Trace Driven Dynamic Deadlock Detection and Reproduction

Malavika Samak  
Indian Institute of Science, Bangalore  
malavika@csa.iisc.ernet.in

Murali Krishna Ramanathan  
Indian Institute of Science, Bangalore  
muralikrishna@csa.iisc.ernet.in

Abstract
Dynamic analysis techniques have been proposed to detect potential deadlocks. Analyzing and comprehending each potential deadlock to determine whether the deadlock is feasible in a real execution requires significant programmer effort. Moreover, empirical evidence shows that existing analyses are quite imprecise. This imprecision of the analyses further void the manual effort invested in reasoning about non-existent defects.

In this paper, we address the problems of imprecision of existing analyses and the subsequent manual effort necessary to reason about deadlocks. We propose a novel approach for deadlock detection by designing a dynamic analysis that intelligently leverages execution traces. To reduce the manual effort, we replay the program by making the execution follow a schedule derived based on the observed trace. For a real deadlock, its feasibility is automatically verified if the replay causes the execution to deadlock.

We have implemented our approach as part of WOLF and have analyzed many large (upto 160KLoC) Java programs. Our experimental results show that we are able to identify 74% of the reported defects as true (or false) positives automatically leaving very few defects for manual analysis. The overhead of our approach is negligible making it a compelling tool for practical adoption.

Categories and Subject Descriptors  D.2.5 [Software Engineering]: Testing and Debugging; D.2.4 [Software Engineering]: Software/Program Verification

Keywords  deadlock detection; dynamic analysis; concurrency

1. Introduction

Traditional software testing to ensure the correctness of multi-threaded programs is inadequate due to the limitations associated with checking the correctness of the program across all possible interleavings [20]. This has resulted in the design of a number of program analysis techniques for detecting concurrency bugs [3, 8, 9, 14, 17, 22, 26, 29, 35]. The defects detected by dynamic analyses are highly likely to be real bugs because the analyses operate on real program execution data. Nevertheless, the analyses still suffer from false positives i.e., they report a defect when there is indeed no problem. Therefore, a programmer needs to analyze and comprehend a defect report to determine its correctness which can be time consuming. Practical experience [2, 19] shows that programmers adapt program analysis tools widely if the results are precise and actionable. Hence, the combination of imprecise analysis and the subsequent manual effort required to verify the correctness of the reported defects can lead to poor deployment of dynamic analysis tools, in spite of the sophisticated machinery employed to detect rare problems.

Deadlocks in multi-threaded programs are frustrating due to their ability to bring the system to a grinding halt in a non-deterministic fashion [16]. Existing state of the art dynamic analyses [3, 14] use cycle detection on a global lock graph to detect a deadlock. The nodes in the graph represent the lock instances. An edge, labelled \( t \), between any two nodes \( u \) and \( v \), represents the acquisition of lock \( v \) while holding lock \( u \) by thread \( t \). A cycle in the global lock graph is considered a potential deadlock if the edge labels in the cycle are unique.

Even though cycles in the lock graph point to potential deadlocks, these deadlocks may never manifest in a real execution. For example, if two threads \( t_1 \) and \( t_2 \) acquire locks \( \ell_1 \) and \( \ell_2 \) in opposite order respectively, then the lock graph contains two edges – an edge from \( \ell_1 \) to \( \ell_2 \) and an edge from \( \ell_2 \) to \( \ell_1 \). Observe that a deadlock is only possible if the acquisitions by \( t_1 \) and \( t_2 \) overlap in some schedule. Because existing detectors do not consider the complete history prior to the relevant lock acquisitions, they report any cycle as a deadlock, even though the lock acquisitions by \( t_1 \) and \( t_2 \) never overlap. Identifying overlaps can be challenging as the corresponding threads may not communicate directly with each other. Even if there is an overlap, the detected dead-
lock may still not be real for a variety of reasons including infeasible interleavings.

As we mentioned above, significant programmer effort is also necessary to comprehend the defects reported by these approaches. Therefore, to address this problem, apart from detecting potential deadlocks, DeadlockFuzzer [14] goes a step further and attempts to reproduce the deadlocks in a real execution. For each potential deadlock, the program is re-executed by fuzzing the schedule appropriately to make the execution deadlock. If the re-execution deadlocks, then the deadlock is considered a real defect. Otherwise, the classification of the deadlock is unknown. While a necessary first step in reducing the tedious process of comprehending the reported defects, their approach manages to automatically reproduce only 8% of the reported defects [14].

A detected deadlock can be easily classified as real (or true positive) in the presence of just one deadlocking re-execution. Unlike true positives, for a detected deadlock to be classified as a false positive, a re-execution of the program with any schedule should not cause the execution to deadlock. Unfortunately, this goes back to the original problem of exhaustive search of the interleavings. Moreover, even for real defects, because the trace data generated during the detection process is not leveraged, random schedules are generated to cause a deadlocking execution. This strategy may be ineffective as it may not result in the intended deadlock. This ineffective exploration of schedules associated with a deadlock, along with the presence of a large number of false positives, leaves a significant number of unknown defects to be comprehended manually.

In this paper, we address both the problems: (a) imprecision of existing dynamic deadlock detection approaches and (b) poor automatic reproducibility of a potential deadlock. We design a novel, scalable and more precise dynamic deadlock detection approach that intelligently leverages trace data to identify real deadlocks. We extend the instrumentation for deadlock detection to keep track of time stamps of lock acquisitions and releases, apart from recording the operations. Each thread also maintains a vector of timestamp pairs, that presents the thread’s view of the execution times of the other threads.

We build a Pruner that effectively uses this additional data to identify code regions from different threads that can never overlap, thereby pruning potential deadlocks reported in these regions. The remaining deadlocks along with the trace data are consumed by the Generator. The Generator generates a synchronization dependency graph, \( G_s \), for each deadlock. The lock acquisitions leading to the deadlocking acquisitions made in the original execution of the program form the nodes of this graph. An edge between two nodes \( u \) and \( v \) in the graph denotes that the lock acquisition at \( u \) must happen before the acquisition at \( v \). The edges are generated systematically so that following the order specified in \( G_s \) will result in a deadlocking re-execution. If \( G_s \) contains a cycle, then the associated deadlock can never manifest in a real execution for the path explored (i.e., any different schedule on the same trace will never lead to a deadlocking execution). Therefore, potential deadlocks with a cycle in their corresponding synchronization dependency graphs are also eliminated as false positives automatically.

For the potential set of deadlocks that are output by the Generator, we attempt to re-execute the program on the same input so that all dependencies in \( G_s \) corresponding to the deadlock under consideration are satisfied. We build a Replayer for this purpose that monitors the thread operations and systematically pauses or enables different threads according to the dependencies in \( G_s \). If the Replayer attempts to replay a potential deadlock and the re-execution results in the same deadlock, then it classifies the potential deadlock as real. If it is unable to reproduce the deadlock after a pre-determined number of attempts, the deadlock is classified as unknown and left for manual comprehension.

