Sherlock: Scalable Deadlock Detection for Concurrent Programs

Mahdi Eslamimehr Jens Palsberg

UCLA, University of California, Los Angeles, USA
{mahdi,palsberg}@cs.ucla.edu

ABSTRACT

We present a new technique to find real deadlocks in concurrent programs that use locks. For 4.5 million lines of Java, our technique found almost twice as many real deadlocks as four previous techniques combined. Among those, 33 deadlocks happened after more than one million computation steps, including 27 new deadlocks. We first use a known technique to find 1275 deadlock candidates and then we determine that 146 of them are real deadlocks. Our technique combines previous work on concolic execution with a new constraint-based approach that iteratively drives an execution towards a deadlock candidate.

Categories and Subject Descriptors

D.2.5 Software Engineering [**Testing and Debugging**]

Keywords

Concurrency; deadlocks

1. INTRODUCTION

Java has a concurrent programming model with threads, shared memory, and locks. The shared memory enables threads to exchange data efficiently, and the locks can help control memory access and prevent concurrency bugs such as data races.

In Java, the statement synchronized(e){ s } first evaluates the expression e to an object, then acquires the lock of that object, then executes the statement s, and finally releases the lock.

Locks enable deadlocks, which can happen when two or more threads wait on each other forever [49]. For example, suppose one thread executes:

synchronized(A) { synchronized(B) { ... }}

while another thread concurrently executes:

synchronized(B) { synchronized(A) { ... }}

One possible schedule of the program lets the first thread acquire the lock of A and lets the other thread acquire the lock of B. Now the program is deadlocked: the first thread waits for the lock of B, while the second thread waits for the lock of A.

Usually a deadlock is a bug and programmers should avoid deadlocks. However, programmers may make mistakes so we have a bug-finding problem: provide tool support to find as many deadlocks as possible in a given program.

Researchers have developed many techniques to help find deadlocks. Some require program annotations that typically must be supplied by a programmer; examples include [18, 47, 6, 54, 66, 60, 42, 23, 35]. Other techniques work with unannotated programs and thus they are easier to use. In this paper we focus on techniques that work with unannotated Java programs. We use 22 open-source benchmarks that have a total of more than 4.5 million lines of code, which we use "straight of the box" without annotations.

We can divide deadlock-detection techniques into three categories: static, dynamic, and hybrid. A static technique examines the text of a program without running it. One of the best static tools is Chord [46, 45] which for our benchmarks reports 570 deadlocks, which include both false positives and false negatives. A dynamic technique gathers information about a program during one or more runs. Until now, four of best dynamic tools are DeadlockFuzzer [36], IBM ConTest [16, 20], Jcarder [17] and Java HotSpot [48], which together for our benchmarks report 75 real deadlocks. Finally, hybrid techniques may be able to combine the best of both worlds, static and dynamic. One of the best hybrid tools is GoodLock [29] which is highly efficient and for our benchmarks report a total 1275 deadlocks, which may include both false positives and false negatives.

In this paper we focus on dynamic techniques. The advantage of a dynamic technique is that it reports only real deadlocks. The main shortcoming of the previous dynamic techniques is that they mostly find deadlocks that occur after *few* steps of computation. Our experiments show that those techniques leave undetected many deadlocks that occur after one million steps of computations. We believe that this shortcoming stems from their approach to search for executable *schedules*. A schedule is a sequence of events that must be executed in order. A real deadlock is a combination of deadlock pattern, such as the one in the example above, and an executable schedule that leads to the deadlock. We will show how to do a better search for executable schedules.

The challenge. Help programmers find more reproducible deadlocks than with previous techniques.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
FSE'14 , November 16–22, 2014, Hong Kong, China
Copyright 2014 ACM 978-1-4503-3056-5/14/11 ...$15.00.

Our result. We present a technique that for a deadlock candidate searches for an input and a schedule that lead to the deadlock.

We use GoodLock [29] to quickly produce a manageable number of deadlock candidates. Our technique combines previous work on concolic execution with a new constraint-based approach to drive an execution towards a deadlock candidate. We have implemented our technique in a tool called Sherlock that finds real deadlocks in Java programs. For our benchmarks, our tool found almost twice as many real deadlocks as four previous techniques combined. Our technique is particularly good at finding deadlocks that happen after many execution steps: we found 33 deadlocks that happened after more than one million computation steps, including 27 new deadlocks. Our tool is fully automatic and its user needs no expertise on deadlocks. Once our tool reports a deadlock, our tool can replay the execution that leads to the deadlock.

In summary, the two main contributions of this paper are:

- an effective and easy-to-use tool for dynamic deadlock detection and
- a large-scale experimental comparison of seven deadlock detectors.

The rest of the paper. In the following section we present our approach and in Section 3 we present the key innovation that makes our approach work. In Section 4 we present our experimental results, in Section 5 we discuss limitations, in Section 6 we discuss related work.

2. OUR TECHNIQUE

Overview. In a nutshell, we first produce a set of deadlocks candidates and then we do a separate search for each of the deadlock candidates. The key idea is to turn each search for a deadlock into a search for a schedule that leads to the deadlock. We structure those searches in a particular manner that Eslamimehr and Palsberg used in their work on data race detection [19] and that we illustrate in Figure 1. Each circle in Figure 1 is a schedule. The search is an alternating sequence of execute and permute steps:

$$(\mathsf{execute} \cdot \mathsf{permute})^i \cdot \mathsf{execute}$$

where i is a nonnegative integer. The execute function attempts to execute a given schedule and determine whether it leads to a deadlock, and the permute function permutes a given schedule. The search begin with an initial schedule found simply by executing the program. The search fails if execute cannot execute a given schedule, if permute cannot find a better permutation, or if the search times out. Our key innovation is a permute function that works well for deadlock detection.

Each call to execute may produce a more promising schedule, after which a call to permute will further improve that schedule. In more detail, each call to execute will both try to execute the given schedule *and* continue execution beyond that schedule, typically until termination of the program. Part of the continued execution may make progress towards the desired deadlock. The call to permute will permute the events in the schedule to make the next call to execute have a better chance to succeed.

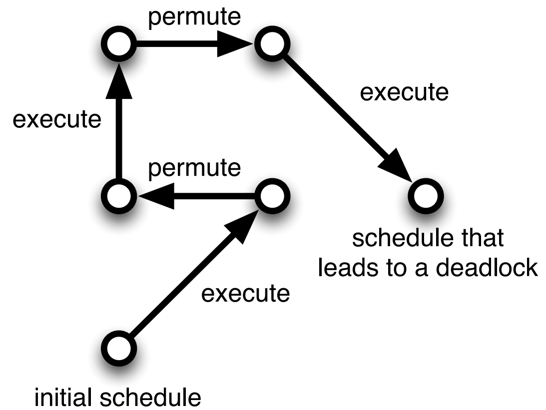

Figure 1: An illustration of the basic technique.

