \rightarrowtail \varepsilon(c_4, t_2)$ and $s_{20} \rightarrowtail \varepsilon(c_4, t_2)$, we then get $s_{19} \rightsquigarrow s_{09}$ and $s_{20} \rightsquigarrow s_{09}$.

Based on *Execution* 2, we can identify a set of six SHBRs for cycle c_4: $\{s_{10} \rightsquigarrow s_{21}, s_{11} \rightsquigarrow s_{21}, s_{21} \rightsquigarrow s_{15}, s_{14} \rightsquigarrow s_{22}, s_{19} \rightsquigarrow s_{09}, s_{20} \rightsquigarrow s_{09}\}$, as depicted in Fig. 5(a) by dotted arrows.

Execution 2 fails to trigger a deadlock, and its execution path is $\langle s_{19}, s_{20}, s_{09}, s_{21}\text{–}s_{24}, s_{10}\text{–}s_{18}\rangle$. This path does not follow three of the SHBRs described above (i.e., $s_{10} \rightsquigarrow s_{21}$, $s_{11} \rightsquigarrow s_{21}$, $s_{14} \rightsquigarrow s_{22}$) for cycle c_4. In fact, any execution path that does not follow all of these SHBRs fails to trigger the deadlock.

Execution 1 triggers a deadlock occurrence, and its execution path is $\langle s_{19}, s_{20}, s_{09}\text{–}s_{14}, s_{21}\rangle$ before the two deadlocking sites s_{15} and s_{22}. This execution path satisfies all SHBRs of cycle c_4.

In Section 3, we have illustrated that both *MS* and *DF* suffer from the occurrence of thrashing. This thrashing occurs because thread t_2 acquires lock n at site s_{21} before thread t_1 attempts to acquire the same lock at site s_{10}, and yet thread t_2 is suspended by *MS* and *DF* at site s_{22}. The above set of SHBR highlights that this execution does not follow the SHBR $s_{10} \rightsquigarrow s_{21}$. The thrashing that occurs using *ASN* (see Section 3) is also caused by not following the SHBR $s_{19} \rightsquigarrow s_{09}$ on lock q.

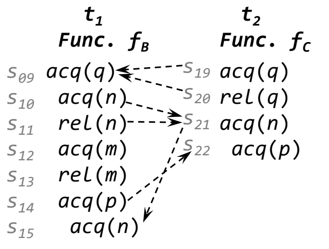
4.2.3 Generation of the Should-Happen-Before Relationship

ConLock+ treats each identified SHBR as a scheduling ***constraint*** in the confirmation run. Algorithm 2 shows the constraint set generation algorithm (Ψ-Generator for short).

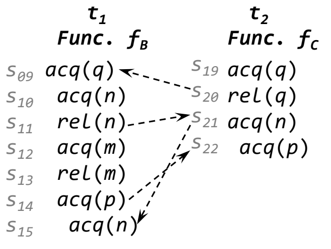
Given an execution trace σ and a cycle *c*, Algorithm 2 firstly extracts all locks in WLOCK$_c$ and HLOCK$_c$ (lines 02–05). Then, it checks each event in the projection σ_t of trace σ for each thread *t* with respect to the two rules (lines 10–21). The checking is performed in a reverse program order, starting from the deadlocking site of thread *t* (lines 07–09). For each event $e = \langle t, op, l@s, ls\rangle$, the algorithm checks whether the lock *l* is in the set WLOCK$_c$ (line 10). If so, the algorithm further checks *e* against e_α to determine whether the pair of events *e* and e_α forms an SHBR based on *Rule* 1 (lines 11–12). If this is also the case, the relationship $e \rightsquigarrow e_\alpha$ is added to the set Ψc (line 12). The algorithm then checks whether the lock *l* is in the set HLOCK$_c$ (line 15). If so, it further checks whether there is an event e_β operating on lock *l* such that $l@s_\beta$ of the event e_β is in the lockset ls' of $\varepsilon(c, t_\beta)$ (lines 16–18), indicating that the site s_β is HSITE$_c(l)$. If such an event e_β exists, the algorithm adds the relationship $e \rightsquigarrow e_\beta$ into Ψc (line 18) based on *Rule* 2.

ConLock+ can identify all such SHBRs for each cycle, and it can actually easily be proved based on the two rules that each constraint identified by *ConLock*+ is a necessary condition to trigger the corresponding cycle. Hence, it can improve the probability of triggering a deadlock by avoiding the occurrence of thrashing.

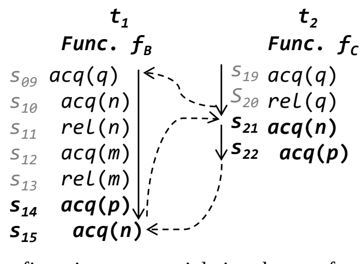
ConLock+ schedules a confirmation run by enforcing all of the constraints. However, there may be a high runtime overhead incurred by scheduling an execution with such a large set of constraints, which may in certain cases become infeasible, as has been experienced previously with `MySQL` (e.g., where more than 2,000 constraints were produced). Thus, in the next subsection, we present an equivalent constraint reduction algorithm to reduce the number of constraints.



(a) The generated set of constraints.



(b) The reduced set of constraints.



(c) A confirmation run not violating the set of constraints and triggering a deadlock occurrence.

Fig. 5. Generation and Reduction of constraints of the cycle c_4 as well as a scheduling not violating the generated set of constraints.

4.2.4 Reduction of Constraints

The equivalent reduction of constraints is based on the following three properties as illustrated in Fig. 6:

Property 1: If a constraint set includes both $e_1 \rightsquigarrow e_2$ and $e_2 \rightsquigarrow e_3$, then the constraint $e_1 \rightsquigarrow e_3$ is redundant.

Property 2: If a constraint set includes both $e_1 \rightsquigarrow e_3$ and $e_2 \rightsquigarrow e_3$ such that e_1 happens before e_2 ($e_1 \rightarrowtail e_2$) within

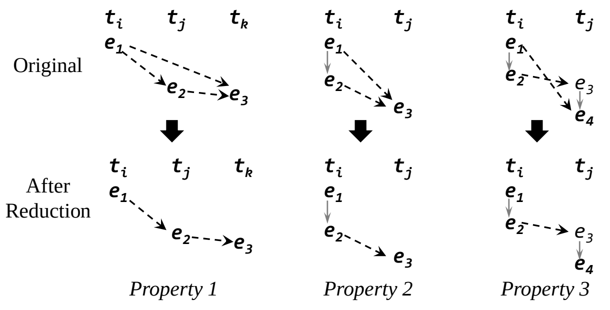


Fig. 6. Illustration on three properties.

the same thread, then the constraint $e_1 \rightsquigarrow e_3$ is redundant. (A special case of this is when e_2 is a lock release event and e_1 is the corresponding lock acquisition event of e_2.)

***Property* 3**: If a constraint set includes both $e_1 \rightsquigarrow e_4$ and $e_2 \rightsquigarrow e_3$ such that e_1 happens before e_2 ($e_1 \rightarrowtail e_2$) within the same thread and e_3 happens before e_4 ($e_3 \rightarrowtail e_4$) within the same thread, then constraint $e_1 \rightsquigarrow e_4$ is redundant.

For each of the three properties, the redundant constraint is already implicitly enforced by both the happened-before relationship and the should-happen-before relationship, as illustrated in Fig. 6.

Given a set of constraints generated by Algorithm 2, these three properties can be applied on the set to produce a smaller but equivalent set. The reduction algorithm is straightforward: each of the three properties is recursively applied to every triple of constraints until no more reductions in constraints are possible. Applying these three properties on the constraint set of cycle c_4 (originally produced by Algorithm 2) removes two of the constraints: $\{s_{10} \rightsquigarrow s_{21}, s_{19} \rightsquigarrow s_{09}\}$, as shown in Fig. 5(b).

By including this reduction algorithm, there is no requirement for *ConLock*+ to adopt any scheduling points to start its scheduling. A scheduling point is an event which happens before the deadlocking site of a thread, when the thread holds no lock. It is used by *ConLock* [16] but it suffers from the scalability issue, because it has to check a large number of constraints repeatedly during its confirmation runs. Although an equivalent reduction can be made based on two properties, even after reduction, the number of constraints remains relatively large. Therefore, *ConLock* skips all of the constraints with a first event that happens before the scheduling point [16]. For these points, the number of constraints to be checked can be non-equivalently reduced to an acceptable level (e.g., less than 10 in our experiment). However, during scheduling, *ConLock* also has to wait for all threads to reach their scheduling points (see lines 04–12 of *ConLock* scheduler [16]). This may decrease the probability of deadlock confirmation, especially for large-scale programs (e.g., `MySQL`).

The generalized reduction algorithm can equivalently reduce the number of constraints for each cycle to an acceptable level (e.g., less than 10). This generalization will also be evaluated in our experiment in Section 5.4.4.

