Magiclock: Scalable Detection of Potential Deadlocks in Large-Scale Multithreaded Programs

Yan Cai and W.K. Chan†

Abstract—We present *Magiclock*, a novel potential deadlock detection technique by analyzing execution traces (containing no deadlock occurrence) of large-scale multithreaded programs. *Magiclock* iteratively eliminates removable lock dependencies before potential deadlock localization. It divides lock dependencies into thread specific partitions, consolidates equivalent lock dependencies, and searches over the set of lock dependency chains without the need to examine any duplicated permutations of the same lock dependency chains. We validate *Magiclock* through a suite of real-world, large-scale multithreaded programs. The experimental results show that *Magiclock* is significantly more scalable and efficient than existing dynamic detectors in analyzing and detecting potential deadlocks in large-scale execution traces from large-scale multithreaded programs.

Index Terms—deadlock detection; multithreaded programs; concurrency; lock order graph; scalability

1 Introduction

Many real-world large-scale multithreaded programs are error-prone. They suffer from concurrency bugs [34] such as data races [18], [19], [46], atomicity violations [27], [32], [34], and deadlocks [14], [15], [29], [36]. For instance, a deadlock occurrence in an execution may prevent (a part of) the program execution from making further progress.

Resource deadlock [14], [29] and *communication deadlock* [28], [31] are two broad kinds of deadlocks. A *resource deadlock* occurs when a set of threads is holding some resources (locks) and is waiting for the other resources held by the threads in the same set. A *communication deadlock* occurs when some threads wait for some messages but they never receive these messages. Previous works (e.g., [28]) have illustrated that it could be infeasible to precisely detect all kinds of deadlocks by the same technique. In this paper, we study the detection of resource deadlocks in multithreaded programs, where locks are resources.

Many predictive deadlock detection techniques have been proposed, such as static analysis [23], [41], [45], dynamic analysis [15], [29], model checking [26], runtime monitoring [44], and their integrations [14], [28]. Some studied lock order graphs [36] and their integrations [15] with the happened before relation [33]; others studied confirmation of potential deadlocks [16], [24], [29], or deadlock avoidance/healing [30], [40], [44].

Among these techniques, static analysis and model checking techniques can analyze the whole program including open frameworks. They either report many false positives [45] or are unable to scale up to handle large-scale programs [28]. Dynamic analysis analyzes a given program execution trace and may reduce false positives but its scopes is restricted by the given input (i.e., reporting false negatives). Dynamic confirmation techniques are able to automatically confirm a potential deadlock if it is a real one, but they cannot guarantee that a cycle will never deadlock. Avoidance and healing techniques are often pattern based, which may imprecisely quantify deadlock triggering conditions, producing incomplete solutions. Besides, they slow down the program executions further, and may not prevent the same deadlock to re-occur.

Modern dynamic deadlock detection techniques [36] use lockset based strategies to analyze an execution trace consisting of threads locking behaviors (which does not contain any deadlock occurrence) and predict potential deadlocks in other executions. Once a potential deadlock is found, deadlock confirmation, avoidance, or healing strategies can be applied. However, without successfully analyzing the execution trace, no potential deadlock can be reported for subsequence steps to take actions.

At the heart of the preliminary version [21] of this paper is *Magiclock*, a novel algorithm for potential deadlock detection. In this paper, we present the generalized *Magiclock* algorithm. To ease our presentation, we refer to the version of *Magiclock* in [21] as *ML1*, and refer to the generalized version proposed by this paper as *Magiclock*.

Magiclock monitors a set of critical events in a program execution and generates a trace, consisting of a sequence of *lock dependencies* [21], [29] (Section 3.2). It then analyzes the trace to detect potential deadlocks, each of which is in the form of lock dependency sequence such that in the sequence, (1) the (i+1)-th lock dependency depends on the i-th lock dependency, and (2) the first one depends on the last one. To ease our presentation, we also refer to a potential deadlock as a deadlock warning or a *cycle*.

- *Yan Cai is with Department of Computer Science, City University of Hong Kong, Tat Chee Avenue, Hong Kong. E-mail: ycai.mail@gmail.com.*
- *W.K. Chan is with Department of Computer Science, City University of Hong Kong, Tat Chee Avenue, Hong Kong. E-mail: wkchan@cityu.edu.hk.*

† Correspondence author.

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

Magiclock then classifies all the locks appearing in a trace into four sets. We show that if a potential deadlock appears in the trace, then all the involved locks must reside in exactly one of the four sets (denoted by *Cyclic-set*).

Moreover, we exploit the insights that (1) any thread can only occur *once* in any potential deadlock, (2) the sequence of all threads in a same permutation are the same, and (3) detecting one permutation of the same potential deadlock suffices to confirm the presence of the potential deadlock in the trace. *Magiclock* partitions the subset of all lock dependencies in a relation whose locks also appear in *Cyclic-set* into thread specific partitions. It arranges such partitions into a fixed order so that only one permutation of each potential deadlock needs to be explored and the remaining are eliminated.

We further exploit the insights that (1) many lock dependencies in a trace can be regarded as equivalent from the viewpoint of potential deadlock detection, and (2) detecting one cycle among these non-equivalent classes of partitions suffices to infer the presence of the other equivalent cycles. As such, *Magiclock* selects only one lock dependency among all its equivalent ones for cycle localization. The net result is a new algorithm that traverses each reduced and thread specific lock dependency relation to locate each set of lock dependencies at most once, and reports all inferred cycles equivalent to the located cycles.

Magiclock generalizes *ML1* in multiple aspects: (1) it formulates a generalized lock classification scheme (see Algorithm 1). As we will illustrate via Fig. 3 in Section 4, where *ML1* produces the graph in Fig. 3(a), this generalized scheme can produce a significantly much smaller set of lock dependencies (see Fig. 3(c)) to be considered for cycle localization. (2) It develops a new lock dependency equivalency reduction strategy and a new cycle inference strategy (in Algorithm 5). (3) *Magiclock* has been further optimized to divide the set of lock dependencies produced by Algorithm 1 into disjoint subsets, and runs Algorithm 5 over each of these disjoint subsets.

We have conducted a comprehensive validation experiment that includes 11 benchmarks with more than 10 real-world deadlock cases, and evaluates *Magiclock* in multiple dimensions. The experimental results show that *Magiclock* can scale up significantly better than existing techniques including *ML1*.

The main contribution of this paper is threefold. (i) We propose a generalized *Magiclock* to address the scalability challenges in analyzing traces and detecting potential deadlocks in large-scale multithreaded programs. (ii) We implement a prototype to show the feasibility of this generalized version of *Magiclock*. (iii) Last, but not the least, we report an experiment on a suite of real-world large-scale multithreaded benchmarks. The experimental results show that *Magiclock* can be significantly more efficient and scalable than *MulticoreSDK*, *iGoodlock*, and *ML1* in handling large-scale programs.

The rest of this paper is organized as follows. Section 2 gives a motivating example and Section 3 presents the preliminaries. Section 4 presents *Magiclock* followed by its validation experiment in Section 5. We review the related work in Section 6. Section 7 concludes the paper.

Locks: $l_1, l_2, l_3, l_4, l_5, l_6, l_7, l_8, l_9$;

```
s01  funA (lock m, lock n) {             s19  funD (lock m, lock n) {
s02      acquire (m);                    s20      acquire (m);
s03          acquire (n);                s21          acquire (n);
s04          release (n);                s22          release (n);
s05      release (m);                    s23      release (m);
s06  }                                   s24  }
s07  funB (lock m, lock n) {             s25  funE (lock m, lock n) {
s08      acquire (m);                    s26      acquire (l1);
s09          acquire (n);                s27          acquire (m);
s10          release (n);                s28              acquire (n);
s11      release (m);                    s29              release (n);
s12  }                                   s30          release (m);
s13  funC (lock m, lock n) {             s31      release (l1);
s14      acquire (m);                    s32  }
s15          acquire (n);                s33  funF (lock m, lock n) {
s16          release (n);                s34      acquire (m);
s17      release (m);                    s35          acquire (n);
s18  }                                   s36          release (n);
                                         s37      release (m);
                                         s38  }
```

Thread 1 (t_1)	**Thread 2** (t_2)	**Thread 3** (t_3)
funA (l_1, l_2);	*funC* (l_2, l_1);	*funB* (l_4, l_5);
funC (l_2, l_3);	*funE* (l_8, l_9);	*funB* (l_6, l_7);
funA (l_3, l_4);	*funA* (l_1, l_6);	*funD* (l_5, l_4);
funF (l_7, l_3);		

Fig. 1. Example program (Note that *FunA*, *FunB, FunC*, *FunD*, and *FunF* have the same locking code but they could be different in their non-locking code, such as memory accesses. Similar code existed in multiple functions may also have impacts on the performance of different techniques.)

2 Motivating Example

Example A: We motivate our work via an example program as shown in Fig. 1. The example program includes six functions (*funA* to *funF*), three threads (denoted by t_1, t_2, and t_3), and nine locks (denoted by l_1 to l_9).

A deadlock in the example occurs as follows: the thread t_1 firstly calls *funA*(l_1, l_2) and acquires l_1 at s_{02}. At this moment, suppose that t_2 calls *funC*(l_2, l_1) and acquires l_2 at s_{14}. Then, when t_1 attempts to acquire l_2 at s_{03}, it is blocked by t_2. Similarly, when t_2 attempts to acquire l_1 at s_{15}, it is blocked by t_1. Now, both threads t_1 and t_2 are mutually blocked and a deadlock occurs. After the thread t_3 terminates, the entire execution ceases to proceed further.

A *lock order graph* [14], [15], [26] is a directed multigraph and describes the lock acquisition relations among threads and locks. In such a graph, a node represents a lock. For instance, in Fig. 2 (a), the two nodes labeled as l_1 and l_2 represent the two locks l_1 and l_2, respectively. A directed edge from the node l_1 to the node l_2 annotated with a set of labels (e.g., t_1 as a label) represents that, in the course of execution, t_1 acquires l_2 while holding l_1.

Fig. 2(a) shows the lock order graph generated by analyzing the execution trace that fully executes t_1 followed by t_2 and finally t_3. We also highlight the illustrated deadlock using dotted edges.

The *Goodlock* algorithm [14], [15] directly constructs a lock order graph to detect all cycles on it (e.g., Fig. 2(a)). However, *Goodlock* is not scalable enough to handle large-scale programs. For instance, Luo et al. [36] report that such a graph for an IBM in-house program (i.e., *ITCAM*) consists of more than 300K nodes and 600K edges; and the *Goodlock* algorithm spent 48 hours and 13.6 GByte of memory to find all cycles on it [36].

Lock Reduction: *MulticoreSDK* [36] is the latest technique based on lock order graph. It uses locations (where the locks are acquired) information to reduce the lock

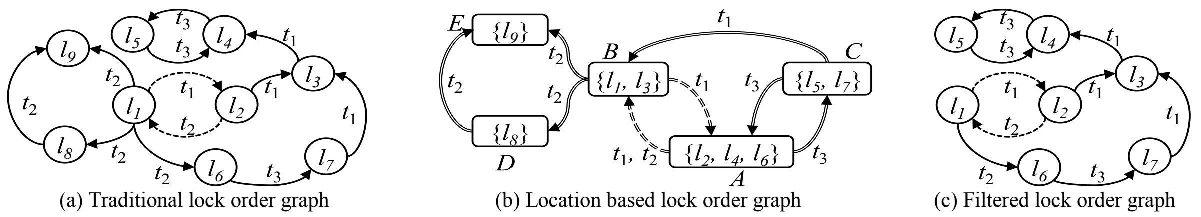

(a) Traditional lock order graph
(b) Location based lock order graph
(c) Filtered lock order graph

Fig. 2. Lock order graph example. Edges that indicate a deadlock are shown in dotted lines.

order graph and locate cycles on the reduced graph. It firstly groups the locks acquired at the same code location into one group and then merges multiple groups into one if they share any lock (because a lock may be acquired at different locations). These two phases result in a location based lock order graph, as shown in Fig. 2(b), where a node is a group and there is an edge from a group G_1 to another group G_2 if there is an edge from a lock in G_1 to some lock in G_2. In Fig. 2(b), groups *A*, *B*, and *C* form three cycles. Then, *MulticoreSDK* only considers the locks in these located groups (i.e., groups *A*, *B*, and *C*) in its second phase, where it constructs an ordinary lock order graph (see Fig. 2(c)). Because the locks in a group that does not involve in any cycle in a location based lock order graph also does not appear in the final lock order graph, this approach alleviates the scalability problem in cycle detection. Nonetheless, from Fig. 2(c), the resultant graph to locate cycles may not prune many nodes irrelevant to any cycle.