The alternation of permute and execute steps is more powerful than either one alone. For our benchmarks, our technique finds 146 deadlocks, while execute alone finds only 63 deadlocks, and permute alone finds only 22 deadlocks.

Eslamimehr and Palsberg's work on data race detection [19] showed how to implement *execute* via a series of concolic executions, as we will summarize below. In Section 3 we show how to define a *permute* function that successfully helps to find deadlocks.

Data types. We use these data types in Sherlock:

$$\begin{array}{rcl}
\mathsf{Program} & = & \text{a Java 6 program} \\
\mathsf{Input} & = & \text{input to a Java 6 program} \\
\mathsf{Lock} & = & \text{a Java 6 object} \\
\mathsf{Event} & = & \mathsf{threadId} \times \mathsf{statementLabel} \\
\mathsf{Schedule} & = & \mathsf{Event\ sequence} \\
\mathsf{Link} & = & \mathsf{threadId} \times (\mathsf{statementLabel} \times \mathsf{Lock}) \\
 & & \quad \times (\mathsf{statementLabel} \times \mathsf{Lock}) \\
\mathsf{Cycle} & = & \mathsf{Link\ set} \\
\mathsf{Deadlock} & = & \mathsf{Cycle} \times \mathsf{Input} \times \mathsf{Schedule}
\end{array}$$

Sherlock works for Java 6 programs, which have the type Program. The input to such programs is a vector of values; we use Input to denote the type of input vectors. Each object in Java contains a lock; for simplicity we refer to each object as a lock and use Lock to denote the type of locks.

We have one threadId for each program point that creates a thread. Notice that one threadId may cover multiple dynamic threads. When a program execution executes a particular statement in a thread with a particular threadId, we refer to that as an *event* that has type Event.

The standard notion of *schedule* is here the data type Schedule, which is a sequence of events.

In the context of deadlock detection, two key data types are Link and Cycle. We use Link to describe that a thread in a particular statement has acquired a lock and now wants to acquire another lock. We use Cycle, which is a set of links, to describe a deadlock.

A Deadlock is the type of information that we need to replay an execution that leads to a deadlock. A Deadlock has three components, namely the Cycle that is the deadlock, the Input that we should supply at the beginning of the execution, and the Schedule that the execution should follow to reach the deadlock.

Deadlock candidates. Our technique relies on access to a set of deadlock candidates. We use Havelund's technique GoodLock [29] to produce 1275 deadlock candidates for our benchmarks of more than 4.5 million lines of code. Those 1275 deadlock candidates are an excellent starting point for our search. GoodLock combines model checking and dynamic analysis into an efficient deadlock detector that can produce both false positives and false negatives. Here is the interface to GoodLock:

$$\textsf{GoodLock} : \quad \textsf{Program} \rightarrow (\textsf{Cycle set})$$

We use GoodLock as a "black box", that is, as an unmodified component for which we rely only on its input-output behavior. Notice that GoodLock maps a Java program to a set of eventSets, that is, a set of deadlock candidates. We use an extension of GoodLock that can handle deadlocks of any number of threads [1]. Havelund reported that deadlocks that involve three or more threads are extremely rare in practice, and indeed for our benchmarks GoodLock found only deadlock candidates that involve two threads.

The InitialRun function. Here is the interface to the initialRun function:

$$\textsf{initialRun} \;:\; \textsf{Program} \rightarrow \textsf{Schedule}$$

A call to initialRun simply executes the program with some particular input and records the schedule. Our benchmarks are drawn from open source repositories and each one comes with a specific input. For each benchmark, we use the predetermined input in initialRunbecause those inputs appear to exercise the code well. Alternatively, we could pick some other input (for example, at random). We leave to future work to investigate whether the effectiveness of our approach would be significantly affected by the quality of the inputs.

The Execute function. Here is the interface to the *execute* function:

$$\begin{aligned}\textsf{execute} : \quad & (\textsf{Program} \times \textsf{Schedule} \times \textsf{Cycle}) \rightarrow \\ & ((\textsf{Input} \times \textsf{Schedule} \times \textsf{boolean}) \oplus \{\textsf{none}\})\end{aligned}$$

The arguments to execute are a program, a schedule, and a deadlock candidate. A call to execute will attempt to execute the given schedule, determine whether it leads to a deadlock, and try to execute a longer schedule that contains the events embodied in the deadlock candidate. Consider the call:

$$(a, \mathit{trace}, \mathit{found}) = \textsf{execute}(p, s, c)$$

Here, *found* is a boolean that is true if the given schedule s leads to a deadlock and that is false otherwise. If *found* is true, then a is the input to the program that was used to execute the schedule. Additionally, *trace* is the schedule that was actually executed.

The implementation of execute uses concolic execution [41, 28, 8, 9, 56, 55]. Our explanation of execute is in two steps: first we summarize the idea of concolic execution and then we explain the implementation of execute.

Consider the sequential program

$x = 6$; if $(y > 4)$ { s }

which has input y, and which contains a statement s. How can we find an input that leads to execution of s? A good answer is to use *directed testing* [28] that executes the program multiple times with different inputs and each time hopefully gets closer to execute s. The insight of directed testing is to use information from one execution to generate a more promising input to the next execution. Specifically, we need a listing of all the assignments and conditions (and similar constructs) encountered. For example, for the program above, suppose the first run uses input $y = 0$. The execution encounters one assignment $x = 6$ and one condition $y > 4$. We can read those as constraints and form the conjunction $(\ x = 6 \ \wedge \ y > 4\ )$. The last condition encountered ($y > 4$ in this case) led us *off* the path towards s. To get a more promising input, we solve the constraints and might get the solution $\{\ x = 6, \quad y = 10\ \}$. Here we see that we can try input $y = 10$, which indeed leads to execution of s. Such an execution that records constraints is called a *concolic execution*. The word concolic stems from that the execution is both *conc*rete (it executes as usual) and symb*olic* (it record constraints). If the statement s is nested deeply, we may need many concolic executions before we either find an input that leads to execution of s or we give up.

Our implementation of execute uses Eslamimehr and Palsberg's generalization [19] of directed testing to a concurrent program and a sequence of events. The idea is to find an input that leads to execution of all of the events in the sequence in order. We can do that by iterating the above idea and by controlling the thread scheduler. Each iteration leads to execution of one of the events and the next iteration takes the constraints from the previous iteration as its starting point. We control the thread scheduler to ensure that we execute the events in the right order. If we can match the event sequence, then we continue exploration in an attempt to execute as many of the events embodied in the deadlock candidate as possible. However, if we cannot

```
(Deadlock set) Sherlock(Program p) {
  (Cycle set) candidates = GoodLock(p)
  Schedule s0 = initialRun(p)
  (Deadlock set) dlocks = ∅

  for each Cycle c ∈ candidates do {
    boolean found = false
    boolean stalled = false
    int i = 0
    Schedule s = s0
    while (¬ found) ∧ (¬ stalled) ∧ (i ≤ 1000) {
      case execute(p, s, c) of
        (Input × Schedule × boolean) (a, trace, true) : {
          dlocks = dlocks ∪ {(c, a, trace)}
          found = true
        }
        (Input × Schedule × boolean) (a, trace, false) : {
          case permute(trace, c) of
            Schedule s′ : {s = s′}
            none : {stalled = true}
        }
        none : {stalled = true}
      }
      i = i + 1
    }
  }

  return dlocks
}
```

Figure 2: Sherlock.

match the event sequence, then execute returns none, which may happen for a variety of reasons including nondeterminism that stems from system calls and external events.

