ConLock: A Constraint-Based Approach to Dynamic Checking on Deadlocks in Multithreaded Programs

Yan Cai
Department of Computer Science
City University of Hong Kong
Hong Kong, China
ycai.mail@gmail.com

Shangru Wu
Department of Computer Science
City University of Hong Kong
Hong Kong, China
shangru.wu@my.cityu.edu.hk

W. K. Chan
Department of Computer Science
City University of Hong Kong
Hong Kong, China
wkchan@cityu.edu.hk

ABSTRACT
Many predictive deadlock detection techniques analyze multithreaded programs to suggest potential deadlocks (referred to as *cycles* or *deadlock warnings*). Nonetheless, many of such cycles are false positives. On checking these cycles, existing dynamic deadlock confirmation techniques may frequently encounter thrashing or result in a low confirmation probability. This paper presents a novel technique entitled *ConLock* to address these problems. *ConLock* firstly analyzes a given cycle and the execution trace that produces the cycle. It identifies a set of thread scheduling constraints based on a novel *should-happen-before* relation. *ConLock* then manipulates a confirmation run with the aim to not violate a reduced set of scheduling constraints and to trigger an occurrence of the deadlock if the cycle is a real deadlock. If the cycle is a false positive, *ConLock* reports scheduling violations. We have validated *ConLock* using a suite of real-world programs with 11 deadlocks. The result shows that among all 741 cycles reported by *Magiclock*, *ConLock* confirms all 11 deadlocks with a probability of 71%−100%. On the remaining 730 cycles, *ConLock* reports scheduling violations on each. We have systematically sampled 87 out of the 730 cycles and confirmed that all these cycles are false positives.

Categories and Subject Descriptors
D.2.4 [**Software Engineering**]: Software/Program Verification – *reliability*, *correctness proofs*, *validation*. D.2.5 [**Software Engineering**]: Testing and Debugging – *testing tools*. D.4.1 [**General**]: Processing Management – *concurrency*, *deadlocks*.

General Terms
Reliability, Verification

Keywords
Deadlock, confirmation, should-happen-before relation.

1. INTRODUCTION
Many multithreaded programs use various locking mechanisms [32] to coordinate how their threads produce the program outputs. Improper sequences of lock acquisitions and releases performed by these threads may result in concurrency bugs such as data races [11][14][46], atomicity violations [30], or deadlocks [5][7][15] [25]. A *deadlock* [15][25] occurs when every thread in a thread set waits for acquiring a lock that another thread in the same set is holding. Each occurrence of a deadlock stops the threads involved in it from making further progress. Deadlock is a critical failure.

ICSE'14, May 31 – June 7, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2756-5/14/05...$15.00
http://dx.doi.org/10.1145/2568225.2568312

Once a deadlock has occurred in an execution trace, it is not difficult to report the occurrence and reproduce it [45]. In general, deadlocks rarely occur in the program executions of real-world programs, but may reveal their presences in some other execution traces. To suggest potential deadlocks, many static techniques (e.g., [40][43]) and dynamic techniques (e.g., [5][12][34]) have been proposed. Static techniques analyze the program code to infer the existence of cyclic lock acquisition (i.e., *cycles*) among threads as potential deadlocks. They generally suffer from reporting many false positives. For instance, the experiment in [43] reported more than 100,000 potential cases when analyzing the Java JDK; and yet only 7 of them could finally be confirmed as real deadlocks (after applying various unsound heuristics). Dynamic predictive techniques [7][8] also suffer from reporting false positives, albeit less serious than the static counterparts. Tracking the happened-before relations [29] or constructing a segmentation graph [7] on the corresponding execution trace may eliminate some kinds of false positives, but may also eliminate certain true positives due to different thread schedules [25], which is risky. Confirming each given cycle to be a real deadlock or not by executing the program with respect to the cycle is desirable.

Latest techniques that can automatically *confirm* cycles as real deadlocks include *DeadlockFuzzer* [25] and *MagicScheduler* [15]. A minor adaptation of *PCT* [9] is also an alternative. However, in Section 5, our experiment shows that they either are unable to confirm a real deadlock at all or can only achieve a low confirmation probability. Besides, existing dynamic techniques such as [15][25] have no strategy to handle cycles that are false positives.

To ease our presentation, we refer to an execution used to suggest cycles as a *predictive run*. Similarly, we refer to an execution that is used to confirm whether a suggested cycle c is a real deadlock or not as a *confirmation run*. We also suppose that cycles have been suggested by a predictive technique on a predictive run.

In this paper, we propose ***ConLock***, a novel constraint-based approach to dynamic confirmation of deadlocks and handling false positives. *ConLock* consists of two phases: (1) In Phase I, *ConLock* analyzes the predictive run, and generates a set of scheduling constraints with respect to the given cycle c. Each constraint specifies the order of a pair of lock acquisition/release events in a confirmation run between the corresponding pair of threads involved in the cycle c. (2) In Phase II, *ConLock* manipulates a confirmation run with the attempt to not violate the reduced set of constraints produced in Phase I so as to trigger the deadlock if the cycle c is a real deadlock; or else, it reports a scheduling violation against the given set of constraints, which indicates that the current run is no longer meaningful to confirm the cycle c. In either case, *ConLock* terminates the current confirmation run.

We have implemented a prototype of *ConLock* to validate *ConLock* on a suite of real-world programs. We compared *ConLock* with *MagicScheduler* [15], *DeadlockFuzzer* [25], and *PCT* [9] in terms of confirmation probability, consistency in confirmation, and the amount of time taken to check against all given cycles. In the experiment, *ConLock* achieved a consistently higher probability (71%–100%) in confirming all 11 real deadlocks; whereas, other techniques either missed to confirm 5 to 7 cycles as real deadlocks in every confirmation run or only achieved a lower probability on remaining real deadlock cases. We systematically sampled a subset (87 cycles in total) of the remaining 730 cycles (on which *ConLock* reported scheduling violations) for careful manual code inspection, and confirmed that they were all false positives.

The main contribution of this paper is threefold:

- This paper proposes *ConLock*, a novel dynamic constraint-based deadlock confirmation technique to isolating real deadlocks from the given set of cycles with a high probability and a low slowdown overhead.
- To the best of our knowledge, *ConLock* is the first technique that can terminate confirmation runs on false positive cycles by reporting scheduling violations.
- We report an experiment, which confirms that *ConLock* can be effective and efficient.

In the rest of this paper, Section 2 revisits the preliminaries of this work. Section 3 motivates our work by an example. Section 4 presents the *ConLock* algorithm. Section 5 describes a validation experiment, and reports the experimental results. Section 6 reviews the closely related work. Section 7 concludes this paper.

2. PRELIMINARIES

2.1. Events and Traces

Our model monitors an execution trace over a set of critical operations $\Gamma = \{acq, rel\}$ performed on locks, where *acq* represents *lock acquisition* and *rel* represents *lock release*. The extension to handle other synchronization primitives (e.g., barriers) is straightforward [11][14][20].

Definition 1. An ***event*** $e = \langle t, op, m@s, ls\rangle$ denotes that a thread t performs an operation $op \in \Gamma$ on a lock m, which occurs at the *site* s, and at the same time, t is holding a set of locks (called *lockset*) ls, each of which is associated with the site where t acquires the corresponding lock.

Definition 1 extends the definition of *lock dependency* in [15][25] by including lock release *rel* in Γ. A site is an execution context [16][25] (e.g., the triple ⟨call stack, statement number, the latest occurrence count of the couple ⟨call stack, statement number⟩⟩ can be used to denote an execution context).

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

2.2. Cycle as Potential Deadlock

Definition 2. A sequence of k events denoted by $c = \langle e_1, e_2, \ldots, e_k\rangle$, where $e_i = \langle t_i, acq, m_i@s_i, ls_i\rangle$ for $1 \le i \le k$, is called a ***cycle*** [15] if both of the following two conditions are satisfied:

(1) for $1 \le i \le k - 1$, $m_i \in ls_{i+1}$, and $m_k \in ls_1$; and,
(2) for $1 \le i < j \le k$, $t_i \ne t_j$, $m_i \ne m_j$, $m_i \notin ls_i$, and $ls_i \cap ls_j = \emptyset$.

A cycle models a ***potential deadlock***: The site s_i in the event e_i involved in a cycle c is referred to as a ***deadlocking site*** of the thread t_i. The lock m_i of an event e_i is the lock that the thread t_i waits to acquire.

