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

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.

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 \( \Gamma = [\text{acq}, \text{rel}] \) performed on locks, where \( \text{acq} \) represents lock acquisition and \( \text{rel} \) represents lock release. The extension to handle other synchronization primitives (e.g., barriers) is straightforward [11][14][20].

Definition 1. An event \( e = (t, op, m@s, ls) \) 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 \( \text{rel} \) in \( \Gamma \). A site is an execution context [16][25] (e.g., the triple \( \langle \text{call stack, statement number, the latest occurrence count of the couple (call stack, statement number) \rangle} \) can be used to denote an execution context).

An execution trace \( \sigma \) of a program \( p \) is a sequence of events, and \( \sigma \) is the projection of a trace \( \sigma \) on a thread \( t \) of the same trace.

2.2. Cycle as Potential Deadlock

Definition 2. A sequence of \( k \) events denoted by \( e = (e_1, e_2, ..., e_k) \), where \( e_i = (t_i, \text{acq}, m@s_i, ls_i) \) for \( 1 \leq i \leq k \), is called a cycle [15] if both of the following two conditions are satisfied:

1. for \( 1 \leq i \leq k - 1 \), \( m_i \in ls_{i+1} \), and \( m_i \in ls_j \); and,

2. for \( 1 \leq i \leq j \leq k, i \neq j, t_i \neq t_j, m_i \neq m_j, m_i \in ls_i \), 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 \( s_{06} \), 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 \( c_0 = \{(t_1, \text{acq}, m@s_{06}, \{d@s_3, p@s_4, m@s_{07}\}), (t_2, \text{acq}, p@s_{15}, \{n@s_{15}\})\} \).

We denote the set \( \{m_i | e_i = (t_i, \text{acq}, m@s_i, ls_i) \} \) \( \in \emptyset \) by \( \text{WL}(t_i) \).

Figure 2 shows a bug that can be triggered by using two threads operating on four locks. The operations \( \text{acq}(x) \) and \( \text{rel}(x) \) 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 \( \langle s_{13}, s_{14}, s_{15}, s_{16}, s_{17}, s_{18}, s_{19}, s_{20}, s_{21}, s_{22}, s_{23}, s_{24}, s_{25}, s_{26}, s_{27}, s_{28} \rangle \), resulting in a deadlock occurrence: Specifically, the thread \( t_1 \) 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 \( s_{13}, s_{14}, s_{15} \) to get the lock \( m \) at the site \( s_{16} \) and \( s_{17} \), respectively. When the thread \( t_1 \) is about to acquire the lock \( m \) at the site \( s_{18} \), 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 \( \langle s_{13}, s_{14}, s_{15}, s_{16}, s_{17}, s_{18}, s_{19}, s_{20}, s_{21}, s_{22}, s_{23}, s_{24}, s_{25}, s_{26}, s_{27}, s_{28}, s_{29}, s_{30}, s_{31}, s_{32} \rangle \), 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 \( s_{50} \) 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 \( n \) and \( p \) 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 \( c_0 = \{(t_1, \text{acq}, m@s_{06}, \{d@s_3, p@s_4, m@s_{07}\}), (t_2, \text{acq}, p@s_{15}, \{n@s_{15}\})\} \), however, without confirming the cycle \( c_0 \), this cycle \( c_0 \) is unknown to be a
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 t1 when t2 is right before executing the operation at the site s18, and suspend the thread t2 when t1 is right before executing the operation at the site s16.

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 t2 right before executing the operation at the site s16 (after the thread t2 has acquired the lock n at the site s15). To trigger the deadlock with respect to the cycle c0, MS aims to wait for the thread t1 to be suspended at the site s08. This target is nonetheless impossible to achieve because the thread t1 has been blocked at the site s31 (or the site s3b) as the lock n is being held by t2, and yet t2 has been suspended. This kind of problem is known as thrashing [25]. To resolve this occurrence of thrashing, MS resumes the thread t2, which runs to complete the execution of the operations up to the site s18 and releases the lock n. Nonetheless, the deadlocking site s16 for t2 has been passed. So, the cycle c0 could not be confirmed.