Algorithm 3: *ConLock*+**_Scheduler**

```
    Input: $p$ – a program
    Input: $c$ – a cycle
    Input: $\Psi c$ – a set of constraints of the cycle $c$
01  $EnabledSet \coloneqq$ all threads in $c$, $SuspendedSet \coloneqq \varnothing$
02  for each $h \in \Psi c$, $State(h) \coloneqq idle$
03  for each thread $t$ do
04  |  $State(t) \coloneqq Enabled$, $LS(t) \coloneqq \varnothing$, $Req(t) \coloneqq \varnothing$, $Site(t) \coloneqq \varnothing$
05  end for
06  while $\exists\, t \in c \wedge State(t) = Enabled$ do
07  |  let $e \coloneqq \langle t, op, m@s, ls\rangle$ be the next event of the thread $t$
08  |  //check $e$ against each constraint in $\Psi c$
09  |  if $\exists\, h = e_a \rightsquigarrow e_b \in \Psi c, e_b = e \wedge State(h) = Idle$ then
10  |  |  $State(h) \coloneqq Active$, $State(t) \coloneqq Waiting$ on $h$
11  |  |  if a steering failure occurs then
12  |  |  |  print "A steering failure occurs."
13  |  |  |  halt //Early termination of this confirmation run
14  |  |  end if
15  |  |  continue //postpone the execution of the event $e$
16  |  else if $\exists\, h = e_a \rightsquigarrow e_b \in \Psi c, e_a = e \wedge State(h) = Active$ then
17  |  |  $State(h) \coloneqq Used$, $Notify(h)$ //$State(t')$: = $Enabled$
18  |  else if $\exists\, h = e_a \rightsquigarrow e_b \in \Psi c, e_a = e \wedge State(h) = Idle$ then
19  |  |  $State(h) \coloneqq Used$
20  |  end if
21  |  if $op = acq$ then //execute $e$ and check for deadlock
22  |  |  $Req(t) \coloneqq m$, $Site(t) \coloneqq s$
23  |  |  call $CheckDeadlock()$
24  |  |  $Req(t) \coloneqq \varnothing$, $LS(t) \coloneqq LS(t) \cup \{m@s\}$
25  |  else if $op = rel$ then
26  |  |  $LS(t) \coloneqq LS(t) \setminus \{m@s'\}$ for some $s'$
27  |  end if
28  |  $execute(e)$
29  end while
30  Function $CheckDeadlock()$
31  |  if $\exists$ a sequence $\langle e_1, e_2, \ldots, e_n\rangle$, where $e_i = \langle t_i, acq, Req(t_i)@Site(t_i),$
    |  $LS(t_i)\rangle$, for $1 \le i \le n$, is a cycle then
32  |  |  print "a deadlock occurs." halt
33  |  end if
34  end Function
```

4.3 Phase II: ConLock+ Scheduler

This section presents the *ConLock*+ scheduler for a single cycle. Given a program p, a cycle c, and a set of constraints Ψc of the cycle c, *ConLock*+ schedules the confirmation run by enforcing the given constraint set Ψc. It immediately stops the current confirmation run whenever it detects a steering failure, indicating that the current run can no longer meaningfully confirm the cycle. A few auxiliary concepts are firstly presented before the scheduling algorithm is introduced.

State of a constraint. Each constraint $h = e_a \rightsquigarrow e_b$ has the following state (denoted as `State(`h`)`):

- `Idle`: if both events e_a and e_b have not been executed. This state means that the constraint h has to be checked later after events e_a and e_b have occurred.
- `Active`: if event e_b is about to be executed but event e_a has not been executed. To enforce this constraint, the thread involved in event e_b has to wait until the event e_a occurs.
- `Used`: if event e_a has been executed. This state means that there is no need to further track this constraint as it has already been enforced in the current run.

State of a thread. Each thread t has the following state (denoted as `State(`t`)`):

- `Enabled`: if thread t can be scheduled to execute its next event.

- `Waiting`: if thread t is about to execute an event e, but there is a constraint, say $e' \rightsquigarrow e$ and the event e' has not been executed. To enforce this constraint, *ConLock*$^+$ suspends thread t until event e' has been executed. In such cases, we say that thread t is *waiting* on the constraint, and is thus in the `Waiting` state.
- `Disabled`: if thread t has not started, or has terminated, or has been suspended by the OS. In this case, thread t cannot be scheduled anymore.

A ***steering failure*** occurs in a confirmation run with respect to a cycle c if:

- $\nexists\, t \in c$, such that `State(`t`)` = `Enabled`, and,
- $\exists\, t \in c$, such that `State(`t`)` = `Waiting`.

The occurrence of a steering failure indicates that none of the threads in cycle c can be scheduled to execute their next event; otherwise, a constraint cannot be enforced. In other words, since no thread in c is in an `Enabled` state, all threads in c are either in a `Disabled` or in a `Waiting` state. For any thread that is in a `Disabled` state, it cannot be scheduled to execute its next event. The only way to continue the execution of these threads is to continue threads in the `Waiting` state. However, each `Waiting` thread t is waiting on a constraint, say $h = e' \rightsquigarrow e$ where the event e is the next event of thread t and the event e' is an expected (i.e., not executed) event from a different thread t'. To continue thread t, the constraint h will not be enforced. Since each constraint is a necessary condition to trigger the corresponding deadlock, the deadlock occurrence will fail to be triggered if a constraint is not enforced. Hence, *ConLock*$^+$ terminates the execution since continuation is pointless.

The *ConLock*$^+$ scheduler is shown in Algorithm 3. A program p, a cycle c, and a set of constraints Ψc are taken as the inputs. The scheduler firstly enables all threads; no thread is suspended (line 01). The state of each constraint is set as `Idle` (line 02). The status and other data for each thread are then initialized (line 04 and 05) including three maps: from thread t to a lockset as `LS(`t`)`, from t to its requested lock as `Req(`t`)`, and from t to its requested site as `Site(`t`)`.

ConLock$^+$ then starts its guided scheduling (lines 07–29). It randomly fetches the next event e from a random and `Enabled` thread (line 07). Before executing event e, *ConLock*$^+$ checks e against each constraint in Ψc that is not in the `Used` state, and determines the states of the selected constraint and the current thread t (lines 10–20) to enforce all constraints. There are three cases:

- If there is any constraint $h = e_a \rightsquigarrow e_b$ such that `State(`h`)` = `Idle` and the current event $e = e_b$, the execution of event e will be postponed until event e_a is executed. *ConLock*$^+$ then sets `State(`h`)` = `Active` and `State(`t`)` = `Waiting` on h (lines 10). It further checks whether any steering failure occurs, and reports any violation that occurs (lines 11–14).
- If there is any constraint $h = e_a \rightsquigarrow e_b$ such that `State(`h`)` = `Active` and the current event $e = e_a$, *ConLock*$^+$ sets `State(`h`)` = `Used` and updates the state of every thread that is `Waiting` on h to be `Enabled` (lines 16–17). On line 17, `Notify(`h`)` is used to indicate the change in state of each thread (say t') waiting on this constraint h from `Waiting` to `Enabled`.
- If there is any constraint $h = e_a \rightsquigarrow e_b$ such that `State(`h`)` = `Idle` and the current event $e = e_a$, *ConLock*$^+$ sets `State(`h`)` = `Used` (lines 18–19).

ConLock$^+$ then checks the type of event e. If it is a lock acquisition, *ConLock*$^+$ updates the three maps `Req`, `Site`, and `LS`, and calls the function `CheckDeadlock()`(lines 21–24). Otherwise, if event e is a lock release, *ConLock*$^+$ updates the map `LS` only (lines 25–26). *ConLock*$^+$ then executes event e.

If the function `CheckDeadlock()` (lines 30–34) finds any cycle, *ConLock*$^+$ reports the occurrence of a real deadlock and terminates the current run.

Fig. 5(c) shows the scheduling for cycle c_4 by enforcing the set of constraints shown in Fig. 5(b). The scheduling successfully triggers a deadlock occurrence.

Examples of false positives: we have discussed how *ConLock*$^+$ guides confirmation runs to trigger deadlocks. In the remainder of this subsection, we show how *ConLock*$^+$ handles these false positives (i.e., cycles c_1, c_2 and c_6 in TABLE 1), which distinguishes this technique from other existing active schedulers.

Cycle c_1 involves lock acquisitions at sites s_{03} and s_{04} by thread t_1 and at sites s_{29} and s_{30} by thread t_3. This cycle is a false positive because the lock acquisitions by thread t_1 occur before the start of thread t_3. Given its constraint set: $\{s_{03} \rightsquigarrow s_{30}, s_{28} \rightsquigarrow s_{03}, s_{29} \rightsquigarrow s_{04}\}$ (after reduction), the second constraint $s_{28} \rightsquigarrow s_{03}$ (which specifies a lock acquisition order on lock q) cannot be enforced by *ConLock*$^+$. In other words, upon lock acquisition of q by thread t_1 at s_{03}, *ConLock*$^+$ detects that there is an `Idle` constraint $s_{28} \rightsquigarrow s_{03}$ (line 09 of Algorithm 3). It then updates the state of this constraint to be `Active` and the state of thread t_1 to be `Waiting` (line 11). Next it detects a steering failure: there is only one thread (i.e., thread t_1) and that is in `Waiting` state. For cycle c_2, the case is similar to that of cycle c_1.