Search Strategy: *iGoodlock* [29] is the core algorithm in *DeadlockFuzzer* that searches for cycles on the full permutations of the whole set of lock dependencies generated from an execution trace (with a heuristic pruning strategy). *iGoodlock* is the same as *Goodlock* [26] except that it is more efficient but may consume much more memory [29]. But, *iGoodlock* still incurs undesirable features. For example, by the nature of its algorithmic design, it cannot avoid locating the same cycle multiple times in its search process. Hence, it uses a less desirable strategy, which is to suppress the reporting of the duplicated cycles or chains (rather than preventing them by design). Note that this problem is also suffered by the traditional lock order graph (e.g., *Goodlock* and *MulticoreSDK*).

In addition, although the total number of cycles predictable from an execution trace could be small, yet there could be a large number of lock dependency chains (see the definition in Section 3.3). In our experiment (Section 5), on a majority of the large-scale benchmarks, *iGoodlock* consumed all the memory that a Linux process was allowed to consume before reporting any/all cycles.

3 Preliminaries

3.1 Events and Execution Trace

Following [20], [29], [36], *Magiclock* monitors three types of *critical events* involving threads and locks: (1) *create*(t, t'): thread t creates a new thread t'; (2) *acquire*(t, m): thread t acquires a lock m; (3) *release*(t, m): thread t releases a lock m.

We use *Thread* and *Lock* to denote all threads and all locks, respectively.

An ***execution trace*** σ_p is a sequence of critical events.

3.2 Lock Dependency

Following [21], [29], we use the lock dependency relation to describe an execution trace.

A ***lock dependency*** $\tau = \langle t, m, L\rangle$ is a triple containing a thread t, a lock m, and a lockset L such that the thread t acquires a lock m while holding all the locks in the lockset L. In *Example A*, at the execution step where t_1 calls *funA*(l_1, l_2) and acquires the lock l_2 at s_{03} while holding the lockset $\{l_1\}$, the corresponding lock dependency is $\langle t_1, l_2, \{l_1\}\rangle$. Each lock dependency $\langle t, m, L\rangle$ corresponds to a set of edges, one for each $n_i \in L$ to m in a corresponding lock order graph and each edge is labeled with t.

A ***lock dependency relation*** D on the execution trace σ_p is a sequence of lock dependencies. To ease our presentation, we may simply refer to a *lock dependency* as a ***dependency*** and a *lock dependency relation* as a ***relation***.

Moreover, we say that two dependencies $\langle t_1, m_1, L_1\rangle$ and $\langle t_2, m_2, L_2\rangle$ are ***equivalent*** whenever $t_1 = t_2 \wedge m_1 = m_2 \wedge L_1 = L_2$. If two dependencies are equivalent, we say that they belong to the same ***lock acquisition pattern***.

Note that there is a *site* information [22] associate with each dependency. However, a site is not used in cycle detection but is only kept to report cycles for subsequent analyses (e.g., deadlock confirmation or avoidance / healing) to take actions. That is, when *Magiclock* reports a cycle, it also outputs the site information associated with all dependencies in the cycle. As such, we do not show such information along with a dependency.

3.3 Lock Dependency Chain

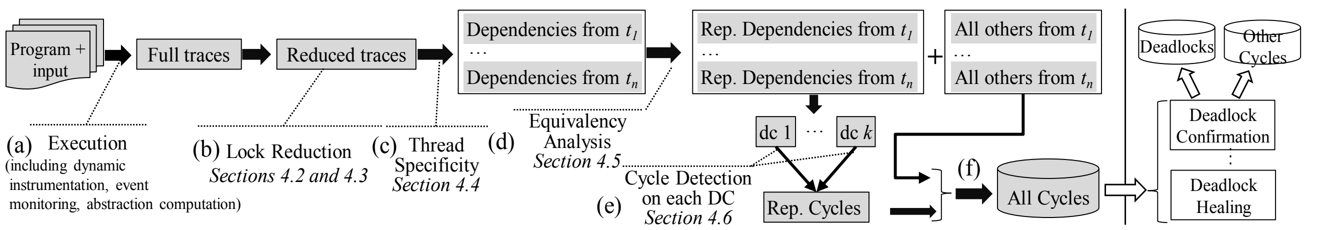
Given a sequence of k (where $k > 1$) dependencies $d = \langle \tau_1, \tau_2 \ldots \tau_k\rangle$ where $\tau_i = \langle t_i, m_i, L_i\rangle$, if $m_1 \in L_2 \ldots m_{k-1} \in L_k$, $t_i \neq t_j$, and $L_i \cap L_j = \emptyset$ for $1 \le i, j \le k$ ($i \neq j$), we refer to d as a ***lock dependency chain*** (or ***chain*** *for simplicity*). In particular, if $m_k \in L_1$, d is called a ***cyclic lock dependency chain*** (or **cyclic *chain***, **cycle** *for simplicity*). A cyclic chain represents a potential deadlock.

For example, the cycle for the dotted edges in Fig. 2 (a) is $\langle\langle t_1, l_2, \{l_1\}\rangle, \langle t_2, l_1, \{l_2\}\rangle\rangle$, which forms a real deadlock as illustrated by *Example A*.

3.4 Removable and Irremovable Locks

This section presents a few elementary definitions necessary for our technique to be presented in Section 4. The *indegree* and *outdegree* of a node n are total number of incoming edges to the node n and total number of outgoing edges from the node n, respectively; and *edgesFromTo* is the total number of edges from one node to another node. Formally, the three concepts are defined as follows:

- ***indegree*** $(m) = \Sigma\,|L'|, \forall L' \in \{L \mid \langle t, m', L\rangle \in D \wedge m = m'\}$.
- ***outdegree*** $(n) = |S|, S = \{\tau \mid \tau = \langle t, m', L\rangle \in D, n \in L\}$.



Program + input
Full traces
Reduced traces
Dependencies from t_1
…
Dependencies from t_n
Rep. Dependencies from t_1
…
Rep. Dependencies from t_n
+
All others from t_1
…
All others from t_n
Deadlocks
Other Cycles
Deadlock Confirmation
:
Deadlock Healing
dc 1
…
dc k
Rep. Cycles
(f)
All Cycles
(a) Execution (including dynamic instrumentation, event monitoring, abstraction computation)
(b) Lock Reduction Sections 4.2 and 4.3
(c) Thread Specificity Section 4.4
(d) Equivalency Analysis Section 4.5
(e) Cycle Detection on each DC Section 4.6

Fig. 3. Overview of *Magiclock* (Note: "Rep." stands for "Representative" and "dc" stands for "Disjoint Component".)

- ***edgesFromTo*** $(n, m) = |S|$, $S = \{\tau \mid \tau = \langle t, m', L\rangle \in D,\ n \in L \wedge m = m'\}$.

A lock m is said to be ***removable*** if it does not appear in any cycle. Similarly, if a lock appears in at least one cycle, it is said to be ***irremovable***. Eliminating a removable lock as well as the edges directly connected to this lock does not affect the presence of any cycle in a given set of lock dependencies. However, eliminating an irremovable lock destroys all cycles that contain this lock, compromising the cycle detection ability of a technique.

4 MAGICLOCK

4.1 Overview

Magiclock aims at efficiently and effectively analyzing an execution trace (that does not contain any deadlock occurrence) to report cycles as depicted in Fig. 3. Given a program with an input, it firstly collects the execution trace as follows (Fig. 3(a)):

Let w be an empty execution trace. Whenever an event *create* (t) occurs, *Magiclock* allocates a new thread identifier and an *empty* lockset L_t for the thread t. Whenever an event *acquire* (t, m) occurs, it firstly appends the values of the triple $\langle t, m, L_t\rangle$ to w, and then adds m to L_t (i.e., $L_t := L_t \cup \{m\}$). Also, whenever an event *release* (t, m) occurs, it removes the lock m from L_t (i.e., $L_t := L_t \backslash \{m\}$).

After collecting an execution trace, it begins to perform its cycle localization. *Magiclock* first reduces the locks as well as edges directly connecting to these locks and generates a reduced trace (Sections 4.2 and 4.3). On the reduced trace, it uses the thread specificity strategy to arrange all lock dependencies according to their thread IDs into n partitions (where n is the total number of threads) so that during searching, only one lock dependency is selected from each partition (Section 4.4). Next, it further selects a set of representative lock dependencies from each partition through equivalency analysis among lock dependencies so that during searching, when a lock dependency is selected from each partition, the dependency is the representative one among all its equivalent ones (Section 4.5). Finally, *Magiclock* groups all representative lock dependencies into different disjoint components and searches for cycles on each disjoint component (Section 4.6). Whenever it reports a cycle, all non-representative lock dependencies are considered. In this way, *Magiclock* reports all cycles in the collected execution traces. After a set of cycles have been identified, developers may further use deadlock confirmation techniques (e.g., *MagicScheduler* [21]) to attempt to confirm them as real deadlocks. Fig. 3 depicts the whole as stated above.

4.2 Analysis: Reduction of Locks

Our lock reduction relies on two insights. The first one is: a necessary condition for the lock dependency $\langle t, m, L\rangle$ in a relation D to be a part of a cyclic chain is that both the indegree and outdegree of the lock m cannot be zero. Negating this necessary condition means that a lock m with either a zero indegree or a zero outdegree is not a part of any cyclic chain in D. Such a lock m must be *removable*; and the lock dependency $\langle t, m, L\rangle$ can be eliminated from D without hampering the number of cycles that can be detected based on the reduced relation D' from D.

By so doing, the indegree and outdegree of the remaining locks in D can be reduced by not counting its connection to all eliminated removable locks without affecting the precision of cycle detection. It is because the reduced amount of indegree or outdegree of any lock should not be related to any cycle existing in D. A consequence is that, by not counting such edges, a lock may then have a zero indegree or a zero outdegree (in D'), indicating that the lock only connects to or from the locks marked as removable. Such a lock can also be marked as removable and the corresponding lock dependency can be removed from D'. As such, more lock dependencies can be iteratively removed from D', and the indegree and the outdegree of more locks will be iteratively reduced.

Our second insight in lock reduction is that, in a cyclic chain $d = \langle \tau_1, \tau_2 \ldots \tau_n\rangle$, each lock m_i in $\tau_i = \langle t_i, m_i, L_i\rangle$ for $1 \le i \le n$ requires the lock itself to have been acquired by at least two threads (i.e., any two threads t_i and t_{i+1}, where $t_{n+1} = t_1$). Therefore, if a lock is only acquired and released by only one thread, this lock can also be removed without compromising the effectiveness of potential deadlock detections. Moreover, once such a lock has been eliminated, it will open up a new opportunity to eliminate other locks based on the first insight above.

4.3 Algorithm: Reduction of Locks

To identify removable locks as many as possible, *Magiclock* iteratively classifies each lock in the set of locks *Lock* on a relation D into one of the following four sets.