**Execution 2** starts with the thread t2 at the site s15. On Execution 2, according to the scheduling strategy of MS, MS always results in thrashing and fails to trigger the cycle c0 as a real deadlock. An execution scenario that starts with the thread t1 would still result in thrashing caused by MS. For instance, suppose that MS has successfully suspended the thread t1 at the site s16 (before acquiring the lock n), and then the thread t2 starts. The thread t2 cannot acquire the lock a at s13 because t1 is still holding the lock a. As a result, thrashing occurs. MS resumes t1 to acquire the lock n. As such, no deadlock could be triggered. This also illustrates that merely applying the active thread scheduling at the sites s08 and s16 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 c0 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 $\Psi$ (as depicted in Figure 2(b)). The generation of the constraint set $\Psi$ 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 $\Psi$, 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 $\Psi$ 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
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 \( \Rightarrow \) 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 \( e_1 \Rightarrow e_2 \). (ii) Lock acquire and release: if (1) \( e_1 \) is a lock release on a lock \( m \) by a thread \( t_1 \), \( e_2 \) is a lock acquisition on the same lock \( m \) by a thread \( t_2 \) where \( t_1 \neq t_2 \) and (3) \( e_1 \) appears prior to \( e_2 \) in the execution trace, then \( e_1 \Rightarrow e_2 \). (iii) Transitivity: if \( e_1 \Rightarrow e_2 \) and \( e_2 \Rightarrow e_3 \), then \( e_1 \Rightarrow e_3 \).

We proceed to present the definition of should-happen-before relation. We use \( \Rightarrow \) 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 \( e(i, t_i) \), and use the site of an event \( e \) to denote \( e \) when describing the \( \Rightarrow \) and \( \Rightarrow \) relations.

**Definition 3.** Given an execution trace \( \sigma \), a cycle \( c \) on \( \sigma \), suppose that \( t_1, t_2, \) and \( t_3 \) are threads involved in the cycle \( c \), where \( t_i \neq t_j \) and \( t_i \neq t_k \), the should-happen-before relation is defined as:

**Rule 1:** Suppose that \( e \) and \( e_0 \) are two events performed by two threads \( t_0 \) and \( t_0 \), respectively, and they both operate on the same lock \( m \). If the three conditions (1) \( m \in \text{Lock} \), (2) \( e \Rightarrow e(c, t) \), and (3) \( e_0 \Rightarrow e(c, t_0) \) are satisfied, then \( e \Rightarrow e_0 \).

**Rule 2:** Suppose that \( e \) and \( e_0 \) are two events performed by two threads \( t_0 \) and \( t_0 \), respectively, and they both operate on the same lock \( m \). If the three conditions (1) \( m \in \text{Lock} \), (2) \( e \Rightarrow e(c, t) \), and (3) \( e_0 \Rightarrow e(c, t_0) \) for some \( e \) are satisfied, then \( e \Rightarrow e_0 \). (Note that \( e_0 \neq (c, t_0) \) and \( e_0 \neq (c, t_k) \).

**Rule 1** defines a condition to prevent predictable thrashing to occur on these locks in the set \( \text{Lock} \). Figure 3(a) uses Execution 2 to illustrate this rule via the lock \( p \) and the cycle \( \delta \). In Figure 3(a), the lock \( p \) is in \( \text{Lock} \), the site \( s_{08} \) is the deadlocking site for the thread \( t_1 \) (i.e., \( t_1 \) in Rule 1) that operates on this lock \( p \), and the deadlock site for the thread \( t_2 \) (i.e., the thread \( t \) in Rule 1) is the site \( s_{16} \). Rule 1 specifies that any lock acquisition or release event on this lock \( p \) performed by the thread \( t \) (e.g., the event \( e \) at the site \( s_{08} \)) that happened-before the event \( e(c, t_1) \) at the site \( s_{08} \) should happen-before the event (i.e., \( e_0 \)) performed by the thread \( t_2 \) at its deadlocking site \( s_{16} \). Thus, by Rule 1, we get \( s_{08} \Rightarrow s_{16} \).