For instance, Figure 1(b) (to be described in Section 3) depicts that a thread t_1 is holding the lockset $\{a, p, m\}$ and is waiting to acquire the lock n at site s_{08}; and a thread t_2 is holding the lockset $\{n\}$, and is waiting to acquire the lock p at site s_{16}. The four boxed operations represent a deadlock bug that has *not* been triggered in the scenario; and this deadlock can be modeled as a cycle $\boldsymbol{c_0 = \langle\langle t_1, acq, n@s_{08}, \{a@s_{03}, p@s_{06}, m@s_{07}\}\rangle, \langle t_2, acq, p@s_{16}, \{n@s_{15}\}\rangle\rangle}$.

We denote the set $\{m_i \mid e_i = \langle t_i, acq, m_i@s_i, ls_i\rangle \wedge e_i \in c\}$ by $\mathbf{WLock_c}$. It means that each lock in $\mathbf{WLock_c}$ is a lock *waiting* to be acquired by a thread involved in the cycle c at its deadlocking site. Similarly, we denote the set of all locks, each of which is being *held* by a thread involved in c at the deadlocking site, by the set $\mathbf{HLock_c}$ (i.e., $\mathbf{HLock_c} = \{n_j \mid e_i = \langle t_i, acq, m_i@s_i, ls_i\rangle \wedge e_i \in c \wedge n_j@s_j \in ls_i$ for some site $s_j\}$). Moreover, the site to acquire a lock $m \in \mathrm{WLock_c}$ ($n \in \mathrm{HLock_c}$, respectively) is denoted by $\mathbf{WSite_c}(m)$ ($\mathbf{HSite_c}(n)$, respectively). For the above cycle c_0 in Figure 1(b), we have $\mathrm{WLock_{c0}} = \{n, p\}$, $\mathrm{WSite_{c0}}(n) = s_{08}$, $\mathrm{WSite_{c0}}(p) = s_{16}$, $\mathrm{HLock_{c0}} = \{n, p, a, m\}$, $\mathrm{HSite_{c0}}(n) = s_{15}$, $\mathrm{HSite_{c0}}(p) = s_{06}$, $\mathrm{HSite_{c0}}(a) = s_{03}$, and $\mathrm{HSite_{c0}}(m) = s_{07}$.

3. MOTIVATING EXAMPLE

Figure 1(a) shows a bug that can be triggered by using two threads operating on four locks. The operations `acq`(x) and `rel`(x) in the figure depict a lock acquisition event and a lock release event on the lock x, respectively. The program in **Figure 1**(a) illustrates a deadlock bug as shown by the four boxed operations.

Execution 1, depicted in **Figure 1**(a), passes through the path $\langle s_{13}, s_{14}, s_{01}, s_{02}, s_{03}, s_{04}, s_{05}, s_{06}, s_{07}, s_{15}\rangle$, resulting in a deadlock occurrence: Specifically, the thread t_2 firstly acquires the lock a at the site s_{13} and then releases the lock a at the site s_{14}. When the thread t_2 is about to acquire the lock n at the site s_{15}, the thread t_2 is suspended. Then, the thread t_1 executes the operations at sites s_{01} to s_{07} to acquire three locks a, p, and m, at the sites s_{03}, s_{06} and s_{07}, respectively. When t_1 is about to acquire the lock n at the site s_{08}, it is suspended, and the thread t_2 is resumed to successfully acquire the lock n at the site s_{15}. Then, the thread t_2 is suspended when it is about to acquire the lock p at the site s_{16} because the lock p is being held by t_1 at this moment. As such, the thread t_1 resumes its execution. Nonetheless, the thread t_1 has to wait for the thread t_2 to release the lock n so that the thread t_1 can acquire this lock n. The two threads now mutually wait for each other to release their waiting locks. The execution triggers a deadlock.

Execution 2, depicted in Figure 1(b), passes through the path $\langle s_{13}, s_{14}, s_{15}, s_{16}, s_{17}, s_{18}, s_{01}, s_{02}, s_{03}, s_{04}, s_{05}, s_{06}, s_{07}, s_{08}, s_{09}, s_{10}, s_{11}, s_{12}\rangle$, failing to trigger any deadlock: Suppose that the thread t_2 has acquired the lock n at the site s_{15} (which is different from the *Execution* 1), and is about to acquire the lock p at the site s_{16}. At this moment, the thread t_2 is suspended, and the thread t_1 is resumed. However, the thread t_1 cannot successfully acquire the lock n at the site s_{01} because the thread t_2 is holding the lock n. Hence, t_1 is suspended. The thread t_2 is then resumed, and acquires the lock p. It finally releases the two locks p and n at the sites s_{17} and s_{18}, respectively. Next, the thread t_1 is resumed, and completes its remaining execution. No deadlock has been triggered.

Existing dynamic *predictive* techniques (e.g., [15][25][34]) may analyze *Execution* 2 to *suggest* the cycle $c_0 = \langle\langle t_1, acq, n@s_{08}, \{a@s_{03}, p@s_{06}, m@s_{07}\}\rangle, \langle t_2, acq, p@s_{16}, \{n@s_{15}\}\rangle\rangle$. However, without confirming the cycle c_0, this cycle c_0 is unknown to be a

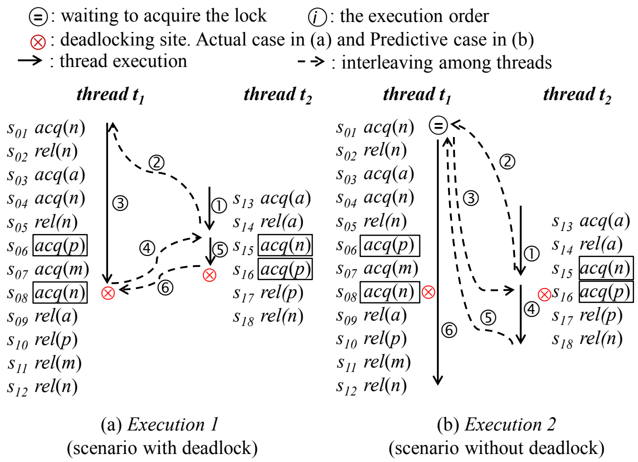


Figure 1. Example deadlock adapted from `JDBC Connector 5.0` [2] (Bug ID: 2147). The acronym `n`, `a`, `p`, and `m` are `Connection`, `Statement`, `ServerPreparedStatement`, and `Connection.Mutex`, respectively.

real deadlock or just a false positive. Manually confirming every such cycle can be tedious and error prone.

The latest state-of-the-art techniques on automatic confirmation of cycles as real deadlocks include *DeadlockFuzzer* [25] and *MagicScheduler* [15]. *PCT* [9] is not designed for deadlock confirmation, but it provides a probabilistic guarantee to detect real deadlocks if they exist. We review them to motivate our work.

MagicScheduler (***MS***) [15]: *MS* is the latest dynamic deadlock confirmation technique. It uses a heuristic to randomly schedule each individual thread in a given program against a set of given cycles and *suspend a thread if the thread holds a set of locks and requests another lock at the deadlocking site of this thread specified by a given cycle* [15]. Consider the example in **Figure 1**(b). *MS* aims to suspend the thread t_1 when t_1 is right before executing the operation at the site s_{08}, and suspend the thread t_2 when t_2 is right before executing the operation at the site s_{16}.

Directly applying the above heuristic can be challenging to schedule the two threads in a confirmation run to trigger a real deadlock. Suppose that *MS* firstly suspends the thread t_2 right before executing the operation at the site s_{16} (after the thread t_2 has acquired the lock n at the site s_{15}). To trigger the deadlock with respect to the cycle c_0, *MS* aims to wait for the thread t_1 to be suspended at the site s_{08}. This target is nonetheless impossible to achieve because the thread t_1 has been blocked at the site s_{01} (or the site s_{04}) as the lock n is being held by t_2, and yet t_2 has been suspended. This kind of problem is known as *thrashing* [25]. To resolve this occurrence of thrashing, *MS* resumes the thread t_2, which runs to complete the execution of the operations up to the site s_{18} and releases the lock n. Nonetheless, the deadlocking site s_{16} for t_2 has been passed. So, the cycle c_0 could not be confirmed.

Execution 2 starts with the thread t_2 at the site s_{13}. On *Execution* 2, according to the scheduling strategy of *MS*, *MS* always results in thrashing and fails to trigger the cycle c_0 as a real deadlock. An execution scenario that starts with the thread t_1 would still result in thrashing caused by *MS*. For instance, suppose that *MS* has successfully suspended the thread t_1 at the site s_{08} (before acquiring the lock n), and then the thread t_2 starts. The thread t_2 cannot acquire the lock a at s_{13} because t_1 is still holding the lock a. As a result, thrashing occurs. *MS* resumes t_1 to acquire the lock n. As such, no deadlock could be triggered. This also illustrates that merely applies the active thread scheduling at the sites s_{08} and s_{16} is unlikely to trigger the deadlock bug with a high probability.

