A Randomized Dynamic Program Analysis Technique for Detecting Real Deadlocks

Pallavi Joshi Chang-Seo Park
Koushik Sen
EECS Department, UC Berkeley, USA
{pallavi,parkcs,ksen}@cs.berkeley.edu

Mayur Naik
Intel Research, Berkeley, USA
mayur.naik@intel.com

Abstract

We present a novel dynamic analysis technique that finds real deadlocks in multi-threaded programs. Our technique runs in two stages. In the first stage, we use an imprecise dynamic analysis technique to find potential deadlocks in a multi-threaded program by observing an execution of the program. In the second stage, we control a random thread scheduler to create the potential deadlocks with high probability. Unlike other dynamic analysis techniques, our approach has the advantage that it does not give any false warnings. We have implemented the technique in a prototype tool for Java, and have experimented on a number of large multi-threaded Java programs. We report a number of previously known and unknown real deadlocks that were found in these benchmarks.

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

General Terms Languages, Reliability, Verification

Keywords deadlock detection, dynamic program analysis, concurrency

1. Introduction

A common cause for unresponsiveness in software systems is a deadlock situation. In shared-memory multi-threaded systems, a deadlock is a liveness failure that happens when a set of threads blocks forever because each thread in the set is waiting to acquire a lock held by another thread in the set. Deadlock is a common form of bug in today's software—Sun's bug database at `http://bugs.sun.com/` shows that 6,500 bug reports out of 198,000 contain the keyword 'deadlock'. There are a few reasons for the existence of deadlock bugs in multi-threaded programs. First, software systems are often written by many programmers; therefore, it becomes difficult to follow a lock order discipline that could avoid deadlock. Second, programmers often introduce deadlocks when they fix race conditions by adding new locks. Third, software systems can allow incorporation of third-party software (e.g. plugins); third-party software may not follow the locking discipline followed by the parent software and this sometimes results in deadlock bugs [17].

Deadlocks are often difficult to find during the testing phase because they happen under very specific thread schedules. Coming up with these subtle thread schedules through stress testing or random testing is often difficult. Model checking [15, 11, 7, 14, 6] removes these limitations of testing by systematically exploring all thread schedules. However, model checking fails to scale for large multi-threaded programs due to the exponential increase in the number of thread schedules with execution length.

Several program analysis techniques, both static [19, 10, 2, 9, 27, 29, 21] and dynamic [12, 13, 4, 1], have been developed to detect and predict deadlocks in multi-threaded programs. Static techniques often give no false negatives, but they often report many false positives. For example, the static deadlock detector developed by Williams et al. [29] reports 100,000 deadlocks in Sun's JDK 1.4 [1], out of which only 7 are real deadlocks. Type and annotation based techniques [5, 10] help to avoid deadlocks during coding, but they impose the burden of annotation on programmers. Predictive dynamic techniques such as Goodlock [13] and its improvements [4, 1] give both false negatives and false positives. For example, in our experiments we have found that an improved Goodlock can report as many as 254 false positives for our Jigsaw web server. Being imprecise in nature, most of these tools require manual inspection to see if a deadlock is real or not. Nevertheless, these techniques are effective in finding deadlocks because they can predict deadlocks that could potentially happen during a real execution—for such a prediction, static analyses do not need to see an actual execution and dynamic analyses need to see only one multi-threaded execution.

Dynamic analysis based deadlock detection can be made precise by taking the happens-before relation [18] into account. However, it has several problems. First, it reduces the predictive power of dynamic techniques—it fails to report deadlocks that could happen in a significantly different thread schedule. Second, it can perturb an execution significantly and can fail to report a deadlock that can happen when no dynamic analysis is performed.

We propose a new dynamic technique for detecting real deadlocks in multi-threaded programs, called DEADLOCKFUZZER, which combines an imprecise dynamic deadlock detection technique with a randomized thread scheduler to create real deadlocks with high probability. The technique works in two phases. In the first phase, we use an informative and a simple variant of the Goodlock algorithm, called *informative Goodlock*, or simply iGoodlock, to discover potential deadlock cycles in a multi-threaded program. For example, iGoodlock could report a cycle

[1] They reduce the number of reports to 70 after applying various unsound heuristics

PLDI'09, June 15–20, 2009, Dublin, Ireland.

of the form $(t_1, l_1, l_2, [c_1, c_2])(t_2, l_2, l_1, [c_1', c_2'])$, which says that there could be a deadlock if thread t_1 tries to acquire lock l_2 at program location c_2 after acquiring lock l_1 at program location c_1 and thread t_2 tries to acquire lock l_1 at program location c_2' after acquiring lock l_2 at program location c_1'. In the second phase, DEADLOCKFUZZER executes the program with a random schedule in order to create a real deadlock corresponding to a cycle reported in the previous phase. For example, consider the cycle $(t_1, l_1, l_2, [c_1, c_2])(t_2, l_2, l_1, [c_1', c_2'])$ again. At each program state, the random scheduler picks a thread and executes its next statement with the following exception. If t_1 is about to acquire lock l_2 at location c_2 after acquiring lock l_1 at location c_1, then the random scheduler pauses the execution of thread t_1. Similarly, the random scheduler pauses the execution of thread t_2 if it is about to acquire lock l_1 at location c_2' after acquiring lock l_2 at location c_1'. In this biased random schedule, it is very likely that both the threads will reach a state where t_1 is trying to acquire l_2 while holding l_1 and t_2 is trying to acquire l_1 while holding l_2. This results in a real deadlock. In summary, DEADLOCKFUZZER actively controls a randomized thread scheduler based on a potential deadlock cycle reported by an imprecise deadlock detection technique.

The above technique poses the following *key challenge*. Phase II assumes that Phase I can provide it with precise knowledge about the thread and lock objects involved in the deadlock cycle. Unfortunately, since thread and lock objects are created dynamically at runtime, their addresses cannot be used to identify them across executions, i.e. in the above example, addresses of t_1, t_2, l_1, l_2 do not remain the same between Phase I and Phase II executions. Therefore, we need some mechanism to identify the same objects across executions. Specifically, we need a form of object abstraction such that if two dynamic objects in different executions are the same, they must have the same abstraction. For example, the label of a statement at which an object is created can be used as its abstraction. Such an abstraction of an object does not change across executions. However, an abstraction could be the same for several objects (e.g. if both l_1 and l_2 in the above example are created by the same statement). In this paper, we propose two techniques for computing the abstraction of an object that helps us to distinguish between different objects more precisely—the first technique is motivated by the notion of *k-object-sensitivity* in static analysis [20] and the second technique is motivated by the notion of execution indexing [30]. We show that both these abstractions are better than the trivial abstraction where all objects have the same abstraction. We also empirically show that the abstraction based on execution indexing is better than the abstraction based on k-object-sensitivity in most benchmarks.

We have implemented DEADLOCKFUZZER for multi-threaded Java programs in a prototype tool. We have applied the tool to a large number of benchmarks having a total of over 600K lines of code. The results of our experiments show that DEADLOCKFUZZER can create real deadlocks with high probability and DEADLOCKFUZZER can detect all previously known real deadlocks.

We make the following contributions in this paper.

- We propose a simple and informative variant of the Goodlock algorithm, called *iGoodlock*. Unlike existing Goodlock algorithms [13, 4, 1], iGoodlock does not use lock graphs or depth-first search, but reports the same deadlocks as the existing algorithms. Due to this modification, iGoodlock uses more memory, but reduces runtime complexity. We also attach context information with each cycle that helps in debugging and in biasing the random scheduler. iGoodlock is iterative in nature—it finds all cycles of length k before finding any cycle of length $k+1$. Our experiments show that all real deadlocks in our benchmarks have length two. Therefore, if we have a limited time budget, we can run iGoodlock for one iteration so that it only reports deadlock cycles of length 2.
- *Our key contribution* is an active random deadlock detecting scheduler that can create real deadlocks with high probability (we show this claim empirically) based on information provided by iGoodlock. This phase prevents us from reporting any false positives and creates real deadlocks which are useful for debugging. This relieves the manual inspection burden associated with other imprecise techniques such as Goodlock.
- We propose two kinds of object abstraction techniques that help us correlate thread and lock objects between iGoodlock and the randomized scheduling algorithm.
- We have implemented DEADLOCKFUZZER in a tool for Java and have discovered subtle previously known and unknown deadlocks in large applications. To the best of our knowledge, DEADLOCKFUZZER is the first precise dynamic deadlock analysis tool for Java that has been applied to large Java applications.