For cycle c_6, its constraint set is: $\{s_{13} \rightsquigarrow s_{29}, s_{31} \rightsquigarrow s_{09}, s_{32} \rightsquigarrow s_{15}, s_{14} \rightsquigarrow s_{33}\}$ (after reduction). The first two constraints $s_{13} \rightsquigarrow s_{29}$ and $s_{31} \rightsquigarrow s_{09}$ cannot be enforced at the same time. In the first case, where thread t_1 reaches site s_{09} before thread t_3 reaches site s_{31}, *ConLock*$^+$ updates the state of thread t_1 to be `Waiting` based on checking the constraint $s_{31} \rightsquigarrow s_{09}$ (lines 10–11). Once thread t_3 reaches site s_{29}, its state is also updated to `Waiting`, based on checking the constraint $s_{13} \rightsquigarrow s_{29}$. A steering failure then occurs: no thread is `Enabled` and the two threads t_1 and t_3 are in the `Waiting` state. In the second case, where thread t_3 reaches site s_{29} before thread t_1 reaches site s_{09}, a similar scenario occurs. The states of both threads are also finally updated to be `Waiting` and thus a steering failure occurs.

4.4 Phase III (1): Inferring Infeasible Scheduling

Predictive deadlock detectors may report many cycles within an execution, Usually, only a small number of these reported cycles are real deadlocks. However, active schedulers have to schedule each program at least once, to attempt to trigger each cycle as a deadlock occurrence.

An interesting fact that we have discovered is that many predicted cycles share the same set of constraints,

TABLE 2
CONSTRAINT SET OF CYCLES c_1 AND c_2

Ψc_1	Ψc_2
$s_{01} \rightsquigarrow s_{30}$	$s_{01} \rightsquigarrow s_{38}$
$s_{02} \rightsquigarrow s_{30}$	$s_{02} \rightsquigarrow s_{38}$
$s_{03} \rightsquigarrow s_{30}$	$s_{03} \rightsquigarrow s_{38}$
$s_{25} \rightsquigarrow s_{04}$	$s_{25} \rightsquigarrow s_{04}$
$s_{26} \rightsquigarrow s_{04}$	$s_{26} \rightsquigarrow s_{04}$
$s_{27} \rightsquigarrow s_{03}$	$s_{27} \rightsquigarrow s_{03}$
$s_{28} \rightsquigarrow s_{03}$	$s_{28} \rightsquigarrow s_{03}$
$s_{29} \rightsquigarrow s_{04}$	$s_{29} \rightsquigarrow s_{04}$
	$s_{36} \rightsquigarrow s_{04}$
	$s_{37} \rightsquigarrow s_{04}$
	$s_{30} \rightsquigarrow s_{03}$
	$s_{31} \rightsquigarrow s_{03}$

The site s_{30} is a deadlocking site of cycle c_1 on lock q; the site s_{38} is also a deadlocking site of cycle c_2 on lock q.

Algorithm 4: ***Find_akinCycles***

```
    Input: € – a set of cycles
    Input: c – a cycle
    Input: Ψ – a set of constraints for each cycle in €
    Output: €' – a set of akin cycles of the cycle c
01  €' := ∅
02  for each cycle c' ∈ € ∧ c' ≠ c do
03      Ψc := Ψ(c), Ψc' := Ψ(c')
04      is_akinCycle := true
05      for each h = $e_i \rightsquigarrow e_j$ ∈ Ψc do
06          if ¬ (h ∈ Ψc' ∨ $e_j$ occurs at a deadlocking site of c) then
07              is_akinCycle := false
08          end if
09      end for
10      if is_akinCycle = true then
11          €' := €' ∪ { c'}
12      end if
13  end for
```

due to a combination of executions of the same or similar pieces of program code. If a steering failure occurs while confirming one such cycle, the same steering failure is usually incurred while confirming all the other such cycles (i.e., the same set of constraints cannot be enforced).

For instance, in our example program, the constraint sets of cycles c_1 and c_2 are shown in TABLE 2. We observe that the last five constraints (denoted as Ψ^*) from Ψc_1 also appear in Ψc_2; and, if we ignore the events that occur at the deadlocking sites (i.e., s_{04} and s_{30}), all of the remaining constraints from Ψc_1 (i.e., Ψ^*) are a subset of Ψc_2. Note that, given two cycles, different constraints must exist involving events happening at some deadlocking sites (e.g., the first three constraints of the two cycles c_1 and c_2).

During confirmation of cycle c_1, the first three constraints (i.e., $s_{01} \rightsquigarrow s_{30}$, $s_{02} \rightsquigarrow s_{30}$, and $s_{03} \rightsquigarrow s_{30}$) can be enforced. However, the remaining constraints (i.e., all of the constraints from Ψ^*) cannot be enforced, as thread t_1 starts its execution only after the execution of events at s_{03} and s_{04}. Therefore, *ConLock*$^+$ reports a steering failure. Similarly, during confirmation of cycle c_2, its first three constraints from Ψc_2 are also enforced and the remainder cannot be enforced. Again, *ConLock*$^+$ reports a steering failure.

The two cycles c_1 and c_2 actually both involve the same set of two threads t_1 and t_3, and they also share the same event from the two cycles (i.e., $\langle t_1, acq, m@s_{04}, \{q@s_{03}\}\rangle$, see TABLE 1). The difference between the two cycles is that, for cycle c_2, its second event (i.e., $\langle t_3, acq, q@s_{38}, \{m@s_{37}\}\rangle$) happens after the second event (i.e., $\langle t_3, acq, q@s_{30}, \{m@s_{29}\}\rangle$) of cycle c_1. Therefore, if events occurring at the deadlocking sites of c_1 are not considered, Ψc_1 is a subset of Ψc_2; hence, if cycle c_1 cannot be triggered, cycle c_2 then also cannot be triggered for the same reason.

We call the above cycle c_2 an ***akin cycle*** of cycle c_1. A cycle c' is said to be an akin cycle of another cycle c if the constraint set Ψc is a subset of the constraint set $\Psi c'$, excluding constraints with an event that occurs at a deadlocking site (e.g., the first three constraints of Ψc_1 shown in TABLE 2). We denote the constraints shared by Ψc and $\Psi c'$ as Ψ^* (or a set of common constraints of the two cycles c and c'). As seen from the above cycles c_1 and c_2, if a cycle cannot be triggered, confirmation of all of its akin cycles can be skipped. The correctness of this can be easily proven based on the following point: if a set (Ψ^*) cannot be enforced, any of its super sets ($\Psi c'$) also cannot be enforced. If there are many akin cycles, both the number of confirmation runs and the total confirmation time to check all cycles can be reduced.

In our definition of akin cycles, we do not consider the constraints where one event occurs at a deadlocking site. However, it is possible that a steering failure occurs on these constraints. In this case, no akin cycles should be skipped. (Alternatively, a detailed analysis should be performed to ensure the correctness of the use of akin cycles.)

An important point to note is that the relationship between akin cycles is not symmetrical. That is, given two cycles c and c', if c' is an akin cycle of c, there is no guarantee that c is also an akin cycle of c'. This is because the constraint set of c' (i.e., $\Psi c'$) may contain additional constraints; if any of these additional constraints cannot be enforced, it gives no indication of whether or not a constraint from Ψc could be enforced.

Additionally, an equivalent[1] reduction on constraint sets is adopted (see Section 4.2.4), which may also reduce the content of the set of common constraints of the two cycles. Therefore, the original constraint sets should be used to identify akin cycles if a steering failure is reported.

Algorithm 4 finds all akin cycles of a given cycle c. For each cycle c' from all cycles €, the algorithm checks whether the original constraint set Ψc is a subset of $\Psi c'$ (lines 05–09), excluding constraints with an event occurring at a deadlocking site. If this is true, the cycle c' is added into the akin cycle set €' of cycle c. At the end of the algorithm, the set €' contains all akin cycles of the given cycle c.

4.5 Phase III (2): Inferring Deadlock Causes

It is well-known that deadlock occurrences are caused by inconsistent lock acquisition orders. However, rather than

[1] If the reduction is not equivalent or the checked constraint set is not equivalent to the original one (e.g., our previous proposal *ConLock*), the use of akin cycles may produce imprecise results. This is because, in this case, the constraints enforced in a confirmation run may be the ones that are not enforced in the original set.

```
              inMutex = 0;
        Thread t1                Thread t2
   1 acq(m);                 6 acq(n);
   2 inMutex++;              7 if(inMutex == 0)
   3 acq(n);                 8 { acq(m);
   4 rel(m);                 9   inMutex++;
   5 inMutex--;             10   rel(m)
                            11   inMutex--; }
```

Fig. 7. A deadlock simplified from `SQLite` (see file "`os_unix.c`")

just detecting or triggering them, it would be more helpful to understand how these inconsistent lock acquisition orders are introduced.