DeadlockFuzzer (***DF***) [25] uses a heuristic strategy that is identical to *MS* except that *DF* tries to confirm one cycle per run instead of a set of cycles per run. The running example has only one cycle. *DF* suffers from the same problem experienced by *MS*.

Probabilistic Scheduler: *PCT* [9] probabilistically generates a sequence of priority changing points. From the probabilistic theory, *PCT* can generate a thread schedule (e.g., *Execution 1* in **Figure 1**(a)) that results in triggering a deadlock occurrence. According to [9], its guaranteed probability is $1 / (n \times k^{d-1})$ for a concurrency bug of depth d involving n threads that executes a total of k steps. For the running example, the guaranteed probability is $1 / (2 \times 18^{2-1})$ or 0.02778, which is low, despite that *PCT* can detect the deadlock bug without needing any predictive run or any information about a given cycle.

In the next section, we present *ConLock* and illustrate how *ConLock* confirms the cycle c_0 in Figure 1.

4. CONLOCK

4.1. Overview

ConLock is a novel constraint-based dynamic approach to deadlock checking. It consists of two phases with respect to a given cycle [5][7][15] as depicted in Figure 2.

A predictive technique firstly suggests a cycle c (as depicted in Figure 2(a)). *ConLock* then starts its two phases.

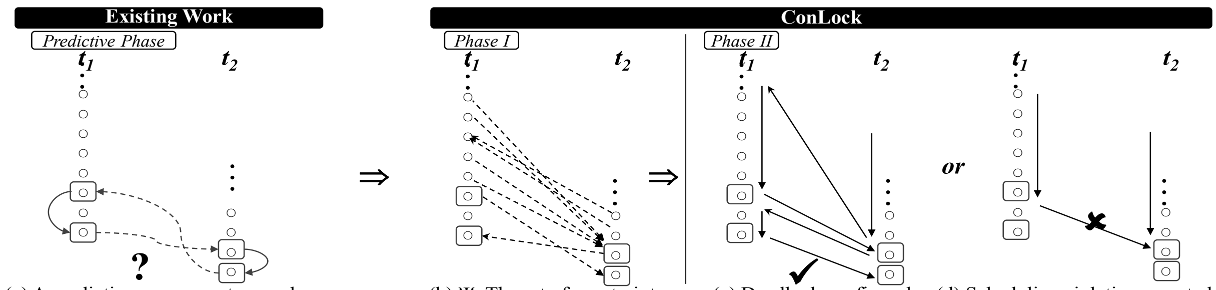
In Phase I, given a cycle from a predictive run, *ConLock* generates a set of constraints Ψ (as depicted in Figure 2(b)). The generation of the constraint set Ψ is based on the novel *should-happen-before relation* (proposed in Section 4.2.1).

In Phase II, *ConLock* actively schedules a confirmation run with respect to a subset of constraints Ψ, and produces two important consequences: (1) if the given cycle is a real deadlock (as depicted in Figure 2(c)), *ConLock* tries to confirm it. As shown in our experiment, its confirmation probability is high; (2) if the given cycle is a false positive (as depicted in Figure 2(d)), *ConLock* reports a scheduling violation. The two consequences significantly distinguish *ConLock* from the existing techniques.

4.2. Phase I: Generation of Constraint Set Ψ and Scheduling Points

To schedule a confirmation run that successfully confirms a given cycle as a real deadlock, each thread involved in a cycle should be precisely suspended at its deadlocking site. Many existing dynamic active testing techniques [15][25] have used this insight to extract information from a predictive run to guide the manipulation of a confirmation run. Moreover, we observe that at the same time, a confirmation technique should avoid occurrences of thrashing as much as possible. Hence, our goal is that each thread involved in a cycle should not be artificially blocked by any other thread involved in the same cycle before the former thread is about to acquire the lock at its deadlocking site as much as possible.

Based on the above two observations, we formulate a novel relation entitled the ***should-happen-before*** relation to (1) effectively prevent occurrence of thrashing and (2) precisely suspend each thread involved in a cycle at its deadlocking site. We note that the



(a) A predictive run suggests a cycle c (b) Ψ: The set of constraints (c) Deadlock confirmed (d) Scheduling violation reported

Figure 2. An overview of *ConLock*.

should-happen-before relation is a relation between two events in the execution trace of a predictive run (where the run itself has no deadlock occurrence). It denotes that the two related events *should* occur in a specified order in the confirmation run.

4.2.1 Should-Happen-Before Relation

We firstly revisit the *happened-before-relation*. We use $\rightarrowtail$ to denote the *happened-before relation* between two events.

In our problem context, the *happened-before relation* [29] describes a relation between two events over the given execution trace of the predictive run. The happened-before relation [29] is defined as follows: (i) *Program order*: if two events e_1 and e_2 are performed by the same thread, and e_1 appeared before e_2 in the execution trace, then $e_1 \rightarrowtail e_2$. (ii) *Lock acquire and release*: if (1) e_r is a lock release on a lock m by a thread t_1, (2) e_a is a lock acquisition on the same lock m by a thread t_2, where $t_1 \neq t_2$ and (3) e_r appears prior to e_a in the execution trace, then $e_r \rightarrowtail e_a$. (iii) *Transitivity*: if $e_1 \rightarrowtail e_2$ and $e_2 \rightarrowtail e_3$, then $e_1 \rightarrowtail e_3$.

We proceed to present the definition of *should-happen-before* relation. We use $\rightsquigarrow$ to represent this relation over two events. To ease our subsequent presentation, sometimes, we refer to the event e_i by the thread t_i involved in the cycle c as $\boldsymbol{\varepsilon(c, t_i)}$, and use the site of an event e to denote e when describing the $\rightsquigarrow$ and $\rightarrowtail$ relations.

Definition 3. Given an execution trace σ, a cycle c on σ, suppose that t, t_α, and t_β are threads involved in the cycle c, where $t \neq t_\alpha$ and $t \neq t_\beta$, the ***should-happen-before relation*** is defined as:

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

Rule 1 defines a condition to ***prevent*** predictable thrashing to occur on these locks in the set $\texttt{WLock}_c$. Figure 3(a) uses *Execution* 2 to illustrate this rule via the lock p and the cycle c_0. In Figure 3(a), the lock p is in $\texttt{WLock}_{c0}$, the site s_{16} is the deadlocking site for the thread t_2 (i.e., t_α in the *Rule* 1) that operates on this lock p, and the deadlocking site for the thread t_1 (i.e., the thread t in *Rule* 1) is the site s_{08}. *Rule* 1 specifies that any lock acquisition or release event on this lock p performed by the thread t_1 (e.g., the event e at the site s_{06}) that happened-before the event $\varepsilon(c_0, t_1)$ at the site s_{08} *should-happen-before* the event (i.e., e_α) performed by the thread t_2 at its deadlocking site s_{16}. Thus, by *Rule* 1, we get $s_{06} \rightsquigarrow s_{16}$.

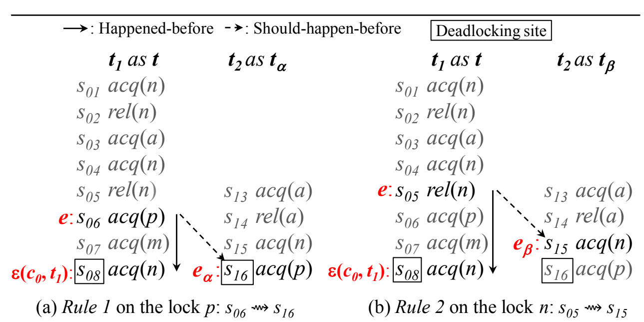
Similarly, *Rule* 2 defines a condition that prevents predictable thrashing on these locks in the set $\texttt{HLock}_c$. Figure 3(b) uses *Execution* 2 to illustrate this rule via the lock n. In Figure 3(b), the lock n is in $\texttt{HLock}_{c0}$, and the thread t_2 (i.e., the thread t_β in *Rule* 2) holds a lockset $\{n@s_{15}\}$ when t_2 is about to acquire the lock p at its deadlocking site s_{16}. We also recall that the deadlocking site for the thread t_1 (i.e., the thread t in *Rule* 2) is the site s_{08}. *Rule* 2 specifies that any lock acquisition or release event on n performed by t_1 that happened-before the event occurred at its deadlocking site s_{08} should-happen-before the lock acquisition event on n at site s_{15} (i.e., the event e_β). Thus, by *Rule* 2, we get $s_{05} \rightsquigarrow s_{15}$. The lock n has also been acquired or released by the thread t_1 at sites s_{01}, s_{02}, and s_{04}. So, we get $s_{01} \rightsquigarrow s_{15}$, $s_{02} \rightsquigarrow s_{15}$, and $s_{04} \rightsquigarrow s_{15}$, accordingly.