2. Algorithm

The DEADLOCKFUZZER algorithm consists of two phases. In the first phase, we execute a multi-threaded program and find potential deadlocks that could happen in some execution of the program. This phase uses a modified Goodlock algorithm, called *informative Goodlock*, or simply iGoodlock, which identifies potential deadlocks even if the observed execution does not deadlock. We call the modified algorithm *informative* because we provide suitable debugging information to identify the cause of the deadlock—this debugging information is used by the second phase to create real deadlocks with high probability. A limitation of iGoodlock is that it can give false positives because it does not consider the happens-before relation between the transitions in an execution. As a result the user is required to manually inspect such potential deadlocks. The second phase removes this burden from the user. In this phase, a random thread scheduler is biased to generate an execution that creates a real deadlock reported in the previous phase with high probability. We next describe these two phases in more detail.

2.1 Background Definitions

We use a general and simple model of a concurrent system to describe our dynamic deadlock checking algorithm. We consider a concurrent system to be composed of a finite set of threads. Each thread executes a sequence of labeled statements. A thread communicates with other threads using shared objects. At any point during program execution, a concurrent system is in a *state*. Let s_0 be the initial state. A concurrent system evolves from one state to another state when a thread executes a statement. In our algorithms, we will consider the following dynamic instances of labeled program statements:

1. c: `Acquire`(l), denoting the acquire of the dynamic lock object l. c is the label of the statement (same for below).
2. c: `Release`(l), denoting the release of the dynamic lock object l.
3. c: `Call`(m), denoting a call to the method m.
4. c: `Return`(m), denoting the return from the method m.
5. c: $o =$`new` (o', T), where the statement occurs in the body of a method m and when the `this` argument of m evaluates to object o', then o is the dynamic object of type T allocated by the statement.

In several languages including Java, locks are re-entrant, i.e., a thread may re-acquire a lock it already holds. In our algorithm,

we ignore the execution of c: `Acquire`(l) or c: `Release`(l) statements by a thread t, if t re-acquires the lock l or does not release the lock l, respectively[2]. To simplify exposition, we also assume that locks are acquired and released in a nested way, i.e., if a thread acquires l_2 after acquiring l_1, then it has to release l_2 before releasing l_1. Our algorithm can easily be extended to handle languages where locks can be acquired and released in an arbitrary order.

Next we introduce some definitions that we will use to describe our algorithms.

- `Enabled`(s) denotes the set of threads that are enabled in the state s. A thread is disabled if it is waiting to acquire a lock already held by some other thread (or waiting on a `join` or a `wait` in Java.)
- `Alive`(s) denotes the set of threads whose executions have not terminated in the state s. A state s is in a *stall state* if the set of enabled threads in s (i.e. `Enabled`(s)) is empty and the set of threads that are alive (i.e. `Alive`(s)) is non-empty.
- `Execute`(s, t) returns the state after executing the next statement of the thread t in the state s.

2.2 Phase I: iGoodlock

In this section, we present a formal description of iGoodlock. The algorithm observes the execution of a multi-threaded program and computes a *lock dependency relation* (defined below) and uses a transitive closure of this relation to compute potential deadlock cycles. The algorithm improves over generalized Goodlock algorithms [4, 1] in two ways. First, it adds context information to a computed potential deadlock cycle. This information helps to identify the program locations where the deadlock could happen and also to statically identify the lock and thread objects involved in the deadlock cycle. Second, we simplify the generalized Goodlock algorithm by avoiding the construction of a lock graph, where locks form the vertices and a labeled edge is added from one lock to another lock if a thread acquires the latter lock while holding the former lock in some program state. Unlike existing Goodlock algorithms, iGoodlock does not perform a depth-first search, but computes transitive closure of the lock dependency relation. As such it uses more memory, but has better runtime complexity. We next introduce some formal definitions before we describe the algorithm.

Given a multi-threaded execution σ, let L_σ be the set of lock objects that were held by any thread in the execution and T_σ be the set of threads executed in the execution. Let $\mathcal{C}$ be the set of all statement labels in the multi-threaded program. We next define *the lock dependency relation of a multi-threaded execution* as follows.

DEFINITION 1. *Given an execution σ, a* lock dependency relation *D_σ of σ is a subset of $T_\sigma \times 2^{L_\sigma} \times L_\sigma \times \mathcal{C}^*$ such that $(t, L, l, C) \in D_\sigma$ iff in the execution σ, in some state, thread t acquires lock l while holding the locks in the set L, and C is the sequence of labels of `Acquire` statements that were executed by t to acquire the locks in $L \cup \{l\}$.*

DEFINITION 2. *Given a lock dependency relation D,* a lock dependency chain *$\tau = \langle (t_1, L_1, l_1, C_1), \ldots, (t_m, L_m, l_m, C_m)\rangle$ is a sequence in D^* such that the following properties hold.*

1. *for all distinct $i, j \in [1, m]$, $t_i \neq t_j$, i.e. the threads $t_1, t_2, \ldots, t_m$ are all distinct objects,*
2. *for all distinct $i, j \in [1, m]$, $l_i \neq l_j$, i.e. the lock objects $l_1, l_2, \ldots, l_m$ are distinct,*
3. *for all $i \in [1, m-1]$, $l_i \in L_{i+1}$, i.e. each thread could potentially wait to acquire a lock that is held by the next thread,*
4. *for all distinct $i, j \in [1, m]$, $L_i \cap L_j = \emptyset$, i.e., each thread t_i should be able to acquire the locks in L_i without waiting.*

DEFINITION 3. *A lock dependency chain* $\tau = \langle (t_1, L_1, l_1, C_1), \ldots, (t_m, L_m, l_m, C_m)\rangle$ *is* a potential deadlock cycle *if $l_m \in L_1$.*

Note that the definition of a potential deadlock cycle never uses any of the C_i's in D_σ to compute a potential deadlock cycle. Each C_i of a potential deadlock cycle provides us with information about program locations where the locks involved in the cycle were acquired. This is useful for debugging and is also used by the active random deadlock checker to determine the program locations where it needs to pause a thread.

Each lock and thread object involved in a potential deadlock cycle is identified by its unique id, which is typically the *address* of the object. The unique id of an object, being based on dynamic information, can change from execution to execution. Therefore, the unique id of an object cannot be used by the active random checker to identify a thread or a lock object across executions. In order to overcome this limitation, we compute an abstraction of each object. An abstraction of an object identifies an object by static program information. For example, the label of a statement at which an object is created could be used as its abstraction. We describe two better (i.e. more precise) object abstraction computation techniques in Section 2.4. In this section, we assume that `abs`(o) returns some abstraction of the object o.

Given a potential deadlock cycle $\langle (t_1, L_1, l_1, C_1), \ldots, (t_m, L_m, l_m, C_m)\rangle$, iGoodlock reports the abstract deadlock cycle $\langle (\texttt{abs}(t_1), \texttt{abs}(l_1), C_1), \ldots, (\texttt{abs}(t_m), \texttt{abs}(l_m), C_m)\rangle$. The active random checker takes such an abstract deadlock cycle and biases a random scheduler so that a real deadlock corresponding to the cycle gets created with high probability.