Consider a simplified deadlock from `SQLite` (one of our benchmark programs) as shown in Fig. 7. The deadlock involves two threads (t_1 and t_2) and two locks (m and n). Additionally, there is a conditional *inMutex* with an initial value of 0. The conditional is used to indicate whether any thread is holding lock m. However, this conditional does not correctly reflect the facts under multi-threaded executions, i.e. if immediately after thread t_1 acquires lock m (i.e., thread t_1 has not increased the value of *inMutex* and this value is still 0), thread t_2 acquires lock n, reads the value of *inMutex*, and passes its check since "*inMutex == 0*". Then, thread t_2 tries to acquire lock m which is blocked. Next, thread t_1 increases the value of *inMutex* and tries to acquire lock n, which is also blocked. Thus, a deadlock occurs.

In this example, it seems that the developers were aware of the possible deadlock occurrences and hence have introduced a conditional *inMutex* to avoid them. However, the conditional fails to prevent a deadlock from occurring. Therefore, besides constraints on lock acquisitions, access to conditionals may also play an important role in the occurrence of deadlocks. We denote these accesses as **deadlock sensitive accesses**.

To determine whether a deadlock that has been confirmed by *ConLock*+ also involves deadlock sensitive accesses, we design a static analysis to extract any potential ones. The key of our analysis is based on two simple but effective heuristics: (1) if there are any deadlock sensitive accesses, the accessed conditionals usually share a common static (variable) name; and (2) these accesses are usually conflicting (e.g., a read and a write to the same conditional).

For example, from the deadlock shown in Fig. 7, both threads have static access to the same conditional named "*inMutex*" and the deadlock involves a pair of conflicting accesses to the conditional (i.e., the write at line 2 in thread t_1 and the read at line 7 in thread t_2).

Our static analysis firstly collects all memory accesses for each thread t in a cycle c from the beginning of function f_h to the end of function f_w. Function f_h contains the site where thread t acquires the lock that another thread in cycle c is waiting for (i.e., line 1 in thread t_1 and line 6 in thread t_2 in Fig. 7). F$_w$ contains the deadlocking site of thread t (i.e., line 3 in thread t_1 and line 8 in thread t_2 in Fig. 7). Our algorithm then checks for any conflicting accesses to the same static memory location (variable).

Algorithm 5 outlines our detailed analysis. Given a cycle

Algorithm 5: *Find_deadlockSensitiveAccesses*

```
    Input: $p$ – a program
    Input: $c$ – a cycle as a deadlock
01  $Acc := \varnothing$, $Inst := \varnothing$
02  for each $\langle t, acq, m@s, ls\rangle \in c$ do
03      Let $f_w$ and $f_h$ be the function containing the deadlocking site
04          and be the function containing the site where thread $t$
            acquires a lock waited by another thread in $c$, respectively.
05      $Inst(t) := FindAllPaths(f_h, f_w)$
06      //all instructions on all paths from $f_h$ to $f_w$ based on a DFS search
07      for each $ins \in Inst(t)$ do
08          if $Opcode(ins) \in \{read, write\}$ then
09              $Acc(t) := Acc(t) \cup \{\langle Operand(ins), Opcode(ins), SLine(ins)\rangle\}$
10          end if
11      end for
12  end for
13  for any two threads $t_1$, $t_2$ from $c$ do
14      if $\exists$ $a_1 = \langle var_1, op_1, line_1\rangle \in Acc(t_1)$, $a_2 = \langle var_2, op_2, line_2\rangle \in Acc(t_2)$,
15          such that $var_1 = var_2 \wedge \{op_1, op_2\} = \{read, write\}$ then
16              Print "A potential conditional is found at" $a_1$ and $a_2$!
17      end if
18  end for
```

c, for each thread t in c, the functions f_h and f_w are firstly identified (lines 03–04). It then calls *FindAllPaths*() to extract all of the instructions from the beginning of f_h to the end of f_w based on a DFS search (line 05). Next, it extracts all memory accesses including the variable name (*Operand*()), access type i.e. *read* or *write* (*Opcode*(), lines 07–11), and the source code line NO (*SLine*()). Finally, it checks whether there are two threads from the given cycle c that have a pair of conflicting accesses to a variable of the same name (lines 13–18). If so, it reports a deadlock sensitive access.

Applying *Algorithm* 5 on the deadlock in Fig. 7, two sets of accesses are otained:

- for thread t_1: $\{\langle inMutex, read, 2\rangle, \langle inMutex, write, 2\rangle, \langle inMutex, read, 5\rangle, \langle inMutex, write, 5\rangle\}$ and,
- for thread t_2: $\{\langle inMutex, read, 7\rangle, \langle inMutex, read, 9\rangle, \langle inMutex, write, 9\rangle, \langle inMutex, read, 11\rangle, \langle inMutex, write, 11\rangle\}$.

Since a pair of two conflicting accesses $\langle inMutex, write, 2\rangle$ and $\langle inMutex, read, 7\rangle$ exist to the same variable *inMutex*, Algorithm 5 reports these two accesses as deadlock sensitive accesses.

Note that there may still be some imprecision to our static analysis above and it could be further improved by combining various other techniques (e.g., static alias analysis and even dynamic analysis).

5 Experimental Results

We have implemented *ConLock*+ (*CL*+) using both Java and C/C++ programs. The Java implementation used `ASM 3.2` [1] to identify all "`synchronized`" operations of each loaded class and generate wrapper code to produce events. The C/C++ implementation is based on Pin 2.10 [37] in Probe mode. *ConLock*+ used a C/C++ binary program to produce events by wrapping the Pthread libraries.

PCT, *MagicScheduler* (*MS*), *DeadlockFuzzer* (*DF*), and *ASN* have also been implemented using the same framework. Although *DF* is available through the current release of *Calfuzzer* [25], it works only on Java programs and addi-

TABLE 3
DESCRIPTIVE STATISTICS AND EXECUTION STATISTICS OF THE BENCHMARKS

Benchmark	SLOC (× 1,000)	Bug ID	Deadlock Description	# of threads / locks	# of cycles	# of real deadlocks (cycle ID)	# of events
JDBC (5.0)	36.3	14927	`Connection.prepareStatement`() and `Statement.close`()	3/131*	10	1 (c1)	5,050
		31136	`PreparedStatement.executeQuery`() and `Connection.close`()	3/134*	16	1 (c2)	5,080
		17709	`Statement.executeQuery`() and `Conenction.prepareStatement`()	3/134*	18	2 (c3–c4)	5,090
Hawknl (.6b3)	9.3	n/a	`Nlshutdown`() and `nlclose`()	3/9	2	1 (c5)	33
SQLite (3.3.3)	74.0	1672	`sqlite3UnixEnterMutex`() and `sqlite3UnixLeaveMutex`()	3/3	2	2 (c6–c7)	16
MySQL (6.0.4)	1,093.6	34567	`Alter` on a temporary table and a non-temporary table	16/205	462	4 (c8–c11)	19,300
		37080	`Insert` and `Truncate` on a same table using falcon engine	17/206	183	1 (c12)	15,066
MySQL (5.5.17)	1,282.7	62614	`PUGE BINARY LOG` acquires locks in wrong order	22/55,251	8	2 (c13–c14)	444,621
MySQL (5.1.57)	1,146.7	60682	`SHOW INNODB STATUS` deadlocks if `LOCK_thd_data` points to `LOCK_open`	19/32,964	110	2 (c15–c16)	406,117
Total	3,642.6	-	-	89/89,037	811	16	900,373

*Note: * the # of locks is the # of objects in Java programs.*

tionally, when we tried to use it on a Java benchmark, only the test harness programs worked, and not the library files (i.e., the program code that contains the deadlocks) preventing us from profiling any event that could detect a deadlock. We managed to reliably implement *DF* using a method based on [28] in conjunction with *Calfuzzer* [25] (including all its optimizations), rather than modifying *Calfuzzer*. The original *PCT* tools were unavailable for download at the time the experiment was conducted, so its scheduling algorithms for deadlocks were implemented according to [10]. We have checked the accuracy of our implementations using several programs. A static analysis of *CL*$^+$ for C/C++ programs within LLVM [34] was implemented on bitcode.

5.1 Engineering Challenges

There are two main challenges to implementing our framework. The first one is how memory abstractions can be computed in order to identify the same lock or thread, as well as the same site for each event across multiple executions.

To address this, we implemented an improved object frequency abstraction algorithm [17] to model both memory objects and event sites by considering the program call stacks. This modeling also maintains a precise relationship between a parent thread and its child thread. For Java programs, this can easily be done by tracking the *start*() method of a Java `Thread` or `Runnable` instance. For C/C++ binaries, it becomes more difficult. We had to invoke some core functions of the Linux kernel, including the *clone*() and the *start_thread*() functions to maintain a parent-child relationship for the thread creation. Our current implementation is correct for single-process programs.