Similarly, Rule 2 defines a condition that prevents predictable thrashing on these locks in the set \( \text{Lock} \). Figure 3(b) uses Execution 2 to illustrate this rule via the lock \( n \). In Figure 3(b), the lock \( n \) is in \( \text{Lock} \), and the thread \( t_2 \) (i.e., the thread \( t_2 \) 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 \( t_2 \) (i.e., the thread \( t \) in Rule 2) is the site \( s_{15} \). Rule 2 specifies that any lock acquisition or release event on \( p \) performed by \( t_2 \) that happened-before the event occurred at its deadlocking site \( s_{15} \) should happen-before the lock acquisition event on \( p \) at site \( s_{15} \) (i.e., the event \( e_p \)). Thus, by Rule 2, we get \( s_{15} \Rightarrow s_{16} \). The lock \( n \) has also been acquired or released by the thread \( t_1 \) at sites \( s_{10}, s_{23}, \) and \( s_{24} \). So, we get \( s_{10} \Rightarrow s_{15}, s_{23} \Rightarrow s_{15}, \) and \( s_{24} \Rightarrow s_{15} \), 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 \( \{a, b, c, d\} \). The cycle \( c_0 \) has two deadlocking sites \( s_{08} \) and \( s_{15} \) for the thread \( t_2 \). \( \text{Lock} \) is \( \{a, b, c, d\} \). 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 \( n \) to identify \( s_{08} \Rightarrow s_{15}, s_{23} \Rightarrow s_{15}, s_{24} \Rightarrow s_{15}, \) and \( s_{15} \Rightarrow s_{16} \) in the above illustration of Rule 2. The thread \( t_2 \) performs the event on the lock \( n \) at its deadlocking site \( s_{15} \) which is also denoted by \( e(c_0, t_2) \). For the thread \( t_2 \), there is only one event \( e \Rightarrow e(c_0, t_2) \). For the lock \( n \) and \( e \Rightarrow e(c_0, t_2) \). By Rule 1, we get \( s_{15} \Rightarrow s_{16} \).

Consider the lock \( p \). We have applied Rule 1 on this lock to have identified \( s_{08} \Rightarrow s_{15}, s_{23} \Rightarrow s_{15}, s_{24} \Rightarrow s_{15}, \) and \( s_{15} \Rightarrow s_{16} \) in the above illustration of Rule 2. The thread \( t_2 \) performs the event on the lock \( p \) at its deadlocking site \( s_{15} \) that happened-before the event \( e(c_0, t_2) \) at the site \( s_{16} \). Thus, Rule 2 produces no further should-happen-before relation for the lock \( p \).

![Figure 2. An overview of ConLock.](image)

![Figure 3. Examples of Rule 1 and Rule 2 on Execution 2.](image)
Consider the lock a. Rule 1 gives no should-happen-before relation on this lock because the lock a is not in \( WLOCK_c \). 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 \( \mathcal{E}(c_0, t_2) \) should-have-before the lock acquisition event on a performed by the thread \( t_1 \) at the site \( s_{10} \). As for the thread \( t_2, s_{11} \Rightarrow \mathcal{E}(c_0, t_2) \) and \( s_{12} \Rightarrow \mathcal{E}(c_0, t_2) \), we get \( s_{12} \Rightarrow s_{30} \) and \( s_{11} \Rightarrow s_{30} \).

In total, based on Execution 2 and the cycle \( c_0 \), we identify a set of eight should-have-before relations \( \{ s_{30} \Rightarrow s_{15}, s_{26} \Rightarrow s_{15}, s_{26} \Rightarrow s_{10}, s_{21}, s_{12} \Rightarrow s_{10}, s_{12} \Rightarrow s_{15}, s_{26} \Rightarrow s_{10} \} \). They are depicted as dotted arrows in Figure 4(a).

Execution 2 fails to trigger the deadlock, and its execution path is \( \{ s_{15}, s_{26}, s_{26}, s_{26}, s_{21}, s_{12}, s_{10}, s_{12}, s_{12} \} \). This path violates 5 out of these eight should-have-before relations (each has been highlighted in the last paragraph). In fact, any other execution path violating at least one of these eight should-have-before relations misses to trigger the deadlock.

Execution 1 triggers a deadlock occurrence, and its execution path is \( \{ s_{15}, s_{10}, s_{10}, s_{10}, s_{10}, s_{10}, s_{10}, s_{10}, s_{10} \} \) before deadlocking at the site \( s_{10} \) for the thread \( t_1 \) and the site \( s_{10} \) for the thread \( t_2 \). We observe that this execution path satisfies all eight should-have-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 \( t_1 \) having acquired the lock \( n \) at the site \( s_{10} \) before the thread \( t_1 \) attempts to acquire the same lock at the site \( s_{10} \), and yet the thread \( t_2 \) is actively suspended by the technique (e.g., MS) at the site \( s_{10} \). The above set of should-have-before relations has pointed out that the execution under active scheduling has already violated the relation \( s_{30} \Rightarrow s_{15} \), irrespective to whether or not the technique suspends \( t_1 \) at \( s_{10} \).