We have implemented all the above components as part of WOLF and analyzed large Java programs (upto 160KLoC). Applying WOLF on these benchmarks shows that it detects a number of deadlocks. It eliminates 18.5% (12 out of 65) of the reported defects as false positives and automatically confirms 68% (36 out of 53 defects) of the remaining potential defects as real deadlocks. Our experimental results also show the ability of the Replayer to reproduce a given deadlock reliably. The runtime overhead of WOLF for detecting the deadlocks is quite negligible. The average time to automatically reproduce a deadlock ranges from 3 to 50 seconds approximately.

This paper makes the following technical contributions:

- We design a novel and scalable algorithm that uses vector clocks and cycle detection effectively and leverages trace data to detect deadlocks precisely.
- We propose a replay algorithm that runs the program using the synchronization dependency graph generated from the recorded trace so as to deadlock the execution. The reproduction of the deadlock automatically confirms the correctness of the reported defect.
- We implement these algorithms as part of WOLF which takes as input the program under consideration and a set of test inputs and outputs a list of automatically confirmed deadlocks.
- We demonstrate the effectiveness of WOLF by applying it on large benchmarks (upto 160KLoC) and show that it not only reports deadlocks but also identifies 74% of the overall reported defects as true (or false) positives automatically.

---

1 In our experiments, their approach is able to reproduce approximately 35% of the reported defects as we count the number of problematic source locations instead of the number of cycles. Refer Section 4.3 for more details.
2. Motivation

In this section, we demonstrate using examples that existing dynamic deadlock detection approaches, that are trace agnostic, suffer from false positives. We also show that using a randomized schedule to reproduce a real deadlock need not always succeed leaving even real bugs for manual comprehension and motivate the problems addressed in this paper.

![Diagram of two threads synchronized on different instances of ThreadCache](image)

**Figure 1.** False deadlock reported by iGoodLock [14]. \( t_1 \) acquires lock on \( \text{TC} \), an instance of ThreadCache, at line 401 and on \( \text{CT} \), an instance of ThreadCache, at line 75. \( t_2 \) acquires the locks on the same objects in the opposite order at lines 24 and 175 respectively. Deadlock can never manifest because \( t_1 \) always starts \( t_2 \).

Figure 1 presents code fragment from Jigsaw 2.2.6, a web server, executed by two threads \( t_1 \) and \( t_2 \). Initially, \( t_1 \) executes and acquires a lock on an instance of ThreadCache, \( \text{TC} \), at line 401. Subsequently, it acquires a lock on an instance of CachedThread, \( \text{CT} \), at line 75 while still holding \( \text{TC} \). Deadlock detectors based on cycle detection will create two nodes \( \text{TC} \) and \( \text{CT} \) corresponding to the two lock acquisitions and add an edge, labelled \( t_1 \), between them to represent the nested lock acquisition. Eventually, \( t_2 \) also executes and acquires locks on \( \text{CT} \) and \( \text{TC} \) at lines 24 and 175 respectively. An edge between \( \text{CT} \) and \( \text{TC} \) is added and is labelled \( t_2 \). Due to the presence of a cycle in the lock graph and the edges carrying labels of different threads, a potential deadlock is reported by existing state of the art detectors.

On closer examination, however, we observe that this deadlock can never happen in reality. This is because \( t_1 \) starts thread \( t_2 \) at line 76 while holding locks \( \text{TC} \) and \( \text{CT} \). For the deadlock under consideration to manifest between \( t_1 \) and \( t_2 \), it is necessary that one of the locks must be held by \( t_2 \). This is impossible on any interleaving of the two threads as both the locks are already held by \( t_1 \). \( t_2 \) can acquire \( \text{CT} \) only after \( t_1 \) relinquishes it. Therefore, the deadlock detected on this set of lock acquisitions can be eliminated as a false positive. Moreover, we emphasize that any attempt to reproduce the detected (false) deadlock will obviously fail leaving this defect to be manually evaluated.

Even when there is no total ordering of lock acquisitions, the sequence of lock acquisitions leading to the deadlocking acquisitions can make the deadlock infeasible. This is demonstrated by the code fragment given in Figure 2. Consider cycle \( \theta_4 \) shown in the Figure. Let \( \text{SM}_1 \) and \( \text{SM}_2 \) be instances of SynchronizedMap with instance field \( \text{mutex} \). Thread \( t_1 \) acquires locks on \( \text{SM}_1.\text{mutex} \) and \( \text{SM}_2.\text{mutex} \) at lines 2024 and 522 respectively. When thread \( t_2 \) acquires the same locks in the opposite order at lines 2024 and 522 respectively, then existing deadlock detectors report the presence of a potential deadlock. This indeed could be a real deadlock but for the presence of an interim lock acquisition in method \( \text{size} \) invoked at line 509 that lies in between these two lock acquisitions.

Thread \( t_1 \) acquires and releases a lock on \( \text{SM}_2.\text{mutex} \), when the method \( \text{size} \) is invoked in AbstractMap at line 509. Therefore, if the underlying detector analyzes the trace leading to \( \theta_4 \) it can potentially eliminate the reported deadlock as false because it will never manifest in a real execution. To elaborate further, let us assume \( t_1 \) acquires the lock on \( \text{SM}_1.\text{mutex} \) at line 2024. Thread \( t_2 \) cannot go beyond line 509 resulting in cycle \( \theta_1 \) or cycle \( \theta_3 \) manifesting based on whether \( t_1 \) already invoked \( \text{size} \) or not. A similar argument can be made to show that even when \( t_2 \) is executed first, only cycles \( \theta_1 \) and \( \theta_2 \) can manifest. In other words, there is no schedule where \( \theta_4 \) can be reproduced and thus the potential deadlock indicated by it can be safely eliminated as a false positive.

Because of the presence of false positives discussed above, any attempt to reproduce these non-existent bugs will obviously fail. These defects need to be comprehended.
Consider a real deadlock between threads $t_1$ and $t_2$ indicated by cycle $\theta_2$ in Figure 2. In the process of re-executing the program to reproduce a deadlock based on cycle $\theta_2$, if a context switch happens such that $t_1$ always acquires lock on $SM_1$ mutex at line 2024 before $t_2$ can acquire lock at line 509, then thread $t_2$ will always be blocked at line 509. This prevents the deadlock indicated by cycle $\theta_2$ from manifesting and the deadlock indicated by cycle $\theta_1$ to be reproduced. While the deadlock is indeed real, the randomized scheduling can potentially bias the schedules in favor of deadlocks that happen earlier in the code to manifest always, thus motivating the need for a systematic approach. We show, empirically, that maintaining a dependency of the lock acquisitions across threads indeed results in a higher number of real deadlocks being reproduced automatically.

### 3. Design

The overall system architecture of our trace aware dynamic deadlock detection tool named WOLF is shown in Figure 3. We broadly divide the system into four major components – Extended Dynamic Cycle Detector, Pruner, Generator and Replayer.