The second challenge is precisely determining the state of a thread during cycle confirmation. The basic concept used to address this is to track a set of functions (e.g., *sleep*(), *wait*(), and *barrier*()) that are related to the thread scheduling, other than the lock acquisitions and releases. This approach is relatively precise and has been adopted by *PCT* [10].

5.2 Benchmarks

We have selected a suite of widely-used real-world Java and C/C++ programs, including `JDBC` connector [3], `HawkNL` [2], `SQLite` [5], and three versions of the `MySQL` database server [4]. These benchmarks have been used in previous deadlock-related experiments [15], [29] and are available online. All of our test cases on these benchmarks have been taken from either [29] or their Bugzilla repositories.

TABLE 3 shows the descriptive statistics for the benchmarks used in the experiments. The first three columns show the benchmark name, the size of each benchmark in terms of SLOC (×1000), and the available bug report number, respectively. The fourth column shows the functions or operations that can lead to the corresponding deadlock state. The next three columns show the number of threads and locks, the total number of cycles, and the cycle ID for each real deadlock, respectively. The last column shows the number of events within the predictive runs.

5.3 Experimental Setup

The experiments were run on a virtual machine installed with Ubuntu Linux 10.04, hosted on a Microsoft Windows 7 system with 3.6 GHz Duo2 processor and 16GB physical memory. For each benchmark, *MagicLock* [15] was used to generate the set of cycles, based on the execution traces that had been collected. Each cycle was input into each technique (i.e., *PCT*, *MS*, *DF*, *ASN*, and *CL*$^+$) for each test case and was run 100 times [15], [28]. Since *PCT* shows insensitivity to a given cycle, if a benchmark showed the presence of *k* cycles, *PCT* was run 100 × *k* times.

As each execution may produce many cycles, each technique was run against cycles based on the generation order of the events in each cycle: cycles with events that appear earlier in an execution were checked first.

5.4 Results Analysis

The effectiveness of *ConLock*$^+$ on both real deadlocks and false positives was evaluated by comparing this method with other techniques.

5.4.1 Effectiveness on Real Deadlocks

TABLE 4 summarizes the overall experimental results. The first column shows the cycle ID, followed by the number of threads and locks, and the number of constraints before and after constraint reduction (i.e., "`#Con.`

TABLE 4
EXPERIMENTAL RESULTS COMPARING *PCT*, *MAGICSCHEDULER* (*MS*), *DEADLOCKFUZZER* (*DF*), *ASN*, AND *CONLOCK*$^+$ (*CL*$^+$)

Cycle ID	#Threads / #Locks	#Con. bef./aft.	Probability					#Thrashing					Time (in seconds)						
			PCT	*MS*	*DF*	*ASN*	*CL*$^+$	*PCT*	*MS*	*DF*	*ASN*	*CL*$^+$	*Native*	*Pred*	*PCT*	*MS*	*DF*	*ASN*	*CL*$^+$
c1	2 2	2 2	0.09	0.41	0.38	1.00	1.00	-	51	60	0	0	0.85	1.03	1.45	1.58	1.71	1.75	1.54
c2	2 5	2 2	0	0.58	0.49	1.00	1.00	-	39	46	0	0	0.92	1.18	-	1.53	1.48	1.65	1.52
c3	2 4	4 2	0	0.66	0.6	1.00	1.00	-	31	33	0	0	0.88	1.06	-	1.67	1.43	1.67	1.48
c4	2 4	2 3	0.17	0.61	0.54	1.00	1.00	-	33	37	0	0			1.45	1.42	1.55	1.61	1.51
c5	2 3	2 2	0.33	1.00	1.00	1.00	1.00	-	0	0	0	0	1.01	1.10	2.41	1.37	1.22	1.82	1.67
c6	2 2	4 3	0.27	0	0	1.00	1.00	-	100	100	0	0	2.01	2.16	2.63	-	-	3.08	2.13
c7	2 2	4 3	0.21	0	0	0	1.00	-	100	100	100	0			2.78	-	-	-	2.15
c8	2 3	2,050 3	0	0	0	0.71	1.00	-	90	92	0	0	-	-	-	-	-	3.8	1.91
c9	2 3	2,156 3	0	0.27	0.31	0.77	0.83	-	68	62	13	0			-	2.48	2.01	3.82	3.17
c10	2 3	2,166 3	0	0	0	0.92	0.80	-	93	86	2	0			-	-	-	3.56	3.94
c11	2 3	2,216 3	0	0	0	1.00	0.91	-	97	89	5	2			-	-	-	2.42	2.65
c12	2 8	58 2	0	0.71	0.73	0.91	0.88	-	17	14	2	0	-	-	-	1.07	0.94	1.59	1.21
c13	2 3	29 6	0	0.91	0.94	0.95	0.96	-	4	2	0	0	-	-	-	8.96	8.24	9.12	9.03
c14	3 3	37 6	0	0.87	0.93	0.21	0.99	-	8	2	73	0			-	9.17	8.43	9.21	9.12
c15	2 4	68 3	0	0.64	0.62	0.86	0.87	-	31	30	6	0	-	-	-	5.62	5.19	8.07	6.21
c16	2 4	70 4	0	0	0	0	0.89	-	88	82	95	2			-	-	-	-	6.27

Highlighted cells are those with a value of zero under the column "Probability" and with values larger than five under the column "# of thrashings".

`bef./aft.`") on each cycle. The next three major columns show the confirmation probability, the number of thrashings that occur, and the time taken by each technique to confirm each cycle, respectively, including the time ("`Native`") of the native executions and the time ("`Pred`") of the predictive runs. Note that the time consumption that is reported is the time taken by each technique to successfully confirm one corresponding cycle as a real deadlock. The entry "-" indicates that the corresponding run has timed out after 60 seconds. On cycles `c8`–`c16`, the normal execution time and the time needed by *PCT* could not be precisely collected, because these cycles are from `MySQL` which does not stop when used with the test harness. The symbol "-" is used to indicate these cases. The confirmation probability is computed using the formula: `sc` ÷ `tr`, where `sc` is the number of runs that successfully confirm the cycle, and `tr` is the total number of confirmation runs. Note that there may not be a direct relationship between the number of thrashing occurrences and the confirmation probability [28].

From TABLE 4, we observe that *CL*$^+$ confirmed 16 cycles as real deadlocks with probabilities ranging from 80% to 100%. On cycles `c1` to `c8`, *CL*$^+$ confirmed each cycle as a real deadlock in every run (i.e., with 100% probability); whereas other techniques were significantly less effective on these cycles. On cycles `c9`–`c11` and `c16`, all techniques, except *ASN* and *CL*$^+$, achieved quite low or zero confirmation probability. For the remaining four cycles (`c12`–`c15`), all techniques, except *ASN* on cycle `c14`, confirmed them correctly with high probabilities.

Overall, *PCT*, *MS*, and *DF* each had very low probabilities of confirming 6 to 11 cycles as real deadlocks, and the corresponding cells have been highlighted in TABLE 4. *ASN* and *CL*$^+$ both have high confirmation probabilities on some cycles (i.e., `c9`–`c13`, `c15`–`c16`), but *ASN* has a low or even zero probability on other cycles (i.e., `c7`, `c14`, and `c16`). In contrast, *CL*$^+$ consistently achieved high probabilities.

It is worth emphasizing that *PCT* does not rely on any given cycle to detect it as a real deadlock. Hence, the comparison with *PCT* should be considered for reference only.

The column "`#Thrashing`" shows that *MS* and *DF* encountered thrashing quite frequently, except on cycles `c12`–`c14`. In TABLE 4, the cells with values larger than five have been highlighted. *ASN* and *CL*$^+$ encountered a small number of thrashings except on `c7`, `c14`, and `c16` (on which *ASN* encountered a larger number of thrashings while *CL*$^+$ encountered almost none).

For *CL*$^+$, as shown in the third column of TABLE 4, the numbers of constraints before reduction was generally larger (e.g., >2,000 on `MySQL`); but were reduced to between 2 and 6 by our approach. This confirms that reducing the number of constraints has no negative impact on triggering real deadlocks. This observation is also consistent with the results of an empirical study, which show that concurrency bugs usually appear on short execution paths [36].

5.4.2 *Effectiveness on False Positives*

We also validated the ability of *ConLock*$^+$ on cycles that were false positives: a sample of 131 cycles out of all 795 cycles were manually inspected. These 131 cycles were selected using the following rules: (1) We selected all 40 (i.e., 9+15+16) remaining cycles on `JDBC`. (2) On `SQLite`, there was no false positive and, on `HawkNL`, there was only one. (3) On `MySQL`, we selected all cycles that *CL*$^+$ reported as steering failures (note that once *CL*$^+$ reports a steering failure when confirming a cycle, it starts the process of finding all of its akin cycles so that they can be excluded). We manually inspected and verified that all of these 131 cycles were false positives. Due to time limits, we did not manually verify whether or not all of the other 664 cycles were also false positives.