- ***Independent-set*** = $\{m \mid m \in Lock,\ indegree\ (m) = 0 \wedge outdegree\ (m) = 0\}$.
- ***Intermediate-set*** = $\{m \mid m \in Lock,\ (indegree\ (m) = 0 \vee outdegree\ (m) = 0) \wedge \neg\ (indegree\ (m) = 0 \wedge outdegree\ (m) = 0)\}$.
- ***Inner-set*** = $\{m \mid m \in Lock,\ (\exists\ \langle t, m, L\rangle \in D,\ \forall\ n \in L,\ n \in Intermediate\text{-}set \cup Inner\text{-}set) \vee (\exists\ \langle t, n, L\rangle \in D,\ m \in L \wedge n \in Intermediate\text{-}set \cup Inner\text{-}set)\}$
- ***Cyclic-set*** = $\{m \mid m \in Lock,\ m \notin Independent\text{-}set \cup Intermediate\text{-}set \cup Inner\text{-}set\}$.

Algorithm 1: *LockReduction* (*D*)

```
1  call LockClassfication (D)
2  for each lock $m \in$ Cyclic-set do
3      if $mode(m, D) \neq -1$ then
4          remove m from Cyclic-set // further reduce locks in cyclic-set
5          for each lock $n \in$ Cyclic-set do
6              if $edgesFromTo(m, n) \neq 0$ do
7                  $indegree(n) := indegree(n) - edgesFromTo(m, n)$
8                  $edgesFromTo(m, n) := 0$
9              end if
10         end for
11         for each lock $n \in$ Cyclic-set do
12             if $edgesFromTo(n, m) \neq 0$ do
13                 $outdegree(n) := outdegree(n) - edgesFromTo(n, m)$
14                 $edgesFromTo(n, m) := 0$
15             end if
16         end for
17     end if
18 end for
19 $D'$ := projection of D on the locks in Cyclic-set
20 if $D' \neq D$
21     call LockReduction ($D'$)
22 end if
```

We define a function *mode*(*m, D*) to identify whether the lock *m* has been acquired by exactly one thread in the relation *D*. The possible values of *mode*(*m, D*) are:

- **0**: the lock *m* has never been acquired by any thread.
- **−1**: the lock *m* has been acquired by two or more threads.
- ***t***: the lock *m* has been acquired by exactly one thread, which is the thread *t*.

We firstly present the lock reduction algorithm (*LockReduction*) and then illustrate it using *Example B*.

LockReduction (Algorithm 1) firstly calls *LockClassification* to classify all locks appearing in *D* into the above four sets. From line 2 to line 18, *LockReduction* removes all the locks that each has been used by exactly one thread through checking the usage mode *mode*(*m, D*) for each lock *m* in *Cyclic-set*. After the removal, there might be additional locks that can be further removed. *LockReduction* then projects the relation *D* into a new relation *D*' by taking out each dependency ⟨*t, m*, L_t⟩ in *D* into *D*' such that *m* ∈ *Cyclic-set*. The new relation *D*' will be further checked by calling *LockReduction* recursively (at line 21 of Algorithm 1) to remove locks that cannot appear in any cycle. After the termination of Algorithm 1, the locks in *Cyclic-set* will be used to search for potential deadlock cycles (which will be presented in Section 4.6).

Before invoking *LockReduction*, the data structures are initialized in *InitClassification* (Algorithm 2). In *InitClassification*, *indegree*, *outdegree*, and *mode* are arrays that each maps each lock (as an index) to a number. *edgesFromTo* is a two-dimensional array (a sparse matrix), where an empty record indicates a value of zero.

LockClassification (Algorithm 3) firstly identifies all the locks that should belong to *independent-set* by checking, for each lock *m*, whether the *indegree*(*m*) and *outdegree*(*m*) are both zero (lines 3–4). Then, it further identifies all the locks that should belong to *intermediate-set* by checking, for each lock *m*, whether one of *indegree*(*m*) and *outdegree*(*m*) is zero (lines 6–7). Such an identified lock must be removable. Hence, all such locks and their edges can be removed from the subsequent consideration of cycle detection. Then, for each lock that belongs to *intermediate-set*, *LockClassification* also pushes it into a stack *S* (line 8).

Algorithm 2: *InitClassification*(*D*)

```
1  for each $m \in D.Lock$ do
2      $indegree(m) := 0$
3      $outdegree(m) := 0$
4      $mode(m, D) := 0$
5  end for
6  $edgesFromTo := \varnothing$
7  for each lock dependency $\langle t, m, L \rangle \in D$ do
8      if $mode(m, D) \neq t \wedge mode(m, D) \neq 0$ then
9          $mode(m, D) := -1$
10     else
11         $mode(m) := t$
12     end if
13     for each lock $n \in L$
14         $indegree(m) := indegree(m) + 1$
15         $outdegree(n) := outdegree(n) + 1$
16         $edgesFromTo(n, m) := edgesFromTo(n, m) + 1$
17     end for
18 end for
```

Algorithm 3: *LockClassification* (*D*)

```
1  Stack $S := \varnothing$; Independent-set $:= \varnothing$; Intermediate-set $:= \varnothing$;
       Inner-set $:= \varnothing$; Cyclic-set $:= \varnothing$
2  for each lock $m \in D.Lock$
3      if $indegree(m) = 0$ and $outdegree(m) = 0$ then
4          add m to Independent-set  // keep in independent-set
5      else
6          if $indegree(m) = 0$ or $outdegree(m) = 0$ then
7              add m into Intermediate-set  // keep in intermediate-set
8              push m into S
9          end if
10     end if
11 end for
12 while S is non-empty do
13     pop m from S
14     if $indegree(m) = 0$ then
15         for each $n \in D.Lock \wedge n \neq m$ do
16             $indegree(n) := indegree(n) - edgesFromTo(m, n)$
17             $outdegree(m) := outdegree(m) - edgesFromTo(m, n)$
18             $edgesFromTo(m, n) := 0$
19             if $indegree(n) = 0$ then
20                 push n into S
21                 add n into inner-set  // keep in inner-set
22             end if
23         end for
24     end if
25     if $outdegree(m) = 0$ then
26         for each $n \in D.Lock$ and $n \neq m$ do
27             $outdegree(n) := outdegree(m) - edgesFromTo(n, m)$
28             $indegree(m) := indegree(m) - edgesFromTo(n, m)$
29             $edgesFromTo(n, m) := 0$
30             if $outdegree(n) = 0$ then
31                 push n into S
32                 add n into Inner-set  //keep in inner-set
33             end if
34         end for
35     end if
36 end while
37 for each lock $m \in D.Lock$ do
38     if $m \notin$ Independent-set $\cup$ Intermediate-set $\cup$ Inner-set then
39         add m to Cyclic-set  // keep in cyclic-set
40     end if
41 end for
```

After the classification of the locks to the first two sets, *LockClassification* enumerates the content of the stack *S* to identify the locks that should belong to *inner-set*. The procedure is as follows: for each lock *m* in *S*, there are two cases: (*Case* 1) *indegree*(*m*) = 0 and (*Case* 2) *outdegree*(*m*) = 0. For *Case* 1, *LockClassification* subtracts both *indegree*(*n*) and *outdegree*(*m*) from *edgesFromTo*(*m, n*), respectively, for each *n* connected from *m*. It then resets *edgesFromTo*(*m, n*) to be 0, indicating that all edges from the lock *m* to the lock *n* have been labeled as "removed". After the subtraction and reset (if any), if *indegree*(*n*) becomes zero, the lock *n* will be classified to *inner-set* and also be pushed into *S* (lines 14–24) for further inference in subsequent iterations. For *Case* 2, *LockClassification* performs the similar actions as what it does to handle *Case* 1 (lines 25–35).

For the remaining locks, *LockClassification* classifies them into *Cyclic-set* (lines 37–41). In Section 4.7, we present a theorem to show that *LockClassification* correctly classifies all irremovable locks into *Cycle-set*.

Example B: Take the lock order graph in Fig. 2(a) for our illustration purpose. TABLE 1 shows the *indegree* and *outdegree* of each lock for the lock order graph in Fig. 2(a).

TABLE 1
THE INDEGREES AND OUTDEGREES FOR THE LOCKS ON THE GRAPH SHOWN IN FIG. 2(a)

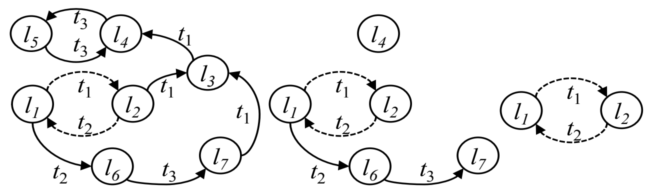
Lock instance	l_1	l_2	l_3	l_4	l_5	l_6	l_7	l_8	l_9
indegree	1	1	2	2	1	1	1	1	2
outdegree	4	2	1	1	1	1	1	1	0

After the initialization of *indegree*, *outdegree*, *mode*, and *edgesFromTo* for every lock, *LockClassification* aims to classify locks to *independent-set*. As shown in TABLE 1, no lock has 0 in both the `indegree` and the `outdegree` rows; hence, no lock is classified into *independent-set*. Then, it classifies the lock l_9 into *intermediate-set* because it has a value of 0 in its *outdegree* row, and the algorithm pushes l_9 into the stack S (initially empty). Then, no lock has zero-indegree or zero-outdegree. Next, *LockClassification* iteratively pops each lock in S and reduces indegree or outdegree of other locks connecting to the popped lock. The lock l_9 is firstly popped out and the outdegree of the locks l_1 and l_8 are reduced by 1 and 1, respectively; however, the outdegree of the lock l_1 is non-zero and it is not pushed into S; but the outdegree of the lock l_8 becomes 0. It is then classified into *inner-set* and is pushed into S. *LockClassification* further pops l_8 from S. Similarly, it reduces the outdegree of l_1 by 1. After that, l_1 has non-zero indegree or non-zero outdegree. The stack S becomes empty. *LockClassification* terminates its iteration on S. Next, *LockClassification* classifies all locks that has not been classifies into the first three sets into *Cyclic-set*. Finally *LockClassification* terminates and the *Cyclic-set* includes the locks $\{l_1, l_2, l_3, l_4, l_5, l_6, l_7\}$, as shown in Fig. 4(a).

Next, *LockReduction* further removes the locks l_3 and l_5 from *Cyclic-set* because these two locks have only been acquired and released by the thread t_1 and the thread t_3, respectively, which results in Fig. 4(b). Then, by only considering the lock dependencies for the set of locks identified by the current *Cyclic-set* (i.e., the locks $\{l_1, l_2, l_4, l_6, l_7\}$), *LockReduction* calls itself recursively, which invokes *LockClassification* again. *LockClassification* removes three more locks l_4, l_6 and l_7 by classifying them into *Independent-set*, *Intermediate-set*, and *Inner-set*, respectively, in this round of lock classification. Finally, only the locks $\{l_1, l_2\}$ are remained in the final *Cyclic-set,* as shown in Fig. 4(c).

Compared to *Magiclock*, *ML1* only works on the inputted relation D once and hence can only produce the result corresponding to the lock order graph in Fig. 4(a). This graph is much larger than that in Fig. 4(c).

We emphasize that Algorithm 1 uses *Cyclic-set* to identify a set of irremovable locks and does not alter the contents of any lock dependencies. By the definition of cyclic chain (Section 3.3), the locksets of lock dependencies of a chain should be pairwise disjoint, leaving no room for a gate lock (or a guarding lock) [44] to appear in cyclic chain.



(a) Lock order graph after the first call to *LockClassification* (b) Lock order graph after removing the locks l_3 and l_5 (c) The final lock order graph

Fig. 4. Illustration of *LockReduction* on the example in Fig. 1.

4.4 Analysis: From Non-Equivalent Many to One

Let us firstly consider an example on the relation $D_{eg} = \{\tau_1, \tau_2, \tau_3, \tau_4\}$, and the lock dependency sequence that we want to discuss is $d = \langle \tau_4, \tau_2, \tau_1, \tau_3 \rangle$. To ease our discussion, we further suppose that the thread ID for the lock dependency τ_i is t_i. We distinguish two cases for $\langle \tau_3, \tau_4 \rangle$ to be a chain and not to be a chain. If $\langle \tau_3, \tau_4 \rangle$ is a chain, then d is a cyclic chain; otherwise, d is not cyclic.