Extended Dynamic Cycle Detector analyzes the execution of the program on a set of test inputs and identifies cycles in the global lock graph as potential deadlocks. In this work, we use iGoodLock [14] to build the base Detector and extend it to get better precision. In the extended detector, the trace containing the lock acquisitions for each test input is also recorded. Pruner takes the potential deadlocks as input and filters the false positives based on the timestamps. The filtered set of potential deadlocks along with the recorded trace are input to the Generator. For each deadlock and its associated trace, the Generator builds a synchronization dependency graph. The presence of a cycle in this graph indicates that the corresponding deadlock is false. Otherwise, the Replayer attempts to reproduce the potential deadlock by executing the program on the test input that enabled the Detector to detect the deadlock in the first place. Replayer executes the program by ensuring that the dependencies in the synchronization dependency graph associated with the potential deadlock are satisfied to cause the execution to deadlock appropriately. If the deadlock cannot be reproduced after multiple trials, the tool indicates that the defect needs to be analyzed manually. The rest of this section describes each component in detail.

### 3.1 Background

We initially explain the operations in the program that are handled by our analysis and give a brief background on iGoodLock [14] which is used as the base detector in WOLF.

- **Lock($\ell$):** Executing thread acquires lock on $\ell$.
- **Unlock($\ell$):** Executing thread releases lock on $\ell$.
- **t.start():** Executing thread starts thread $t$ and both the threads can execute in parallel thereon.
- **t.join():** Executing thread waits till thread $t$ finishes executing all its instructions.

We consider a thread $t$ makes a deadlocking acquisition if it attempts to acquire a lock and potentially gets into a cyclic dependency with one or more threads resulting in all the threads waiting on each other.

To detect cycles in an execution $\sigma$, the detector maintains a lock dependency relation $D_{\sigma}$, where $D_{\sigma}$ is a set of tuples $(t, L_t, \ell, C_t)$. During execution $\sigma$, if thread $t$ acquires a lock on $\ell$ while holding locks in $L_t$ creating a context $C_t$, then a tuple $\eta = (t, L_t, \ell, C_t)$ is added to $D_{\sigma}$. The context $C_{t}$ is a sequence of execution indices\(^2\) representing instructions that correspond to the acquisition of currently held locks given by $L_t$. We define functions $\text{thread}(\eta)$, $\text{lock}(\eta)$ and $\text{lockset}(\eta)$ that return $t$, $\ell$ and $L_t$ respectively. For any tuple $\eta_i$, we define $\mu_i$ as a function which maps each lock contained in lockset $L_t$ of $\eta_i$ to its corresponding execution index in $C_t$.

The detector uses $D_{\sigma}$ to detect potential deadlocks. We define a potential deadlock, $\theta = \{\eta_1, \eta_2 \ldots \eta_n\}$, where

- For every $\eta_i, \eta_{i+1}$ in $\theta$, $\text{lock}(\eta_i) \in \text{lockset}(\eta_{i+1})$ and $\text{lock}(\eta_i) \in \text{lockset}(\eta_i)$. Every thread in cycle attempts to acquire a lock that is acquired by some other thread.

---

\(^2\)Identifies instructions, objects and threads across runs.
For every \( \eta_i, \eta_j \) in \( \theta \), where \( i \neq j \), \( \text{lockset}(\eta_i) \cap \text{lockset}(\eta_j) = \emptyset \) and \( \text{thread}(\eta_i) \neq \text{thread}(\eta_j) \).

There are no guard locks present for the cycle and each thread contributes to only one edge in the cycle.

For more details on cycle detection and how execution indices are determined, we refer the reader to [14]. We elucidate the basic deadlock detection process using an example and use it as a running example to demonstrate the application of our approach subsequently.

\[
\begin{align*}
\text{t_1} & \quad \text{t_2} & \quad \text{t_3} \\
11 : \text{Lock (t_1)} & \quad 21 : \text{t_3.start()} \quad \rightarrow & \quad 31 : \text{Lock (t_3)} \\
12 : \text{Lock (t_2)} & \quad 32 : \text{Lock (t_2)} \\
13 : \text{Unlock (t_2)} & \quad 33 : \text{Lock (t_1)} \\
14 : \text{Unlock (t_1)} & \quad 34 : \text{Unlock (t_1)} \\
15 : \text{t_2.start()} & \quad 35 : \text{Unlock (t_2)} \\
16 : \text{Lock (t_2)} & \quad \ldots & \quad 36 : \text{Unlock (t_3)} \\
17 : \text{Unlock (t_3)} \\
18 : \text{Lock (t_1)} \\
19 : \text{Lock (t_2)} \\
\ldots
\end{align*}
\]

**Figure 4.** Illustrative example

\[
\eta_1 = (1, \{1, 4\}) \\
\eta_2 = (1, \{1, 6\}) \\
\eta_3 = (3, \{1, 3\}) \\
\eta_4 = (3, \{3, 5\}) \\
\eta_5 = (3, \{3, 5\}) \\
\eta_6 = (1, \{1, 6\}) \\
\eta_7 = (1, \{1, 8\}) \\
\eta_8 = (1, \{1, 8\})
\]

\[
\eta'_1 = (1, \{1, 4, 11\}) \\
\eta'_2 = (1, \{1, 4, 11\}) \\
\eta'_3 = (3, \{1, 3\}) \\
\eta'_4 = (3, \{3, 5\}) \\
\eta'_5 = (3, \{3, 5\}) \\
\eta'_6 = (1, \{1, 6\}) \\
\eta'_7 = (1, \{1, 8\}) \\
\eta'_8 = (1, \{1, 8\})
\]

**Figure 5.** Left: \( D_\sigma \) generated by iGoodLock. Right: \( D_\sigma \) generated by Extended Dynamic Cycle Detector.

Figure 4 presents a code fragment with threads \( t_1, t_2 \) and \( t_3 \), and locks \( \ell_1, \ell_2 \) and \( \ell_3 \). Assume the numbers to the left of the instructions are the execution indices (and not line numbers). According to the Figure, \( t_1 \) acquires a lock on \( \ell_1 \) by executing instruction with index 11. Tuple \( \eta_1 \) is added to \( D_\sigma \) to capture this acquisition. Later, \( t_1 \) acquires a lock on \( \ell_2 \) and \( \eta_2 \) is added to \( D_\sigma \). The other instructions are executed and subsequently \( t_3 \) acquires locks on \( \ell_3, \ell_2 \) and \( \ell_1 \) in that order and tuples \( \eta_3, \eta_4, \eta_5 \) are added to \( D_\sigma \) accordingly. The state of \( D_\sigma \) after all instructions are executed is shown on the left side of Figure 5. When the program terminates, the detector analyzes \( D_\sigma \) and detects two cycles \( \theta_1 \) and \( \theta_2 \), where \( \theta_1 = \{\eta_2, \eta_3\} \) and \( \theta_2 = \{\eta_4, \eta_5\} \).