The Permute function. Here is the interface to the permute function, which maps a schedule and a deadlock candidate to a better schedule or else to none if no better schedule was found:

$$\text{permute}: \quad (\text{Schedule} \times \text{Cycle}) \rightarrow (\text{Schedule} \oplus \{\text{none}\})$$

In the following section we describe permute in detail.

Sherlock pseudo-code. Figure 2 shows pseudo-code for Sherlock, which we will go over in detail. We hope our pseudo-code and explanation will enable practitioners to implement our technique easily.

The input to the Sherlock procedure is a program while the output is a set of real deadlocks. The first three lines of Sherlock declares these three variables: (1) a set of deadlock candidates, called *candidates*, that we initialize by a call to GoodLock, (2) a schedule, called s_0, that we initialize to the trace produced by an initial run of the program, and (3) a set of deadlocks, called *dlocks*, that initially is the empty set and that we eventually return as the result of the procedure.

The main body of the pseudo-code consists of a for-each-loop that tries each of the event sets in the set of candidates. The body of the for-each loop declares these four variables: (1) a boolean *found* that tells whether we have found a schedule that leads to the desired deadlock, (2) a boolean *stalled* that tells whether permute was able to improve a given schedule and whether execute was able to match the trace and execute a longer trace with the events embodied in the deadlock candidate, (3) an integer i that counts the number of pairs of calls to permute and execute, and (4) a schedule, called s, that we initialize to s_0. For each deadlock candidate we use a while-loop to do an alternation of calls to execute and permute, as illustrated in Figure 1. Intuitively, the while-loop terminates if either we find the deadlock, we give up, or we time out. The time-out condition ($i \leq 1000$) was never exercised in our experiments; the highest number of iterations of the while-loop for our benchmarks was 726.

In the body of the while-loop, we first call execute to match the given schedule, after which either we declare success, or proceed with a call to permute, or abandon the search. Similarly, after the call to permute, we either continue with the next iteration of the while-loop or we abandon the search. Notice how each iteration of the while-loop begins with s, extends it to *trace* and then improves it to a new value of s.

If we find a deadlock, then we record the input and the trace that lead to the deadlock. If we abandon the search, then the deadlock candidate may still be a real deadlock.

Example. We now present an example in which we walk through a run of Sherlock on the following program with four shared variables and two threads:

A, B are shared variables that contain objects
x, y are shared variables that contain integers
y has an initial value received from user input

Thread 1:	Thread 2:
l_1: $x = 6$	l_5: $x = 2$
l_2: synchronized(A) {	l_6: synchronized(B) {
l_3: if $(y > 4)$ {	l_7: if $(y^2 + 5 < x^2)$ {
l_4: synchronized(B) { }	l_8: synchronized(A) { }
}	}
}	}

The example is a refined version of the example in Section 1: we have added two assignments and two if-statements. The point of the example is that the program enters a deadlock only when it executes the bodies of both if-statements. For a deadlock to happen, y must be 5 and the program must execute a particular schedule that lets x be 6 at the time the program evaluates the condition at l_7. So, while a deadlock is possible, most executions are deadlock free. We will explain how our technique finds the deadlock.

We use these abbreviations for events: $e_1 = (1, l_1)$, $e_2 = (1, l_2)$, $e_3 = (1, l_3)$, $e_4 = (1, l_4)$, $e_5 = (2, l_5)$, $e_6 = (2, l_6)$, $e_7 = (2, l_7)$, $e_8 = (2, l_8)$.

The call to GoodLock produces a single deadlock candidate, namely the following cycle, which in the for-each loop will be called c:

$$c = \{ (\text{Thread } 1, (l_2, A), (l_4, B)), \ (\text{Thread } 2, (l_6, B), (l_8, A)) \}$$

Now we do an initial run of the program. Suppose that the initial input, which becomes the value of the shared variable y, is 0. We get

$$s = e_1, e_2, e_3, e_5, e_6, e_7$$

Now we run the first iteration of the while-loop. First we run execute which matches the schedule and finds out that with input $y = 5$, it can add the event e_4. So we have:

$$trace = e_1, e_2, e_3, e_5, e_6, e_7, e_4$$

The call to permute on *trace* gives:

$$s = e_5, e_6, e_7, e_1, e_2, e_3, e_4$$

Now we run the second iteration of the while-loop. The call to execute matches the schedule with input $y = 5$ so we have:

$$trace = e_5, e_6, e_7, e_1, e_2, e_3, e_4$$

The call to permute on *trace* gives:

$$s = e_5, e_6, e_1, e_2, e_7, e_3, e_4$$

Now we run the third iteration of while-loop. The call to execute matches the schedule with input $y = 5$, adds the event e_8, and enters a deadlock. The schedule is:

$$trace = e_5, e_6, e_1, e_2, e_7, e_3, e_4, e_8$$

Our key innovation is permute, which we explain next.

3. OUR PERMUTE FUNCTION

Our permute function combines ideas from static analysis and dynamic analysis.

Background: dynamic race detection. Many researchers have studied how to extract information from execution traces. A pinnacle of this area is the paper by Serbanuta, Chen, and Rosu [57] that presented a sound and maximal model of execution traces: it subsumes all other sound models that rely solely on information from an execution trace. They also showed how to use the model to do dynamic race detection. Their race detector works in two steps: first run the program to get a trace, then find an executable permutation of the trace that leads to a race. Their model helps guarantee that the chosen permutation is executable.