We firstly take *iGoodlock* as the algorithm to illustrate the challenges in existing approaches. *iGoodlock* uses a breadth first search (BFS) strategy. At the first iteration level, *iGoodlock* checks every pair of lock dependencies in D_{eg}, and produces all chains of length 2, and there are 3 such chains in total if d is not a cyclic chain and 4 in total if d is a cyclic chain. The former case has one fewer chain (i.e., without $\langle \tau_3, \tau_4 \rangle$). With this set of chains, *iGoodlock* continues to its second iteration level. It checks each chain produced at the first iteration level against each lock dependency in D_{eg}, and produces 2 and 4 chains of length 3 each for the chain d not being cyclic and d being cyclic, respectively. Then, *iGoodlock* continues to produce 1 and 4 chains of length 4 each at the third iteration level for the two cases, respectively. It produces no chain at the fourth iteration level, and hence the algorithm terminates.

As a whole, if d is not a cyclic chain, *iGoodlock* produces up to 3 chains at an iteration level (i.e., at the first iteration level), iterates on D_{eg} for 10 times, visits 44 nodes in total, and does not report any cyclic chain. If d is a cyclic chain, *iGoodlock* produces 4 chains at the third iteration level, iterates on D_{eg} for 16 times, visits 52 nodes in total, and finally reports one cyclic chain as well as suppresses the reporting of 3 other duplicated cyclic chains (i.e., $\langle \tau_1, \tau_3, \tau_4, \tau_2 \rangle$, $\langle \tau_2, \tau_1, \tau_3, \tau_4 \rangle$, and $\langle \tau_3, \tau_4, \tau_2, \tau_1 \rangle$).

From above, we observe the following: for each cyclic chain d, the existing algorithm produces many redundant chains, which are nonetheless treated as the same chain in cycle reporting at the end. Moreover, suppose only one (say $\langle \tau_4, \tau_2, \tau_1, \tau_3 \rangle$) of the four cyclic chains at the third iteration level is the cycle to be reported. Then, there is indeed no need to produce some of the prefixes of all other three cyclic chains at the first two iteration levels, which are used to produce $\langle \tau_1, \tau_3, \tau_4, \tau_2 \rangle$, $\langle \tau_2, \tau_1, \tau_3, \tau_4 \rangle$, or $\langle \tau_3, \tau_4, \tau_2, \tau_1 \rangle$. If d is not a cyclic chain, the situation is better because in this case, $\langle \tau_3, \tau_4 \rangle$ is not a chain, and so, less redundant chains will be generated.

To address these problems, we propose the ***Thread Specificity*** strategy as follows: *Magiclock* firstly partitions the set of lock dependencies (that are produced by Algorithm 1) by their thread IDs and sorts the partitions in the ascending order (or any other fixed order) of their thread

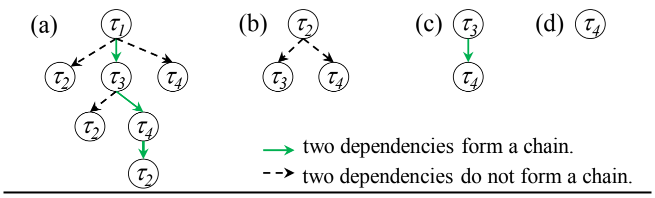

(a)
(b)
(c)
(d)
two dependencies form a chain.
two dependencies do not form a chain.

Fig. 6. Search Process on the Relation $D_{eg} = \{\tau_1, \tau_2, \tau_3, \tau_4\}$

s_{01}	***funA***(lock *m*, lock *n*) {	**Thread 1** (t_1)
s_{02}	*acquire* (*m*);	*funA* (l_1, l_2);
s_{03}	*acquire* (*n*);	*funA* (l_1, l_2);
s_{04}	*release* (*n*);	**Thread 2** (t_2)
s_{05}	*release* (*m*);	*funA* (l_2, l_1);
s_{06}	}	*funA* (l_1, l_2);

Fig. 5. Example for Equivalent Lock Dependency Identification

IDs. It then searches one specific permutation of every potential deadlock cycle such that a lock dependency with a lowest thread ID is always searched first in the permutation. Because each thread can only occur once in a cycle, there is no need to pick more than one lock dependency from each such partition. Besides, *Magiclock* avoids exploring any next sub-tree whose root (the sub-root) is in the nodes from the root node to the current node in the search tree. If the permutation is a cyclic dependency chain, it is reported as a cycle.

Each lock dependency τ_l in D_{eg} refers to one specific thread t_i. Therefore, *Magiclock* firstly divides D_{eg} into four partitions, denoted by D_1, D_2, D_3, and D_4, respectively, where $D_i = \{\tau_i\}$. We show the search process using the thread specificity strategy as follows with help of Fig. 6:

- During the search for cycles, *Magiclock* firstly checks D_1 against D_2. It finds no chain as shown in Fig. 6(a) by the dotted arrow between the node τ_1 and the node τ_2. It then checks D_1 against D_3 and finds a chain $\langle\tau_1, \tau_3\rangle$. Because *Magiclock* uses a depth first search (DFS) approach, when $\langle\tau_1, \tau_3\rangle$ is found, it further checks $\langle\tau_1, \tau_3\rangle$ against D_2 and finds no chain. Next, it checks $\langle\tau_1, \tau_3\rangle$ against D_4 and finds $\langle\tau_1, \tau_3, \tau_4\rangle$. Similarly, it further checks $\langle\tau_1, \tau_3, \tau_4\rangle$ against D_2, and gets $\{\langle\tau_1, \tau_3, \tau_4, \tau_2\rangle\}$, which is a cycle. It then terminates searching on this path and goes back. It further checks D_1 against D_4 and finds no chain.
- Then, it searches for cycles starting from D_2 and skips checking against D_1. When it checks D_2 against D_3, no chain is found as shown in Fig. 6(b). It further checks D_2 against D_4 and still no chain is found.
- Similarly, it searches starting from D_3 and skips checking D_1 and D_2, as shown in Fig. 6(c). When it checks D_3 against D_4, a chain $\langle\tau_3, \tau_4\rangle$ is found. However, there is no more partition to be checked along this path.
- Finally, it searches starting from D_4, but there is no partition to be checked with, as shown in Fig. 6(d).

When the whole searching terminates, *Magiclock* visits 13 nodes in total (i.e., the sum of all nodes in Fig. 6) no matter the dependency *d* is a cyclic chain or not, which is much fewer than that visited by *iGoodlock* (i.e., 44 or 52).

Magiclock uses a depth first search among the partitions. It needs to keep only one intermediate result at each iteration level, and needs not to check the intermediate chain against any partition that one of its lock dependencies has appeared in the intermediate result denoted by the current search path. By so doing, it saves many unnecessary comparisons incurred by existing algorithms.

Magiclock searches the whole tree, and hence does not miss to report any cycles. For a cyclic chain with a length of k, it only searches for one permutation which starts with the thread with the lowest thread ID and avoids (instead of suppressing) the generation of other $k - 1$ permutations. (Note that the thread IDs of other dependencies except the first one in a reported cycle may not appear in ascending order.) Its algorithmic design also avoids comparing any two or more dependencies sharing the same thread ID (i.e., two dependencies from the same partition), which saves computational time.

4.5 Analysis: From Equivalent Many to One

A longer execution trace means that more critical events have been monitored during the execution of a program. The time to search for cycles over a longer trace may tend to grow exponentially. It is because the number of edges in a lock order graph (or dependencies in the corresponding relation) is usually much larger than the number of locks in the same execution trace. On the other hand, a technique needs to search over a permutation of these locks/dependencies in order to locate cycles.

We observe that a thread may repeat its lock acquisition procedure (e.g., in a loop to process an array of shared data) [19]. As such, the same lock acquisition pattern (see its definition in Section 3.2) may appear multiple times in a trace. Because multiple lock dependencies can be regarded as equivalent (that suffices for dependency based potential deadlock detection), we exploit this insight to scale up *Magiclock* further.

We firstly give an example to further motivate our work. Fig. 5 shows an example program with two threads t_1 and t_2. The thread t_1 calls *funA* twice, where each call results in lock acquisitions and releases on the locks l_1 and l_2 in a nested manner. The thread t_2 also calls *funA* twice but two calls results in two different nested orders on acquisition and releases of two locks l_1 and l_2.

The generated sequences of lock dependencies with respect to each thread are: $\sigma_{t1} = \langle\tau_1, \tau_2\rangle$ and $\sigma_{t2} = \langle\tau_3, \tau_4\rangle$. Note that we do not show any lock dependency with an empty lockset because such a dependency is irrelevant to any cyclic chain. According to the above scenario, we have $\tau_1 = \langle t_1, l_2, \{l_1\}\rangle$ which models the lock acquisition at the location s_{03} via the first invocation to *funA* by t_1. Also, we have $\tau_2 = \langle t_1, l_2, \{l_1\}\rangle$, $\tau_3 = \langle t_2, l_1, \{l_2\}\rangle$, and $\tau_4 = \langle t_2, l_2, \{l_1\}\rangle$, which can be interpreted similarly. As the locking orders on the two locks by the two threads are not the same, there are two potential deadlocks $\langle\tau_1, \tau_3\rangle$ and $\langle\tau_2, \tau_3\rangle$.

We observe that τ_1 and τ_2 are equivalent dependencies with each other. For a set X of equivalent dependencies, if one of them appears in a lock dependency chain, any other dependencies in X can be a substitute of this lock dependency to construct a new chain. Most importantly, if one of them does not appear in any cycle, all other dependencies in X cannot appear in any cycle. Deadlock in a real-world multithreaded program does not frequently occur. Besides, most lock dependencies in the relation D should be irrelevant to any cycle. Hence they should fall within this case.

Algorithm 4: *DisjointComponentsFinder* (*Cyclic-set*)

```
 1  DCS dcs := ∅   //Disjoint Component Set
 2  DC  dc  := ∅   //Disjoint Component
 3  visited (m) := false, for each m ∈ Cyclic-set
 4  for each m ∈ Cyclic-set do
 5    if visited(m) = false then
 6      visitEdgesFrom (m, dc)
 7      dcs := dcs ∩ {dc}
 8      dc := ∅
 9    end if
10  end for
11  Function visitEdgesFrom (m)
12    if visited(m) = false then
13      dc := dc ∪ {m} //add m to the current disjoint component
14      visited(m) := true
15      for each edge <m, n> from m do
16        //i.e., ∀ n ∈ Cyclic-set, edgesFromTo(m, n) ≠ 0
17        visitEdgesFrom (n)
18      end for
19    end if
20  end Function
```

Algorithm 5: *CycleDetection*(*dc*, *D*)

```
 1  k := |D.Thread|, Group := ∅
 2  isTraversed(i) := false, D_i := ∅, for each i from 1 to k
 3  for each dependency τ = ⟨t, m, L⟩ ∈ D do
 4    if m ∈ dc ∧ L ≠ ∅ then
 5      if findEquDepGroup(D_i, τ) returns a Group g then
 6        g := g ∪ {τ}   //equivalent dependency
 7      else
 8        add τ into D_i  //non-equivalent dependency
 9        Group(τ) := ∅
10      end if
11    end if
12  end for
13  Stack S := ∅
14  for each i from 1 to k do
15    for each τ ∈ D_i do
16      isTraversed(i) := true
17      call DFS_Traverse(i, S, τ)
18    end for
19  end for
20  Function DFS_Traverse(i, S, τ)
21    push τ into S
22    for each j from i+1 to k do  //repeated cycles elimination
23      if isTraversed(j) = false then //otherwise, skip all visited D_j
24        for each τ' ∈ D_j do
25          θ := S
26          push τ' into θ
27          if θ forms a chain then
28            if θ forms a cyclic chain then
29              call reportCycles(θ, 0, an empty chain)
30            else
31              isTraversed (j) := true
32              call DFS_Traverse(i, S, τ')
33              isTraversed (j) := false
34            end if
35          end if
36        end for
37      end if
38    end for
39    pop τ from S
40  end Function
41  Function findEquDepGroup(chain D, dependency d)
42    for each d_i in D do
43      if d_i is equivalent to d then
44        return Group(d_i)
45      end if
46    end for
47    return ∅
48  end Function
49  Function reportCycles(Cycle θ, Size s, Chain equCycle)
50    if s = sizeof(θ) then
51      report equCycle as a potential deadlock!
52    else
53      for each dependency d in Group(θ [s+1]) do
54        equCycle := equCycle + {d}  //concatenation of chains
55        reportCycles(θ, s+1, equCycle)
56        equCycle := equCycle - {d}  //subtraction of chains
57      end for
58    end if
59  end Function
```

The basic idea of our equivalence reduction strategy is as follows: for each thread specific partition, we put all lock dependencies equivalent to one another into the same group. During cycle detection, we only select one lock dependency from each group as the *representative* case to stand for the entire group to be searched for cycles. When a cycle is located, we report all inferred cycles by substituting the each representative case by each dependency in the same group. Otherwise, if the representative case does not appear in any cycle, all other dependencies in the same group need not be searched.