\( \theta_1 \) is a false deadlock because there is no feasible schedule where \( t_3 \) begins its execution before \( t_1 \) acquires lock \( \ell_2 \) at instruction 12. This is because there is a total order between instructions at 21 and 31 due to \( \ell_3 . \text{start()} \) and the total order between instructions at 31 and 32 due to program order. \( \theta_2 \) can become a real deadlock, even though it involves the same threads and locks as \( \theta_1 \). The existing detector is ordering agnostic and outputs both cycles as potential deadlocks.

We address this limitation by extending the cycle detection and describe it in the next section.

### 3.2 Extended Dynamic Cycle Detector

To improve the precision of detection, each thread maintains a time stamp, which is initially \( \bot \). We use \( \bot \) to indicate that the thread has not started. The value of the time stamp is updated to one when the thread begins its execution. The time stamp of thread \( t \) is increased by one whenever \( t \) executes a \( t . \text{start()} \) or \( t . \text{join()} \) for some thread \( t' \). This represents the creation of a total order among a few instructions of \( t \) and \( t' \). If \( t \) starts \( t' \), all the instructions executed by \( t \) prior to the start precedes every instruction executed by \( t' \). Similarly, on a \( t' . \text{join()} \) by \( t \), all the instructions executed by \( t' \) always happens before every instruction executed by \( t \) after the join.

We use the timestamps to identify non-overlapping range of instructions.

For every thread pair \( (t, t') \), there can be a maximum of two non-overlapping regions between \( t \) and \( t' \). Therefore, to identify the boundaries of the non-overlapping regions, we maintain an ordered pair of timestamps. Each thread maintains a vector of these ordered pairs presenting the boundaries of the non-overlapping regions.

We extend the definition of \( \eta \in D_\sigma \) from the previous section to encode the timestamp as well. \( \eta \) is now defined as \( (t, L_t, \ell, C_t, \tau_t) \), where \( \tau_t \) is the time stamp of thread \( t \) when it acquired a lock on \( \ell \), while holding locks in set \( L_t \) with context \( C_t \). We index each thread in the system to a unique value in \( \{1, \ldots |T|\} \). Also, every thread \( t \) maintains a vector clock \( V_t \), a set of ordered pairs, whose cardinality is \( |T| \). The ordered pair \( (S, J) = V_t(t') \) represents \( t' \)'s view of thread \( t \) and is defined as follows:

1. \( S \): the time stamp of \( t' \) such that all operations in \( t' \) at timestamp less than \( S \) will always be completed before thread \( t \) begins its execution. In other words, no instruction in \( t' \) with timestamp less than \( S \) will ever overlap with any instruction in \( t \).
2. \( J \): the time stamp of \( t \) such that all operations in \( t \) at timestamp greater than or equal to \( J \) will always execute after thread \( t' \) has joined. In other words, no instruction in \( t \) with timestamp greater than or equal to \( J \) will ever overlap with any instruction in \( t' \).

The detector is also extended to maintain two more global states, \( \tau \) and \( V \).

- \( \tau : T \rightarrow N \cup \{\bot\} \), a map from a thread to its timestamp, which is initially \( \bot \).
- \( V : T \rightarrow (\{N \cup \bot\} \times \{N \cup \bot\})^{|T|} \), a map from a thread to its vector clock, where each ordered pair is initially \( (\bot, \bot) \).
Algorithm 1 Extended Dynamic Cycle Detector

Input: Instrumented program, Test input
Output: Set of potential deadlocks, \( \Theta \)

1: \( D_\sigma \leftarrow \emptyset \), \( Enabled \leftarrow T \), \( Terminated \leftarrow \emptyset \);
2: for each \( i \) in \([1, |T|]\) do
3: \( C_i \leftarrow \emptyset \); \( L_i \leftarrow \emptyset \); \( \tau_i = \bot \);
4: end for
5: for each \( i, j \) in \([1, |T|]\) do
6: \( V_i(j) \leftarrow (\bot, \bot) \);
7: end for
8: while \( Enabled \neq \emptyset \) do
9: \( t_p \leftarrow \) a random thread from \( Enabled \)
10: \( s \leftarrow \) next statement to be executed by \( t_p \)
11: if \( \tau_p = \bot \) then \( \tau_p \leftarrow 1 \) end if
12: if \( s \) is nil then Move \( t_p \) from \( Enabled \) to \( Terminated \)
13: else if \( s \) is \( t_c \).start() then
14: \( \tau_p \leftarrow \tau_p + 1 \); \( \tau_c \leftarrow 1 \)
15: for each \( i \) in \([1, |T|]\) do /* updating \( V_e \) */
16: /* Has thread \( t_c \), already joined?*/
17: if \( V_p[i].J \neq \bot \) then \( V_p[i].J = \tau_c \) end if
18: /* Time stamps of \( t_c \), that are always in past for \( t_e \)*/
19: if \( i = p \) then \( V_c[p].S \leftarrow \tau_p \)
20: else \( V_c[i].S \leftarrow V_p[i].S \) end if
21: end for
22: else if \( s \) is \( t_c \).join() then
23: \( \tau_p \leftarrow \tau_p + 1 \)
24: for each \( i \) in \([1, |T|]\) do /* Has \( t_c \) already joined?*/
25: if \( i = c \) or \( (V_c[i].J \neq \bot \) and \( V_p[i].J = \bot) \) then
26: \( V_p[i].J \leftarrow \tau_p \)
27: end if
28: end for
29: else if \( s \) is lock() then
30: \( idx \leftarrow \) execution index of \( s \).
31: Push \( \ell \) to \( L_c \) and \( idx \) to \( C_p \).
32: Add \( \eta \leftarrow \{t_p, L_p, \ell_c, C_p, \tau_p\} \) to \( D_\sigma \).
33: else if \( s \) is unlock() then
34: Pop from \( L_p \) and \( C_p \).
35: end if
36: Execute statement \( s \)
37: end while
38: \( \Theta \leftarrow \) Detected cycles in \( D_\sigma /*standard cycle detection*/ \)

Algorithm 1 presents the set of actions that are performed by the detector during execution. The set \( Enabled \) is initialized to include all the active threads in the system. When a thread \( t_p \) starts another thread \( t_c \), the time stamps of \( t_p \) and \( t_c \) are incremented by one and set to one respectively. The child thread \( t_c \)'s vector clock \( V_c \) is also updated. The child thread \( t_c \)'s \( S \) values are copied from the parent thread \( t_p \)'s vector clock. We perform this operation because the instructions by threads that finished before \( t_p \) began its execution are obviously completed before \( t_c \) begins its execution. If \( J \) value is set to a value other than \( \bot \) in \( V_p(t') \), for some thread \( t' \), then the \( J \) value in \( V_c(t') \) is set to \( \tau_c \). We perform this operation because if \( t_p \) cannot currently overlap with \( t' \) due to an execution of \( t' \).join() in some thread (not necessarily by \( t_p \) due to transitivity), then any thread started by \( t_p \) subsequently can never overlap with the instructions in \( t' \). In a similar fashion, other actions are designed for the rest of the operations. Once the program exits, the cycles in the dependency set \( D_\sigma \) are detected.