As shown later by Said, Wang, Yang, and Sakallah [52], one can phrase the problem to find an executable permutation of a trace as a constraint-solving problem, and one

$$\alpha_\pi = [\bigwedge_{t=1}^{T} o_{e_1^t.idx} < \ldots < o_{e_n^t.idx}] \wedge [\bigwedge_{e \in FORK} o_{e.idx} < o_{(e.val).first.idx}] \wedge [\bigwedge_{e \in JOIN} o_{(t_{e.val}).last.idx} < o_{e.idx}]$$

$$\beta_\pi = \bigwedge_{e \in \pi \wedge e.type=read} \left(\begin{array}{l} \left(\bigwedge_{(e.tiwp=null) \wedge (e.val=e.var.init)} \bigwedge_{e1 \in e.pws} o_{e.idx} < o_{e1.idx}\right) \vee \\ \left(\bigvee_{e1 \in e.pwsv} \bigwedge_{e2 \in e.pws \wedge e2 \neq e1} (o_{e1.idx} < o_{e.idx}) \wedge (o_{e.idx} < o_{e2.idx} \vee o_{e2.idx} < o_{e1.idx})\right) \end{array} \right)$$

$$\Psi_{(V,E)} = \bigwedge_{(e_i,e_j) \in E} o_i < o_j$$

$$\delta_c = (o_{i_1} < o_{j_2}) \wedge (o_{i_2} < o_{j_1}) \quad \text{where } c = \{ (t_1, (l_{i_1}, L_A), (l_{j_1}, L_B)), \ (t_2, (l_{i_2}, L_B), (l_{j_2}, L_A)) \}$$
$$\text{and } e_{i_1} = (t_1, l_{i_1}) \wedge e_{j_1} = (t_1, l_{j_1}) \wedge e_{i_2} = (t_2, l_{i_2}) \wedge e_{j_2} = (t_2, l_{j_2})$$

Figure 3: Constraints for our permute function

can use an SMT-solver to produce that permutation. In essence, Said et al. presented a permute function that works well for race detection. Eslamimehr and Palsberg [19] combined Said et al.'s permute function with concolic execution and thereby obtained an efficient and useful dynamic race detector. We will present a permute function that works well for deadlock detection.

A static characterization of potential deadlocks. Deshmukh, Emerson, and Sankaranarayanan [15] presented a static analysis of library code that identifies potential deadlocks. Their analysis delivers a library interface that describes how to call library functions with deadlock-safe alias relationships among library objects. In outline, their approach has two steps.

First, from the text of a library, their static analysis builds a lock-order graph and a representation of alias information. The lock-order graph describes the order in which the code acquires locks. For example, for the statement

$$\text{synchronized}(A) \ \{ \ \text{synchronized}(B) \ \{ \ \ldots \ \}\}$$

the graph contains an edge from a node "synchronized(A)" to a node "synchronized(B)".

Second, from the lock-order graph and the alias information, they derive constraints and show that the constraints are solvable if and only if the lock-order graph is acyclic. In other words, the constraints are solvable if and only the library code cannot deadlock.

They use an SMT-solver to solve the constraints. We will use their approach to handle lock orders in our definition of permute.

A memory-less Permute function for deadlock. Now we give an overview of a baseline version of our permute function that we call the *memory-less* permute function; later we give a constraint-based definition. Our memory-less permute function leads to a deadlock detector that finds 121 deadlocks in our benchmarks, which is already better than the previous dynamic techniques with which we compare. At the end of this section, we present an enhanced permute function that leads us to find an additional 25 deadlocks.

Our memory-less permute function combines aspects of Said et al.'s permute function [52] with aspects of Deshmukh et al.'s static analysis [15] and a constraint that encodes a deadlock candidate. Let us now explain the key observation that makes the combination work.

Said et al. generates a constraint that at the top level has two conjuncts: 1) a constraint that guarantees that the permutation of a trace will be sequentially consistent, and 2) a constraint that represents a data race. We replace (2) with a representation of deadlock candidate; let us now take a closer look at (1). The constraint about sequential consistency has three conjuncts that represent that the permuted trace must: 1.1) preserve the *happens-before relation* for each thread, 1.2) satisfy *write-read consistency*, and 1.3) satisfy *synchronization consistency*. Write-read consistency means that a read event must read the value written by the most recent write event to that location, and synchronization consistency means that the permuted trace is consistent with the semantics of the synchronization events.

Our observation is that we can use (1.1) and (1.2), and then replace (1.3) with the Deshmukh et al.'s lock-order constraints. Intuitively, we replace dynamic information about synchronization and lock order from *a single trace* with static lock-order information about *the entire program*. The whole-program view of lock order makes our permute function efficient and powerful.

The grand total is a constraint that consists of the constraints (1.1) and (1.2) from Said et al., Deshmukh et al.'s lock-order constraints, and a representation of a deadlock candidate. This constraint, if solvable, represents a permuted trace. If the input trace contains all the events embodied in the deadlock candidate, and permuted trace is executable, then the execution leads to the deadlock. We use an SMT-solver to solve the constraint, and, as explained earlier, right after the call to permute, we run execute on the permuted trace to find out whether it is executable.

Constraints. Now we give full details of the constraints that we use in our permute function. Suppose we have a program, a trace $\pi = \langle e_1, \ldots, e_n \rangle$ of an execution of the program, a lock-order graph (V, E) produced by a standard interprocedural static analysis of the program [15], and a deadlock candidate c. The constraints use n position variables $o_1, \ldots, o_n$. The idea is that the value of o_i is the position of e_i in the permuted trace. The symbol $<$ denotes the happens-before relation. The constraints are of the form

$$\alpha_\pi \ \wedge \ \beta_\pi \ \wedge \ \Psi_{(V,E)} \ \wedge \ \delta_c \qquad (1)$$

where each of the four conjuncts is defined in Figure 3. Here α_π is Said et al.'s constraint (1.1), β_π is Said et al.'s constraint (1.2), $\Psi_{(V,E)}$ is Deshmukh et al.'s lock-order constraints, and δ_c is a representation of a deadlock candidate. Figure 3 uses helper functions that we explain below.

The constraint (1) is formed by conjunctions and disjunctions of inequalities of the form $(o_i < o_j)$. A solution to the

constraints is an injective function

$$S \quad : \quad \{o_1, \ldots, o_n\} \to \{1, \ldots, n\}$$

First we explain α_π. We use the same notation as Said et al. [52]. In particular, for a trace π and a threadId t, we let $\langle e_1^t, \ldots, e_n^t$ be subsequence of t-events in π, and we let $t.first$ denote e_1^t and we let $t.last$ denote e_n^t. We assume that the set of threadIds is $1..T$. For each event e in π, we use $e.idx$ to denote its index in π, we use $e.type$ to denote the event type, which ranges over $\{read, write, fork, join, acquire, release\}$, and we use $e.tid$ to denote the threadId. Additionally, we use $e.var$ to denote either (in read or write) a shared variable, or (in fork or join) a synchronization object. Further, we use $e.val$ to denote either (in read or write) a concrete value, or (in fork or join) the child threadId. $FORK$ denotes the set of fork events in π, and similarly $JOIN$ denotes the set of join events in π.

The first conjunct of α_π expresses that the happens-before relation per thread must be preserved. The second conjunct of α_π expresses that a fork event happens before the first event of the forked thread. The third conjunct of α_π expresses that the last event of a thread happens before the thread participates in a join event.