The Whole Set of Should-Happen-Before Relations in the Running Examples: We now apply *Rule* 1 and *Rule* 2 to identify a complete set of should-happen-before relations with respect to the cycle c_0. We recall that *Execution* 2 in Figure 1(b) operates on four locks $\{n, a, p, m\}$. The cycle c_0 has two deadlocking sites s_{08} of the thread t_1 and s_{16} for the thread t_2. $\texttt{WLock}_{c0}$ is $\{n, p\}$, and $\texttt{HLock}_{c0}$ is $\{n, a, p, m\}$.

The lock m is only acquired once. There is no should-happen-before relation on it (because the should-happen-before relation is defined over two events performed by different threads).

Consider the lock n. We have applied *Rule* 2 on it to have identified $s_{01} \rightsquigarrow s_{15}$, $s_{02} \rightsquigarrow s_{15}$, $s_{04} \rightsquigarrow s_{15}$, and $s_{05} \rightsquigarrow s_{15}$ in the above illustration of *Rule* 2. The thread t_1 performs the event on the lock n at its deadlocking site s_{08}, which is also denoted by $\varepsilon(c_0, t_1)$. For the thread t_2, there is only one event $e = \langle t_2, acq, n@s_{15}, \{\}\rangle$ operating on the lock n and $e \rightarrowtail \varepsilon(c_0, t_2)$. By *Rule* 1, we get $s_{15} \rightsquigarrow s_{08}$.

Consider the lock p. We have applied *Rule* 1 on this lock to have identified $s_{06} \rightsquigarrow s_{16}$. We recall that $\texttt{HSite}_c(p)$ is the site s_{06}, but there is no event operating on the lock p by the thread t_2 that happened-before the event $\varepsilon(c_0, t_2)$ at the site s_{16}. Thus, *Rule* 2 produces no further should-happen-before relation for the lock p.



(a) *Rule 1* on the lock p: $s_{06} \rightsquigarrow s_{16}$ (b) *Rule 2* on the lock n: $s_{05} \rightsquigarrow s_{15}$

Figure 3. Examples of *Rule* 1 and *Rule* 2 on *Execution* 2.

Consider the lock a. *Rule* 1 gives no should-happen-before relation on this lock because the lock a is not in $\texttt{WLock}_{c0}$. In the cycle c_0, the lock a is in a lockset of an event for thread t_1. By *Rule* 2, any lock acquisition or release event on the lock a that happened-before $\varepsilon(c_0, t_2)$ should-happen-before the lock acquisition event on a performed by the thread t_1 at the site s_{03}. As for the thread t_2, $s_{13} \rightarrowtail \varepsilon(c_0, t_2)$ and $s_{14} \rightarrowtail \varepsilon(c_0, t_2)$, we get $s_{13} \rightsquigarrow s_{03}$ and $s_{14} \rightsquigarrow s_{03}$.

In total, based on *Execution* 2 and the cycle c_0, we identify a set of eight should-happen-before relations $\{\boldsymbol{s_{01} \rightsquigarrow s_{15}}, \boldsymbol{s_{02} \rightsquigarrow s_{15}}, \boldsymbol{s_{04} \rightsquigarrow s_{15}}, \boldsymbol{s_{05} \rightsquigarrow s_{15}}, \boldsymbol{s_{06} \rightsquigarrow s_{16}}, s_{13} \rightsquigarrow s_{03}, s_{14} \rightsquigarrow s_{03}, s_{15} \rightsquigarrow s_{08}\}$. They are depicted as dotted arrows in Figure 4(a).

Execution 2 fails to trigger the deadlock, and its execution path is $\langle s_{13}, s_{14}, s_{15}, s_{16}, s_{17}, s_{18}, s_{01}, s_{02}, s_{03}, s_{04}, s_{05}, s_{06}, s_{07}, s_{08}, s_{09}, s_{10}, s_{11}, s_{12}\rangle$. This path violates 5 out of these eight should-happen-before relations (each has been highlighted in the last paragraph). In fact, any other execution path violating at least one of these eight should-happen-before relations misses to trigger the deadlock.

Execution 1 triggers a deadlock occurrence, and its execution path is $\langle s_{13}, s_{14}, s_{01}, s_{02}, s_{03}, s_{04}, s_{05}, s_{06}, s_{07}, s_{15}\rangle$ before deadlocking at the site s_{08} for the thread t_1 and the site s_{16} for the thread t_2. We observe that this execution path satisfies all eight should-happen-before relations.

In Section 3, we have illustrated an occurrence of thrashing suffered by both *MS* and *DF*. This thrashing occurrence is due to the thread t_2 having acquired the lock n at the site s_{15} before the thread t_1 attempts to acquire the same lock at the site s_{01}, and yet the thread t_2 is actively suspended by the technique (e.g., *MS*) at the site s_{16}. The above set of should-happen-before relations has pointed out that the execution under active scheduling has already violated the relation $s_{01} \rightsquigarrow s_{15}$, irrespective to whether or not the technique suspends t_2 at s_{16}.

ConLock can identify all such should-happen-before relations before scheduling a confirmation run. As such, it has the ability to guide a thread scheduler to avoid occurrence of thrashing.

4.2.2 Generation of Should-Happen-Before Relations

ConLock treats each identified should-happen-before relation as a scheduling ***constraint*** in a confirmation run. Algorithm 1 shows the constraint set generation algorithm (Ψ-Generator for short).

Given an execution trace σ and a cycle c, Algorithm 1 firstly identifies all the locks in $\texttt{WLock}_c$ and $\texttt{HLock}_c$ and all threads in c as $\texttt{Threads}(c)$ (lines 02–06). Then, it checks each event in the projection σ_t of the trace σ over each thread t in the reversed program order starting from the deadlocking site of the thread t (lines 09–11) with respect to the two rules (lines 12–27). The set $\texttt{Threads}(c)$ at line 8 keeps all the threads involved in the cycle c (computed at line 03). For each event $e = \langle t, op, l@s, ls\rangle$ from σ_t, the algorithm checks whether the lock l is in the set $\texttt{WLock}_c$ (line 12). If this is the case, the algorithm further checks e against e_α to determine whether the pair of events e and e_α forms a should-happen-before relation based on *Rule* 1 (lines 13–14). If this is the case, it adds the relation $e \rightsquigarrow e_\alpha$ into the set Ψ (line 15). Next, the algorithm checks whether the lock l is in the set $\texttt{HLock}_c$ (line 19). If this is the case, it checks whether or not there is an event e_β operating on the lock l such that $l@s_\beta$ of the event e_β is in the lockset ls' of $\varepsilon(c, t_\beta)$ (lines 20–22), which indicates the site s_β is $\texttt{HSite}_c(l)$. If there is such an event e_β, the algorithm adds the relation $e \rightsquigarrow e_\beta$ into Ψ (line 23) based on *Rule* 2.

Algorithm 1: Ψ-Generator

```
    Input: σ : an execution trace
    Input: c : a cycle
    Output: Ψ : a constraint set with respect to c on σ
01  Ψ := ∅, WLock_c := ∅, HLock_c := ∅
02  for each event ⟨t, req, m@s, ls⟩ in c do
03  |  WLock_c := WLock_c ∪ {m}, Threads(c) := Threads(c) ∪ {t}
04  |  for each n@s_n ∈ ls
05  |  |  HLock_c := HLock_c ∪ {n}
06  |  end for
07  end for
08  for each t ∈ Threads (c) do
09  |  let i := p such that σ_t[p] = ε(c, t)
10  |  while i -- > 0 do
11  |  |  let σ_t[i] be e = ⟨t, op, l@s, ls⟩ //e ↣ ε(c, t), op ∈ {acq, rel}
12  |  |  if l ∈WLock then //By Rule 1
13  |  |  |  for each e_α = ⟨t_α, acq, m@s_α, ls_α⟩ = ε(c, t_α) ∧ t_α ≠ t do
14  |  |  |  |  if l = m then // m ∈ WLock_c
15  |  |  |  |  |  Ψ := Ψ ∪ {e ⇝ e_α}
16  |  |  |  |  end if
17  |  |  |  end for
18  |  |  end if
19  |  |  if l ∈ HLock_c then //By Rule 2
20  |  |  |  let e_β = ⟨t_β, acq, l@s_β, ls_β⟩ where t_β ≠ t
21  |  |  |  let ε(c, t_β) = ⟨t_β, acq, m@s', ls'⟩ //the deadlocking site of t_β
22  |  |  |  if l@s_β ∈ ls' then //thus, we have s_β= HSite_c(l)
23  |  |  |  |  Ψ := Ψ ∪ {e ⇝ e_β}
24  |  |  |  end if
25  |  |  end if
26  |  end while
27  end for
```

ConLock can schedule a confirmation run with the aim of not violating any constraint thus produced. However, if the size of the set constraint Ψ is large, scheduling a program execution against such a large set of constraints from the beginning may incur a high runtime overhead. In the following two subsections, we present a precise constraint reduction algorithm and an optimization by selecting a nearest scheduling point for each thread.