In the above example, there is one group $\{\tau_1, \tau_2\}$ in σ_{t1} and two groups $\{\tau_3\}$ and $\{\tau_4\}$ in σ_{t2}. On cycle detection, there are only two combinations to be considered: $\langle \tau_1, \tau_3\rangle$ and $\langle \tau_1, \tau_4\rangle$ if we select τ_1 from the σ_{t1} (alternatively, $\langle \tau_2, \tau_3\rangle$ and $\langle \tau_2, \tau_4\rangle$ if we select τ_2 from σ_{t1}). As such, we locate one cycle $\langle \tau_1, \tau_3\rangle$, and then infer another cycle $\langle \tau_2, \tau_3\rangle$ because τ_2 is equivalent to τ_1, and τ_3 is in a singleton group. As a result, two cycles are reported. Moreover, τ_2 has not been used in searching at all.

4.6 Algorithm: Cycle Detection Algorithm

A relation D produced by Algorithm 1 may contain disjoint components (which corresponds to disjoint subgraphs in a lock order graph). Instead of simply searching cycles on the whole relation D, *Magiclock* splits D into a set of disjoint components, and then searches cycles on each of these components. We use the *edgesFromTo* information in Algorithm 2 to split the chains into disjoint components such that the dependencies of each cycle must retain in the same disjoint component. Algorithm 4 shows the disjoint component finder algorithm. It is adopted from the well-known *Tarjan Graph* algorithm [43] to find strongly connected components, except that Algorithm 4 iterates on each edge in the maintained *edgesFromTo* data structure (see Algorithms 1 and 2).

Algorithm 5 shows the cycle detection algorithm of *Magiclock* on each disjoint component. To ease our presentation, let i to be the thread id of t_i, for $0 \le i \le k$.

Lines 2–12 in *CycleDetection* show the lock dependency partitioning and equivalent dependencies reduction process on each disjoint component (*dc*) reported by Algorithm 4. *Magiclock* only needs to examine the lock dependencies having locks in the *cyclic-set* and with non-empty lockset at line 4. Before adding a lock dependency τ into a partition D_i, a checking on equivalent lock dependency is performed through the function *findEquDepGroup*. If this lock dependency is equivalent to another lock dependency that has already been added into the partition D_i, the group g associated with this lock dependency τ is returned and the lock dependency τ is added to the group (line 6). Otherwise, the lock dependency τ is added into the current partition D_i, and a new empty group is associated with this lock dependency τ (lines 8–9).

Next, *CycleDetection* searches for cycles on each disjoint component. The array *isTraversed*(i) tracks whether the thread t_i has already been included on the current path. *CycleDetection* iteratively (at lines 17 and 32) searches lock dependency chains as follows: on visiting the partition D_i, it restricts itself to further explore other partitions D_j for $i+1 \le j \le k$, where k is the number of threads in D (line 22), skipping those visited (line 23) in its depth first search.

Note that the first parameter of *DFS_Traverse* (i.e., *i* at line 20) is in an increasing order whenever it is called (lines 14 and 16), which determines each reported cycle always starts with a thread with lowest thread ID. Also, at line 32, the first parameter of *DFS_Traverse* is the same as that at line 20. If a cycle θ is detected, the cycle θ and all cycles equivalent to this one are reported by *reportCycles*. For all dependencies in the input cycle θ (lines 49–59), *reportCycles* substitutes them by every possible combination of equivalent lock dependencies in their corresponding groups, and reports each substituted cycle.

4.7 Correctness Proof of Lock Classification

In this section, we present the theorem to show that Algorithm 3 correctly classifies all irremovable locks on the given relation *D* into the *Cycle-set*.

Lemma 1. Given a lock dependency relation *D* and a set of removable locks *K* on *D*, if a lock *n* on a new lock dependency relation *D*' is a removable lock, where *D*' is constructed from *D* by deleting all locks in *K* and all edges from or to any lock in *K*, then the lock *n* is also a removable lock in *D*.

Proof. Suppose that the lock *n* is an irremovable lock in *D*. By definition, there must exist a cycle *c* that contains the lock *n*. Let's denote all locks in the cycle *c* as a set L_c and the edges on the cycle *c* by E_c which are only from the locks in L_c and to the locks in L_c. As any lock in a cycle is an irremovable lock by definition, all locks in L_c are irremovable locks in *D*. Therefore, L_c and *K* do not share any lock (i.e., $L_c \cap K = \varnothing$). Hence, all locks in L_c must appear in *D*'. Besides, all edges E_c also appear in *D*' as each of them links two locks in L_c only. Therefore, the cycle *c* (i.e., all locks in L_c and all edges in E_c) also appears in *D*'. By definition, any locks in *c* should be an irremovable lock. So, the lock *n* in the cycle *c* is also an irremovable lock in *D*', which contradicts the given condition that the lock *n* is a removable lock in *D*'. Hence, the lock *n* is a removable lock in *D*. □

Lemma 2. Given a lock dependency relation *D* and a removable lock *k* in it. If the indegree of a lock *n* is the same as *edgesFromTo*(*k*, *n*) or the outdegree of a lock *n* is the same as *edgesFromTo*(*n*, *k*), then *n* is removable.

Proof. Consider the case that the lock *n* only associates with edges incoming from the lock *k*. Suppose that the lock *n* appears in a cycle (say c_n), then the lock *k* must also appear in the cycle c_n because the lock *n* has no other incoming edge except edges from the lock *k*. This contradicts the given condition that *k* is not in any cycle as *k* is a removable lock. Hence, *n* is a removable node. Similarly, if *n* only associates with edges outgoing to *k*, the lock *n* is also a removable lock. □

Lemma 3. After executing Algorithm 3, every lock *m* in *independent-set* or *intermediate-set* is removable.

Proof. When a lock *m* is added into *independent-set* (at line 4) or *intermediate-set* (at line 7), either the indegree or the outdegree of the lock *m* is zero according to the corresponding conditions at lines 3 and 6, respectively. So, at least one original value of the indegree and outdegree of *m* is zero. If one of the original values of the indegree and outdegree of *m* is zero, the lock *m* has been classified into the *independent-set* at line 4 or the *intermediate-set* at line 7 because *m* has no dependency with other locks and hence it cannot be a part of any cyclic chain. By definition, *m* is a removable lock. □

Lemma 4. If a lock *m* ever appears in the stack *S* (at lines 8, 20, and 31) in Algorithm 3, then *m* is removable.

Proof. We prove Lemma 4 by mathematical induction on the order of locks pushed into the stack *S*. We firstly prove the base case: when the first lock is pushed into *S*, it is a removable lock. It is because the first lock must be pushed into *S* at line 8 (otherwise, the stack *S* is empty at line 12 and no lock is pushed into the stack *S*) and this lock also belongs to *intermediate-set* which contains removable locks only by Lemma 3. The base case is proved. Now, suppose that the first *q* lock(s) pushed into the stack *S* are removable locks. Consider the (*q*+1)-*th* lock (denoted by the lock *n′*) pushed into *S*. According to Algorithm 3, the only changes are that: the indegree or the outdegree of the node *n′* are reduced by the number of edges from the lock *m* or to the lock *m* at lines 16 or 27. Because the lock *m* is a removable lock, by Lemma 1, the lock *n*', which is the (*q*+1)-th lock pushed into the stack *S*, is also a removable lock. By mathematical induction, the result follows. □

Lemma 5. Given a lock dependency relation *D,* after applying Algorithm 3 on *D*, all the locks in the sets *independent-set, intermediate-set,* and *inner-set* are removable locks.

Proof. Suppose that the lock *m* is in *independent-set* ∪ *intermediate-set*. By Lemma 3, *m* will not appear in any cycle. Similarly, suppose that *m* is in *inner-set*. Because all these locks are pushed into *S* by Algorithm 3 (at lines 8, 20, and 31), by Lemma 4, the lock *m* is removable. □

Theorem 1. Given a lock dependency relation *D,* after applying Algorithm 3 on *D,* all locks that are parts of any cycle in *D* are in the set *cyclic-set*.

Proof. By Lemma 5, no lock that is not removable is classified into sets *independent-set, intermediate-set,* and *inner-set*. By lines 38–40 of Algorithm 3, the result follows. □

5 Experiment

5.1 Implementation and Benchmarks

Implementation. We implemented our tool for C/C++ programs with Pthread and used *Pin* tool 2.9 [35] in Probe mode. For each event, a location is also generated as needed by *MulticoreSDK*. To compare with our tool, we also faithfully implemented *iGoodlock* [29] and *MulticoreSDK* [36] based on their papers and downloadable artifacts because their original tools are either unavailable or can handle Java programs only. We used the abstraction algorithm presented in [22] to compute the site information for each lock dependency.

Benchmarks. We have selected a set of 11 large-scale, real-world, and open-source multithreaded benchmarks. Six of them have, in total, 11 real deadlock cycles. They

TABLE 2
DESCRIPTIVE STATISTICS OF BENCHMARKS

Benchmark	Version	Bug ID	SLOC (k)	# of threads / locks	Trace size: *File size (MB)*	Trace size: *# of a/r events*	Test input and/or deadlock descriptions
HawkNL	1.6b3	n/a	9,300	3 / 8	0.84K	28	`nlshutdown`() and `nlclose`()
SQLite	3.3.3	1672	74.0	5 / 3	0.0467	1,920	4 working-threads, each sends 10 queries `sqlite3UnixEnterMutex`() and `sqlite3UnixLeaveMutex`()
MySQL 1	5.5.17	62614	1,282.7	21 / 55,297	12.1	444,846	`PUGE BINARY LOG` acquires locks in the wrong order.
MySQL 2	5.1.57	60682	1,146.7	26 / 33,458	13.1	480,790	`SHOW INNODB STATUS` deadlocks when `LOCK_thd_data` points to `LOCK open`
MySQL 3	6.0.4	37080	1,093.6	17 / 231	0.478	15,860	`alter` on a temporary table and a non-temporary table using falcon engine
MySQL 4	6.0.4	34567	1,093.6	17 / 367	0.783	25,918	`insert` and `truncate` on a same table using falcon engine
Chromium	24.0	n/a	8,397.0	66 / 35,117	270.9	9,396,752	Open 9 pages: 3 "`cnn.com`", 3 "`bbc.com`", 3 "`sohu.com`" (>14.8MB)
Firefox	3.0	n/a	2,601.3	52 / 8,726	975.1	33,321,814	Open 9 pages: 3 "`cnn.com`", 3 "`bbc.com`", 3 "`sohu.com`" (>14.8MB)
OpenOffice	3.2	n/a	5,445.8	10 / 9,629	133.0	4,524,654	Open a Doc file (paper draft) with 226.5KB.
Evolution	2.28.3	n/a	420.4	84 / 1,903	445.6	15,271,570	Fetch 212 e-mails from a Gmail.com account
Thunderbird	3.0.1	n/a	3,039.2	17 / 2,497	373.6	13,110,044	Fetch 212 e-mails from a Gmail.com account

Note: ***a/r events*** *refer to acquisition and release events; n/a means no bug ID available.*

allow us to validate *Magiclock* on scenarios with and without potential deadlocks. The set of benchmarks includes `HawkNL` [4], `SQLite` [10], three versions of `MySQL` [5] (one version has been used on two different test cases as `MySQL 3` and `MySQL 4`), `Chromium` [1], `Firefox` [3], `Open Office` [8], `Evolution` [2], and `Thunderbird` [12]. For `HawkNL`, we used the test case from [30]. `SQLite` is an embedded database program and we wrote a test harness program with 4 client threads to concurrently send SQL queries to it. For all versions of `MySQL`, we used the test cases based on their bug reports [10]. For `Firefox` and `Chromium`, we opened 9 web pages, which have a total size of more than 14.8MB. For `OpenOffice`, we opened a WinWord *.doc file with a size of 226.5KB, containing text, tables, and figures (which actually is a paper draft). For `Thunderbird` and `Evolution`, we configured them to fetch all emails from a Gmail.com account (212 emails in total). The details of test inputs and/or the bug descriptions are shown in the last column of TABLE 2.