We next describe iGoodlock. Specifically, we describe how we compute the lock dependency relation during a multi-threaded execution and how we compute all potential deadlock cycles given a lock dependency relation.

2.2.1 Computing the lock dependency relation of a multi-threaded execution

In order to compute the lock dependency relation during a multi-threaded execution, we instrument the program to maintain the following three data structures:

- `LockSet` that maps each thread to a stack of locks held by the thread
- `Context` that maps each thread to a stack of the labels of statements where the thread acquired the currently held locks
- D is the lock dependence relation

We update the above three data structures during a multi-threaded execution as follows:

- Initialization:
  - for all t, both `LockSet`[t] and `Context`[t] map to an empty stack
  - D is an empty set
- If thread t executes the statement c: `Acquire`(l)
  - push c to `Context`[t]
  - add (t,`LockSet`[t], l, `Context`[t]) to D
  - push l to `LockSet`[t]
- If thread t executes the statement c: `Release`(l)

[2] This is implemented by associating a usage counter with a lock which is incremented whenever a thread acquires or re-acquires the lock and decremented whenever a thread releases the lock. Execution of `Acquire`(l) by t is considered whenever the thread t acquires or re-acquires the lock l and the usage counter associated with l is incremented from 0 to 1.

- pop from `Context`$[t]$
- pop from `LockSet`$[t]$

At the end of the execution, we output D as the lock dependency relation of the execution.

2.2.2 Computing *potential* deadlock cycles iteratively

Let D^k denote the set of all lock dependency chains of D that has length k. Therefore, $D^1 = D$. iGoodlock computes potential deadlock cycles by iteratively computing $D^2, D^3, D^4, \ldots$ and finding deadlock cycles in those sets. The iterative algorithm for computing potential deadlock cycles is described in Algorithm 1.

Algorithm 1 `iGoodlock`(D)

```
1: INPUTS: lock dependency relation D
2: i ⇐ 1
3: D^i ⇐ D
4: while D^i ≠ ∅ do
5:    for each (t, L, l, C) ∈ D and each τ in D^i do
6:       if τ, (t, L, l, C) is a dependency chain by Definition 2 then
7:          if τ, (t, L, l, C) is a potential deadlock cycle by Definition 3
            then
8:             report abs(τ, (t, L, l, C)) as a potential deadlock cycle
9:          else
10:            add τ, (t, L, l, C) to D^{i+1}
11:         end if
12:      end if
13:   end for
14:   i ⇐ i + 1
15: end while
```

Note that in `iGoodlock`(D) we do not add a lock dependency chain to D^{i+1} if it is a deadlock cycle. This ensures that we do not report complex deadlock cycles, i.e. deadlock cycles that can be decomposed into simpler cycles.

2.2.3 Avoiding duplicate deadlock cycles

In Algorithm 1, a deadlock cycle of length k gets reported k times. For example, if

$\langle (t_1, L_1, l_1, C_1), (t_2, L_2, l_2, C_2), \ldots, (t_m, L_m, l_m, C_m) \rangle$

is reported as a deadlock cycle, then

$\langle (t_2, L_2, l_2, C_2), \ldots, (t_m, L_m, l_m, C_m), (t_1, L_1, l_1, C_1) \rangle$

is also reported as a cycle. In order to avoid such duplicates, we put another constraint in Definition 2: the unique id of thread t_1 must be less than the unique id of threads $t_2, \ldots, t_m$.

2.3 Phase II: The Active Random Deadlock Checking Algorithm

DEADLOCKFUZZER executes a multi-threaded program using a random scheduler. A simple randomized execution algorithm is shown in Algorithm 2. Starting from the initial state s_0, this algorithm, at every state, randomly picks an enabled thread and executes its next statement. The algorithm terminates when the system reaches a state that has no enabled threads. At termination, if there is at least one thread that is alive, the algorithm reports a system stall. A stall could happen due to a resource deadlock (i.e. deadlocks that happen due to locks) or a communication deadlock (i.e. a deadlock that happens when each thread is waiting for a signal from some other thread in the set). We only consider resource deadlocks in this paper.

A key limitation of this simple random scheduling algorithm is that it may not create real deadlocks very often. DEADLOCKFUZZER biases the random scheduler so that potential deadlock cycles reported by iGoodlock get created with high probability. The active random deadlock checking algorithm is shown in Algorithm 3. Specifically, the algorithm takes an initial state s_0 and a potential deadlock cycle `Cycle` as inputs. It then executes the multi-threaded program using the simple random scheduler, except that it performs some extra work when it encounters a lock acquire or lock release statement. If a thread t is about to acquire a lock l in the context C, then if $(\texttt{abs}(t), \texttt{abs}(l), C)$ is present in `Cycle`, the scheduler pauses thread t before t acquires lock l, giving a chance to another thread, which is involved in the potential deadlock cycle, to acquire lock l subsequently. This ensures that the system creates the potential deadlock cycle `Cycle` with high probability.

Algorithm 2 simpleRandomChecker(s_0)

```
1: INPUTS: the initial state s_0
2: s ⇐ s_0
3: while Enabled(s) ≠ ∅ do
4:    t ⇐ a random thread in Enabled(s)
5:    s ⇐ Execute(s, t)
6: end while
7: if Alive(s) ≠ ∅ then
8:    print 'System Stall!'
9: end if
```

Algorithm 3 DEADLOCKFUZZER(s_0, Cycle)

```
1: INPUTS: the initial state s_0, a potential deadlock cycle Cycle
2: s ⇐ s_0
3: Paused ⇐ ∅
4: LockSet and Context map each thread to an empty stack
5: while Enabled(s) ≠ ∅ do
6:    t ⇐ a random thread in Enabled(s) \ Paused
7:    Stmt ⇐ next statement to be executed by t
8:    if Stmt = c: Acquire(l) then
9:       push l to LockSet[t]
10:      push c to Context[t]
11:      checkRealDeadlock(LockSet) // see Algorithm 4
12:      if (( abs(t), abs(l), Context[t] ) ∉ Cycle) then
13:         s ⇐ Execute(s, t)
14:      else
15:         pop from LockSet[t]
16:         pop from Context[t]
17:         add t to Paused
18:      end if
19:   else if Stmt = c: Release(l) then
20:      pop from LockSet[t]
21:      pop from Context[t]
22:      s ⇐ Execute(s, t)
23:   else
24:      s ⇐ Execute(s, t)
25:   end if
26:   if |Paused| = |Enabled(s)| then
27:      remove a random thread from Paused
28:   end if
29: end while
30: if Active(s) ≠ ∅ then
31:    print 'System Stall!'
32: end if
```

Algorithm 4 checkRealDeadlock(LockSet)

```
1: INPUTS: LockSet mapping each thread to its current stack of locks
2: if there exist distinct t_1, t_2, ..., t_m and l_1, l_2, ..., l_m such that l_m
   appears before l_1 in LockSet[t_m] and for each i ∈ [1, m − 1], l_i
   appears before l_{i+1} in LockSet[t_i] then
3:    print 'Real Deadlock Found!'
4: end if
```

Algorithm 3 maintains three data structures: `LockSet` that maps each thread to a stack of locks that are currently held by the thread, `Context` that maps each thread to a stack of statement labels where the thread has acquired the currently held locks,

and `Paused` which is a set of threads that has been paused by DEADLOCKFUZZER. `Paused` is initialized to an empty set, and `LockSet` and `Context` are initialized to map each thread to an empty stack.