4.2.3 Reduction of Constraints

We first give two properties of the should-happen-before relation:

> ***Property*** 1 (*Transitivity*): If the constraint set Ψ has included both $e_1 \rightsquigarrow e_2$ and $e_2 \rightsquigarrow e_3$, then Ψ needs not to include $e_1 \rightsquigarrow e_3$ because the event order specified by $e_1 \rightsquigarrow e_3$ has been implicitly and jointly specified by the relations $e_1 \rightsquigarrow e_2$ and $e_2 \rightsquigarrow e_3$.
>
> ***Property*** 2 (*Program Locking Order*): If the constraint set Ψ has included $e_a \rightsquigarrow e_x$ and $e_r \rightsquigarrow e_x$ such that e_a is the corresponding lock acquisition event of e_r performed by the same thread t, then Ψ needs not to include $e_a \rightsquigarrow e_x$ because $e_a \rightsquigarrow e_x$ is enforced by the program order of the thread t and $e_r \rightsquigarrow e_x$.

Applying both properties produces a smaller but equivalent set of constraints generated by Algorithm 1. The reduction algorithm is straightforward: recursively applying the two properties on every triple of constraints until no more constraint can be reduced.

For the running example, applying these two properties on the constraint set produced by Algorithm 1 removes the following four constraints from the original constraint set: $s_{01} \rightsquigarrow s_{15}$, $s_{02} \rightsquigarrow s_{15}$, $s_{04} \rightsquigarrow s_{15}$, $s_{13} \rightsquigarrow s_{03}$ (see Figure 4(b)).

4.2.4 Identifying Scheduling Points

Lu et al. [32] empirically conclude that a concurrency bug in real-world large-scale multithreaded programs usually needs a "short

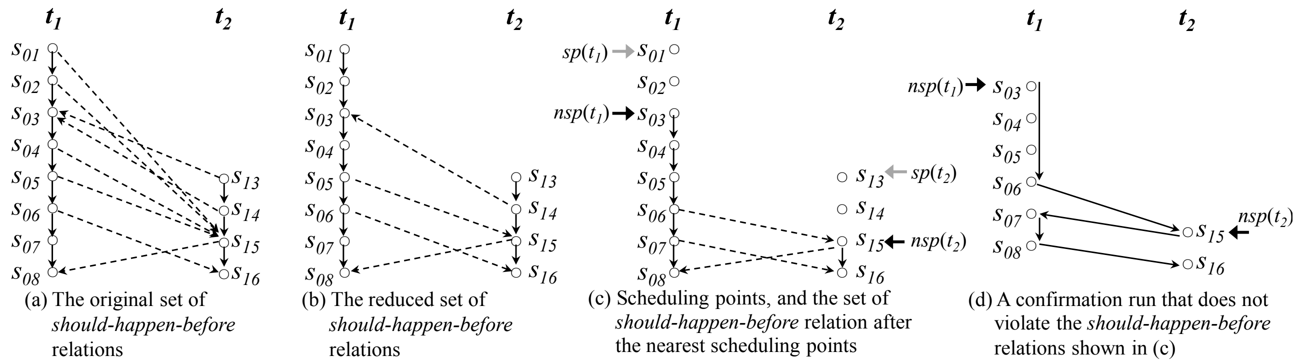


Figure 4. Reduction of constraints and selection of nearest scheduling points with respect to the cycle c_0 and *Execution* 2

depth" to manifest itself in an execution. In other words, it is empirically enough to explicitly schedule only parts of an execution to manifest a deadlock. This indicates the existence of a set of points (events) from which *ConLock* can start to schedule the involved threads. (Such a point may be the beginning of each thread in the worst case.) We refer to such a point as a *scheduling point*.

A scheduling point should-happen-before the deadlocking site of the same thread. Besides, the lockset held by a thread at such a point must be empty; otherwise, suspending a thread at its scheduling point may prevent other threads to acquire locks at their corresponding scheduling points (,which is akin to the occurrences of thrashing). In general, a thread may have one or more scheduling points. *ConLock* selects the scheduling point nearest to the deadlocking site of the same thread. We formulate a scheduling point as an event and denote all scheduling points and the nearest one of a thread t in σ as ***sp*(*t*)** and ***nsp*(*t*)**, respectively. Figure 4(c) shows four scheduling points (two for the thread t_1 and two for the thread t_2) denoted by the horizontal arrows.

The algorithm to select the nearest scheduling point for each thread t (i.e., $nsp(t)$) can be revised from Algorithm 1 by inserting the following four lines ($z_1 - z_4$) to the position in between line 25 and line 26 in Algorithm 1. For brevity and owing to its simplicity, we do not show the whole revised algorithm here.

z_1	**if** $ls = \varnothing$ **then**
z_2	$nsp(t) := e$
z_3	**break** while
z_4	**end if**

For the running example, Figure 4(d) shows an execution schedule fragment that starts from the nearest scheduling point of each thread and satisfies the constraints in Figure 4(c). In the confirmation run for the program in Figure 1, *ConLock* is able to confirm the cycle c_0 predicated from *Execution* 2 (Figure 1(b)) as a real deadlock with a certainty, and produces no thrashing occurrence.

4.3. Phase II: ConLock Scheduler

4.3.1 Confirmation Algorithm

ConLock accepts a program p, a cycle c, a set of nearest scheduling points nsp (one for each thread in c), and a set of constraints Ψ as inputs. It firstly executes the program using randomized scheduling, and monitors the events until any thread, say t, involved in c reaches (i.e., is the same as) its scheduling point. Then, *ConLock* suspends t (without executing the event), and waits for other threads involved in c to reach their corresponding nearest scheduling points. Next, *ConLock* schedules all subsequent events with the aim of not violating the reduced constraints set Ψ, and checks for deadlock occurrence. It stops the current confirmation run immediately whenever it detects a scheduling violation. We proceed to present a few auxiliary concepts before presenting the scheduling algorithm of *ConLock*.

State of a constraint. Given a constraint $h = e_a \rightsquigarrow e_b$, the state of the constraint h (denoted as `State(`h`)`) is one of the followings:

- `Idle`: if both e_a and e_b are not executed.
- `Active`: if e_b is about to be executed, and e_a is not executed.
- `Used`: if e_a is executed.

State of a thread. Given a thread t, the state of the thread t (denoted as `State(`t`)`) is one of the followings:

- `Enabled`: if t can be scheduled to execute its next event.
- `Waiting`: if t is waiting on a constraint. (Note: if t is about to execute an event e, but there is a constraint, say, $h = e' \rightsquigarrow e$ on which e' has not been executed. To avoid violating the constraint h, *ConLock* suspends the thread t until the event e' has been executed. In such cases, we say that the thread t is *waiting* on the constraint h, and is in the `Waiting` state.)
- `Suspended`: if t is suspended by *ConLock*.
- `Disabled`: if t has terminated or suspended by OS.

Definition 4. A ***scheduling violation*** occurs in a confirmation run with respect to a cycle c if the two conditions below are satisfied:

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

A scheduling violation means that no any thread in `Threads(`c`)` is in the `Enabled` state, and each thread in `Threads(`c`)` is either `Disabled` or `Waiting` on a constraint. Each `Waiting` thread t waits on a constraint, say $e' \rightsquigarrow e$, to be fulfilled (i.e., the event e' from a different thread (i.e., $\neq t$) should be executed before the execution of the event e by t). Because there is no thread in the `Enabled` state, no any event can be further executed. To continue the whole execution, at least one constraint will be violated in the current scheduling (or else a deadlock has been triggered). Because a constraint has been violated, the current confirmation run is no longer meaningful to be further scheduled not to violate other constraints in view of triggering the deadlock with respect to the given cycle. Hence, we can terminate the confirmation run.