Next, we explain β_π. We use the same notation as Said et al. [52]. In particular, for a read event e, we use $e.tiwp$ to denote the *thread immediate write predecessor*, which is a write event that comes before e in π that has the same threadId as e, has the same variable as e, and for which no other such write event occurs in π between $e.tiwp$ and e. If no such write event exists, then we write $e.tiwp = null$. Additionally, for a read event e, we use $e.liwp$ to denote the *linearization immediate write predecessor*, which is a write event that comes before e in π that has a possibly different threadId than e, has the same variable as e, and for which no other such write event occurs in π between $e.liwp$ and e. If no such write event exists, then we write $e.val = e.var.init$, where $e.var.init$ is the initial value of variable $e.var$. Finally, for a read event e, we use $e.pws$ to denote the *predecessor write set*, which is the set of write events e' such that $e'.var = e.var$ and either $e'.tid \neq e.tid$ or both $e'.tid = e.tid$ and $e' = e.tiwp$. We use $e.pwsv$ to denote the subset of $e.pws$ for which for each element $e' \in e.pwsv$ we have $e'.val = e.val$.

The constraint β_π has a conjunct for each read event in π. Each of the conjuncts is a disjunction of two disjuncts, one per line of the definition of β_π. The first disjunct says that if the read event doesn't have thread immediate write predecessor, then the value read is the initial value and all writes to that variable come after that read in π. The second disjunct says that if π contains a linearization immediate write predecessor, then every possible write event $e1$ for the value read must come before the read event, and no other such write event $e2$ can come between $e1$ and the read event.

Next, let us explain $\Psi_{(V,E)}$. Following Deshmukh [15] we first use a standard interprocedural analysis to compute a lock-order graph (V, E) for the entire program. The nodes V are events that acquire locks, and the edges E express nested relationships between the nodes: $(e_1, e_2) \in E$ if and only if the body of e_1 contains e_2. Now $\Psi_{(V,E)}$ is a conjunction of one constraint per edge $(e_i, e_j) \in E$, namely $(o_i < o_j)$, that expresses that e_i must happen before e_j.

Finally, let us explain how δ_c represents a deadlock candidate c with two links in the cycle. The definition and explanation generalize easily to cycles with more than two links though we haven't found any programs with such deadlocks. The deadlock candidate can be understood as a collection of the four events $e_{i_1}, e_{j_1}, e_{i_2}, e_{j_2}$ listed in Figure 3. A precondition for the deadlock candidate to be a real deadlock is that e_{i_1} and e_{i_2} must both happen before both of e_{j_1} and e_{j_2}. We have that $\Psi_{(V,E)}$ contains $(o_{i_1} < o_{j_1})$ and o_{i_2}, o_{j_2}, so to ensure that the precondition is met, we let δ_c be $(o_{i_1} < o_{j_2}) \land (o_{i_2} < o_{j_1})$. In case some or all of the four events occur multiple times in the trace, we let the candidate refer to the *first* occurrence of each event in the trace.

Example. Let us return to the example from Section 2 and explain details of the call to permute in the first iteration of the while-loop. That call is permute($trace, c$) where

$$trace = e_1, e_2, e_3, e_5, e_6, e_7, e_4$$

and c is the deadlock candidate:

$$c = \{\ (\text{Thread } 1, (l_2, A), (l_4, B)),\ (\text{Thread } 2, (l_6, B), (l_8, A))\ \}$$

Here are the constraints used by the permute function. First we list α_{trace}, which preserves the happens-before relation for each thread:

$$o_1 < o_2 \ \land \ o_2 < o_3 \ \land \ o_3 < o_4 \ \land \ o_5 < o_6 \ \land \ o_6 < o_7$$

Next we list β_{trace}, which ensures write-read consistency:

$$o_5 < o_7$$

Next we list $\Psi_{(V,E)}$ which represents Deshmukh et al.'s lock-order constraints:

$$o_2 < o_4 \ \land \ o_6 < o_8$$

Finally, δ_c encodes the deadlock candidate c:

$$o_2 < o_8 \ \land \ o_6 < o_4$$

One possible solution is:

$$s = e_5, e_6, e_7, e_1, e_2, e_3, e_4$$

which ignores the constraints that involve e_8 because e_8 doesn't occur in *trace*. So, we can return s as the result of the call to permute in the first iteration of the while-loop.

An enhanced Permute function for deadlock. The full version of our permute function has "memory" and takes advantage of the schedules that have been submitted in all previous calls. The idea is to use the schedules that have been submitted earlier to relax the happens-before relation. We do the relaxation by taking the union of the happens-before relations from all those schedules. The result is a constraint system that is more likely to be satisfiable and that leads us to find 25 more deadlocks in our benchmarks.

One final enhancement of our permute function is based on partial order reduction. The issue is that permute might produce a permuted trace that is semantically equivalent with the input trace and therefore must fail to lead to the deadlock candidate. We use Flanagan and Godefroid's approach [22] to partial order reduction to avoid such a situation.

Our implementation uses Flanagan and Godefroid's approach as a checker that determines whether an input trace and the permuted trace are equivalent. In case the input trace and the permuted trace are equivalent, we repeatedly ask permute for a different output until we get one we want.

4. EXPERIMENTAL RESULTS

Our implementation of GoodLock is an extension of Java PathFinder [30]. In our implementation, events are at the Java bytecode level. We use Soot [62] to instrument bytecodes to implement `execute` on top of the Lime concolic execution engine; `http://www.tcs.hut.fi/Software/lime`. We ran all our experiments on a Linux CentOs machine with two 2.4 GHz Xeon quad core processors and 32 GB RAM.

4.1 Benchmarks

Figure 4 lists our 22 benchmarks which we have collected from six sources:

- From ETH Zurich [64]: Sor, TSP, Hedc, Elevator.
- From `java.util`, Oracle's JDK 1.4.2: ArrayList, TreeSet, HashSet, Vector.
- From Java Grande [50]: RayTracer, MolDyn, MonteCarlo.
- From the Apache Software Foundation [26]: Derby.
- From CERN [24]: Colt.
- From DaCapo [4]: Avrora, Tomcat, Batic, Eclipse, FOP, H2, PMD, Sunflow, Xalan.

The sizes of the benchmarks vary widely: we have 2 huge (1M+ LOC), 10 large (20K–1M LOC), 8 medium (1K–8K LOC), and 2 small (less than 1K LOC) benchmarks.

Figure 4 also lists the high watermark of how many threads each benchmark runs, and the input size in bytes for each benchmark. Most of the benchmarks come with a specific input, except the four benchmarks from Oracle's JDK 1.4.2 for which we use a test harness from previous work [36, 19].

Each benchmark is supposed to terminate so every real deadlock is a bug.

4.2 Deadlock Detectors

We compare Sherlock with one static deadlock detector, namely Chord [46, 45], one hybrid deadlock detector that we call GoodLock [29], and four dynamic deadlock detectors, namely DeadlockFuzzer [36], IBM ConTest [16, 20], Jcarder [17], and Java HotSpot [48]. Additionally we compare with a combined dynamic technique that we call DIJJ.

Chord is a static technique, and by design it may report false positives; its main objective is to report all real deadlocks (or as many as possible).

GoodLock monitors an execution, computes a lock dependency relation, and uses the transitive closure of this relation to suggest potential deadlocks.