We revisit the example from Figure 4 to show the modified \( \eta \) values. The modified \( \eta \) values (denoted as \( \eta' \)) of \( D_\sigma \) are given on the right side of Figure 5. Figure 6 details the change in the timestamps and the vector clocks of each thread. For ease of presentation, we shorthand the ordered pair \( <\bot, \bot> \) to \( \bot \). We also do not show changes to \( C_t \), \( L_t \) and \( D_\sigma \) as the operations on them are the same as in the base detector algorithm. Furthermore, we elide showing the uninteresting operations where timestamps or vector clocks are not affected.

When \( t_1 \) starts, its timestamp is set to one. When \( t_1 \) calls \( t_2 \).start(), the timestamp of \( t_1 \) is increased to two and the timestamp of \( t_2 \) is set to one. \( t_2 \)'s vector clock is also updated such that the value of \( V_2(1) \) is \( <2, \bot> \) and the rest of the ordered pairs remain unchanged. Subsequently when \( t_2 \) performs \( t_3 \).start(), the time stamps of \( t_2 \) and \( t_3 \) are incremented and the vector clock of \( t_3 \) is updated as shown in the Figure. Even though \( t_2 \) starts \( t_3 \), the latter manages to get the information pertaining to the timestamp of \( t_1 \). Moreover, there are no changes in \( V_1 \) as shown by \( \bot \) values. The updates to the remaining timestamp and vector clock values follow Algorithm 1.

The detector detects the same cycles that were detected previously (as discussed in Section 3.1), albeit with additional information. We now have two cycles \( \theta_1' \) and \( \theta_2' \) where \( \theta_1' = \{\eta_2', \eta_3'\} \) and \( \theta_2' = \{\eta_6', \eta_0'\} \).
3.3 Pruner

Armed with the additional information, Pruner applies Algorithm 2 on the potential defects and filters false positives.

Algorithm 2 Pruner

**Input:** Potential deadlocks $\Theta$ and vector clock $V^{\prime}$

**Output:** Result Map, R: $\Theta \rightarrow \{\text{False, Unknown}\}$

1. for each cycle $\theta = \{\eta_1, \eta_2, \ldots, \eta_n\}$ in $\Theta$ do
2. \hspace{0.5cm} $\text{R}\{\theta\} \leftarrow \text{Unknown}$
3. \hspace{0.5cm} for each $(\eta_i, \eta_j)$ in $\theta$ where $i \neq j$ do
4. \hspace{1.0cm} Let $t_i$ be thread($\eta_i$) and $t_j$ be thread($\eta_j$)
5. \hspace{1.0cm} if $V_i(j).S > \eta_i.\tau$ then
6. \hspace{1.5cm} /* Thread $t_j$ hasn’t started */
7. \hspace{1.0cm} $\text{R}\{\theta\} \leftarrow \text{False}$
8. \hspace{1.0cm} else if $V_i(j).J \neq \perp$ and $V_i(j).J \leq \eta_i.\tau$ then
9. \hspace{1.5cm} /* Thread $t_j$ has already joined */
10. \hspace{1.0cm} $\text{R}\{\theta\} \leftarrow \text{False}$
11. \hspace{1.0cm} end if
12. \hspace{1.0cm} end for
13. \hspace{0.5cm} end for

For a given cycle $\theta = \{\eta_1, \eta_2, \ldots, \eta_n\}$, every pair $(\eta_i, \eta_j)$ is checked to find if the cycle is feasible. If $t_i = \text{thread}(\eta_i)$ and $t_j = \text{thread}(\eta_j)$, then it is initially checked whether thread $t_i$ always starts only after thread $t_j$ made the deadlocking acquisition. Next we check whether thread $t_j$ has always exited before thread $t_i$ made its deadlocking acquisition. If either of these checks succeed, then the potential deadlock is marked as False. If both the checks fail for every such pair then the cycle is classified as a potential deadlock by Pruner.

Cycle $\theta'_2$ is marked as false positive because $V_3(1).S$ equals 2 and $\eta'_2.\tau$ equals one and therefore the first conditional check in Algorithm 2 succeeds. The other potential deadlock, $\theta'_2$ is still Unknown and forms the input to the Generator.

3.4 Generator

The potential deadlocks output by the Pruner along with the corresponding execution traces form the input to the Generator. Generator captures all the synchronization dependencies from the trace that need to be satisfied for a potential deadlock to become a real deadlock and generates a synchronization dependency graph accordingly. If a cycle is detected in this graph, then the deadlock is eliminated as a false positive. For an acyclic graph, the Generator outputs the synchronization dependency graph.

We now describe the process of generating a synchronization dependency graph, $G_s$. The execution indices of the lock acquisitions in $D_{\sigma}$, the execution trace, form the nodes in $G_s$. An edge $(u, v)$ in $G_s$ indicates that the acquisition at $u$ must execute before acquisition at $v$. Let $\theta$ be a potential deadlock containing tuples $\{\eta_1, \eta_2, \ldots, \eta_n\}$ found in $D_{\sigma}$. Let $T_0 = \{t_1, t_2, \ldots, t_n\}$ be the threads involved in cycle $\theta$. We define $D'_{\sigma} \subseteq D_{\sigma}$ as the set of tuples leading up to the deadlocking acquisition in every thread $t$ in $T_0$. $D'_{\sigma}$ is defined because all acquisitions by a thread $t$ after the potentially deadlocking acquisition in the original run are irrelevant for the deadlock under consideration to be reproduced. Three types of edges are added to $G_s$.

1. **type-D edge:** For every $\eta_i, \eta_{i+1}$ in cycle $\theta$ such that $\ell_i = \text{lock}(\eta_i)$ and $\ell_{i+1} \in \text{lockset}(\eta_{i+1})$, add edge $\langle \mu_{i+1}(\ell_{i}), \mu_i(\ell_i) \rangle$. This represents the necessary condition for the deadlock to happen. Informally, each thread in the cycle needs to acquire a lock and wait for another lock for the deadlock to manifest.

2. **type-C edge:** For each lock $\ell_j$ in lockset($\eta_j$), $\forall t_i \in T_0$, $\forall t_k$ in $D'_{\sigma}$, if $\text{thread}(\eta_k) \neq t_i$ and $\text{lock}(\eta_k) = \ell_j$, then add edge $\langle \mu_{k}(\ell_j), \mu_{i}(\ell_j) \rangle$. This captures the fact that a lock $\ell_j$ in the lockset should be acquired by $t_i$ only after every thread $t_j \in T_0$ has made necessary acquisitions on $\ell_j$, setting up the deadlocking context.