Algorithm 2 presents the confirmation scheduler of *ConLock*. It takes a program p, a cycle c, a set of constraints Ψ, and a set of nearest scheduling points nsp (one for each thread involved in c)

Algorithm 2: ConLock Scheduler

```
   Input: p – a program
   Input: c – a cycle
   Input: Ψ – a set of constraints
   Input: nsp – the nearest scheduling points
01 for each h ∈ Ψ, State(h) := idle
02 for each thread t in p, State(t) := Enabled
03 EnabledSet:= all threads in p, SuspendedSet := ∅
04 while EnabledSet ≠ ∅ ∧ Threads(c) ≠ SuspendedSet do
05 |  e := the next event from a thread t
06 |  if e = nsp(t) then
07 |  |  SuspendedSet := SuspendedSet ∪ {t}, State(t) := Suspended.
08 |  |  EnabledSet := EnabledSet \ {t}.
09 |  else
10 |  |  execute (e)
11 |  end if
12 end while
13 EnabledSet := EnabledSet ∪ SuspendedSet //resume all threads
14 SuspendedSet := ∅
15 for each thread t do
16 |  State(t) := Enabled
17 |  LS(t) := ∅, Req(t) := ∅, Site (t) := ∅
18 end for
19 while ∃ t ∈ Threads(c) ∧ State(t) = Enabled do
20 |  let e := ⟨t, op, m@s, ls⟩ be the next event of the thread t
   |  //check e against each constraint in Ψ
21 |  if ∃ h = e_a ⇝ e_b ∈Ψ, e_b = e ∧ State(h) = Idle then
22 |  |  State(h) := Active, State(t) := Waiting on h
23 |  |  if a scheduling violation occurs by Definition 4 then
24 |  |  |  print "A scheduling violation occurs."
25 |  |  |  halt //Early termination of confirmation run
26 |  |  end if
27 |  |  continue
28 |  else if ∃ h = e_a ⇝ e_b ∈Ψ, e_a = e ∧ State(h) = Active then
29 |  |  State(h) := Used, Notify(h) //State(t'): = Enabled
30 |  else if ∃ h = e_a ⇝ e_b ∈Ψ, e_a = e ∧ State(h) = Idle then
31 |  |  State(h) := Used
32 |  end if
33 |  //else execute e and check for deadlock
34 |  switch (op)
35 |  |  case acq:
36 |  |  |  Req(t) := m, Site (t) := s
37 |  |  |  call CheckDeadlock()
38 |  |  |  Req(t) := ∅, LS(t) := LS(t) ∪ {m@s}
39 |  |  case rel:
40 |  |  |  LS(t) := LS(t) \ {m@s'} for some s'
41 |  end switch
42 |  execute (e) //other event, e.g., thread termination
43 end while
44 Function CheckDeadlock()
45 |  if ∃ a sequence of events ⟨e_1, e_2, …, e_n⟩, where e_i =⟨t_i, acq,
   Req(t_i)@Site(t_i), LS(t_i)⟩ for 1≤ i ≤ n, is a cycle by Definition 2 then
46 |  |  print "a deadlock occurs."
47 |  |  halt
48 |  end if
49 end Function
```

as inputs. The scheduler firstly initializes the state of each constraint as `Idle` (line 01). It also updates the state of each thread as `Enabled` (lines 02–03). Then, it uses OS scheduling to execute the operation of a randomly selected instruction (lines 04–12). If a thread t is about to execute an event that is the nearest scheduling point of the same thread, the scheduler suspends t, moves t from `EnabledSet` into `SuspendedSet`, and sets `State(t)= Suspended` (lines 07–08). Otherwise, the instruction is executed (line 10).

After all these threads reach their corresponding nearest scheduling points (by checking whether `Threads(c) ≠ SuspendedSet` (line 04)), *ConLock* enables all these threads (lines 13, 14, and 16).

In order to check for the occurrence of a real deadlock, *ConLock* maintains some necessary data for each thread. These data are three maps: from a 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)`, which are all initialized to be empty (line 17).

Next, *ConLock* starts its guided scheduling (lines 19–43). It randomly fetches the next event e from a random and `Enabled` thread (line 20). Before executing the event e, *ConLock* checks e against each constraint in Ψ that is not in the `Used` state, and determines the states of both the selected constraint and the current thread t (lines 21–32) such that no constraint is violated. There are three cases to consider:

- 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 e_a has been executed. *ConLock* sets `State(h) = Active` and `State(t) = Waiting` on h (lines 22). It then checks whether any scheduling violation occurs, and reports the violation if any (lines 23–26).
- If there is any constraint $h = e_a \rightsquigarrow e_b$ and `State(h) = Active`, such that the current event $e = e_a$, *ConLock* sets `State(h) = Used` and updates the state of every thread (say t') that is `Waiting` on h to be `Enabled` (lines 28–29). At line 29, we use `Notify(h)` to indicate the change of the state of each thread (say t') waiting on this constraint h from `Waiting` to `Enabled`.
- If there is any constraint $h = e_a \rightsquigarrow e_b$ and `State(h) = Idle`, such that the current event $e = e_a$, *ConLock* sets `State(h) = Used` (lines 30–31).

Next, *ConLock* checks the type of the event e, and performs a corresponding action. If e is a lock acquisition, *ConLock* updates the three maps `Req`, `Site`, and `LS`, and calls the function `CheckDeadlock()` (lines 36–38). If e is a lock release, *ConLock* updates the map `LS` only (line 40). For any other event, *ConLock* directly executes the event. Algorithm 2 then handles the next instruction.

If the function `CheckDeadlock()` (lines 44–49) finds any cycle according to *Definition 2*, *ConLock* reports the occurrence of a real deadlock, and terminates the confirmation run.

4.3.2 Discussions

ConLock can report both real deadlock occurrences and scheduling violations. This feature makes *ConLock* significantly different from existing active randomized schedulers.

Take confirming a cycle on the `MySQL` database server as an example. `MySQL` is a server program that accepts a query and returns a dataset. However, after serving this query, the program will wait for the next input instead of program termination. As such, there is always at least one active thread once `MySQL` has been started.

Existing schedulers (e.g., *MagicScheduler* and *DeadlockFuzzer*) will not terminate the confirmation run by their algorithmic design. We also recall from the motivating example that once an occurrence of thrashing happens, they will activate a previously suspended thread. Because the deadlocking site for the previously suspended thread has been passed in the run, the given cycle could no longer be confirmed.

Table 1. Descriptive statistics and execution statistics of the benchmarks (Note: * the # of locks is the # of objects)

	Benchmark	Bug ID	SLOC	Deadlock Description	# of threads/ locks	# of cycles	# of real deadlocks (cycle ID)	# of data races	# of events
Java	JDBC Connector 5.0	14927	36,300	`Connection.prepareStatement()` and `Statement.close()`	3 / 131*	10	1 (c1)	0	5,050
		31136		`PreparedStatement.executeQuery()` and `Connection.close()`	3 / 134*	16	1 (c2)	0	5,080
		17709		`Statement.executeQuery()` and `Conenction.prepareStatement()`	3 / 134*	18	2 (c3, c4)	0	5,090
C/C++	SQLite 3.3.3	1672	74,000	`sqlite3UnixEnterMutex()` and `sqlite3UnixLeaveMutex()`	3 / 3	2	2 (c5, c6)	1	16
	MySQL Server 6.0.4	34567	1,093,600	`Alter` on a temporary table and a non-temporary table	17 / 292	322	4 (c7– c10)	405	15,670
		37080		`Insert` and `Truncate` on a same table using falcon engine	17 / 211	373	1 (c11)	241	15,170

5. EXPERIMENT

5.1. Implementation and Benchmarks

Implementation. We implemented *ConLock* to handle both Java and C/C++ programs. The Java implementation used `ASM 3.2` [1] to identify all "`synchronized`" operations of each loaded class and wrap them to produce events. Following the mechanism in Java, we take each "`Object`" as a lock instance. The C/C++ implementation was based on Pin 2.10 (45467) [33] on Linux. We used the Probe mode of Pin because the analysis of deadlock is a high level problem and there is no need to monitor any low level memory access in our case; besides, the Probe mode provides almost native execution performance [33]. *ConLock* via Pin instrumented a C/C++ binary program to produce events by wrapping the Pthread library functions.