DEADLOCKFUZZER runs in a loop until there is no enabled thread. At termination, DEADLOCKFUZZER reports a system stall if there is at least one active thread in the execution. Note that DEADLOCKFUZZER only catches resource deadlocks. In each iteration of the loop, DEADLOCKFUZZER picks a random thread t that is enabled but not in the `Paused` set. If the next statement to be executed by t is not a lock acquire or release, t executes the statement and updates the state as in the simple random scheduling algorithm (see Algorithm 2). If the next statement to be executed by t is c: `Acquire`(l), c and l are pushed to `Context`$[t]$ and `LockSet`$[t]$, respectively. DEADLOCKFUZZER then checks if the acquire of l by t could lead to a deadlock using `checkRealDeadlock` in Algorithm 4. `checkRealDeadlock` goes over the current lock set of each thread and sees if it can find a cycle. If a cycle is discovered, then DEADLOCKFUZZER has created a *real* deadlock. If there is no cycle, then DEADLOCKFUZZER determines if t needs to be paused in order to get into a deadlock state. Specifically, it checks if $(\texttt{abs}(t), \texttt{abs}(l), \texttt{Context}[t])$ is present in `Cycle`. If t is added to `Paused`, then we pop from both `LockSet`$[t]$ and `Context`$[t]$ to reflect the fact that t has not really acquired the lock l. If the next statement to be executed by t is c: `Release`(l), then we pop from both `LockSet`$[t]$ and `Context`$[t]$.

At the end of each iteration, it may happen that the set `Paused` is equal to the set of all enabled threads. This results in a state where DEADLOCKFUZZER has unfortunately paused all the enabled threads and the system cannot make any progress. We call this *thrashing*. DEADLOCKFUZZER handles this situation by removing a random thread from the set `Paused`. A thrash implies that DEADLOCKFUZZER has paused a thread in an unsuitable state. DEADLOCKFUZZER should avoid thrashing as much as possible in order to guarantee better performance and improve the probability of detecting real deadlocks.

2.4 Computing object abstractions

A key requirement of DEADLOCKFUZZER is that it should know where a thread needs to be paused, i.e. it needs to know if a thread t that is trying to acquire a lock l in a context C could lead to a deadlock. DEADLOCKFUZZER gets this information from iGoodlock, but this requires us to identify the lock and thread objects that are the "same" in the iGoodlock and DEADLOCKFUZZER executions. This kind of correlation cannot be done using the address (i.e. the unique id) of an object because object addresses change across executions. Therefore, we propose to use object abstraction—if two objects are same across executions, then they have the same abstraction. We assume `abs`(o) computes the abstraction of an object.

There could be several ways to compute the abstraction of an object. One could use the label of the statement that allocated the object (i.e. the allocation site) as its abstraction. However, that would be too coarse-grained to distinctly identify many objects. For example, if one uses the factory pattern to allocate all thread objects, then all of the threads will have the same abstraction. Therefore, we need more contextual information about an allocation site to identify objects at finer granularity.

Note that if we use a coarse-grained abstraction, then DEADLOCKFUZZER will pause unnecessary threads before they try to acquire some unnecessary locks. This is because all these unnecessary threads and unnecessary locks might have the same abstraction as the relevant thread and lock, respectively. This will in turn reduce the effectiveness of our algorithm as DEADLOCKFUZZER will more often remove a thread from the `Paused` set due to the unavailability of any enabled thread. Note that we call this situation *thrashing*. Our experiments (see Section 5) show that if we use the trivial abstraction, where all objects have the same abstraction, then we get a lot of thrashing. This in turn reduces the probability of creating a real deadlock. On the other hand, if we consider too fine-grained abstraction for objects, then we will not be able to tolerate minor differences between two executions, causing threads to pause at fewer locations and miss deadlocks. We next describe two abstraction techniques for objects that we have found effective in our experiments.

2.4.1 Abstraction based on k-object-sensitivity

Given a multi-threaded execution and a $k > 0$, let $o_1, \ldots o_k$ be the sequence of objects such that for all $i \in [1, k-1]$, o_i is allocated by some method of object o_{i+1}. We define $\texttt{abs}^O_k(o_1)$ as the sequence $\langle c_1, \ldots, c_k \rangle$ where c_i is the label of the statement that allocated o_i. $\texttt{abs}^O_k(o_1)$ can then be used as an abstraction of o_1. We call this *abstraction based on k-object-sensitivity* because of the similarity to k-object-sensitive static analysis [20].

In order to compute $\texttt{abs}^O_k(o)$ for each object o during a multi-threaded execution, we instrument the program to maintain a map `CreationMap` that maps each object o to a pair (o', c) if o is created by a method of object o' at the statement labeled c. This gives the following straightforward runtime algorithm for computing `CreationMap`.

- If a thread t executes the statement c: $o = \texttt{new}\ (o', T)$, then add $o \mapsto (o', c)$ to `CreationMap`.

One can use `CreationMap` to compute $\texttt{abs}^O_k(o)$ using the following recursive definition:

$\texttt{abs}^O_k(o) = \langle\rangle$ if $k = 0$ or `CreationMap`$[o] = \perp$

$\texttt{abs}^O_{k+1}(o) = c :: \texttt{abs}^O_k(o')$ if `CreationMap`$[o] = (o', c)$

When an object is allocated inside a static method, it will not have a mapping in `CreationMap`. Consequently, $\texttt{abs}^O_k(o)$ may have fewer than k elements.

2.4.2 Abstraction based on light-weight execution indexing

Given a multi-threaded execution, a $k > 0$, and an object o, let $m_n, m_{n-1}, \ldots, m_1$ be the call stack when o is created, i.e. o is created inside method m_1 and for all $i \in [1, n-1]$, m_i is called from method m_{i+1}. Let us also assume that c_{i+1} is the label of the statement at which m_{i+1} invokes m_i and q_{i+1} is the number of times m_i is invoked by m_{i+1} in the context $m_n, m_{n-1}, \ldots, m_{i+1}$. Then $\texttt{abs}^I_k(o)$ is defined as the sequence $[c_1, q_1, c_2, q_2, \ldots, c_k, q_k]$, where c_1 is the label of the statement at which o is created and q_1 is the number of times the statement is executed in the context $m_n, m_{n-1}, \ldots, m_1$.

```
 1 main() {
 2   for (int i=0; i<5; i++)
 3     foo();
 4 }
 5 void foo() {
 6   bar();
 7   bar();
 8 }
 9 void bar() {
10   for (int i=0; i<3; i++)
11     Object l = new Object();
12 }
```

For example in the above code, if o is the first object created by the execution of `main`, then $\texttt{abs}^I_3(o)$ is the sequence $[11, 1, 6, 1, 3, 1]$. Similarly, if o is the last object created by the execution of `main`, then $\texttt{abs}^I_3(o)$ is the sequence $[11, 3, 7, 1, 3, 5]$. The idea of computing this kind of abstraction is similar to the idea of execution indexing proposed in [30], except that we ignore

branch statements and loops. This makes our indexing light-weight, but less precise.

In order to compute $\texttt{abs}_k^I(o)$ for each object o during a multi-threaded execution, we instrument the program to maintain a thread-local scalar d to track its depths and two thread-local maps `CallStack` and `Counters`. We use $\texttt{CallStack}_t$ to denote the `CallStack` map of thread t. The above data structures are updated at runtime as follows.

- Initialization:
  - for all t, $d_t \Leftarrow 0$
  - for all t and c, $\texttt{Counters}_t[d_t][c] \Leftarrow 0$
- If a thread t executes the statement c: $\texttt{Call}(m)$
  - $\texttt{Counters}_t[d_t][c] \Leftarrow \texttt{Counters}_t[d_t][c] + 1$
  - push c to $\texttt{CallStack}_t$
  - push $\texttt{Counters}_t[d_t][c]$ to $\texttt{CallStack}_t$
  - $d_t \Leftarrow d_t + 1$
  - for all c, $\texttt{Counters}_t[d_t][c] \Leftarrow 0$
- If a thread t executes the statement c: $\texttt{Return}(m)$
  - $d_t \Leftarrow d_t - 1$
  - pop twice from $\texttt{CallStack}_t$
- If a thread t executes the statement c: $o = \texttt{new}(o', T)$
  - $\texttt{Counters}_t[d_t][c] \Leftarrow \texttt{Counters}_t[d_t][c] + 1$
  - push c to $\texttt{CallStack}_t$
  - push $\texttt{Counters}_t[d_t][c]$ to $\texttt{CallStack}_t$
  - $\texttt{abs}_k^I(o)$ is the top $2k$ elements of $\texttt{CallStack}_t$
  - pop twice from $\texttt{CallStack}_t$

Note that $\texttt{abs}_k^I(o)$ has $2k$ elements, but if the call stack has fewer elements, then $\texttt{abs}_k^I(o)$ returns the full call stack.