To further evaluate the scalability of *Magiclock*, we conducted an additional experiment on `MySQL` by using a system testing tool SysBench [11] to send different SQL queries to `MySQL`. We configured SysBench to produce scenarios of (1) increasing number of requests sent by each thread with a fixed number of threads and (2) increasing number of threads that each sends a fixed number of requests, respectively.

We performed the experiment on a 32-bit Ubuntu Linux 10.04 with four 2.80GHz processors and 3.9GB physical memory.

In the reset of Section 5, we firstly present the generated traces and monitoring overhead of our tool and then give the data analysis in Section 5.3. In Section 5.4, we present the scalability result of *Magiclock* on `MySQL`.

5.2 Traces and Monitoring Overhead

TABLE 2 shows the descriptive statistics of the benchmarks that evaluated *Magiclock*, including the name, version, Bug ID, and code size (SLOC [9]) of each benchmark. The fifth columns show the number of threads and the number of locks, respectively. The next two columns show the execution trace file size and the number of lock acquisitions and releases in a trace. The last column shows the test input the can lead the occurrence of deadlocks and/or the brief description of deadlocks in each benchmark.

Fig. 7 summarizes the time spent on deadlock detection by each component to generate traces (as described in TABLE 2): On each benchmark, we collected the time of native run, Pin base time, instrumentation time, event monitoring time, abstraction computation time, time spent by *Magiclock* (taken from TABLE 3), and time spent by deadlock confirmation. We then computed the percentage of each of these components out of the total time spent. Fig. 7 shows that the trace collection overhead by *Magiclock* is not heavy (less than 4.4% without considering confirmation run). With the introduction of *Magiclock*, the time spent on abstraction computation now becomes noticeable in the process of deadlock detection.

5.3 Data Analysis

5.3.1 *Result Summary*

TABLE 3 summarizes the overall comparisons among *iGoodlock*, *MulticoreSDK* (denoted as `MSDK`), and *Magiclock* in aspects of the memory footprint in Megabytes (MB) (or GB for Gigabytes), the time cost in second (*s*) (or *m* for minutes, and *h* for hours), and the number of unique cycles reported. The last two columns show the number of real deadlock cycles (confirmed by the latest *MagicScheduler* [13], [21]) among the detected cycles and the number of threads in the reported cycles. Due to the out of memory error of *iGoodlock*, we cannot collect its data in full. We mark these cells with ">" indicating that the data in the cell is just the value before the tool has exhausted all the memory (and hence cannot complete) or timed out. We also use this marker in TABLE 4 for the same purpose.

From TABLE 3, we observe that, on `HawkNL` and `SQLite`, the three techniques performed similarly in memory and time consumption, which is not surprising

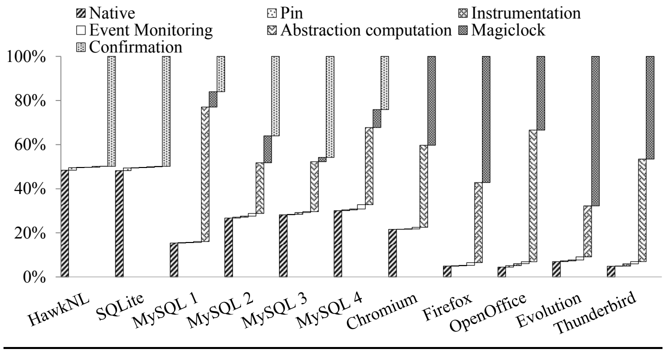


Fig. 7. Descriptive statistics of overhead on trace collection.

TABLE 3
MEMORY AND TIME COMPARISONS AMONG *IGOODLOCK*, *MULTICORESDK*, AND *MAGICLOCK*

Benchmark	Memory (MB)			Time (second)			# of unique cycles			# of real deadlocks cycles	# of threads in cycles
	iGoodlock	*MSDK*	*Magiclock*	*iGoodlock*	*MSDK*	*Magiclock*	*iGoodlock*	*MSDK*	*Magiclock*		
HawkNL	1.098	1.125	1.195	0.001	0.001	0.001	2	2	2	1	2
SQLite	1.367	1.348	1.277	0.016	0.012	0.008	1	1	1	1	2
MySQL 1	12.086	29.633	2.336	0.432	1.408	0.680	2	2	2	2	2
MySQL 2	>2.8G	1.347	2.375	>55	>10h	0.876	>4	>0	4	2	2, 3
MySQL 3	>2.8G	2.0G	2.164	>480.8	>10h	0.112	>10	>0	12	1	2, 3, 4
MySQL 4	>2.8G	2.0G	3.016	>168.6	>10h	0.456	>9	>0	17	4	2, 3, 4
Chromium	>2.8G	601.6	3.953	>268m	>10h	17.809	>0	>0	1	unconfirmed	2
Firefox	>2.8G	2.0G	2.606	>163m	>10h	51.739	>0	>0	0	-	-
OpenOffice	>2.8G	987.4	43.742	>10h	>10h	16.085	>0	>0	0	-	-
Evolution	>2.8G	919.5	1.414	>191m	>10h	22.269	>0	>0	0	-	-
Thunderbird	>2.8G	835.0	2.160	>512m	>10h	19.053	>0	>0	0	-	-

***MSDK** refers to MulticoreSDK; "**>**" means the data is collected before the tool has exhausted all available memory or has timed out (10 hours);* "-" *means no data collected.*

as TABLE 2 shows that their execution traces are quite small. None suffered from the scalability problem. They also reported the same number of cycle. On `MySQL 1`, the three techniques also reported the same number of cycles; *iGoodlock* and *MSDK* consumed much more memory than *Magiclock*; however, *MulticoreSDK* and *Magiclock* consumed more time than *iGoodlock*. We have analyzed this case and found that, though the trace sizes (and the number of threads and locks) are relative large, most of lock acquisitions were not nested in any other lock acquisition (or dependencies with empty lockset). Readers may refer to TABLE 4 (the columns "`# of chains by iGoodlock`", "`# of locks`", and "`# of edges`") and Section 5.3.2 for more detailed analysis. Therefore, *iGoodlock* was able to directly and quickly finish its searching; but *MulticoreSDK* and *Magiclock* both had to filter locks that were not on any cycle. As a result, the filtering strategies in *MulticoreSDK* and *Magiclock* have led them to consume more analysis time.

Except on `HawkNL`, `SQLite`, and `MySQL 1`, *iGoodlock* consumed the most memory, and ran out of memory when analyzing the execution traces of `MySQL 2-4`, `Chromium`, `Firefox`, `OpenOffice`, `Evolution`, and `Thunderbird`. This result is consistent with what the authors stated in their paper [29] that *iGoodlock* consumed more memory than the traditional techniques. *MulticoreSDK* consumed up to hundreds of MB memory or even up to 2.0 GB memory. *Magiclock* consumed the least memory on all benchmarks except on `HawkNL`, `SQLite`, and `MySQL 2`.

The three techniques took less than 1 second on `HawkNL`, `SQLite`, or `MySQL 1`. On all other benchmarks, *MulticoreSDK* did not finish within our time limit of 10 hours. *iGoodlock* had exhausted all the available memory before completing its analyses.

On the reported numbers of cycles, the three techniques reported the same number of cycles on the first three programs (`HawkNL`, `SQLite`, and `MySQL 1`). On the next four programs (`MySQL 2-4` and `Chromium`), *MulticoreSDK* reported no cycle, and *iGoodlock* can report some but not all these cycles. Because *iGoodlock* did not finish its third iteration, it was unable to report any cycle with 4 threads on `MySQL 3-4` due to its memory-consuming search strategy. *Magiclock* was able to finish its search and reported 4, 12, 17, 1 cycle, respectively. On the remaining ones (`Firefox`, `OpenOffice`, `Evolution`, and `Thunderbird`), all three techniques did not report any cycle.

In summary, in terms of memory and time consumptions, *Magiclock* is more scalable than *iGoodlock* and *MulticoreSDK* in our experiment. Besides, the effectiveness of *iGoodlock* and *MulticoreSDK* may be compromised by their inefficiency being unable to analyze the whole given execution trace in time (e.g., on `MySQL 2-4` and `Chromium` in TABLE 3).

5.3.2 *Comparing iGoodlock with Magiclock*

We compared the number of chains produced by *iGoodlock* and *Magiclock* as shown in the second and the third main columns in TABLE 4(a). *iGoodlock* uses an iterative algorithm to find all cycles that has to store all intermediate results [29], TABLE 4(a) thus shows the intermediate results on each benchmark produced by *iGoodlock* (denoted by DF^0 which is the initial set of chains produced, and DF^x ($x \geq 1$) which is the number of chains produced by the *x-th* iteration). If there is no need to iterate, we mark the

TABLE 4
COMPARISONS BETWEEN *iGoodlock* AND *Magiclock AND* BETWEEN *MulticoreSDK* AND *Magiclock*

	(a) *Magiclock* vs. *iGoodlock*					(b) *Magiclock* vs. *MulticoreSDK*					
Benchmark	*Magiclock*	# of chains by *iGoodlock*				# of locks			# of edges		
		DF^0	DF^1	DF^2	$\sum DF^i (i \geq 3)$	*Total*	*MSDK*	*Magiclock*	*Total*	*MSDK*	*Magiclock*
HawkNL	6	6	-	-	-	8	5	3	9	9	9
SQLite	6	290	-	-	-	3	2	2	290	290	6
MySQL 1	24	562	-	-	-	55,297	203	6	682	303	29
MySQL 2	180	45,492	>5,054,890	-	-	33,458	908	5	82,626	8,037	363
MySQL 3	485	7,890	34,668	158,277	>964,101	231	230	39	46,347	46,157	2,494
MySQL 4	1,466	12,846	298,415	>4,440,382	-	367	366	97	65,062	64,761	6,536
Chromium	53	697,953	>3,360,901	-	-	35,117	33,140	2	1,088,229	802,242	193
Firefox	250	3,246,061	>574	-	-	8,726	7,171	32	6,684,710	2,524,145	882
OpenOffice	561	2,240,282	>2,061,186	-	-	9,629	7,632	2	13,662,579	11,681,875	3,793
Evolution	13	507,186	>3,663,748	-	-	1,903	954	5	1,085,958	771,464	24
Thunderbird	93	375,042	1,577,064	>2,432,715	-	2,497	2,255	23	607,711	596,783	207

***MSDK** refers to MulticoreSDK; "**>**" means no data collected in the cell due to crash error; "–" means that the cycle detection has terminated on the previous iterations (not marked with a "–").*

TABLE 5
IMPROVEMENT OF MAGICLOCK

Benchmark	Memory(MB)		Time (second)		# of chains		# of edges		# of locks	
	ML1	*Magiclock*	*ML1*	*Magiclock*	*ML1*	*Magiclock*	*ML1*	*Magiclock*	*ML1*	*Magiclock*
HawkNL	1.216	1.195	0.001	0.001	6	6	9	9	3	3
SQLite	1.277	1.254	0.012	0.008	290	6	290	6	2	2
MySQL 1	2.266	2.336	0.884	0.680	187	24	194	29	6	6
MySQL 2	12.535	2.375	2.992	0.876	2,737	180	4,844	363	5	5
MySQL 3	2.453	2.164	4.708	0.112	4,501	485	25,673	2,494	119	39
MySQL 4	3.324	3.016	284.478	0.456	8,444	1,466	40,503	6,536	195	97
Chromium	9.402	3.953	77.505	17.809	6,348	53	11,210	193	5	2
Firefox	92.723	2.606	1,887.302	51.739	580,241	250	589,222	882	63	32
OpenOffice	260.027	43.742	37.274	16.085	881,677	561	5,924,278	3,793	1,222	2
Evolution	4.481	1.414	147.909	22.269	17,302	13	41,465	24	7	5
Thunderbird	2.262	2.160	20.137	19.053	3,156	93	594	207	29	23

corresponding cell with the marker '–'.