DeadlockFuzzer, IBM ConTest, Jcarder, Java HotSpot, and Sherlock are all dynamic techniques that report only real deadlocks.

DeadlockFuzzer begins with a set of deadlock candidates produced by a variant of GoodLock. For each deadlock candidate, DeadlockFuzzer executes the program with a random scheduler that is biased towards executing the events in the deadlock candidate. The idea to use a random scheduler for Java can be traced back to Stoller [59].

IBM ConTest uses heuristics to perturbate the schedule and thereby hopefully reach a deadlock. One of the techniques is to insert time-outs.

Jcarder instruments Java byte code dynamically and looks for cycles in the graph of acquired locks. The instrumented code records information about the locks at run time. A later, separate phase of Jcarder post-processes the recorded information to search for deadlocks.

The Java HotSpot Virtual Machine from Oracle can track the use of locks and detect cyclic lock dependences. The utility detects Java-platform-level deadlocks, including locking done from the Java Native Interface (JNI), the Java Virtual Machine Profiler Interface (JVMPI), and Java Virtual Machine Debug Interface (JVMDI).

We use DIJJ to stand for the union of DeadlockFuzzer, IBM ConTest, Jcarder, and Java HotSpot in following sense. We can implement DIJJ as a tool that for a given benchmark starts runs of DeadlockFuzzer, IBM ConTest, Jcarder, and Java HotSpot in four separate threads, and if any one of them reports a deadlock, then DIJJ reports a deadlock.

4.3 How we handle Reflection

Many of the benchmarks use reflection, and IBM ConTest and Java HotSpot handle reflection well. We enable the other deadlock detectors to handle reflection with the help of the tool chain TamiFlex [5]. The core of the problem is that reflection is at odds with static analysis and bytecode instrumentation: reflection may make make static analysis unsound and may load uninstrumented classes. TamiFlex solves these problems in a manner that is sound with respect to a set of recorded program runs. If a later program run deviates from the recorded runs, TamiFlex issues a warning.

We have combined each of Chord, GoodLock, DeadlockFuzzer, and Jcarder with TamiFlex and we have run all our experiments without warnings. As a result, all the deadlock detectors all handle reflection correctly.

4.4 Measurements

Figure 5 shows the numbers of deadlocks found in 22 benchmarks by 7 techniques. When we compare deadlocks, we focus on the program points where locks are acquired. For each benchmark and each tool, the reported deadlocks turns out to be disjoint, that is, any two deadlocks have nonoverlapping program points. Even across tools, we found no cases of partially overlapping deadlocks; each pair of reported deadlocks are either identical or disjoint. As a result we can easily compare tools. We have manually inspected all the deadlocks reported by the dynamic tools and we believe that the deadlocks are rather unrelated. While one can imagine that a code revision may remove multiple deadlocks, we found no obvious signs of correlation between the reported deadlocks.

Figure 6 shows the time each of the runs took in minutes and seconds, and it shows the geometrical mean for each technique. We made no attempt to throttle the amount of time that the tools can use.

4.5 Evaluation

We now present our findings based both on the measurements listed above and on additional analysis of the deadlocks that were found.

Sherlock versus other dynamic techniques. We can see in Figure 5 that Sherlock finds the most deadlocks (146) of all the dynamic techniques. Among those 146 deadlocks, 86 deadlocks were found only by Sherlock and are entirely novel to this paper, while 60 were also found by DIJJ. Du-

Name	LOC	# threads	input size (bytes)	Brief description
Sor	1270	5	404	A successive order-relaxation benchmark
TSP	713	10	58	Traveling Salesman Problem solver
Hedc	30K	10	220	A web-crawler application kernel
Elevator	2840	5	60	A real-time discrete event simulator
ArrayList	5866	26	116	ArrayList from `java.util`
TreeSet	7532	21	64	TreeSet from `java.util`
HashSet	7086	21	288	HashSet from `java.util`
Vector	709	10	128	Vector from `java.util`
RayTracer	1942	5	412	Measures the performance of a 3D raytracer
MolDyn	1351	5	240	N-Body code modeling dynamic
MonteCarlo	3619	4	26	A financial simulator, using Monte Carlo techniques to price products
Derby	1.6M	64	564	Apache RDBMS
Colt	110K	11	804	Open source libraries for high perf. scientific and technical computing
Avrora	140K	6	74	AVR microcontroller simulator
Tomcat	535K	16	88	Tomcat Apache web application server
Batic	354K	5	366	Produces Scalable Vector Graphics images based on Apache Batic
Eclipse	1.2M	16	206	Non-GUI Eclipse IDE
FOP	21K	8	34	XSL-FO to PDF converter
H2	20K	16	658	Executes a JDBCbench-like in-memory benchmark
PMD	81K	4	116	Java Static Analyzer
Sunflow	108K	16	24	Tool for rendering image with raytracer
Xalan	355K	9	616	XML to HTML transformer
TOTAL	4587K			

Figure 4: Our benchmarks.

ally, 15 deadlocks were found only by DIJJ. In summary, we have that the combination of DIJJ and Sherlock found 161 deadlocks in the 22 benchmarks.

Found only by DIJJ:	15
Found by both:	60
Found only by Sherlock:	86
Total:	161

Let us consider the 15 deadlocks that DIJJ found but Sherlock missed. Those deadlocks were in ArrayList (4), TreeSet (3), Vector (1), Derby (1), Tomcat (3), Eclipse (2), PMD (1). DeadlockFuzzer found eleven of those, and IBM ConTest found the remaining four (and also four of the eleven found by DeadlockFuzzer).

For example, DeadlockFuzzer found the following deadlock in Tomcat, while Sherlock missed it. The deadlock happens when Tomcat uses OracleDataSourceFactory. The nature of the deadlock is much like the example in Section 1. If we use the notation of that example, then A is an object of class java.util.Properties, while B is an object of class java.util.logging.Logger. Two threads execute synchronized-operations on those objects in the pattern of the example in Section 1, hence they may deadlock.

The reason why DeadlockFuzzer found deadlocks that Sherlock missed is that DeadlockFuzzer uses a random scheduler while the initial run of Sherlock uses the standard scheduler.

We conclude that Sherlock finds the most deadlocks, and that DeadlockFuzzer and IBM ConTest remain worthwhile techniques that each finds deadlocks that the other dynamic techniques don't find.

DIJJ details. The combined dynamic technique DIJJ found 75 deadlocks. We notice that, intuitively:

$$\text{Jcarder} \subseteq (\text{DeadlockFuzzer} \cup \text{IBM ConTest})$$

In words, if Jcarder finds a deadlock, then DeadlockFuzzer or IBM ConTest (or both) also finds that deadlock. We also notice that if Java HotSpot finds a deadlock, then either DeadlockFuzzer or IBM ConTest (or both) also finds that deadlock or the deadlock is one particular deadlock in Elevator (which Sherlock finds too).