3. Examples Illustrating DEADLOCKFUZZER

Consider the multi-threaded Java program in Figure 1. The program defines a `MyThread` class that has two locks `l1` and `l2` and a boolean `flag`. The `run` method of `MyThread` invokes a number of long running methods `f1, f2, f3, f4` if `flag` is true and then it acquires locks `l1` and `l2` in order. The body of `run` shows a common pattern, where a thread runs several statements and then acquires several locks in a nested way. The `main` method creates two lock objects `o1` and `o2`. It also creates two threads (i.e. instances of `MyThread`). In the first instance `l1` and `l2` are set to `o1` and `o2`, respectively, and `flag` is set to true. Therefore, a call to `start` on this instance will create a new thread which will first execute several long running methods and then acquire `o1` and `o2` in order. A call to `start` on the second instance of `MyThread` will create a new thread which will acquire `o2` and `o1` in order. We have commented out lines 24 and 27, because they are not relevant for the current example—we will uncomment them in the next example.

The example has a deadlock because the locks `o1` and `o2` are acquired in different orders by the two threads. However, this deadlock will rarely occur during normal testing because the second thread will acquire `o2` and `o1` immediately after start, whereas the first thread will acquire `o1` and `o2` after executing the four long running methods. iGoodlock will report this deadlock as a potential one by observing a single execution that does not deadlock. If we use the abstraction in Section 2.4.2 with, say $k = 10$, the report will be as follows:

$$([25, 1], [23, 1], [15, 16]), ([26, 1], [22, 1], [15, 16])$$

```
 1 class MyThread extends Thread {
 2   Object l1, l2;
 3   boolean flag;
 4   MyThread(Object l1, Object l2, boolean b){
 5     this.l1 = l1; this.l2 = l2; this.flag = b;
 6   }
 7
 8   public void run() {
 9     if (flag) { // some long running methods
10       f1();
11       f2();
12       f3();
13       f4();
14     }
15     synchronized(l1) {
16       synchronized(l2) {
17       }
18     }
19   }
20
21   public static void main (String[] args) {
22     Object o1 = new Object();
23     Object o2 = new Object();
24     // Object o3 = new Object();
25     (new MyThread(o1,o2,true)).start();
26     (new MyThread(o2,o1,false)).start();
27     // (new MyThread(o2,o3,false)).start();
28   }
29 }
```

Figure 1. Simple Example of a Deadlock

where $[25, 1]$, $[26, 1]$, $[22, 1]$, $[23, 1]$ are the abstractions of the first thread, the second thread, `o1`, and `o2`, respectively. $[15, 16]$ denotes the context in which the second lock is acquired by each thread.

The active random deadlock checker will take this report and create the real deadlock with probability 1. Specifically, it will pause both the threads before they try to acquire a lock at line 16.

The above example shows that DEADLOCKFUZZER can create a rare deadlock with high probability. In practice, the actual probability may not be 1—DEADLOCKFUZZER can miss a deadlock because the execution could simply take a different path due to non-determinism and that path may not exhibit a deadlock. However, in our experiments we have found that the probability of creating a deadlock is high on our benchmarks.

The above example does not show the utility of using thread and object abstractions. To illustrate the utility of object and thread abstractions, we uncomment the lines at 24 and 27. Now we create a third lock `o3` and a third thread which acquires `o2` and `o3` in order. iGoodlock as before will report the same deadlock cycle as in the previous example. In DEADLOCKFUZZER, if we do not use thread and object abstractions, then with probability 0.5 (approx), the third thread will pause before acquiring the lock at line 16. This is because, without any knowledge about threads and objects involved in a potential deadlock cycle, DEADLOCKFUZZER will pause any thread that reaches line 16. Therefore, if the third thread pauses before line 16, then the second thread will not be able to acquire lock `o2` at line 15 and it will be blocked. DEADLOCKFUZZER will eventually pause the first thread at line 16. At this point two threads are paused and one thread is blocked. This results in a *thrashing* (see Section 2.3). To get rid of this stall, DEADLOCKFUZZER will "un-pause" the first thread with probability 0.5 and we will miss the deadlock with probability 0.25 (approx). On the other hand, if we use object and thread abstractions, then DEADLOCKFUZZER will never pause the third thread at line 16 and it will create the real deadlock with probability 1. This illustrates that if we do not use

thread and object abstractions, then we get more thrashings and the probability of creating a real deadlock gets reduced.

4. Optimization: avoiding another potential cause for thrashing

We showed that using object and thread abstractions helps reduce thrashing; this in turn helps increase the probability of creating a deadlock. We show another key reason for a lot of thrashings using the following example and propose a solution to partly avoid such thrashings.

```
1: thread1{                    8: thread2{
2:   synchronized(l1){         9:   synchronized(l1){
3:     synchronized(l2){       10:
4:     }                       11:   }
5:   }                         12:   synchronized(l2){
6: }                           13:     synchronized(l1){
                               14:     }
                               15:   }
                               16: }
```

The above code avoids explicit thread creation for simplicity of exposition. iGoodlock will report a potential deadlock cycle in this code. In the active random deadlock checking phase, if `thread1` is paused first (at line 3) and if `thread2` has just started, then `thread2` will get blocked at line 9 because `thread1` is holding the lock `l1` and it has been paused and `thread2` cannot acquire the lock. Since we have one paused and one blocked thread, we get a thrashing. DEADLOCKFUZZER will "un-pause" `thread1` and we will miss the real deadlock. This is a common form of thrashing that we have observed in our benchmarks.

In order to reduce the above pattern of thrashing, we make a thread to yield to other threads before it starts entering a deadlock cycle. Formally, if $(\texttt{abs}(t), \texttt{abs}(l), C)$ is a component of a potential deadlock cycle, then DEADLOCKFUZZER will make any thread t' with $\texttt{abs}(t) = \texttt{abs}(t')$ yield before a statement labeled c where c is the bottommost element in the stack C. For example, in the above code, DEADLOCKFUZZER will make `thread1` yield before it tries to acquire lock `l1` at line 2. This will enable `thread2` to make progress (i.e. acquire and release `l1` at lines 9 and 11, respectively). `thread2` will then yield to any other thread before acquiring lock `l2` at line 12. Therefore, the real deadlock will get created with probability 1.

5. Implementation and Evaluation

DEADLOCKFUZZER can be implemented for any language that supports threads and shared memory programming, such as Java or C/C++ with pthreads. We have implemented DEADLOCKFUZZER for Java by instrumenting Java bytecode to observe various events and to control the thread scheduler. The implementation is a part of the CALFUZZER framework [16]. DEADLOCKFUZZER can go into livelocks. Livelocks happen when all threads of the program end up in the `Paused` set, except for one thread that does something in a loop without synchronizing with other threads. In order to avoid livelocks, DEADLOCKFUZZER creates a monitor thread that periodically removes those threads from the `Paused` set that are paused for a long time.

5.1 Experimental setup

We evaluated DEADLOCKFUZZER on a variety of Java programs and libraries. We ran our experiments on a dual socket Intel Xeon 2GHz quad core server with 8GB of RAM. The following programs were included in our benchmarks: cache4j, a fast thread-safe implementation of a cache for Java objects; sor, a successive over-relaxation benchmark, and hedc, a web-crawler application, both from ETH [28]; jspider, a highly configurable and customizable Web Spider engine; and Jigsaw, W3C's leading-edge Web server platform. We created a test harness for Jigsaw that concurrently generates simultaneous requests to the web server, simulating multiple clients, and administrative commands (such as "shutdown server") to exercise the multi-threaded server in a highly concurrent situation.