From TABLE 4(a), we observe that, except on `HawkNL`, `SQLite`, and `MySQL 1`, *iGoodlock* produced quite many chains at either its initial iteration (DF^0) or the later iterations; whereas, *Magiclock* produced much fewer chains. In particular, on `Evolution`, *iGoodlock* initially produced nearly 39,000 times more chains than that produced by *Magiclock*. The result of the first iteration is shown in column DF^1. Compared to the number of chains in its initial results (DF^0), *iGoodlock* produced quite many chains during its iterations. Moreover, *iGoodlock* had exhausted all the available memory either in the second (DF^2) or in the later (DF^i, $i \geq 3$) iterations on all benchmarks except on `HawkNL`, `SQLite`, and `MySQL 1`.

5.3.3 *Comparing MulticoreSDK with Magiclock*

TABLE 4(b) shows the comparisons between *MulticoreSDK* and *Magiclock* in terms of the numbers of nodes and edges on lock order graph. The second main column shows the sizes of the lock order graph constructed by a traditional graph, by *MulticoreSDK*, and by *Magiclock*, respectively. Note that the first sub-column (`Total`) is the same as the total number of locks because each lock corresponds to a node in a lock order graph. The columns on the right show the numbers of edges produced by a traditional graph, by *MulticoreSDK* (denoted by `MSDK`), and by *Magiclock*, respectively. To facilitate a fair comparison, on counting the number of edges for *Magiclock*, we have converted each lock dependency to a set of edges. For example, a lock dependency $\langle t, m, \{l_1, l_2\}\rangle$ corresponds to two edges (i.e., edges from l_1 to m and from l_2 to m) in a lock order graph.

TABLE 4(b) shows that *MulticoreSDK* only pruned small numbers of nodes and edges except on `HawkNL`, `SQLite`, and `MySQL 1-2`. On these four benchmarks, both *MulticoreSDK* and *Magiclock* pruned many locks, but *Magiclock* simply pruned much more. On the remaining benchmarks, *MulticoreSDK* pruned fewer locks and edges than *Magiclock*. On `Evolution`, *MulticoreSDK* pruned nearly 50% nodes and nearly 30% edges; whereas, *Magiclock* pruned more than 99% nodes and more than 99% edges.

5.3.4 *Improvement of Magiclock*

TABLE 5 shows the comparisons between *ML1* and *Magiclock* in terms of memory consumption, time consumption, # of chains generated, # of locks and # of edges (in the sense of lock order graph) used in cycle detection.

From TABLE 5, we observe a significant improvement made by *Magiclock*. The memory consumption, except on `MySQL 1`, is reduced. On `MySQL 1`, *Magiclock* consumed slightly more memory than *ML1*. Overall, the introduction of new strategy of *Magiclock* did not increase the memory consumption much. For the time consumption, the improvement is significant, especially on the last six benchmarks. *Magiclock* never explores more edges or nodes than *ML1*, and often explores fewer.

5.4 Further Evaluation on MySQL

This section reports a further validation on the scalability of *Magiclock* on `MySQL 1` benchmark. We selected the `MySQL` benchmark because it is a widely-used large-scale server program and its traces are sufficiently large to stress the scalability of the deadlock detection. Besides, on `MySQL`, there are deadlocks that have only been reported by conducting "high concurrency test" (e.g., 200 concurrent connections can discover a real deadlock [6]).

The *SysBench* tool [11] is a widely used automated tool for testing the performance of operating systems and database servers including `MySQL`. We used *SysBench* to send inputs (SQL queries) to `MySQL`. We configured *SysBench* to send requests by increasing the number of *requests* from each client thread as well as the total number of client *threads*, respectively. For each configuration, we collected the time spent, the memory consumption, and the number of dependencies needed to complete the search.

5.4.1 *Scalability with Increasing Number of Requests by Each Thread*

We used *SysBench* with a fixed 16 client threads (which is a default value) to send requests to `MySQL` and increased the requests sent by each thread from 1,000 to 10,000 with step 1,000. The result is summarized in Fig. 8(a)-(c).

Fig. 8(a) shows the time spent by *ML1* and *Magiclock* to search for cycles with respect to different numbers of requests per client thread. We observe that, with increasing number of requests sent per client thread, the time consumed by *ML1* increased significantly and exponentially. It grows from 9.8 seconds to 48008.6 seconds when the number of requests per client thread grows from 1,000 to 10,000. Moreover, *Magiclock* consumes much less time in each case, and only grows quite moderately as the number of requests per client thread increases. It grows from 12.9 seconds to 109.2 seconds when the number of requests per client thread grows from 1,000 to 10,000.

Both *ML1* and *Magiclock* require searching for dependency chains, which the amount of chains to be searched

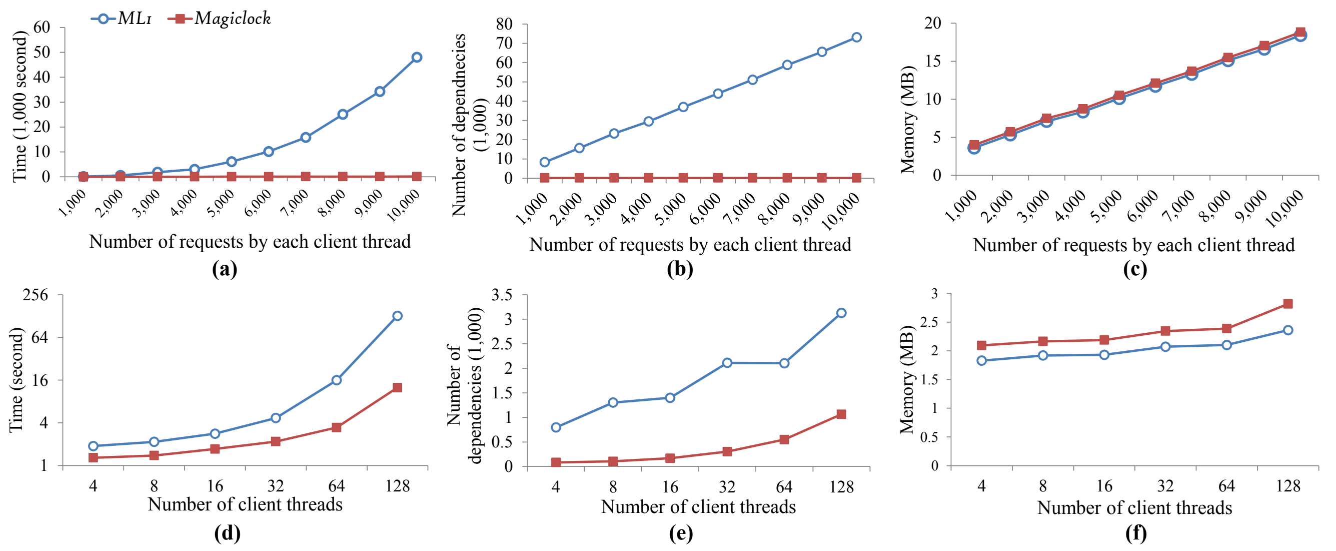


Fig. 8. Scalability Comparisons of *ML1* and *Magiclock* on `MySQL`

grows exponentially as the number of lock dependency increases. Fig. 8(b) shows that with a linear increasing number of requests per thread, for *ML1*, the number of dependencies considered by the cycle detection algorithm, almost increased at the same rate. Fig. 8(b) also shows that the use of the equivalent lock dependencies in *Magiclock* effectively alleviates the growth in the number of lock dependencies to be searched for cycles.

Fig. 8(c) shows the maximum memory consumptions by *ML1* and *Magiclock* with increasing number of requests per client thread. The figure shows that the memory footprints of the two tools were close to each other. Both increased quite linearly when the numbers of requests per client thread increase linearly.

5.4.2 *Scalability with Increasing Number of Threads*

We increased the total number of *SysBench* threads to send requests (100 per-thread) to `MySQL`. The numbers of client threads used were 4, 8, 16, 32, 64, and 128.[1] The summary of the results is shown in Fig. 8(d), (e), and (f).

Fig. 8(d) shows the time consumed by *ML1* and *Magiclock*. Although, the time consumption by both *ML1* and *Magiclock* increased exponentially, the increase by *Magiclock* was much slower. For instance, when there were only 4 client threads, *ML1* and *Magiclock* consumed 1.9 and 1.3 seconds, respectively. Whereas, when there were 128 client threads, *ML1* and *Magiclock* consumed 129.0 and 12.5 seconds, respectively.

Magiclock has not explored how to determine whether two lock dependencies across different threads refer to same usage in the program (where we do not know the program semantics). As such, the equivalent lock dependency strategy used in *Magiclock* does not help to alleviate the scalability issue in this dimension. On the other hand, Fig. 8(e) shows that the number dependencies searched by *Magiclock* increased much slower than *ML1*. It indicates that *Magiclock* is able to reduce the number of dependencies for the same thread by using the equivalent lock dependency strategy.

Fig. 8(f) shows that both *ML1* and *Magiclock* consumed fairly small amounts of memory as the number of client threads grew. The grow trend of either tool is quite moderate. *Magiclock* consumed more memory than *ML1*. However, in each case, the difference was less than 0.5MB. This is the memory needed to keep additional data structures for the equivalency information among lock dependencies used by Algorithm 5.

5.5 Summary

In summary, in the experiment, *Magiclock* shows a significantly improvement over *ML1* and can be significantly more scalable than *iGoodlock* and *MulticoreSDK* to analyze execution traces of large-scale benchmarks[2].

Compared to *ML1*, *Magiclock* can reduce the time used to search for cycles without significantly compromising other studied aspects (e.g., memory consumption). Compared to *iGoodlock* and *MulticoreSDK*, *Magiclock* can be more effective to deal with large-scale benchmarks.

The experiment also shows that *iGoodlock* may consume less memory than *Magiclock* on analyzing small-scale traces such as the trace for `HawkNL`, which consists of 28 lock acquisitions and releases only. In this case, there is no need to apply the innovation made by *Magiclock* to address the scalable challenge.

6 Related Work

Concurrency bugs are difficult to find and reproduce. To detect them, the process can be both time- and memory-consuming, prohibiting detection techniques to scale up to handle real-world large-scale multithreaded programs.

[1] We have found out that the numbers of `MySQL` threads for the corresponding *SysBench* configuration were 20, 24, 32, 48, 80, and 144. That is, `MySQL` needed a baseline of 16 threads. We cannot set to use more threads (e.g., 256 or more) that exceeds the "default setting" of `MySQL`. Once we changed the default setting, errors had occurred due to lost connection, and we cannot to collect the corresponding execution traces to complete the experiment.