ConLock can identify all such should-have-before relations before 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-have-before relation as a scheduling constraint in a confirmation run. Algorithm 1 shows the constraint set generation algorithm (\( \Psi \)-Generator) for short.

Given an execution trace \( \sigma \) and a cycle \( c \), Algorithm 1 firstly identifies all the locks in \( WLOCK \) and \( HLOCK \), and all threads in \( c \) as Threads(\( c \)) (lines 02–06). Then, it checks each event in the projection \( \sigma \) of the trace \( \sigma \) 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 Threads(\( c \)) at line 8 keeps all the threads involved in the cycle \( c \) (computed at line 03). For each event \( e = (t, op, l, b, s, l) \) from \( \sigma \), the algorithm checks whether the lock \( l \) is in the set \( WLOCK \) (line 12). If this is the case, the algorithm further checks \( e \) against \( e_\alpha \) to determine whether the pair of events \( e \) and \( e_\alpha \) forms a should-happen-before relation based on Rule 1 (lines 13–14). If this is the case, it adds the relation \( e \Rightarrow e_\alpha \) into the set \( \Psi \) (line 15). Next, the algorithm checks whether the lock \( l \) is in the set \( HLOCK \) (line 19). If this is the case, it checks whether or not there is an event \( e_\beta \) operating on the lock \( l \) such that \( e_\beta \) of the event \( e_\beta \) is in the lockset \( b' \) of \( e(c, t_\beta) \) (lines 20–22), which indicates the site \( s_{30} \) is \( HSITE(l) \). If there is such an event \( e_\beta \), the algorithm adds the relation \( e \Rightarrow e_\beta \) into \( \Psi \) (line 23) based on Rule 2.

ConLock can schedule a confirmation run with the aim of not violating any constraint thus produced. However, if the size of the set constraint \( \Psi \) 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 \( \Psi \) has included both \( e_1 \Rightarrow e_2 \) and \( e_2 \Rightarrow e_3 \), then \( \Psi \) needs not to include \( e_1 \Rightarrow e_3 \) because the event order specified by \( e_1 \Rightarrow e_3 \) has been implicitly and jointly specified by the relations \( e_1 \Rightarrow e_2 \) and \( e_2 \Rightarrow e_3 \).

**Property 2 (Program Locking Order):** If the constraint set \( \Psi \) has included \( e_1 \Rightarrow e_2 \) and \( e_2 \Rightarrow e_1 \) such that \( e_3 \) is the corresponding lock acquisition event of \( e_1 \) performed by the same thread \( t \), then \( \Psi \) needs not to include \( e_1 \Rightarrow e_3 \) because \( e_3 \Rightarrow e_1 \) is enforced by the program order of the thread \( t \) and \( e_1 \Rightarrow e_3 \).

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: \( s_{91} \Rightarrow s_{15}, s_{30} \Rightarrow s_{12}, s_{26} \Rightarrow s_{15}, s_{15} \Rightarrow s_{30} \) (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
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 in σ as sp(t) and nsp(t), respectively.