The libraries we experimented on include synchronized lists and maps from the Java Collections Framework, Java logging facilities (`java.util.logging`), and the Swing GUI framework (`javax.swing`). Another widely used library included in our benchmarks is the Database Connection Pool (DBCP) component of the Apache Commons project. Each of these libraries contains potential deadlocks that we were able to reproduce using DEADLOCKFUZZER. We created general test harnesses to use these libraries with multiple threads. For example, to test the Java Collections in a concurrent setting, we used the synchronized wrappers in `java.util.Collections`.

5.2 Results

Table 1 shows the results of our analysis. The second column reports the number of lines of source code that was instrumented. If the program uses libraries that are also instrumented, they are included in the count. The third column shows the average runtime of a normal execution of the program without any instrumentation or analysis. The fourth column is the runtime of iGoodlock (Phase I). The fifth column is the average runtime of DEADLOCKFUZZER (Phase II). The table shows that the overhead of our active checker is within a factor of six, even for large programs. Note that runtime for the web server Jigsaw is not reported due to its interactive nature.

The sixth column is the number of potential deadlocks reported by iGoodlock. The seventh column is the number of cycles that correspond to real deadlocks after manual inspection. For Jigsaw, since DEADLOCKFUZZER could reproduce 29 deadlocks, we can say for sure that Jigsaw has 29 or more real deadlocks. With the exception of Jigsaw, iGoodlock was precise enough to report only real deadlocks. The eighth column is the number of deadlock cycles confirmed by DEADLOCKFUZZER. The ninth column is the empirical probability of DEADLOCKFUZZER reproducing the deadlock cycle. We ran DEADLOCKFUZZER 100 times for each cycle and calculated the fraction of executions that deadlocked using DEADLOCKFUZZER. Our experiments show that DEADLOCKFUZZER reproduces the potential deadlock cycles reported by iGoodlock with very high probability. We observed that for some Collections benchmarks, DEADLOCKFUZZER reported a low probability of 0.5 for creating a deadlock. After looking into the report, we found that in the executions where DEADLOCKFUZZER reported no deadlock, DEADLOCKFUZZER created a deadlock which was different from the potential deadlock cycle provided as input to DEADLOCKFUZZER. For comparison, we also ran each of the programs normally without instrumentation for 100 times to observe if these deadlocks could occur under normal testing. None of the runs resulted in a deadlock, as opposed to a run with DEADLOCKFUZZER which almost always went into deadlock. Column 10 shows the average number of thrashings per run. Columns 9 and 10 show that the probability of creating a deadlock decreases as the number of thrashings increases.

We conducted additional experiments to evaluate the effectiveness of various design decisions for DEADLOCKFUZZER. We tried variants of DEADLOCKFUZZER: 1) with abstraction based on k-object-sensitivity, 2) with abstraction based on light-weight execution indexing, 3) with the trivial abstraction, 4) without context information, and 5) with the optimization in Section 4 turned off. Figure 2 summarizes the results of our experiments. Note that the

Program name	Lines of code	Avg. Runtime in msec.			# Deadlock cycles			Probability of reproduction	Avg. # of Thrashes
		Normal	iGoodlock	DF	iGoodlock	Real	Reproduced		
cache4j	3,897	2,045	3,409	-	0	0	-	-	-
sor	17,718	163	396	-	0	0	-	-	-
hedc	25,024	165	1,668	-	0	0	-	-	-
jspider	10,252	4,622	5,020	-	0	0	-	-	-
Jigsaw	160,388	-	-	-	283	$\geq$ 29	29	0.214	18.97
Java Logging	4,248	166	272	493	3	3	3	1.00	0.00
Java Swing	337,291	4,694	9,563	28,052	1	1	1	1.00	4.83
DBCP	27,194	603	1,393	1,393	2	2	2	1.00	0.00
Synchronized Lists (ArrayList, Stack, LinkedList)	17,633	2,862	3,244	7,070	9 + 9 + 9	9 + 9 + 9	9 + 9 + 9	0.99	0.0
Synchronized Maps (HashMap, TreeMap, WeakHashMap, LinkedHashMap, IdentityHashMap)	18,911	2,295	2,596	2898	4 + 4 + 4 + 4 + 4	4 + 4 + 4 + 4 + 4	4 + 4 + 4 + 4 + 4	0.52	0.04

Table 1. Experimental results. (Context + 2nd Abstraction + Yield optimization)

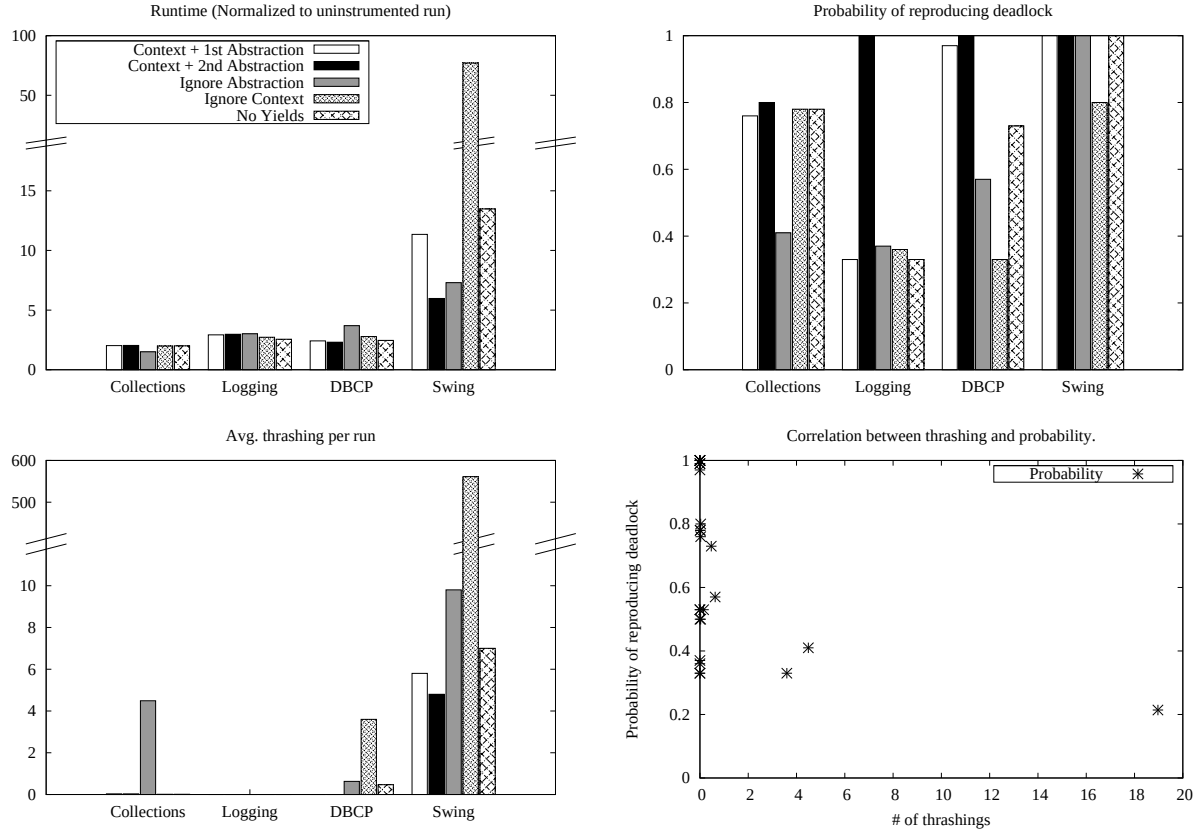

Figure 2. Performance and effectiveness of variations of DEADLOCKFUZZER

results in Table 1 correspond to the variant 2, where we use the light-weight execution indexing abstraction, context information, and the optimization in Section 4. We found this variant to be the best performer: it created deadlocks with higher probability than any other variant and it ran efficiently with minimal number of thrashings.