TABLE 5 summarizes the mean performance of *CL*$^+$ in handling the 131 inspected cycles, and a comparison with

TABLE 5
PERFORMANCE OF *CONLOCK*$^+$ ON FALSE POSITIVES

Benchmark	Bug ID	# false positives inspected	Avg. #cons. bef./aft.		Avg. # thrashing		Avg. Time (in seconds)
					Others	*CL*$^+$	*CL*$^+$
JDBC	14927	9	592	2	100	0	1.57
JDBC	31136	15	479	3	100	0	1.63
JDBC	17709	16	684	3	100	0	1.67
HawkNL	n/a	1	2	2	100	0	1.83
MySQL	34567	46	279	5	93	2	6.85
MySQL	37080	38	158	4	91	0	5.01
MySQL	62614	1	23	5	96	1	11.12
MySQL	60628	5	101	4	98	3	6.96

Note: there is no false warning on SQLite. "Others" includes MS, DF, and ASN. PCT is excluded due to its insensitivity to a given cycle. No time data is available on "Others" since these methods do not handle false positives.

TABLE 6
IMPROVEMENT OF *CONLOCK*$^+$ ON REAL DEADLOCKS FROM `MySQL`

Cycle ID	# constraints				Probability		
	Total	After Reduction					
		CL (-sp)	*CL (+sp)*	*CL*$^+$	*CL*	*CL*$^+$	Δ
c8	2,050	1,026	2	3	1.00	1.00	0.00
c9	2,156	1,079	2	3	0.73	0.83	0.10
c10	2,166	1,084	3	3	0.74	0.8	0.06
c11	2,216	1,109	6	3	0.83	0.91	0.08
c12	58	30	2	2	0.92	0.88	-0.04
c13	29	16	4	6	0.91	0.96	0.05
c14	37	20	4	6	0.93	0.99	0.06
c15	68	35	32	3	0.78	0.87	0.09
c16	70	36	33	4	0.71	0.89	0.18

the other methods. Since all of the other techniques do not aim to handle false positives, the mean data for these methods is shown under the sub-column "`others`". The third column shows the average number of false positives that we manually verified. The fourth major column shows the mean number of constraints before and after reduction ("`Avg. #cons. bef./aft.`"). The last two columns show the mean number of thrashings and the mean time for each technique to perform confirmation runs, respectively.

TABLE 5 shows that in order to confirm cycles that were false positives, all of the other techniques (i.e., *MS*, *DF*, and *ASN*) were very likely to result in thrashing in the experiment; whereas *CL*$^+$ only encountered a small number of thrashing occurrences (e.g., 6 on `MySQL`). *CL*$^+$ completed these confirmations in 11.12 seconds, whereas other methods could not complete until timeout (60 seconds) fired.

Our findings: all of the false positives that we inspected can be classified into three types. The first type is caused by the happened-before relationship (e.g., all false positives from `JDBC`). The second type involves conditional variables, such that it would not be possible for all events in the cycles to be executed concurrently. Conditional variables may be involved even for real deadlocks (see Section 5.4.4). The last type is from `MySQL`, which uses a thread pool to maintain a set of threads, that only become active on SQL connections. The cycles involve two threads (one from the pool and the other not from the pool) that cannot be executed concurrently.

5.4.3 Performance

It can be seen from the column entitled "`Time`" in TABLE 4 that the runtime overheads incurred by *MS*, *DF*, *ASN*, and *CL*$^+$ for successful confirmations are quite close to each other, and they all have an absolute time requirement that is within a practical range.

TABLE 5 shows that *CL*$^+$ can terminate a confirmation run when a false positive is encountered. It also shows that *CL*$^+$ can report a steering failure in this case, except for six confirmation runs where a thrashing has occurred. We have experimented with configuring *CL*$^+$ using the original set of constraints without reduction. However, for large-scale programs (i.e., `MySQL`), this configuration caused a significant slowdown which made it infeasible to schedule executions.

5.4.4 Improvements of ConLock$^+$ over ConLock

ConLock$^+$ offers three main improvements compared with its predecessor *ConLock* [16].

(1) On Real Deadlocks. The first improvement is the reduction of constraints based on a generalized set of three properties. TABLE 6 shows the result of reducing the constraints and the confirmation probability on real deadlocks from `MySQL`. Since there is no scalability issue for other benchmarks, they are not shown as all data remains unchanged.

In TABLE 6, the second major column shows the number of constraints, both before ("`Total`") and after reduction. For the number of constraints after reduction, we show both data without ("`-sp`") and with ("`+sp`") scheduling points for *CL*. The third major column shows the probability of deadlock confirmations as well as the improvement ("`Δ`") of *CL*$^+$ over *CL*.

TABLE 6 shows that *ConLock*$^+$ had a higher reduction in the number of constraints compared to *CL*, without scheduling points. With scheduling points, *CL*$^+$ also reduced the number of constraints to almost the same level (i.e., on cycles `c8`–`c14`) or even a lower level (i.e., on cycles `c15`–`c16`).

For the probabilities, there was an improvement of 4% to 18% except on cycles `c8` and `c12`. On `c8`, both *CL* and *CL*$^+$ had a 100% confirmation probability. On `c12`, the probability was slightly decreased by 4%.

(2) On All Cycles. The second improvement of *CL*$^+$ is its ability to skip akin cycles when the confirmation run of a given cycle encounters a steering failure. This reduces the total number of confirmation runs and the total confirmation time. We show the experimental results in TABLE 7.

The third major column shows the number of groups of akin cycles and the time taken for detection of these groups by *CL*$^+$. A group of akin cycles is generated as follows: during confirmation of a cycle *c*, if a steering failure is reported, all akin cycles of cycle *c* are collected as a group. The last two major columns show the number of runs taken by both *CL* and *CL*$^+$ to confirm all cycles and the corresponding total time for each benchmark, respectively. The reductions in the number of confirmation runs

TABLE 7
IMPROVEMENT OF *CONLOCK*+ ON ALL CYCLES

Benchmark	Bug ID	Akin cycles		The minimal # runs			Total Time		
		# of groups	*Time*	*CL*	*CL*+	Δ	*CL*	*CL*+	Δ
JDBC	14927	2	0.06s	10	3	-70.0%	35.4	9.7	-72.60%
JDBC	31136	4	0.93s	16	3	-81.3%	58.3	17.7	-69.64%
JDBC	17709	3	0.11s	18	5	-72.2%	65.2	17.2	-73.62%
HawkNL	n/a	0	0.01s	2	2	0.0%	7.3	7.5	+2.74%
SQLite	1672	0	0.01s	2	2	0.0%	8.3	8.4	+1.20%
MySQL	34567	46	5.06s	462	50	-89.2%	7,399.6	852.1	-88.48%
MySQL	37080	38	1.32s	183	39	-78.7%	2,186.1	451.2	-79.36%
MySQL	62614	1	0.01s	8	3	-62.5%	181.1	70.8	-60.91%
MySQL	60682	5	0.10s	110	7	-93.6%	1,763.3	112.3	-93.63%

Two runs for each cycle. Two seconds (exactly) are allowed for workspace initialization for each cycle on MySQL. All time data is shown in seconds by default.

and in the total confirmation time are also shown in the last two minor columns ("Δ"). Note that: (1) both *CL* and *CL*+, are configured to run a cycle twice[2] to collect the total time; (2) on all versions of `MySQL`, two additional seconds were allowed to restore database files to be the same on all runs (otherwise, our experience has shown that previous runs may produce bad data that could affect subsequent runs).

As *CL* has to confirm each cycle at least once, the minimum number of runs for each benchmark is the actual total number of cycles. However, since *CL*+ can avoid confirmation runs on akin cycles, it is shown in TABLE 7 that *CL*+ reduced the number of confirmation runs by between 62.5% and 93.6%, with the exception of `HawkNL` and `SQLite` (on which, only four cycles were reported). This can significantly improve the scalability for handling large-scale programs that may produce a larger number of cycles containing few real deadlocks.

On the total time, there was also a significant reduction by 60.91% to 93.63%, with the exception of `HawkNL` and `SQLite` which both incurred a slightly higher time.

(3) On Detection of Deadlock Sensitive Accesses. TABLE 8 shows the ability of *CL*+ to detect conditionals that may also be involved in confirmed deadlocks. The third column shows whether any deadlock sensitive accesses were found, and the fourth column shows the correctness. The last column shows a piece of simplified code indicating how the deadlock sensitive accesses that have been found are related to each deadlock.

Note that, in TABLE 8, we only show the code for each unique deadlock, rather than each cycle. The reason for this is that each unique deadlock may produce two or more similar cycles; however, these cycles share a set of similar static code.