3. **type-P edge:** For every pair of tuples $\eta_i, \eta_j$ in $D'_{\sigma}$ s.t $\text{thread}(\eta_i) = \text{thread}(\eta_j)$, if tuples $\eta_i, \eta_j$ are added successively by thread($\eta_i$), add edge $\langle \mu_i(\text{lock}(\eta_i)), \mu_{j}(\text{lock}(\eta_j)) \rangle$. This represents the program order edges.
We illustrate the different types of edges in $G_s$ with an example. Consider cycle $\theta'_2$ from the running example in Figure 4. $\theta'_2 = \{\eta'_8, \eta'_9\}$ is declared by the Pruner as a potential deadlock. A potential deadlock between $t_1$ and $t_2$ exists when the threads hold locks $\ell_1$ (at 18) and $\ell_2$ (at 32) respectively and attempt to acquire $\ell_2$ (at 19) and $\ell_1$ (at 33) respectively. We add the type-$D$ edges, the necessary condition for the deadlock to manifest, (18, 33) and (32, 19) to $G_s$ as shown in Figure 7(a). Subsequently, we add type-$C$ edges (16, 31), (12, 32) and (11, 33) to the graph. This is because the pairs of locations (16, 31), (12, 32) and (11, 33) acquire the same locks and the locks acquired at indices 31, 32 and 33 are present as part of $t_2$’s context. If the lock at 31 is acquired and held by $t_2$ even before $t_1$ can acquire it at 16, then the deadlock indicated by $\theta'_2$ cannot be reproduced. Lastly, type-$P$ edges (11, 12), (12, 16), (16, 18), (18, 19), (31, 32) and (32, 33) are added to denote the program order.

The algorithm for constructing $G_s$ is given in Algorithm 3. Each vertex in $G_s$ is a tuple with three elements that include the thread identifier, the execution index of the lock operation and the lock. As shown in Algorithm 3, type-$D$, type-$C$ and type-$P$ edges are added in that order and a cycle detection performed on $G_s$. If a cycle exists in $G_s$, then the potential deadlock is declared as a false positive. Otherwise, $G_s$ is used by the Replayer to deadlock the execution.

The absence of a cycle in $G_s$ (shown in Figure 7(a)), corresponding to $\theta'_2$, denotes its feasibility in practice. Potentially, it can be reproduced if the dependencies in $G_s$ are satisfied. Figure 7(b) presents a contrasting scenario and shows a cyclic $G_s$ for the fourth cycle shown in Figure 2. We infer that the fourth cycle can never result in a deadlock because of the cyclic $G_s$.

### Algorithm 4 Replayer

**Input:** $G_s(V_s, E_s)$ : Synchronization dependency graph

**Output:** Result Map, $R: \Theta \rightarrow \{\text{Deadlock}, \text{Unknown}\}$

1. $\text{BlockedInstr} \leftarrow \emptyset$, $\text{Enabled} \leftarrow T$, $\text{Terminated} \leftarrow \emptyset$
2. $\text{BlockedInstr} \leftarrow \emptyset$, $R[\theta] \leftarrow \text{Unknown}$
3. Let $T_\theta$ be the set of threads in cycle $\theta$
4. while $\text{Enabled} \cup \text{Paused} \neq \emptyset$ do
   5. if $\text{Enabled} = \emptyset$ then
      6. Move a random thread $t'$ from $\text{Paused}$ to $\text{Enabled}$.
   7. end if
   8. $t_i \leftarrow$ random thread from $\text{Enabled}$.
   9. $s \leftarrow$ next statement to be executed by $t_i$.
   10. if $s$ is nil then
       11. move $t_i$ from $\text{Enabled}$ to $\text{Terminated}$
   12. end if
   13. if $t_i \not\in T_\theta$ then
       14. Execute s; Goto the beginning of the loop
   15. end if
   16. if $s$ is Lock ($t$) then
      17. Vertex $v \leftarrow (t_i, \text{idx}, \ell)$ / $\text{idx}$ is execution index of $s$
      18. if $\exists a \in V_s$, s.t. $(u, v) \in E_s$ and $u, t \neq v, t$ then
          19. Add $v$ to $\text{BlockedInstr}$
          20. Move $t_i$ to $\text{Paused}$ and goto beginning of the loop
      21. else
          22. Remove every vertex $u$ in $V_s$ that reaches $v$
          23. Remove all the edges incident on $u$
      24. for every $a$ in $\text{BlockedInstr}$ do
         25. if $\exists b \in V_s$, s.t. $(b, a) \in E_s$ and $b, t \neq a, t$ then
            26. Remove vertex $a$ from $\text{BlockedInstr}$
            27. Move thread $a, t$ from $\text{Paused}$ to $\text{Enabled}$
         28. end if
      29. end for
   30. end if
   31. end if
   32. Execute statement $s$
   33. if execution deadlocked at the exact location then
      34. $R[\theta] \leftarrow \text{Deadlock}$
   35. end if
36. end while

![Figure 7.](image-url)
lock was detected. If the thread attempts to acquire a lock at some execution index, and if there is no incoming edge to the node representing the execution index in $G_s$, the lock acquisition is permitted and once after acquisition all edges emanating from the node are removed. If the lock acquisition is not permitted then the thread needs to wait for at least one acquisition by some other thread and the Replayer pauses the current thread until the acquisition happens. The Replayer also ensures that if a thread skips a specific execution index by following a different path, the edges from the node corresponding to the skipped execution index is also removed. Very rarely, all threads are paused when at least one of the threads can make progress. In that unlikely scenario, the Replayer chooses a random thread from the set of paused threads to execute. If none of the threads can make progress indicating a deadlock and if this deadlock corresponds to the potential deadlock that the Replayer set out to reproduce, then the potential deadlock is considered to be automatically verified as a real deadlock.

We now illustrate the application of Algorithm 4 on the running example from Figure 4 using the associated $G_s$ (see Figure 7(a)). The Replayer randomly selects either $t_1$ or $t_3$ to execute. Since $t_2$ is not yet active, it selects $t_1$ and executes up to 15 thus removing edges $(11,12), (12,16), (11,33)$ and $(12,32)$ from $G_s$. At this point, both $t_1$ and $t_3$ are eligible to execute. Assuming $t_2$ is chosen randomly, $t_3$ attempts to acquire a lock on $t_3$ at index 31. However, this acquisition is not allowed until $t_1$ acquires the lock at index 16 because of the dependency denoted by $G_s$. Therefore, the Replayer pauses $t_3$ and chooses $t_1$ for execution. As there are no dependencies in the updated $G_s$ pertaining to index 16, $t_3$ acquires the lock at index 16. Subsequently, the dependencies $(16,31)$ and $(16,18)$ are removed from $G_s$. The removal of these dependencies enable $t_3$ to become active again. Subsequently, if $t_3$ attempts to acquire the lock at index 33 before $t_1$ acquires the lock at index 18, the execution of $t_2$ will be perturbed appropriately. Eventually, the execution proceeds until $t_1$ and $t_3$ are blocked at indices 19 and 33 respectively indicating the presence of a real deadlock.