Chord. Chord is one of the best static deadlock detectors, yet our experiments suggest that Chord produces a large number of false positives. Additionally, Chord missed five real deadlocks, namely one deadlock in each of Elevator, Vector, Raytracer, Batic, and Xalan. We conclude that accurate static deadlock detection remains an open problem.

Timings. The geometrical means of the execution times show that DeadlockFuzzer is the fastest dynamic technique while Sherlock is the slowest. The timings for DeadlockFuzzer and Sherlock include the time to execute GoodLock.

Number of schedules. The number of calls to execute turns out to be rather modest: for every benchmark, it is at most twice the number of deadlock candidates. This shows that the combination of execute and permute is powerful.

Number of steps of execution. This table shows the lengths of the 146 schedules that lead to deadlocks found by Sherlock, including the 86 found only by Sherlock:

	Sherlock		
schedule length	total =	new +	DIJJ
$10^2 - 10^3$	5	0	5
$10^3 - 10^4$	20	9	11
$10^4 - 10^5$	39	12	27
$10^5 - 10^6$	49	38	11
$10^6 - 10^7$	24	18	6
$10^7 - 10^8$	9	9	0
	146	86	60

The schedules can be as long as 34 million events, which

	Static	Hybrid	Dynamic							
benchmarks	Chord	GoodLock	DeadlockFuzzer	IBM ConTest	Jcarder	Java HotSpot	DIJJ	Sherlock		
								total	= new	+ DIJJ
Sor	1	7	0	0	0	0	0	1	1	0
TSP	1	9	0	0	0	0	0	1	1	0
Hedc	24	23	1	0	0	0	1	20	19	1
Elevator	4	13	0	0	0	1	1	5	4	1
ArrayList	9	11	7	6	2	1	7	9	6	3
TreeSet	8	11	7	5	1	3	8	5	0	5
HashSet	11	10	3	1	0	2	5	5	0	5
Vector	3	14	0	1	0	0	1	4	4	0
RayTracer	1	8	0	1	0	0	1	2	1	1
MolDyn	3	6	1	1	1	1	1	1	0	1
MonteCarlo	2	23	0	1	1	1	1	2	1	1
Derby	5	10	2	0	0	0	2	4	3	1
Colt	6	11	0	0	0	0	0	3	3	0
Avrora	78	29	4	2	1	2	4	7	3	4
Tomcat	119	411	9	10	3	4	11	18	10	8
Batic	73	33	5	4	1	3	7	10	3	7
Eclipse	89	389	9	8	4	6	13	23	12	11
FOP	15	11	1	1	0	0	2	4	2	2
H2	25	17	0	1	0	0	1	3	2	1
PMD	20	8	2	2	0	1	3	4	2	2
Sunflow	31	11	1	2	0	2	2	6	4	2
Xalan	42	210	3	4	0	2	4	9	5	4
TOTAL	570	1275	55	50	14	29	75	146	86	60

Figure 5: The numbers of deadlocks found in 22 benchmarks by 7 techniques.

	Static	Hybrid	Dynamic				
benchmarks	Chord	GoodLock	DeadlockFuzzer	IBM ConTest	Jcarder	Java HotSpot	Sherlock
Sor	4:23	0:04	0:05	0:07	0:12	0:15	0:39
TSP	8:09	0:02	0:02	0:06	0:17	0:18	0:50
Hedc	20:11	0:04	0:06	0:08	0:19	0:23	0:44
Elevator	5:19	0:06	0:07	0:11	0:09	0:13	0:51
ArrayList	3:10	0:03	0:04	0:05	0:11	0:19	0:28
TreeSet	2:55	0:02	0:02	0:05	0:11	0:22	0:26
HashSet	2:47	0:04	0:05	0:06	0:10	0:14	0:35
Vector	5:31	0:03	0:03	0:07	0:12	0:17	0:19
RayTracer	4:22	0:02	0:03	0:04	0:19	0:09	0:30
MolDyn	5:34	0:05	0:08	0:12	0:24	0:23	0:49
MonteCarlo	4:48	0:05	0:05	0:13	0:15	0:17	1:02
Derby	46:17	0:12	0:18	0:19	0:48	0:55	1:25
Colt	15:58	0:08	0:13	0:14	0:13	0:20	0:31
Avrora	51:36	0:22	0:24	0:22	0:51	1:02	1:16
Tomcat	58:24	0:20	0:23	0:27	0:49	0:54	4:15
Batic	43:03	0:14	0:19	0:20	0:30	0:41	1:07
Eclipse	59:20	0:29	0:30	0:29	0:38	0:49	3:21
FOP	38:00	0:13	0:19	0:33	0:21	0:33	1:43
H2	27:19	0:10	0:14	0:29	0:29	0:40	0:57
PMD	45:05	0:07	0:10	0:08	0:19	0:23	0:53
Sunflow	39:12	0:16	0:18	0:21	0:32	0:52	1:46
Xalan	40:53	0:14	0:19	0:22	0:27	0:55	3:02
geom. mean	17:39	0:06	0:09	0:12	0:20	0:26	0:59

Figure 6: Timings in minutes and seconds.

shows that the permute method scales to long schedules. For each of seven benchmarks (Derby, Colt, Tomcat, Batic, Eclipse, Sunflow, Xalan), at least one real deadlock happens with a schedule that has more than a million events. Among the 33 deadlocks found after at least a million steps, 27 were found only by Sherlock. Given that Sherlock can reproduce every deadlock, we conclude that Sherlock does a much better job than previous work to find reproducible deadlocks than the previous techniques with which we have compared.

5. LIMITATIONS

Our approach has four main limitations.

First, our current implementation of Sherlock supports synchronized methods and statements, but has no support for other synchronization primitives such as wait, notify, and notify all. We leave support for those to future work.

Second, our approach relies on GoodLock to produce deadlock candidates. In case GoodLock misses a deadlock, so will Sherlock. Note here that GoodLock itself is a partly dynamic analysis that analyzes a particular execution. If we run GoodLock multiple times (perhaps with different inputs) we may in total get a larger set of deadlock candidates and miss fewer deadlocks.

Third, our approach relies on a constraint solver both in permute and execute. The form of constraints that we use in permute has a decidable satisfiability problem, while the form of constraints that we use in execute are derived from expressions in the program text and may be undecidable. So for execute, the power of the constraint solver is critical.

Fourth, our approach has no support for native code.

6. RELATED WORK

In Section 4, we discussed six techniques for deadlock detection, namely Chord [46, 45], GoodLock [29], DeadlockFuzzer [36], IBM ConTest [16, 20], Jcarder [17], and Java HotSpot [48] and we did a large-scale experimental comparison of all six and Sherlock. The goal of this section is to highlight some other notable techniques and tools in the area of deadlock detection for unannotated programs.

Run-time monitoring systems. Arnold, Vechev, and Yahav [2] presented the QVM run-time environment that continuously monitors an execution and potentially detects defects, including deadlocks. Huang, Zhang, and Dolby [34] presented an efficient approach to log execution paths and then do off-line computation in order to reproduce concurrency bugs such as deadlocks. Another idea is to let the operating system detect deadlocks [39]. All three approaches monitor executions but do nothing to drive an execution towards a deadlock.