We implemented *PCT* [9], *MagicScheduler* (*MS*) [15], *DeadlockFuzzer* (*DF*) [25], and *ConLock* (*CL*) on the same framework. Although *DeadlockFuzzer* is available from the current release of *Calfuzzer* [23], yet this tool is for Java programs and cannot handle C/C++ benchmarks; and when we tried it on Java benchmark (i.e., `JDBC Connector`), it only instrumented the test harness programs but not the library files (i.e., the program code that contains the deadlocks) to prevent us from profiling any event to detect the deadlocks. We finally chose to faithfully implement *DF* based on [25] and *Calfuzzer* [23] (to include all its optimizations) instead of modifying *Calfuzzer*. We note here that according to the experiment in [25], *DF* was able to confirm deadlocks in the Java library `List` (i.e., `ArrayList`, `LinkedList`, and `Stack`) and `Map` (i.e., `HashMap`, `WeakHashMap`, `LinkedHashMap`, `IdentityHashMap`, and `TreeMap`) with 100% and 53% probabilities, respectively. The original tools of *PCT* were unavailable for downloading at the time of conducting this experiment. Thus, we implemented its scheduling algorithms for deadlocks according to [9]. We have assured our implementation by a few programs.

Benchmarks. We selected a suite of widely-used real-world Java and C/C++ programs, including `JDBC connector` [2], `SQLite` [4], and `MySQL Database Server` [3]. These benchmarks have been used in previous deadlock related experiments (e.g., [15] [26]) and are available online. All our test cases on these benchmarks are taken from [26] or their Bugzilla repositories.

Site. We used the existing Object Frequency Abstraction [16] to model the site (of an object or an event). The same site of each object or event is used by all techniques (i.e., *PCT*, *MS*, *DF*, *CL*).

5.2. Experimental Setup

We ran the experiment on Ubuntu Linux 10.04 configured with a 3.16GHz Duo2 processor and 3.25GB physical memory, OpenJDK 1.6, and GCC 4.4.3. For each benchmark, we used *MagicLock* [15] to generate the set of cycles based on the collected execution traces. We then inputted each cycle (and other inputs needed by Algorithm 2 if any) to each technique (i.e., *PCT*, *MS*, *DF*, and *CL*) for each test case to run 100 times [15][25]. *PCT* is insensitive to a given cycle. Hence, if a benchmark shows the presence of k cycles, we ran *PCT* for $100 \times k$ times.

Table 1 shows the descriptive statistics of the benchmarks used in the experiment. The column "`Benchmark`", "`Bug ID`", and "`SLOC`" show the benchmark name, the available bug report number, and the size of each benchmark in terms of SLOC, respectively. The "`Deadlock Description`" column shows the functions or operations that can lead to the corresponding deadlock state. The next three columns show the number of threads and the number of locks ("`# of threads/locks`"), the total number of cycles ("`# of cycles`"), and the cycle ID for each real deadlock ("`# of real deadlocks (cycle ID)`"). The last two columns show the number of data races ("`# of data races`") detected by *LOFT* [11][14] configured with *FastTrack* [20] and the number of events ("`# of events`") on the predicative runs, respectively.

5.3. Data Analysis

Table 2 shows the experimental results for all 11 real deadlocks summarized in Table 1. The first column shows the cycle ID ("`Cycle ID`"), followed by the number of threads and the number of locks ("`# of threads/locks in the cycle`") and the number of constraints ("`# of constraints`") before and after constraint reduction generated by *ConLock* on each cycle. (Note that all the constraints before the nearest scheduling points are not counted.) The next three major columns show the confirmation probability ("`Probability`"), the number of thrashing ("`# of thrashing`"), and the time consumption ("`Time`") by each technique to confirm each cycle, respectively. Note the time consumption is that consumed by each technique to successfully confirm the corresponding cycle as a real deadlock or the confirmation run has resulted in a preset timeout for each run (i.e., 60 seconds) as indicated by "-". On cycles `c7–c11`, we cannot precisely collect the normal execution time and the time needed by *PCT* because these cycles are on `MySQL Server` which is non-stopping according to the test harness used. We also use "-" to indicate these cases.

The confirmation probability is computed using the formula: `sc` ÷ `rt`, where `sc` is the number of runs successfully confirming the cycle, and `rt` is the total number of confirmation runs. Note that the number of thrashing occurrence may not be directly related to the confirmation probability [25].

Table 3 lists the total number of real deadlocks in each benchmark ("`# of real deadlocks`") and the total number of such deadlocks confirmed by each technique ("`Confirmed`") by at least one confirmation run.

Table 2. Experimental results comparisons among *PCT*, *MagicScheduler* (*MS*), *DeadlockFuzzer* (*DF*), and *ConLock* (*CL*)

Cycle ID	# of threads / locks in the cycle		# of constraints before / after reduction		Probability				# of thrashing				Time (in seconds)				
					PCT	*MS*	*DF*	*CL*	*PCT*	*MS*	*DF*	*CL*	*Native*	*PCT*	*MS*	*DF*	*CL*
c1	2	2	2	2	0.13	0.47	0.42	1.00	-	53	58	0	0.93	1.49	1.66	1.74	1.60
c2	2	5	2	2	0.00	0.43	0.43	1.00	-	57	57	0	0.97	-	1.55	1.51	1.52
c3	2	4	4	2	0.00	0.56	0.55	1.00	-	44	45	0	0.92	-	1.70	1.49	1.51
c4	2	4	2	3	0.13	0.51	0.49	1.00	-	49	51	0	0.92	1.43	1.44	1.57	1.52
c5	2	2	4	3	0.19	0.00	0.00	1.00	-	100	100	0	2.00	2.56	-	-	2.06
c6	2	2	4	3	0.13	0.00	0.00	1.00	-	100	100	0	2.00	2.76	-	-	2.07
c7	2	3	2,100	2	0.00	0.00	0.00	1.00	-	95	100	0	-	-	-	-	2.36
c8	2	3	2,102	2	0.00	0.16	0.22	0.71	-	78	67	0	-	-	2.65	2.15	3.81
c9	2	3	2,086	3	0.00	0.00	0.00	0.75	-	91	80	0	-	-	-	-	4.62
c10	2	3	2,088	6	0.00	0.00	0.00	0.88	-	92	78	1	-	-	-	-	2.65
c11	2	8	58	2	0.00	0.86	0.85	0.90	-	11	13	0	-	-	0.91	0.84	0.86

Table 3. The # of real deadlocks confirmed by each technique

Benchmark	Bug ID	# of real deadlocks	Confirmed			
			PCT	*MS*	*DF*	*CL*
JDBC Connector 5.0	14927	1	1	1	1	1
	31136	1	0	1	1	1
	17709	2	1	2	2	2
SQLite 3.3.3	1672	2	2	0	0	2
MySQL Server 6.0.4	34567	4	0	1	1	4
	37080	1	0	1	1	1
Total	-	11	4	6	6	11

Table 4. Average performance of *ConLock* on false positives (Note: there is no false warning on `SQLite`; "-" means time out in every run. *PCT* is excluded due to its insensitiveness to a given cycle)

Benchmark	Bug ID	# of false positives inspected	Avg. # of thrashing			Avg. Time (in seconds)		
			MS	*DF*	*CL*	*MS*	*DF*	*CL*
JDBC Connector 5.0	14927	9	100	100	0	-	-	1.66
	31136	15	100	100	0	-	-	1.74
	17709	16	100	100	0	-	-	1.68
MySQL Server 6.0.4	34567	22	91	83	2	-	-	7.85
	37080	25	95	91	0	-	-	5.34

5.3.1 Effectiveness on Real Deadlocks

Table 3 shows that *PCT* only confirmed 4 out of 11 cases as real deadlocks; *MS* and *DF* both confirmed 6 real deadlocks; and, *ConLock* confirmed all 11 deadlocks.

Table 2 shows that *ConLock* confirmed 11 cycles as real deadlocks with a probability from 71% to 100%. On confirming cycles `c1` to `c7`, *ConLock* can always confirm each of these cycles as a real deadlock in every run; whereas, the other techniques were significantly less effective in confirming these cycles as real deadlocks. On confirming cycles `c8` to `c10`, all techniques except *ConLock* can only achieve a quite low or zero confirmation probability. Specifically, *PCT*, *MS*, and *DF* each had a very low probability to confirm 5 to 7 cycles as real deadlocks, and we highlight the corresponding cells in Table 2 to ease readers to reference.

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

The column entitled "`# of thrashing`" shows that both *MS* and *DF* encountered thrashing quite frequently. On confirming each of `c1–c4`, both *MS* and *DF* each encountered thrashing in 44–58 runs out of 100 runs. On each of `c5–c7`, they even guided the corresponding confirmation runs to experience thrashing with very high probabilities. On confirming `c8–c10`, their thrashing probabilities are 0.67 to 0.92, respectively. On confirming `c11`, the number of thrashing (≤ 13 occurrences) seems acceptable.

The `MySQL Server` is the largest benchmark we used in the experiment that has 1,093,600 SLOC. On confirming cycles for this benchmark, *ConLock* encountered almost no occurrence of thrashing in the entire experiment except one on confirming `c10`. However, *MS* and *DF* encountered thrashing much more frequently.