While the above example shows the Replayer following the dependencies in $G_s$, it is also flexible when the dependencies are not satisfactory to ensure progress. For the same example, consider a run where $t_1$ executes 11, 12 and 18 and skips 16 due to a conditional (possible due to other interleavings). In this case, the Replayer detects the change in the control flow of $t_1$ and automatically removes dependencies $(16,31)$ and $(16,18)$ from $G_s$ enabling $t_3$ to execute again.

4. Implementation and Evaluation

We have implemented WOLF in Java to analyze Java applications. The implementation builds on top of the iGoodLock [14] framework which uses Soot [32] for code instrumentation. The implementation of the Extended Detector, Pruner and Generator components of WOLF is straightforward and follows the respective algorithms described in the previous section. For the implementation of the Replayer component, WOLF implements a monitor thread that observes the synchronization operations performed by threads which are expected to deadlock. The threads are assigned a unique identifier during detection. During replay, we use the same assignment strategy to identify corresponding threads across runs. Therefore we monitor only $k$ threads for a deadlock involving $k$ threads. Monitoring only the selected threads enables us to drive the execution along the designated schedule without having to pause more threads than are necessary. The monitor thread pauses the selected threads at different synchronization points based on the synchronization dependency graph.

4.1 Experimental setup

We analyze large Java benchmarks (upto 160KLoC) to evaluate WOLF. All the experiments were conducted on a Ubuntu-12.04 desktop running on a 3.5 Ghz Intel Core i7 processor with 16GB RAM. The benchmarks used for the evaluation include a simple and fast cache for Java objects (cache4j), a leading edge web server platform (Jigsaw), a Java logging library (jakarta-log4j1.2.8) and implementations of Java Collections. To show the effectiveness of our approach, we compare our approach with DeadlockFuzzer [14], a state of the art randomized tool for identifying real deadlocks. For each benchmark and for each test input, the program is executed twice – DeadlockFuzzer analyzes one execution and WOLF analyzes the other.

4.2 Results

The results of our experiments are given in Table 1. It tabulates the benchmarks used for our experiments, their sizes, statistics pertaining to the kinds of deadlocks, the number of deadlocks detected initially and the effectiveness of our approach compared to the state of the art. WOLF classifies approximately 18.5% of the deadlocks as false positives across all benchmarks. We manually verified that the defects reported as false positives are indeed false. The tool also reproduces significantly higher number of deadlocks (55.4%) compared to DeadlockFuzzer (35.4%) as seen from the table. We account this higher reproducibility of our approach to two factors – the use of the synchronization dependency graph to drive the execution and the pausing of only selected threads during execution. In contrast, DeadlockFuzzer uses a completely randomized approach for reproducing the deadlocks and also pauses all threads with the required abstraction. The elimination of false positives and higher reproducibility rate of WOLF leaves only 26% of defects for manual comprehension.

We use hit rate to measure the reliability of our approach to reproduce a given deadlock. A hit is considered to occur
if the replay results in a deadlock on which it was expected
to deadlock. More specifically, if the deadlock that need
to be reproduced has locks that are acquired from specific
source code locations and the reproduced execution also
acquired locks (or attempts to acquire locks) at the same
locations, we consider that a hit. If the execution deadlocks
at a different point, it is not considered a hit. For each
benchmark and for each deadlock reported, we attempt to
reproduce the deadlock by running the Replayer 100 times.
We count the number of times a deadlock that is expected
to be reproduced in the execution. We use this count to calculate
the hit rate. Figure 8 shows our approach has a higher hit rate
than DeadlockFuzzer across all benchmarks. Even when both approaches are able to reproduce the deadlocks (e.g., TreeHashMap, WeakHashMap), the observed hit rate of our approach is higher. We attribute the higher hit rate of our approach to its ability to drive the execution through a previously observed synchronization schedule more reliably. However, observe that the Replayer drives the execution
according to dependencies in $G_s$ and the threads have the freedom to execute other instructions in any order. This non-determinism accounts for the hit rate being less than one with our approach.

We are able to detect many critical deadlocks. For example, not only are we able to detect the defect reported in bug 24159 3 in Java Logging, but also we are able to reproduce the deadlocking execution with a hit rate of one. We use a true deadlock reported in Java Collections to elaborate on the underlying reason for the higher hit rate of our approach. Figure 9 shows a code fragment where a deadlock exists between threads $t_1$ and $t_2$. $SC_1$ and $SC_2$ are two instances of class SynchronizedCollection, where $mutex$ is a field. The deadlock happens when $t_1$ acquires a lock on $SC_1.mutex$ at line 1591, $t_2$ acquires a lock on $SC_2.mutex$ at line 1594 and then each thread attempts to acquire the other lock at lines 1570 and 1567 respectively. Surprisingly, DeadlockFuzzer was never able to reproduce this dead-

3https://issues.apache.org/bugzilla/show_bug.cgi?id=24159
lock (in 100 runs) even though WOLF reliably reproduced this deadlock.

On careful examination, we find that \( t_2 \) executes the same sequence of operations as \( t_1 \) before it executes the operations shown in the Figure. Since, in DeadlockFuzzer, \( t_1 \) and \( t_2 \) have the same abstraction and the approach does not take the dependency constraints into account, \( t_2 \) is incorrectly paused at line 1570 causing a different deadlock (corresponding to different source locations) to be reproduced. Comparatively, as WOLF differentiates \( t_1 \) and \( t_2 \), it does not pause \( t_2 \) at line 1570. Moreover, as it uses the synchronization dependency graph to drive the execution, it pauses \( t_1 \) before line 1591 so that \( t_2 \) executes all the operations up to line 1594 causing the deadlock to be reproduced reliably.

Inspite of higher data generation and consumption, the additional overhead introduced by our tool is minimal. In Table 1, we present the slowdown due to deadlock detection as compared to running the uninstrumented program. We observe that the slowdown varies from 1.07x to 2.3x. Figure 10 also shows the relative detection and reproduction time overheads of WOLF as compared to DeadlockFuzzer. For detection, which includes both Pruner and Generator, our tool introduces approximately 10% relative slowdown across all the benchmarks. The relative overhead of the Replayer ranges from 0.8x for WeakHashMap to 2.1x for Jigsaw. We attribute the better performance to the ability of WOLF to not pause unnecessarily. The extra overhead with Jigsaw is because their approach deadlocks at the same source points on many runs and fails to reproduce many deadlocks. In contrast, our tool explores newer regions to reproduce the deadlock resulting in a higher overhead. In absolute terms, the average amount of time taken to reproduce a deadlock ranges from approximately 3 seconds (for ArrayList) to 50 seconds (for Jigsaw). We believe the overhead incurred in automatically identifying a deadlock is tolerable given the costs associated with manual comprehension.