Figure 4(c) shows four scheduling points (two for the thread t1 and two for the thread t2) 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 (z1 − z4) 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.

```
z1 if ls = ∅ then
z2 nsp(t) := e
z3 break while
z4 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 c0 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 = e0 ⇔ e2 of the state of the constraint h (denoted as State(h)) is one of the followings:

- **Idle:** if both e0 and e2 are not executed.
- **Active:** if e0 is about to be executed, and e2 is not executed.
- **Used:** if e2 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′ ⇔ 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:

- \( ∃ t ∈ \text{Threads}(c), \text{such that State}(t) = \text{Enabled}, \text{and}, ∃ t ∈ \text{Threads}(c), \text{such that State}(t) = \text{Waiting}. \)

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′ ⇔ e, to be fulfilled (i.e., the event e′ from a different thread (i.e., ≠ 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: $e$ — a cycle
Input: $\Psi$ — a set of constraints
Input: $nsp$ — the nearest scheduling points

01 for each $h \in \Psi$, State($h$) := Idle
02 for each thread $t$ in $p$, State($t$) := Enabled
03 EnabledSet := all threads in $p$, SuspendedSet := $\emptyset$
04 while EnabledSet $\neq \emptyset \land$ Threads($c$) $\neq$ SuspendedSet do
05 $e$ := the next event from a thread $t$
06 if $e = nsp(t)$ then
07 SuspendedSet := SuspendedSet $\cup \{t\}$, State($t$) := Suspended.
08 EnabledSet := EnabledSet \ {t}.
09 else
10 execute($e$)
11 end if
12 end while
13 EnabledSet := EnabledSet $\cup$ SuspendedSet //resume all threads
14 SuspendedSet := $\emptyset$
15 for each thread $t$ do
16 State($t$) := Enabled
17 LS($t$) := $\emptyset$, Req($t$) := $\emptyset$, Site($t$) := $\emptyset$
18 end for
19 while $\exists t \in$ Threads($e$) $\land$ State($t$) $\neq$ Enabled do
20 let $e := (t, op, m, s, l)$ be the event of the thread $t$
//check $e$ against each constraint in $\Psi$
21 if $\exists h = e_a \not\rightarrow e_b \in \Psi, e = e \land$ State($h$) $=$ Idle then
22 State($h$) := Active, Site($t$) := 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 $\exists h = e_a \not\rightarrow e_b \in \Psi, e = e \land$ State($h$) $=$ Active then
29 State($h$) := Used, Notify($h$) //Site($r$) := Enabled
30 else if $\exists h = e_a \not\rightarrow e_b \in \Psi, e = e \land$ State($h$) $=$ Idle then
31 State($h$) := Used
32 end if
33 //else execute $e$ and check for deadlock
34 switch (op)
35 case acqu:
36 $\quad$ Req($t$) := $\emptyset$, Site($t$) := $s$
37 call CheckDeadlock()
38 $\quad$ Req($t$) := $\emptyset$, LS($t$) := LS($t$) $\cup$ $\{m@gs\}$
39 case rel:
40 $\quad$ LS($t$) := LS($t$) $\setminus \{m@gs\}$ for some $s$
41 end switch
42 execute($e$) //other event, e.g., thread termination
43 end while
44 Function CheckDeadlock()
45 if $\geq$ a sequence of events ($e_1, e_2, \ldots, e_n$), where $e_i$ = ($t, op, acq, Req(t), Site(t), LS(t))$ for $1 \leq i \leq n$, is a cycle by Definition 2 then
46 print "a deadlock occurs."
47 halt
48 end if
49 end Function

After all these threads reach their corresponding nearest scheduling points (by checking whether Threads($c$) $\neq$ 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: a thread to a lockset as LS($t$), from thread to its requested lock as Req($t$), and from thread 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 $\Psi$ 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 $h = e_a \not\rightarrow e_b$ such that State($h$) = Idle and the current event $e = e_o$, the execution of event $e$ will be postponed until $e_o$ has been executed. ConLock sets State($h$) = Active and State($t$) = Waiting on $h$ (lines 22). Then it checks whether any scheduling violation occurs, and reports the violation if any (lines 23–26).

- If there is any constraint $h = e_a \not\rightarrow e_b$ and State($h$) = Active, such that the current event $e = e_o$, 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 $h = e_a \not\rightarrow e_b$ and State($h$) = Idle, such that the current event $e = e_o$, 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.
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, LinkHashMap, 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 damage of operations or assertions 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 FastTrach [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 ("# of thrashing") 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 1. Descriptive statistics and execution statistics of the benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Bug ID</th>
<th>SLOC</th>