From Table 2, we observe that the number of constraints after reduction ranges from 2 to 6. This is consistent with an empirical study result that a concurrency bug usually needs a "short depth" to manifest it [32]. We note that even though there were 2 constraints for each of 6 cycles, unlike *MS* and *DF*, *ConLock* did not suffer from thrashing on confirming these cycles as real deadlocks.

5.3.2 Effectiveness on False Positives

To validate the ability of *ConLock* on cycles that are false positives, we sampled 87 cycles out of all 730 cycles for manual verification. The 87 cycles were sampled by the following rules: (1) We selected all 40 (i.e., 9+15+16) remaining cycles on `JDBC Connector`. (2) On `SQLite`, there is not false cycle. (3) On `MySQL Server`, we selected 1 out of every 15 consecutive cycles reported by *MagicLock*, which resulted in a total of 47 cycles. We manually inspected and verified that all these 87 cycles were false positives, which had already took us about one whole week to complete this manual task. As such we did not manually verify whether the remaining 643 cycles are false positives.

Table 4 shows the mean performance of *ConLock* on handling the 87 sampled cycles. The first two columns show the benchmark and the bug ID, respectively. The next column ("`# of false positives inspected`") shows the average number of false positives reported by *ConLock* as scheduling violations that we manual verified. The last two columns ("`Avg. # of thrashing`") and ("`Avg. Time`") show the mean number of thrashing and the mean time for each technique on confirmation runs, respectively.

From Table 4, to confirm against cycles that were false positives, *MS* and *DF* were very likely to result in thrashing in the experiment; whereas *ConLock* only encountered a small number (e.g., 2 in the row entitled `MySQL Server`) of thrashing.[1]

5.3.3 *Performance*

From the column entitled "`Time`" in Table 2, the runtime overheads incurred by *MS*, *DF*, and *CL* on successful confirmations are quite close to one another, and the absolute time needed are all practical. Note that there are much more numbers of thrashing occurrences incurred by *MS* and *DF* than *CL* on each row, and on confirming cycles `c5-c6`, *MS* and *DF* simply suspended some threads until the timeout was reached.

From Table 4, we observe that *CL* can terminate a confirmation run against a false positive much earlier than *MS* and *DF*. We also found that *CL* can report a scheduling violation in each case, except in one confirmation run where a thrashing has occurred.

We have experimented to configure *CL* using the whole set of constraints without reduction and scheduling points. However, on large-scale programs (i.e., `MySQL`), this configuraiton encountered many thrashing occurrences and incurred significant slowdown.

5.4. Threats to Validity

We have not manually validated all identified cycles on `MySQL Server` due to our time and effort constraints. The probability, the ratios of thrashing, and the time taken by the techniques may be different if different numbers of runs, different benchmarks, and tool implementations were used to conduct the experiment. Our implementation is based on binary instrumentation. An implementation of *ConLock* through symbolic execution [10][31] 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 to handle large-scale programs as noted in [17] that "*the largest programs that can be symbolically executed today are on the order of thousands of lines of code*". In our benchmarks, `MySQL Server` has millions of source lines of codes (i.e., SLOC), which is far out of the ability of state-of-the-art symbolic execution engines to handle.

6. RELATED WORK

Many predictive deadlock detection techniques [5][12][18][24] [36][40][43] have been proposed. *MagicLock* [12][15] is the state-of-the-art dynamic technique. They all suffer from reporting false positives. Real deadlocks of them should be isolated. Kahlon et al. [28] proposed a static theoretical model for analysis of concurrency bugs in programs with well nested lock acquisitions and releases. However, the lock acquisitions and releases in modern real-world programs (e.g., Java and C/C++) are usually not well-nested and there exists a huge gap between static models and the modern programming languages [21]. Hence, unlike *ConLock*, their model cannot handle the occurrence of thrashing. Marino et al. [35] proposed a static approach for detecting deadlocks in object-oriented programs with data-centric synchronizations. Their approach needs manual annotations to identify the ordering between atomic-sets. *ConLock* is a fully automated dynamic approach.

DeadlockFuzzer [25] is the first technique that proposes to use the lock dependencies (i.e., a variant of *event* in this paper) to detect cycles and to schedule the program execution to confirm cycles as real deadlocks. *MagicScheduler* (the third phase of *MagicFuzzer* [15]) advances *DeadlockFuzzer* by allowing multiple cycles to be confirmed in the same run. We have intensively reviewed these two schedulers and compared them with our *ConLock* technique.

In [13], we proposed *ASN*, the first constraint based real deadlock confirmation technique. *ASN* extracts constraints from the given cycles and formulates them as barriers. However, *ASN* cannot handle false positives. *ConLock* is able to detect scheduling violation to terminate an execution with respect to false positives; on real deadlocks, like *ASN*, it is also able to confirm them with high probabilities and low slowdown overheads.

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. However, these techniques are unable to handle large-scale multithreaded programs (e.g., `MySQL`) even with the use of symbolic execution [17]. Synchronization coverage techniques [22][39][44] may explore multiple schedules of the same input, but they do not handle infeasible coverage requirements adequately.

Dimmunix [26][27] prevents the re-occurrence of each previously occurred deadlock through online monitoring. *Gadara* [42] inserts deadlock avoidance code at the gate position of each deadlock warning via static analysis and then prevents deadlock occurrence at runtime. Nir-Buchbinder et al. [37] used an execution serialization strategy for deadlock healing. These techniques develop and utilize no constraints among different threads and do not choose any nearest scheduling point (needed by *ConLock*). Besides, *Dimmunix* and *Gadara* suffer from false positives; deadlocking healing may introduce new deadlocks [37].

ESD [45] synthesizes an execution from a core dump of a previous execution with deadlock occurrence. *ConLock* can take a cycle (irrespective of whether it is a deadlock) as an input. Both *ConTest* [19] and *CTrigger* [38] inject noise to a run to increase the probability to trigger concurrency bugs. *ConLock* is not completely an active randomized scheduler, and needs not to adopt such a strategy. *PENELOPE* [41] also synthesizes an execution and uses a scheduling strategy similar to *DeadlockFuzzer* and *MagicScheduler* to detect real atomicity violations. It does not use constraints to avoid thrashing. *ConLock* uses constraints and scheduling points and is able to detect false positives.

Replay techniques (e.g., [6]) are able to 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 analyzes a given execution trace and a cycle on this trace to generate a set of constraints and a set of nearest scheduling points. It schedules a confirmation run with the aim to not violate a reduced set of constraints from the chosen nearest scheduling points. *ConLock* not only confirms real deadlocks, but also reports scheduling violations if the given cycles are false positives. The experimental results show that *ConLock* can be both effective and efficient. We will generalize *ConLock* to confirm other types of concurrency bugs effectively and efficiently in the future.

8. ACKNOWLEDGMENTS

We thank anonymous reviewers for their invaluable comments and suggestions. This work is supported in part by the General Research Fund and the Early Career Scheme of the Research Grant Council of Hong Kong (project nos. 111313 and 123512).

[1] We note that on the remaining 643 cycles (which we have not manually verified them to be false positives), *ConLock* reported scheduling violations in at least 60 runs out of 100 on each cycle, and did not report any deadlock occurrence on checking them in any confirmation run.

9. REFERENCES

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

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

[3] MySQL Database Server 6.0.4, *http://www.mysql.com*.

[4] SQLite 3.3.3, *http://www.sqlite.org*. Bug ID: 1672.

[5] 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.

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

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

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

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

[10] 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.

[11] 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.

[12] Y. Cai and W.K. Chan. Magiclock: scalable detection of potential deadlocks in large-scale multithreaded programs. *IEEE Transactions on Software Engineering* (*TSE*), 2014. *http://dx.doi.org/10.1109/TSE.2014.2301725*.

[13] 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*), 2014. *http://dx.doi.org/10.1109/TPDS.2014.2307864*.

[14] Y. Cai and W.K. Chan. LOFT: redundant synchronization event removal for data race detection. In Proc. *ISSRE*, 160–169, 2011.

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

[16] 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.

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

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

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

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

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

[22] 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.

[23] 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.

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

[25] 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.

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

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

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

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

[30] 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.

[31] C. Lattner, V. Adve, LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In Proc. *CGO*, 75−88, 2004.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

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

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

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

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

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

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

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

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

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

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

[46] K. Zhai, B.N. Xu, W.K. Chan, and T.H. Tse. CARISMA: a context-sensitive approach to race-condition sample-instance selection for multithreaded applications. In Proc. *ISSTA*, 221–231, 2012.