The first graph shows the correlation between the various variants of DEADLOCKFUZZER and average runtime. The second graph shows the probability of creating a deadlock by the variants of DEADLOCKFUZZER. The third graph shows the average number of thrashings encountered by each variant of DEADLOCKFUZZER. The fourth graph shows the correlation between the number of thrashings and the probability of creating a deadlock.

The first graph shows that variant 2, which uses execution indexing, performs better than variant 1, which uses k-object-sensitivity. The second graph shows that the probability of creating a dead-

lock is maximum for variant 2 on our benchmarks. The difference is significant for the Logging and DBCP benchmarks. Ignoring abstraction entirely (i.e. variant 3) led to a lot of thrashing in Collections and decreased the probability of creating a deadlock. The third graph on the Swing benchmark shows that variant 2 has minimum thrashing. Ignoring context information increased the thrashing and the runtime overhead for the Swing benchmark. In the Swing benchmark, the same locks are acquired and released many times at many different program locations during the execution. Hence, ignoring the context of lock acquires and releases leads to a huge amount of thrashing.

The first graph which plots average runtime for each variant shows some anomaly. It shows that variant 3 runs faster than variant 2 for Collections—this should not be true given that variant 3 thrashes more than variant 2. We found the following reason for this anomaly. Without the right debugging information provided by iGoodlock, it is possible for DEADLOCKFUZZER to pause at wrong locations but, by chance, introduce a real deadlock which is unrelated to the deadlock cycle it was trying to reproduce. This causes the anomaly in the first graph where the runtime overhead for Collections is lower when abstraction is ignored, but the number of thrashings is more. The runtime is measured as the time it takes from the start of the execution to either normal termination or when a deadlock is found. DEADLOCKFUZZER with our light-weight execution indexing abstraction faithfully reproduces the given cycle, which may happen late in the execution. For more imprecise variants such as the one ignoring abstraction, a deadlock early in the execution may be reproduced wrongfully, thus reducing the runtime.

The fourth graph shows that the probability of creating a deadlock goes down as the number of thrashings increases. This validates our claim that thrashings are not good for creating deadlocks with high probability and our variant 2 tries to reduce such thrashings significantly by considering context information and object abstraction based on execution indexing, and by applying the optimization in Section 4.

5.3 Deadlocks found

DEADLOCKFUZZER found a number of previously unknown and known deadlocks in our benchmarks. We next describe some of them.

Two previously unknown deadlocks were found in Jigsaw. As shown in Figure 3, when the http server shuts down, it calls cleanup code that shuts down the `SocketClientFactory`. The shutdown code holds a lock on the factory at line 867, and in turn attempts to acquire the lock on `csList` at line 872. On the other hand, when a `SocketClient` is closing, it also calls into the factory to update a global count. In this situation, the locks are held in the opposite order: the lock on `csList` is acquired first at line 623, and then on the factory at line 574. Another similar deadlock occurs when a `SocketClient` kills an idle connection. These also involve the same locks, but are acquired at different program locations. iGoodlock provided precise debugging information to distinguish between the two contexts of the lock acquires.

The deadlock in the Java Swing benchmark occurs when a program synchronizes on a `JFrame` object, and invokes the `setCaretPosition()` method on a `JTextArea` object that is a member of the `JFrame` object. The sequence of lock acquires that leads to the deadlock is as follows. The `main` thread obtains a lock on the `JFrame` object, and an `EventQueue` thread which is also running, obtains a lock on a `BasicTextUI$BasicCaret` object at line number 1304 in *javax/swing/text/DefaultCaret.java*. The `main` thread then tries to obtain a lock on the `BasicTextUI$BasicCaret` object at line number 1244 in *javax/swing/text/DefaultCaret.java*,

```
org.w3c.jigsaw.http.httpd {
 384: SocketClientFactory factory;
1442: void cleanup(...) {
1455:   factory.shutdown();}
1711: void run() {
1734:   cleanup(...);}}

org.w3c.jigsaw.http.socket.SocketClient {
 42:  SocketClientFactory pool;
111:  void run() {
152:    pool.clientConnectionFinished(...);}}

org.w3c.jigsaw.http.socket.SocketClientFactory {
130:  SocketClientState csList;
574:  synchronized boolean decrIdleCount() {...}
618:  boolean clientConnectionFinished(...) {
623:    synchronized (csList) {
626:      decrIdleCount();}}
867:  synchronized void killClients(...) {
872:    synchronized (csList) {...}}
902:  void shutdown() {
904:      killClients(...);}
}
```

Figure 3. Deadlock in Jigsaw

but fails to do so since the lock has not been released by the `EventQueue` thread. The `EventQueue` thread tries to acquire the lock on the `JFrame` object at line number 407 in *javax/swing/RepaintManager.java* but cannot since it is still held by the `main` thread. The program goes into a deadlock. This deadlock corresponds to a bug that has been reported at `http://bugs.sun.com/view_bug.do?bug_id=4839713`.

One of the deadlocks that we found in the DBCP benchmark occurs when a thread tries to create a `PreparedStatement`, and another thread simultaneously closes another `PreparedStatement`. The sequence of lock acquires that exhibits this deadlock is as follows. The first thread obtains a lock on a `Connection` object at line number 185 in *org/apache/commons/dbcp/DelegatingConnection.java*. The second thread obtains a lock on a `KeyedObjectPool` object at line number 78 in *org/apache/commons/dbcp/PoolablePreparedStatement.java*. The first thread then tries to obtain a lock on the same `KeyedObjectPool` object at line number 87 in *org/apache/commons/dbcp/PoolingConnection.java*, but cannot obtain it since it is held by the second thread. The second thread tries to obtain a lock on the `Connection` object at line number 106 in *org/apache/commons/dbcp/PoolablePreparedStatement.java*, but cannot acquire it since the lock has not yet been released by the first thread. The program, thus, goes into a deadlock.

The deadlocks in the Java Collections Framework happen when multiple threads are operating on shared collection objects wrapped with the `synchronizedX` classes. For example, in the `synchronizedList` classes, the deadlock can happen if one thread executes `l1.addAll(l2)` concurrently with another thread executing `l2.retainAll(l1)`. There are three methods, `addAll()`, `removeAll()`, and `retainAll()` that obtain locks on both `l1` and `l2` for a total of 9 combinations of deadlock cycles. The `synchronizedMap` classes have 4 combinations with the methods `equals()` and `get()`.

The test cases for Java Collections are artificial in the sense that the deadlocks in those benchmarks arise due to inappropriate use of the API methods. We used these benchmarks because they have been used by researchers in previous work (e.g. Williams et al. [29] and Jula et al. [17]), and we wanted to validate our tool against these benchmarks.

5.4 Imprecision in Goodlock

Since DEADLOCKFUZZER is not complete, if it does not classify a deadlock reported by iGoodlock as a real deadlock, we cannot definitely say that the deadlock is a false warning. For example, in the Jigsaw benchmark, the informative Goodlock algorithm reported 283 deadlocks. Of these 29 were reported as real deadlocks by DEADLOCKFUZZER. We manually looked into the rest of the deadlocks to see if they were false warnings by iGoodlock, or real deadlocks that were not caught by DEADLOCKFUZZER. For 18 of the cycles reported, we can say with a high confidence that they are false warnings reported by the iGoodlock algorithm. These cycles involve locks that are acquired at the same program statements, but by different threads. There is a single reason why all of these deadlocks are false positives. The deadlocks can occur only if a `CachedThread` invokes its `waitForRunner()` method before that `CachedThread` has been started by another thread. This is clearly not possible in an actual execution of Jigsaw. Since iGoodlock does not take the happens-before relation between lock acquires and releases into account, it reports these spurious deadlocks. For the rest of the cycles reported by iGoodlock, we cannot say with reasonable confidence if they are false warnings, or if they are real deadlocks that were missed by DEADLOCKFUZZER.