[2] We chose to execute a program against each cycle twice, in case a real deadlock was not triggered by the first execution. As shown in TABLE 4, the probability of triggering a deadlock over two executions is at least $1 - (1 - 0.80)^2 = 0.96$, which is sufficient for our experiment. By increasing the number of executions per cycle, the total time consumed by *Conlock*+ is decreased compared to *Conlock*. Assuming there are a total of k cycles containing h real deadlocks, and the triggering time for each cycle is $time_i$ $(1 \le i \le k)$, then if each cycle is configured to run x times, the ratio between the total time using *Conlock*+ to that using *Conlock* is $\frac{\sum_{i=1}^{h}(x \times time_i) + time_{akin}}{\sum_{i=1}^{k}(x \times time_i)} = \frac{\sum_{i=1}^{h} time_i + \frac{time_{akin}}{x}}{\sum_{i=1}^{k} time_i}$, which decreases as x increases, where $time_{akin}$ is the total time to find akin cycles and is a constant.

From TABLE 8, we can observe that five out of the nine unique deadlocks involve deadlock sensitive accesses, which were all found by *ConLock*+. From the simplified

TABLE 8
DETECTION OF DEADLOCK SENSITIVE ACCESSES BY *CONLOCK*+

Bench-mark	Bug ID	Found	Correct?	Simplified Code (only showing lock acquisitions and deadlock sensitive accesses)
JDBC	14927	No	✓	Two complicated lock acquisitions.

JDBC	31136	Yes	✓	T1: *acq(m);if(isClosed) return; else acq(n);*
				T2: *acq(n);if(isClosed) return; else acq(m);*
				isClosed=true;

JDBC	17709	No	✓	Two complicated lock acquisitions.

Hawknl	n/a	Yes	✓	T1: *if(valid(socket)){acq(m); acq(n);}*
				T2: *acq(n);if(socket){acq(m); socket=NULL;}*

SQLite	1672	Yes	✓	See Fig. 7.
MySQL	34567	No	✓	A thread acquires two locks consecutively.

MySQL	37080	Yes	✓	T1: *acq(m); if(deleting) return; acq(n);*
				T2: *deleting=true; acq(n); acq(m);*

MySQL	62614	No	✓	A thread acquires three locks consecutively.

MySQL	60682	Yes	✓	T1: `acq(m); if(killed) goto _return_;acq(n);`
				T2: *acq(n); killed=true; acq(k);*
				T3: *acq(k); acq(m);*

code, it is easy to understand how a deadlock occurs. For the remaining four unique deadlocks (i.e., BugIDs: 14927, 17709, 34567, and 62614), no deadlock sensitive accesses are involved, which has been manually confirmed. In detail, two of these four deadlocks (BugIDs: 14927 and 17709) involve complicated lock acquisitions and the remaining two deadlocks (BugIDs: 34557 and 62614) involve a thread that consecutively acquires two locks. We also note this point in TABLE 8.

As well as identifying the deadlock sensitive accesses that are shown in TABLE 8, *ConLock*+ also reported several false positives (i.e., accesses to variables from two threads unrelated to deadlock occurrences). For example, on `MySQL` (BugID=37080), *ConLock*+ also reported two other variables named *section* and *sectionId*. These two variables are defined as method-local variables, but are used as handlers of two shared data structures. However, deadlocks do not occur by accessing these variables, so it is suggested that these variables could be dynamically pruned. We leave this as future work.

5.5 Limitations

Our implementation is based on binary instrumentation. An implementation of *ConLock*+ using symbolic execution [11] might produce more effective results (e.g., higher confirmation probability) as the constraints can be determined more precisely. However, symbolic execution is still not scalable for handling large-scale programs. As noted in [18]: "*the largest programs that can be symbolically executed today are on the order of thousands of lines of code*". In our benchmarks, `MySQL` has millions of lines of source code (i.e., SLOC), which is beyond the ability of state-of-the-art symbolic execution engines.

We have not manually validated all identified cycles on `MySQL` due to time and effort constraints. The probabilities, the ratios of thrashing, and the time taken by each of the techniques may be different if different numbers of runs, different benchmarks, or different tool implementations were used to conduct the experiment.

Another limitation is that *ConLock*+ reports a steering failure to indicate that the cycle that is being confirmed is a (dynamic) false positive. However, it is possible that, the corresponding cycle as well as the inferred akin cycles (if any) might be a real deadlock if different inputs were given to the program. This limitation is actually suffered by many dynamic techniques that analyze concurrency bugs (e.g., data race detection [22]). One possible way to overcome this limitation is still to adopt symbolic execution to search for alternative inputs [20]. Again, symbolic execution is currently not scalable for large-scale programs.

6 Related Work

6.1 Deadlock Detection

Many previous techniques [6], [13], [15], [19], [26], [40], [47], [51] have aimed to predict deadlocks through static or dynamic analyses. However, they all suffer from reporting false positives and it is important that real deadlocks can be identified from the set that is reported. Kahlon et al. [31] proposed a static theoretical model for analyzing concurrency bugs with well-nested lock acquisitions and releases. However, there exists a huge gap between static models and modern programming languages [23]. Hence, unlike *ConLock*+, their model cannot handle the occurrence of thrashing. Marino et al. [39] proposed a static approach for detecting deadlocks in object-oriented programs using data-centric synchronizations. However, their approach needs manual annotations to identify the ordering between atomic-sets. *ConLock*+ is an automated dynamic approach.

We have intensively reviewed the potential of several active schedulers (i.e., *DeadlockFuzzer*, *MagicScheduler*, and *ASN*) to trigger deadlocks, and compared these methods with our *ConLock*+ technique. *WOLF* [45] also predicts deadlocks from execution traces and aims to improve the probability of triggering real deadlock occurrences. However, *WOLF* skips cycles based on a relaxed happened-before relationship, which may also skip real deadlocks [28] before any confirmation is performed. In contrast, *ConLock*+ either schedules a program to confirm each cycle or skips only akin cycles. All of these active scheduling techniques suffer from the limitations of being unable to handle false positives. However, *ConLock*+ can effectively handle false positives through identification of steering failures.

Model checking (e.g., Java Path Finder (JPF)) has the potential to explore all possible schedules from a single input. These schedules can be integrated with a deadlock detector to find deadlocks. Synchronization coverage techniques [24], [44], [53] may also explore more schedules using a single input to detect deadlocks. However, these techniques are unable to handle large-scale multithreaded programs (e.g., `MySQL`) even using symbolic execution [18].

6.2 Deadlock Prevention and Healing

Dimmunix [29], [30] prevents the re-occurrence of each deadlock that has previously occurred through online monitoring executions that are similar to the previous executions that incurred deadlocks. *Gadara* [50] statically predicts deadlocks in a program and inserts corresponding deadlock avoidance code at the gate position of each predicted deadlock to prevent the deadlock occurrence. However, due to the imprecision of static analysis, it may not only miss real deadlocks but also mistakenly insert code to prevent false positives, reducing the parallelism of multithreaded program executions. Nir-Buchbinder et al. [41] use an execution serialization strategy for deadlock healing. While both *Dimmunix* and *Gadara* suffer from false positives, deadlock healing may even introduce new deadlocks [41].

6.3 Deadlock Synthesis

ESD [54] synthesizes an execution using a core dump of a previous execution that contained a deadlock occurrence. *PENELOPE* [48] also synthesizes an execution and uses a scheduling strategy similar to *DeadlockFuzzer* and *MagicScheduler* to detect real atomicity violations. Constraints are not used to avoid thrashing. However, it may not be feasible to execute a synthesized program, due to a lack of concurrent test cases. *ConTeGe* [43] automatically generates concurrent test suites to detect concurrency bugs. *OMEN* [46] further synthesizes program executions for deadlock detection. *Sherlock* [20] actively infers test cases that could trigger deadlocks (predicted under different test cases) through concolic execution [49]. *ConLock*+ has the ability to take any cycle as an input, regardless of whether it is a deadlock or not.

6.4 Others

ConTest [21] and *CTrigger* [42] inject noise into a run to increase the probability of triggering concurrency bugs. Since *ConLock*+ is not a completely randomized scheduler, it does not need to adopt such a strategy. It uses constraints to detect false positives.

Replay techniques (e.g., [7]) can reproduce runs that contain concurrency bugs. However, they are unable to turn a run containing a suggested cycle into a run containing a real deadlock.

7 Conclusion

ConLock⁺ has been proposed to actively check a set of cycles predicted from an execution trace. *ConLock⁺* generates constraints for predicted cycles and schedules confirmation runs in order to enforce constraints. It tries to trigger real deadlock occurrences or report steering failures. If a real deadlock is triggered, it performs a static analysis to identify possible deadlock sensitive accesses that may also contribute to deadlock occurrences. If a steering failure is reported, it avoids further checking of akin cycles in order to reduce the total time, since confirmation of these cycles would also incur steering failures. The experimental results have shown that *ConLock⁺* is both effective and efficient compared with existing techniques for triggering real deadlocks and handling false positives.

Acknowledgments

We sincerely thank the anonymous reviewers for their invaluable comments and suggestions on improving the manuscripts of this paper. We also thank Jian Zhang and Ernst Moritz Hahn for proofreading the manuscripts. This paper is supported in part by the National Basic Research (973) Program of China (2014CB340702) and the National Science Foundation of China (NSFC) (grant No. 61502465 and 91418206). All correspondence should be addressed to Yan Cai.