<th>Deadlock Description</th>
<th># of threads/locks</th>
<th># of cycles</th>
<th># of real deadlocks (cycle ID)</th>
<th># of data races</th>
<th># of events</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDBC</td>
<td>14927</td>
<td>3630</td>
<td>Connection.prepareStatement() and Connection.close()</td>
<td>3 / 131</td>
<td>10</td>
<td>1 (c1)</td>
<td>0</td>
<td>5,050</td>
</tr>
<tr>
<td>Connector</td>
<td>31136</td>
<td></td>
<td>PreparedStatement.executeQuery() and Connection.close()</td>
<td>3 / 134</td>
<td>16</td>
<td>1 (c2)</td>
<td>0</td>
<td>5,080</td>
</tr>
<tr>
<td>5.0</td>
<td>17709</td>
<td></td>
<td>Statement.executeQuery() and PreparedStatement()</td>
<td>3 / 134</td>
<td>18</td>
<td>2 (c3, c4)</td>
<td>0</td>
<td>5,090</td>
</tr>
<tr>
<td>SQLite 3.3.3</td>
<td>1672</td>
<td>74,000</td>
<td>sqlite3UnixEnterMutex() and sqlite3UnixLeaveMutex()</td>
<td>3 / 3</td>
<td>2</td>
<td>2 (c5, c6)</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>MySQL</td>
<td>3457</td>
<td>1,093,600</td>
<td>Alter on a temporary table and a non-temporary table</td>
<td>17 / 292</td>
<td>322</td>
<td>4 (c7-c10)</td>
<td>405</td>
<td>15,670</td>
</tr>
<tr>
<td>Server 6.0.4</td>
<td>37080</td>
<td></td>
<td>Insert and Truncate on a same table using falcon engine</td>
<td>17 / 211</td>
<td>373</td>
<td>1 (c11)</td>
<td>241</td>
<td>15,170</td>
</tr>
</tbody>
</table>
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 real deadlocks 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 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 experi-

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

<table>
<thead>
<tr>
<th>Cycle ID</th>
<th># of threads / locks in the cycle</th>
<th># of constraints before / after reduction</th>
<th>Probability</th>
<th># of thrashing</th>
<th>Time (in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PCT</td>
<td>MS</td>
<td>DF</td>
<td>CL</td>
<td>PCT MS DF CL</td>
</tr>
<tr>
<td>c1</td>
<td>2 2</td>
<td>2 2</td>
<td>0.13</td>
<td>0.47</td>
<td>0.42</td>
</tr>
<tr>
<td>c2</td>
<td>2 5</td>
<td>2 2</td>
<td>0.00</td>
<td>0.43</td>
<td>0.43</td>
</tr>
<tr>
<td>c3</td>
<td>2 4</td>
<td>2 2</td>
<td>0.00</td>
<td>0.56</td>
<td>0.55</td>
</tr>
<tr>
<td>c4</td>
<td>2 4</td>
<td>2 2</td>
<td>0.13</td>
<td>0.51</td>
<td>0.49</td>
</tr>
<tr>
<td>c5</td>
<td>2 2</td>
<td>2 3</td>
<td>0.19</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>c6</td>
<td>2 2</td>
<td>2 3</td>
<td>0.13</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>c7</td>
<td>2 3</td>
<td>2,100 2</td>
<td>0.00</td>
<td>0.16</td>
<td>0.22</td>
</tr>
<tr>
<td>c8</td>
<td>2 3</td>
<td>2,102 2</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>c9</td>
<td>2 3</td>
<td>2,086 3</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>c10</td>
<td>2 3</td>
<td>2,088 6</td>
<td>0.00</td>
<td>0.86</td>
<td>0.85</td>
</tr>
<tr>
<td>c11</td>
<td>2 8</td>
<td>58 2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Bug ID</th>
<th># of real deadlocks</th>
<th>Confirmed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PCT</td>
<td>MS</td>
<td>DF</td>
</tr>
<tr>
<td>JDBC</td>
<td>14927</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Connector</td>
<td>31136</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5.0</td>
<td>17709</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>SQLite 3.3.3</td>
<td>1672</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>MySQL</td>
<td>34567</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Server 6.0.4</td>
<td>37080</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 insensitivity to a given cycle)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Bug ID</th>
<th># of false positives inspected</th>
<th>Avg. # of thrashing</th>
<th>Avg. Time (in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JDBC</td>
<td>14927</td>
<td>9</td>
<td>100 100 0</td>
<td>-</td>
</tr>
<tr>
<td>Connector</td>
<td>31136</td>
<td>15</td>
<td>100 100 0</td>
<td>-</td>
</tr>
<tr>
<td>5.0</td>
<td>17709</td>
<td>16</td>
<td>100 100 0</td>
<td>-</td>
</tr>
<tr>
<td>MySQL</td>
<td>34567</td>
<td>22</td>
<td>91 83 2</td>
<td>-</td>
</tr>
<tr>
<td>Server 6.0.4</td>
<td>37080</td>
<td>25</td>
<td>95 91 0</td>
<td>-</td>
</tr>
</tbody>
</table>
ment; whereas ConLock only encountered a small number (e.g., 2 in the row entitled MySQL Server) of thrashing.\footnote{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.}

### 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 configuration 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. PENLOPE [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.

Repaly 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).
9. REFERENCES