[2] Originally, we have attempted to compare *ML1* and *Magiclock* on the benchmarks larger than `MySQL` 1 for Section 5.4. As shown by Fig 5(a), the time spent by *ML1* has already exceeded 48,000 seconds. TABLE 3 shows that the time needed to complete the analysis by *ML1* on each larger benchmark is much larger than that on `MySQL` 1. As such, we did not further compare the two techniques.

PCT [17] and *PPCT* [39] are randomized schedulers with a probabilistic guarantee to find concurrency bugs. Unlike *Magiclock*, they do not require deadlock detection by analyzing the program execution trace beforehand. However, its theoretically guaranteed probability and the actual values observed from their experiments are low.

Happened-before based predictive data race detectors [18], [20], [25], [37] have been proposed. Recently, researchers mainly focused on the efficiency of the on-the-fly detection approach. By the introduction of *epoch* [25], memory access sampling [18], [37], [46], and the redundancy vector clock elimination due to synchronization events [19], the efficiency of such detectors have been improved. *Magiclock* is also a predictive detector. It deals with the orders of locks in an execution. The epoch-based optimization is inapplicable because the use of epoch only optimizes the use of vector clock in the implementation of a happened-before based detector. Our deadlock prediction approach needs no vector clock implementation. Sampling approach is also inapplicable because if a particular lock dependency is not sampled, any cycle related to this lock dependency could not be detected, making the detector incomplete with respect to the monitored execution trace. The elimination of redundant vector clocks approach only removes identical entities that occur consecutively along the same thread in an execution, which cannot remove the edges between two locks.

Improving the efficiency often compromises other aspects [25], such as completeness, even though tolerable sometimes. For structured parallel languages [41], a scalable and precise dynamic algorithm has been proposed to reduce the memory consumption [41]. *Magiclock* is complete with respect to the monitored execution trace.

Deadlock detection techniques can be static or dynamic. We have compared our *Magiclock* with *iGoodlock* and *MulticoreSDK* extensively, and indirectly compared with *Goodlock* [26], which uses the traditional lock order graph for the detection of potential deadlock cycles.

Many static techniques [14], [23], [38], [41], [42], [45] analyze the source code and infer lock order graphs to find potential deadlock cycles. They have an advantage to apply for software that is not closed such as the Java library. These techniques however suffer from high false positives. For example, an early work [45] reports 1,000 potential deadlock cycles, but only 7 of them are real deadlocks. More recently, Naik et al. [38] combine a suite of static analysis techniques to reduce the false positive rates. However, problems like conditional variables and scalability are still the concerns on using static techniques to analyze large-scale applications. *Magiclock* is a dynamic approach to predict deadlock potentials by analyzing program execution traces, and it has the potential to handle large-scale real-world applications.

Joshi et al. [28] monitor the annotated conditional variables as well as lock synchronization and threading operations in a program to produce a trace program containing not only thread and lock operations but also the values of conditionals. Then they apply a model checker (Java Pathfinder) to check all *abstracted* and inferred execution paths of the trace program to detect both communication deadlocks and resource deadlocks. Their technique however suffers from needing manual effort to design and add annotations, which can be error-prone, and suffers from the scalability issue to handle large-scale programs. *Magiclock* has not been extended to deal with communication deadlocks, which is an interesting future work. Bensalem et al. [15], [16] use the happened-before relation to improve the precision of cycle detection, and use a guided scheduler to confirm deadlocks. Ur and colleagues [24], [39] propose *ConTest* that uses the *Goodlock* approach to identify cycles, and actively introduces noise to increase the probability of deadlock occurrence [24]. *Magiclock* works similar to *iGoodlock* to use lock dependency set to identify cycles. Moreover, as we have presented in this paper, *Magiclock* includes Algorithms 1–5 to address the scalability issue.

Deadlock Immunity [30] prevents the second occurrence of a deadlock by maintaining a database containing all patterns of occurred deadlocks and using online monitoring. Unlike *Magiclock*, it has no potential deadlock cycle detection component but detects deadlocks when they really occur. *Gadara* [44] inserts lock acquisitions at the gate position of statically detected deadlocks. It avoids deadlock occurrence at runtime whenever a statically detected deadlock is like to occur. *Magiclock* detects deadlock dynamically and has not extended to fix deadlocks.

Deadlock confirmation techniques take all reported cycles as their inputs, and attempt to generate thread schedules to trigger real deadlocks. Examples of these techniques include *MagicScheduler* [21] and *DeadlockFuzzer* [29]. The former takes a set of cycles, and schedules a program execution at the lock acquisitions sites specified in the cycles. The use of set of cycles is the improvement over the latter [29]. *Magiclock* uses equivalent lock dependencies to locate cycles to alleviate the runtime overhead in cycle detection. If *Magiclock* reports a located cycle, *Magiclock* also reports every possible cycle containing at least one lock dependency that is equivalent to a corresponding lock dependency of the located cycle. Note that different lock dependencies have their own information (e.g., sites) obtained from the execution trace for potential deadlock detection. Each of these inferred or located cycles can then be confirmed via a deadlock confirmation technique. One may further use cycle segmentation through happened-before relations [14] to eliminate more false positives before applying a confirmation technique.

7 Conclusion

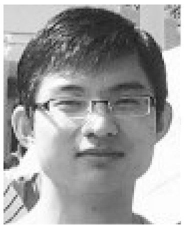
Existing dynamic potential deadlock detection techniques are not scalable enough to handle many real-world large-scale multithreaded programs. This paper has proposed *Magiclock*, a novel dynamic technique to detect potential deadlocks. It is particularly suitable to analyze traces on large-scale multithreaded programs. The experiment has validated that *Magiclock* can be highly efficient and scalable, and has the potential to tackle the challenges in handling large-scale real-world multithreaded programs. In future, we will study how to isolate false positives from all reported cycles because current techniques [21], [29] can only confirm real deadlocks. It is interesting to study more efficient abstraction computation algorithms.

ACKNOWLEDGMENT

We thank the editors and the anonymous reviewers for their invaluable comments and suggestions. This work is supported in part by the Early Career Scheme of the Research Grant Council of Hong Kong (project no.123512).

REFERENCES

[1] Chromium, http://code.google.com/chromium.
[2] Evolution, http://projects.gnome.org/evolution.
[3] Firefox, http://www.mozilla.org/firefox.
[4] HawkNL, http://hawksoft.com/hawknl.
[5] MySQL, http://www.mysql.com.
[6] MySQL bug, http://lists.mysql.com/mysql/209535.
[7] http://bugs.mysql.com/bug.php?id=36526.
[8] OpenOffice, http://www.openoffice.org.
[9] SLOCCount 2.26. http://www.dwheeler.com/sloccount.
[10] SQLite, http://www.sqlite.org.
[11] SysBench, http://sysbench.sourceforge.net.
[12] Thunderbird, http://www.mozilla.org/thunderbird.
[13] http://www.cs.cityu.edu.hk/~51948163/magicfuzzer.
[14] R. Agarwal, S. Bensalem, E. Farchi, K. Havelund, Y. Nir-Buchbinder, S. D. Stoller, S. Ur, and L. Wang. Detection of deadlock potentials in multithreaded programs. *IBM Journal of Research and Development*, Vol. 54 (5), Sep. 2010, 520–534,
[15] S. Bensalem and K. Havelund. Scalable dynamic deadlock analysis of multi-threaded programs. In PADTAD, 2005.
[16] S. Bensalem, J.C. Fernandez, K. Havelund, and L. Mounier. Confirmation of deadlock potential detected by runtime analysis. In Proc. PADTAD, 41−50, 2006.
[17] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In Proc. ASPLOS, 167–178, 2010.
[18] M. D. Bond, K.E. Coons and K.S. Mckinley. PACER: proportional detection of data races. In Proc. PLDI, 255–268, 2010.
[19] Y. Cai and W.K. Chan. Lock trace reduction for multithreaded programs. *IEEE Transactions on Parallel and Distributed Systems*, 24(12), 2407−2417, 2013.
[20] Y. Cai and W.K. Chan. LOFT: redundant synchronization event removal for data race detection. In Proc. ISSRE, 160–169, 2011.
[21] Y. Cai and W.K. Chan. MagicFuzzer: scalable deadlock detection for large-scale applications. In Proc. ICSE, 606−616, 2012.
[22] 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.
[23] J. Deshmukh, E. A. Emerson, and S. Sankaranarayanan. Symbolic deadlock analysis in concurrent libraries and their clients. In Proc. ASE, 480–491, 2009.
[24] E. Farchi, Y. Nir-Buchbinder, and S. Ur. A cross-run lock discipline checker for java. In PADTAD, 2005.
[25] C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. In Proc. PLDI, 121–133, 2009.
[26] K. Havelund, Using runtime analysis to guide model checking of java programs. In Proc. SPIN, 245–264, 2000.
[27] G. Jin, L.H, Song, W. Zhang, S. Lu, B. Liblit. Automated atomicity-violation fixing. In Proc. PLDI, 389–400, 2011.
[28] P. Joshi, M. Naik, K, Sen, and D. Gay. An effective dynamic analysis for detecting generalized deadlocks. In Proc. FSE, 327–336, 2010.
[29] 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.
[30] H. Jula, D. Tralamazza, C. Zamfir, and G.e Candea. Deadlock immunity: enabling systems to defend against deadlocks. In Proc. OSDI, 295–308, 2008.
[31] E. Knapp. Deadlock detection in distributed database systems. *ACM Computing Surveys*, 19(4):303−328, 1987.
[32] 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.
[33] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. of the ACM* 21(7):558–565, 1978.
[34] 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.
[35] 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.
[36] 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.
[37] D. Marino, M. Musuvathi, and S. Narayanasamy. LiteRace: effective sampling for lightweight data-race detection. In Proc. PLDI, 134–143, 2009.
[38] M. Naik, C.S. Park, K. Sen, and D. Gay. Effective static deadlock detection. In Proc. ICSE, 386–396, 2009.
[39] S. Nagarakatte, S. Burckhardt, M. M.K. Martin, M. Musuvathi. Multicore acceleration of priority-based schedulers for concurrency bug detection. In Proc. PLDI, 543–554, 2012.
[40] Y. Nir-Buchbinder, R. Tzoref, and S. Ur. Deadlocks: from exhibiting to healing. In Proc. RV, 104–118, 2008.
[41] R. Raman, J.S. Zhao, V. Sarkar, M. Vechev, and E. Yahav. Scalable and precise dynamic datarace detection for structured parallelism. In Proc. PLDI, 531–542, 2012.
[42] V.K. Shanbhag. Deadlock-detection in java-library using static-analysis. In Proc. APSEC, 361–368, 2008.
[43] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2): 146–160, 1972.
[44] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. Mahlke. Gadara: dynamic deadlock avoidance for multithreaded programs. In Proc. OSDI, 281–294, 2008.
[45] A. Williams, W. Thies, and M.D. Ernst. Static deadlock detection for java libraries. In Proc. ECOOP, 602–629, 2005.
[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.



Yan Cai is a PhD student at Department of Computer Science, City University of Hong Kong. He received his BEng degree in Computer Science and Technology from Shandong University, China in 2009. His current research interest is concurrency bug detection and reproduction in large-scale multithreaded and concurrent systems. His research results have been reported in venues such as ICSE, ISSRE, ICWS, SPE, JWSR, and TPDS.

W.K. Chan is an assistant professor at Department of Computer Science, City University of Hong Kong. He is an editorial board member of *Journal of Systems and Software*, and was guest co-editors of a few international software engineering journals, program co-chairs of AST 2010 and QSIC 2010, and innovative showcase chairs of ICWS and SCC for both 2009 and 2010. He is a program or review committee member of ICSE'15, ICSE'13 DS, and FSE'14. His research results have been reported in many venues including TOSEM, TSE, TPDS, TSC, CACM, COMPUTER, ICSE, FSE, ISSTA, ASE, ICDCS, and WWW. His current research interest includes program analysis and testing of large-scale software systems.