References

[1] ASM 3.2, *http://asm.ow2.org*.

[2] HawkNL, 1.6b3, *http://hawksoft.com/hawknl*.

[3] JDBC Connector 5.0, *http://www.mysql.com*.

[4] MySQL Database Server, *http://www.mysql.com*.

[5] SQLite 3.3.3, *http://www.sqlite.org*.

[6] R. Agarwal, L. Wang, and S. D. Stoller. Detecting potential deadlocks with static analysis and run-time monitoring. In *Proceedings of the 2005 IBM Verification Conference*, 2005.

[7] G. Altekar and I. Stoica. ODR: output-deterministic replay for multicore debugging. In *Proc. SOSP*, 193–206, 2009.

[8] S. Bensalem and K. Havelund. Scalable dynamic deadlock analysis of multi-threaded programs. In *PADTAD*, 2005.

[9] S. Bensalem, J.C. Fernandez, K. Havelund, and L. Mounier. Confirmation of deadlock potentials detected by runtime analysis. In *Proc. PADTAD*, 41−50, 2006.

[10] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proc. ASPLOS*, 167–178, 2010.

[11] C. Cadar, D. Dunbar, D. Engler, KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. OSDI*, 209–224, 2008.

[12] Y. Cai and W.K. Chan. Lock trace reduction for multithreaded programs. *IEEE Transactions on Parallel and Distributed Systems* (*TPDS*), 24(12): 2407−2417, 2013.

[13] Y. Cai and W.K. Chan. Magiclock: scalable detection of potential deadlocks in large-scale multithreaded programs. *IEEE Transactions on Software Engineering* (*TSE*), 40(3):266–281, 2014.

[14] Y. Cai, C.J. Jia, S.R. Wu, K. Zhai, and W.K. Chan. ASN: a dynamic barrier-based approach to confirmation of deadlocks from warnings for large-scale multithreaded programs. *IEEE Transactions on Parallel and Distributed Systems* (*TPDS*), 26(1):13–23, 2015.

[15] Y. Cai and W.K. Chan. MagicFuzzer: scalable deadlock detection for large-scale applications. In *Proc. ICSE*, 606–616, 2012.

[16] Y. Cai, S.R. Wu, and W.K. Chan. ConLock: A Constraint-based approach to dynamic checking on deadlocks in multithreaded programs. In *Proc. ICSE*, 491–502, 2014.

[17] Y. Cai, K. Zhai, S.R. Wu, and W.K. Chan. TeamWork: synchronizing threads globally to detect real deadlocks for multithreaded programs. In Proc. *PPoPP*, 311–312, 2013.

[18] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea. Selective symbolic execution. In *Workshop on Hot Topics in Dependable Systems*, 2009.

[19] J. Deshmukh, E. A. Emerson, and S. Sankaranarayanan. Symbolic deadlock analysis in concurrent libraries and their clients. In Proc. *ASE*, 480–491, 2009.

[20] M. Eslamimehr and J. Palsberg. Sherlock: scalable deadlock detection for concurrent programs. In *Proc. FSE'14*, 353–365, 2014.

[21] E. Farchi, Y. Nir-Buchbinder, and S. Ur. A cross-run lock discipline checker for Java. In *PADTAD*, 2005.

[22] C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. In *Proc. PLDI*, 121–133, 2009.

[23] A. Gupta. Verifying concurrent programs: tutorial talk. In *Proc. FMCAD*, 1, 2011.

[24] S. Hong, J. Ahn, S. Park, M. Kim, and M.J. Harrold. Testing concurrent programs to achieve high synchronization coverage. In *Proc. ISSTA*, 210–220, 2012.

[25] P. Joshi, M. Naik, C.S. Park, and K. Sen. CalFuzzer: an extensible active testing framework for concurrent programs. In *Proc. CAV*, 675–681, 2009.

[26] P. Joshi, M. Naik, K. Sen, and D. Gay. An effective dynamic analysis for detecting generalized deadlocks. In *Proc. FSE*, 327–336, 2010.

[27] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. In *Proc. PLDI*, 22–31, 2007.

[28] P. Joshi, C.S. Park, K. Sen, amd M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *Proc. PLDI*, 110–120, 2009.

[29] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea. Deadlock immunity: enabling systems to defend against deadlocks. In *Proc. OSDI*, 295–308, 2008.

[30] H. Jula, P. Tozun, G. Candea. Communix: A framework for collaborative deadlock immunity. In *Proc. DSN*, 181–188, 2011.

[31] V. Kahlon, F. Ivančić, and A. Gupta. Reasoning about threads communicating via locks. In *Proc. CAV*, 505–518, 2005.

[32] B. Kasikci, C. Zamfir, and G. Candea. RaceMob: crowdsourced data race detection. In *Proc. SOSP*, 406 – 422, 2013.

[33] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7): 558–565, 1978.

[34] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In Proc. CGO, 75–86, 2004.

[35] Z.F. Lai, S.C. Cheung, and W.K. Chan, Detecting atomic-set serializability violations for concurrent programs through active randomized testing. In *Proc. ICSE*, 235–244, 2010.

[36] S. Lu, S. Park, E. Seo, Y.Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proc. ASPLOS*, 329–339, 2008.

[37] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proc. PLDI*, 191–200, 2005.

[38] Z.D. Luo, R. Das, and Y. Qi. MulticoreSDK: a practical and efficient deadlock detector for real-world applications. In *Proc. ICST*, 309–318, 2011.

[39] D. Marino, C. Hammer, J. Dolby, M. Vaziri, F. Tip, and J. Vitek. Detecting deadlock in programs with data-centric synchronization. In *Proc. ICSE*, 322–331, 2013.

[40] M. Naik, C.S. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *Proc. ICSE*, 386–396, 2009.

[41] Y. Nir-Buchbinder, R. Tzoref, and S. Ur. Deadlocks: from exhibiting to healing. In *Proc. RV*, 104–118, 2008.

[42] S. Park, S. Lu, and Y.Y. Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *Proc. ASPLOS*, 25–36, 2009.

[43] M. Pradel and T. R. Gross. Fully automatic and precise detection of thread safety violations. In *Proc. PLDI'12*, 521–530, 2012.

[44] N. Rungta, E.G. Mercer, W. Visser. Efficient testing of concurrent programs with abstraction-guided symbolic execution. In *Proc. SPIN*, 174–191, 2009.

[45] M. Samak and M.K. Ramanthan. Trace driven dynamic deadlock detection and reproduction. In *Proc. PPoPP*, 29–42, 2014.

[46] M. Samak and M.K. Ramanathan. Multithreaded test synthesis for deadlock detection. In *Proc. OOPSLA'14*, 473–489, 2014.

[47] V.K. Shanbhag. Deadlock-detection in java-library using static-analysis. In *Proc. APSEC*, 361–368, 2008.

[48] F. Sorrentino, A. Farzan, and P. Madhusudan. PENELOPE: weaving threads to expose atomicity violations. In *Proc. FSE*, 37–46, 2010.

[49] K. Sen and G. Agha. CUTE and jCUTE: concolic unit testing and explicit path model-checking tools. In *Proc. CAV'06*, 419–423, 2006.

[50] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. Mahlke. Gadara: dynamic deadlock avoidance for multithreaded programs. In *Proc. OSDI*, 281–294, 2008.

[51] A. Williams, W. Thies, and M.D. Ernst. Static deadlock detection for java libraries. In *Proc. ECOOP*, 602–629, 2005.

[52] X.L. Xu, W.J. Wu, Y. Wang, Y.C. Wu. Software crowdsourcing for developing Software-as-a-Service. *Frontiers of Computer Science* (*FCS*), 9 (4):554–565, 2015.

[53] J. Yu, S Narayanasamy, C. Pereira, and G. Pokam. Maple: a coverage-driven testing tool for multithreaded programs. In *Proc*. *OOPSLA*, 485–502, 2012.

[54] C. Zamfir and G. Candea. Execution synthesis: a technique for automated software debugging. In *Proc. EuroSys*, 321–334, 2010.

[55] L. Zheng, X. Liao, S. Wu, X. Fan, and H. Jin. Understanding and identifying latent data races cross-thread interleaving. *Frontiers of Computer Science* (*FCS*), 9(4):524–539, 2015.

Yan Cai received his PhD degree from City University of Hong Kong in 2014. He is currently an associate research professor at the State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences. His current research interest is on concurrency bugs, including detection, reproduction, and fixing, especially in large-scale multithreaded programs. He is a Young Associate Editor of the *Frontiers of Computer Sciences* (*FCS*). His research results have been reported in venues such as *TSE*, *TPDS*, *TSC*, *JSS*, *SPE*, *JWSR*, ICSE, FSE, PPoPP, ISSRE, and ICWS.

Qiong Lu is a master student at Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences. She received her BEng degree in Computer Science and Technology from Beijing University of Posts and Telecommunications, China, in 2013. Her research interest is the analysis and testing of Android apps.