According to Table 1, the average length of the stack trace ranges from 4 to 11. These numbers point to the tediousness associated with manually reasoning about detected deadlocks. Even for a deadlock involving two threads and two locks, there will be two stack traces and a programmer needs to understand the semantics of multiple procedures to identify whether the reported deadlock can happen in a real execution. The average number of nodes in \( G_s \), the synchronization dependency graph, ranges from 4 to 1486. The number gives the number of lock acquisitions that need to be made before a deadlock can happen. A completely random approach for generating a schedule has poor probability of reproducing a deadlock because failing to acquire even one of the locks in \( G_s \) appropriately may result in the deadlock not being detected. These statistics corroborate the underlying reason for designing the various algorithms presented in this paper.

### 4.3 Dynamic Defect Enumeration

The defect count reported in Table 1 corresponds to the unique source code locations of the deadlocking lock acquisitions by the respective threads. In contrast, in [14], the authors report the number of cycles in the lock graph. In other words, if there are two cycles in a lock graph and the nodes
(or lock acquisitions) in the cycles correspond to the same source locations, they count it as two defects whereas we count it as one in this paper. We believe our counting is more appropriate because when the defects need to be fixed, a programmer will have to make changes to a specific source code location. Furthermore, to show a problem exists at a specific source location, it is sufficient to reproduce one deadlocking execution for that source location. Also, using cycles to count the overall defects penalizes both the approaches unnecessarily based on the number of dynamic occurrences of the defective source code location in a run.

For a better understanding, we also provide a comparison of both the tools, where each cycle is counted as a separate defect, in Table 2. Even using this metric of evaluation, WOLF significantly outperforms DeadlockFuzzer. The former classifies 28% of cycles as false positives and reproduces 44.9% of cycles, leaving around 27% of cycles for manual classification. On the other hand, the latter reproduces only 19.1% of cycles leaving 80.9% of cycles for manual analysis.

4.4 Discussion

Our design to reduce the false positives is inspired by other successful program analysis tools [2]. Since our approach is based on dynamic analysis, the quality of the results will be a function of the test inputs. For detecting concurrency bugs, it also becomes a function of the explored schedules. Therefore, a limitation of our approach is that it may consider real bugs as false due to incomplete traces. While our experimental results show that all the false positives reported by WOLF are indeed false, incomplete traces is still a valid concern – a concern that we share with other popular and useful dynamic analysis tools [6]. To elaborate further, in Figure 4, eliminating cycle \( \theta_1 \) can be incorrect if there exists some unexplored execution path where \( t_1 \) is started by some other thread \( t_3 \) (instead of \( t_2 \)). We can address this limitation in two ways. Integrate WOLF with other complementary tools – viz., automatic test input generators [10, 24, 30] and effective schedule explorers [20]. Furthermore, the reported deadlocks can also be ranked based on the output of WOLF, so that the detected false positives are ranked the lowest instead of being eliminated.

WOLF reports 26% of the defects as unknown and a number of these defects are indeed false positives. These false positives exist due to data dependency which ensures that certain code regions never overlap. Currently, WOLF does not explore data dependency associated with the executions and we plan to address this in the future.

5. Related Work

Our approach for developing a precise technique for deadlock detection is inspired by the foundational work of Joshi et al. [14]. As we have shown in our experimental results, we propose an approach that is more precise and is able to reproduce deadlocks more reliably. In MagicFuzzer [3], Cai and Chan improve the scalability and efficiency of cycle detection by pruning the lock dependency graph and apply their algorithm on C/C++ programs. The modifications mentioned in MagicFuzzer can be easily incorporated in WOLF and with appropriate instrumentation, WOLF can be extended to C/C++ programs. Joshi et al. [15] also propose a dynamic analysis for detecting generalized deadlocks that specifically involve communication patterns. We focus on resource deadlocks in this paper. There are a number of static analysis approaches [22, 33] that are designed for detecting deadlocks. Static analysis techniques suffer from higher false positive rate [14]. Furthermore, automatically reproducing a deadlock based on a static analysis report is significantly harder.

Empirical evidence shows that vector clock [18] based concurrency bug detection tools are effective. Pozniansky and Schuster design an algorithm for detecting race conditions in multi-threaded programs [27]. The memory accesses and operations on locks are instrumented to compute the happens-before relation and race conditions are detected. FastTrack [8] makes this process more efficient by identifying appropriate locations where the vector can be reduced to a scalar. The pruning phase of our approach to identify non-overlapping regions of execution is motivated by these algorithms. However, we do not instrument memory accesses unlike these algorithms. Therefore, the overhead introduced by using vector clocks is negligible (approximately 10% overhead).

Replaying the execution by perturbing the schedule to automatically verify the correctness of the reported defects is an active area of research [1, 3, 14, 23, 26, 28]. Narayanasamy et al. [23] provide a replay approach for differentiating between real and benign races by changing the order of problematic memory accesses in the replay. Pradel and Gross [28] execute code sequentially whenever they observe a problem in a concurrent execution to verify whether the problem is due to concurrency. Our approach shares the same ideals of these pioneering approaches, i.e., reduce the amount of manual work that needs to be performed by a programmer to understand defects and concentrate on real bugs.

To improve the reliability of concurrent programs, many techniques based on manipulating schedules are proposed [5, 7, 11, 12, 31]. Cui et al. [4, 5, 34] propose schedule specialization to guard against bugs caused due to nondeterminism. Huang and Zhang [11] design a tool PECAN to generate a feasible schedule for access anomalies in programs that use locks in a nested way. Farzan et al. [7] propose an approach for identifying alternative interleavings that detect null pointer dereferences in concurrent programs using SMT solvers. Sinha and Wang [31] differentiate between intra-thread and inter-thread semantics to reduce the number of possible interleavings that need to be explored. Here, we manipulate the schedule for a different purpose.
Reproducing the bugs that manifest occasionally in multi-threaded executions is an important problem [12, 25]. These approaches are applicable after a bug has manifested in a real execution as opposed to detecting the bugs proactively. CHESS [20] systematically explores thread interleavings by adopting a context-bounded algorithm to detect concurrency bugs. Other testing approaches targeted towards concurrent applications [13, 21, 36] are complementary to the design of WOLF.

6. Conclusions

We present the design and implementation of WOLF, a dynamic analysis tool for detecting deadlocks. WOLF leverages the trace of an execution to automatically identify false positives. Moreover, the tool uses the trace and attempts to reproduce a deadlock so as to verify the correctness of a reported defect without manual intervention. We analyze large Java benchmarks (upto 160KLoC) and our experimental results show that WOLF easily outperforms the state of the art detection tools by identifying three quarters of the reported defects as real (or false) deadlocks automatically.

Acknowledgments

We thank the anonymous reviewers for their useful feedback. We are grateful to Koushik Sen for providing the experimental benchmarks and test harnesses that are used in our experiments. This project is supported in part by Ministry of Human Resources and Development (Government of India), Microsoft Research India and Google India.

References

of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’07.