6. Related Work

We have already compared our proposed technique with several existing techniques for detecting deadlocks in multi-threaded programs. In this section, we discuss several other related approaches, and elaborate on some that we have previously mentioned.

DEADLOCKFUZZER is part of the *active testing framework* [16] that we have earlier developed for finding real races [25] and real atomicity violations [23]. We proposed RACEFUZZER [25] which uses an active randomized scheduler to confirm race conditions with high probability. RACEFUZZER only uses statement locations to identify races and does not use object abstraction or context information to increase the probability of race detection. As shown in Section 5.2, simple location information is not good enough for creating real deadlocks with high probability.

Recently, several random testing techniques have been proposed [8, 26] that introduce noise (using `yield`, `sleep`, `wait` (with timeout)) to a program execution to increase the possibility of the exhibition of a synchronization bug. Although these techniques have successfully detected bugs in many programs, they have a limitation. These techniques are not systematic as the primitives `sleep()`, `yield()`, `priority()` can only advise the scheduler to make a thread switch, but cannot force a thread switch. As such they cannot pause a thread as long as required to create a real deadlock.

More recently, a few techniques have been proposed to confirm potential bugs in concurrent programs using random testing. Havelund et al. [3] uses a directed scheduler to confirm that a potential deadlock cycle could lead to a real deadlock. However, they assume that the thread and object identifiers do not change across executions. Similarly, ConTest [22] uses the idea of introducing noise to increase the probability of the occurrence of a deadlock. It records potential deadlocks using a Goodlock algorithm. To check whether a potential deadlock can actually occur, it introduces noise during program execution to increase the probability of exhibition of the deadlock. Our work differs from ConTest in the following ways. ConTest uses only locations in the program to identify locks. We use context information and object abstractions to identify the run-time threads and locks involved in the deadlocks; therefore, our abstractions give more precise information about run-time objects. Moreover, we explicitly control the thread scheduler to create the potential deadlocks, instead of adding timing noise to program execution. DEADLOCKFUZZER, being explicit in controlling scheduler and in identifying objects across executions, found real deadlocks in large benchmarks with high probability.

A couple of techniques have been proposed to prevent deadlocks from happening during program execution, and to recover from deadlocks during execution. When a buggy program executes and deadlocks, Dimmunix [17] records the deadlock pattern. During program execution, it tries to prevent the occurrence of any of the deadlock patterns that it has previously observed. Rx [24] proposes to recover programs from software failures, including deadlocks, by rolling them back to a recent checkpoint, and re-executing the programs in a modified environment.

7. Conclusion

Existing techniques for deadlock detection, based on static and dynamic analysis, could predict potential deadlocks, but could not verify if they were real deadlocks. Going through all of these warnings and reasoning about them manually could be time consuming. DEADLOCKFUZZER automates such verification—if a real deadlock is created by DEADLOCKFUZZER, the developer no longer needs to verify the deadlock manually. However, DEADLOCKFUZZER is incomplete—if a deadlock is not confirmed to be real by DEADLOCKFUZZER, the developer cannot ignore the deadlock. Nevertheless, DEADLOCKFUZZER has managed to find all previously known deadlocks in large benchmarks and it has discovered previously unknown deadlocks. We believe that DEADLOCKFUZZER is an indispensable and practical tool that complements both static and predictive dynamic analysis.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable comments. This research was supported in part by a generous gift from Intel, by Microsoft and Intel funding (award #20080469), by matching funding by U.C. Discovery (award #DIG07-10227), and by NSF Grant CNS-0720906.

References

[1] R. Agarwal, L. Wang, and S. D. Stoller. Detecting potential deadlocks with static analysis and runtime monitoring. In *Parallel and Distributed Systems: Testing and Debugging 2005*, 2005.

[2] C. Artho and A. Biere. Applying static analysis to large-scale, multi-threaded Java programs. In *Proceedings of the 13th Australian Software Engineering Conference (ASWEC'01)*, pages 68–75, 2001.

[3] S. Bensalem, J.-C. Fernandez, K. Havelund, and L. Mounier. Confirmation of deadlock potentials detected by runtime analysis. In *PADTAD'06*, pages 41–50, 2006.

[4] S. Bensalem and K. Havelund. Scalable dynamic deadlock analysis of multi-threaded programs. In *Parallel and Distributed Systems: Testing and Debugging 2005 (PADTAD'05)*, 2005.

[5] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 211–230, 2002.

[6] S. Chaki, E. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. Concurrent software verification with states, events, and deadlocks. *Formal Aspects of Computing*, 17(4):461–483, 2005.

[7] C. Demartini, R. Iosif, and R. Sisto. A deadlock detection tool for concurrent java programs. *Software - Practice and Experience*, 29(7):577–603, 1999.

[8] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, , and S. Ur. Multithreaded Java program test generation. *IBM Systems Journal*, 41(1):111–125, 2002.

[9] D. R. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 237–252, 2003.

[10] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245. ACM, 2002.

[11] P. Godefroid. Model checking for programming languages using verisoft. In *24th Symposium on Principles of Programming Languages*, pages 174–186, 1997.

[12] J. Harrow. Runtime checking of multithreaded applications with visual threads. In *7th International SPIN Workshop on Model Checking and Software Verification*, pages 331–342, 2000.

[13] K. Havelund. Using runtime analysis to guide model checking of java programs. In *7th International SPIN Workshop on Model Checking and Software Verification*, pages 245–264, 2000.

[14] K. Havelund and T. Pressburger. Model Checking Java Programs using Java PathFinder. *Int. Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[15] G. Holzmann. The Spin model checker. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[16] P. Joshi, M. Naik, C.-S. Park, and K. Sen. An extensible active testing framework for concurrent programs. In *21st International Conference on Computer Aided Verification (CAV'09)*, Lecture Notes in Computer Science. Springer, 2009.

[17] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, 2008.

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

[19] S. Masticola. *Static detection of deadlocks in polynomial time*. PhD thesis, Rutgers University, 1993.

[20] A. Milanova, A. Rountev, and B. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology*, 14(1):1–41, Jan. 2005.

[21] M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *31st International Conference on Software Engineering (ICSE'09)*. IEEE, 2009.

[22] Y. Nir-Buchbinder, R. Tzoref, and S. Ur. Deadlocks: From exhibiting to healing. In *8th Workshop on Runtime Verification*, 2008.

[23] C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *16th International Symposium on Foundations of Software Engineering (FSE'08)*. ACM, 2008.

[24] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: treating bugs as allergies—a safe method to survive software failures. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 235–248. ACM, 2005.

[25] K. Sen. Race directed random testing of concurrent programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, 2008.

[26] S. D. Stoller. Testing concurrent Java programs using randomized scheduling. In *Workshop on Runtime Verification (RV'02)*, volume 70 of *ENTCS*, 2002.

[27] C. von Praun. *Detecting Synchronization Defects in Multi-Threaded Object-Oriented Programs*. PhD thesis, Swiss Federal Institute of Technology, Zurich, 2004.

[28] C. von Praun and T. R. Gross. Object race detection. In *16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications (OOPSLA)*, pages 70–82. ACM, 2001.

[29] A. Williams, W. Thies, and M. Ernst. Static deadlock detection for Java libraries. In *ECOOP 2005 — 19th European Conference on Object-Oriented Programming (ECOOP'05)*, pages 602–629, 2005.

[30] B. Xin, W. N. Sumner, and X. Zhang. Efficient program execution indexing. In *ACM SIGPLAN conference on Programming language design and implementation*, pages 238–248, 2008.