Model checking. Demartini et al. [14] presented a translation from Java source code to Promela that enables deadlock detection via the SPIN model checker [31]. The translator predates Java 6 and would require significant extension to handle our benchmarks. Chaki et al. [13] and Godefroid [27] presented model checkers for C that can find deadlocks.

Static deadlock detectors. Static deadlock detectors [44, 43, 3, 18, 32, 63, 37, 61, 45] have a goal that is dual to our objective to find real deadlocks: they attempt to find *all* deadlocks and possibly some false positives. Chord remains one of the best among the scalable static deadlock detectors for Java to date, hence it was our choice for experimental comparison in this paper.

Dynamic deadlock detectors for Java. ASN [11] first extracts constraints from a deadlock candidate and formulates them as barriers, and then uses a form of random scheduling to trigger real deadlocks with high probability. ConLock [12] first does an initial run and generates scheduling constraint from the trace and from a deadlock candidate, and then uses a form of random scheduling that works within the limits of the generated scheduling constraints. Wolf [53] first does an initial run and does cycle detection, then prunes away cycles that cannot be executed, then generates a synchronization dependency graph, and finally uses a form of scheduling based on that graph.

The papers on ASN and ConLock report on experiments with a single Java benchmark of 36,300 lines of Java, for which the tools found 8 and 4 deadlocks, respectively. They also report on experiments with larger C/C++ benchmarks. The paper on Wolf reports on multiple benchmarks, the largest of which is 160,000 lines of Java and for which the tool found 6 deadlocks. None of those tools use schedule permutation or concolic execution.

Static analyses of traces. Our permute function embodies a static analysis of a trace. Researchers have presented many such analyses [21, 38, 25, 65] that might help define alternative permute functions.

Other dynamic techniques. Penelope [58] is a dynamic tool that detects atomicity violations. Penelope runs a program and then analyzes (with an SMT solver) the recorded reads and writes to predict a schedule that leads to an atomicity violation. Penelope doesn't use concolic execution. Recent papers [51, 33, 40] show how to detect data races in event-driven and reactive programs; their techniques might also be useful for deadlock detection in such programs.

Dynamic deadlock detectors for other languages. PCT (Probabilistic Concurrency Testing) [7] is a concurrency-bug (including deadlock) detector for C and C++ that uses a probabilistic technique to generate a thread schedule. They found one bug in each of eight benchmarks. Their largest benchmark is 245 thousand lines of code.

MagicFuzzer [10] is a deadlock detector for C and C++ that uses a variant of the technique used in DeadlockFuzzer. The novelty is that MagicFuzzer can confirm multiple cycles in the same run. For benchmarks of about 16 million lines of code, MagicFuzzer found 2 real deadlocks.

7. CONCLUSION

We have shown how to detect deadlocks by a combination of concolic execution and a novel approach to schedule permutation. The result is a scalable and useful deadlock detector. For a large benchmark suite, our tool Sherlock found 86 deadlocks that were missed by earlier techniques. Among those 86 deadlocks, about a third namely 27 deadlocks were found with schedules that have more than 1 million events. Our technique can find deadlocks after many steps of computation because the combination of concolic execution and schedule permutation helps drive an execution towards a deadlock candidate.

Our experiments show that DeadlockFuzzer, IBM ConTest, and Sherlock together find a total of 161 real deadlocks in 4.5 million lines of code. As far as we know, this is the most comprehensive list of real deadlocks for those benchmarks that is reported in the literature.

Acknowledgments. We thank the FSE reviewers for insightful comments that helped us improve the paper.

8. REFERENCES

[1] R. Agarwal, L. Wang, and S. D. Stoller. Detecting potential deadlocks with static analysis and runtime monitoring. In *PADTAD*, 2005.

[2] M. Arnold, M. Vechev, and E. Yahav. Qvm: An efficient runtime for detecting defects in deployed systems. In *OOPSLA, ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 143–162, 2008.

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

[4] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee Intel, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA'06, 21st annual ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 169–190, 2006.

[5] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *ICSE, 33rd International Conference on Software Engineering*, May 2011.

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

[7] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding concurrency bugs. In *ASPLOS, International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 167–178, 2010.

[8] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Proc. 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 443–446, 2008.

[9] Cristian Cadar, Paul Twohey, Vijay Ganesh, and Dawson Engler. Exe: A system for automatically generating inputs of death using symbolic execution. In *Proceedings of 13th ACM Conference on Computer and Communications Security*, 2006.

[10] Y. Cai and W. K. Chan. MagicFuzzer: Scalable deadlock detection for large-scale applications. In *ICSE'12, International Conference on Software Engineering*, pages 606–616, 2012.

[11] Y. Cai, C.J. Jia, S.R. Wu, K. Zhai, and W.K. Chan. ASN: a dynamic barrier-based approach to confirmation of deadlocks from warnings for large-scale multithreaded programs. *IEEE Transactions on Parallel and Distributed Systems*, 2014.

[12] Y. Cai, S. Wu, and W. K. Chan. ConLock: A constraint-based approach to dynamic checking on deadlocks in multithreaded programs. In *ICSE'14, International Conference on Software Engineering*, 2014.

[13] 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.

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

[15] Jyotirmoy Deshmukh, E. Allen Emerson, and Sriram Sankaranarayanan. Symbolic deadlock analysis in concurrent libraries and their clients. In *Proceedings of ASE'09, IEEE International Conference on Automated Software Engineering*, pages 480–491, 2009.

[16] Orit Edelstein, Eitan Farchi, Evgeny Goldin, Yarden Nir, Gil Ratsaby, and Shmuel Ur. Framework for testing multi-threaded Java programs. *Concurrency and Computation: Practice and Experience*, 15(3–5):485–499, 2003.

[17] ENEA. Jcarder. `http://www.jcarder.org`.

[18] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *SOSP, Nineteenth ACM Symposium on Operating Systems Principles*, pages 237–252, 2003.

[19] Mahdi Eslamimehr and Jens Palsberg. Race directed scheduling of concurrent programs. In *Proceedings of PPOPP'14, ACM Annual Symposium on Principles and Practice of Parallel Programming*, 2014.

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

[21] Azadeh Farzan and P. Madhusudan. Causal atomicity. In *Proceedings of CAV'06, International Conference on Computer Aided Verification*, pages 315–328, 2006.

[22] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of POPL'05, ACM Symposium on Principles of Programming Languages*, Long Beach, CA, USA, January 2005. ACM Press.

[23] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of PLDI'02, ACM Conference on Programming Language Design and Implementation*, pages 234–245, 2002.

[24] European Organization for Nuclear Research (CERN). Colt. `http://acs.lbl.gov/software/colt/`.

[25] Vojtech Forejt, Daniel Kroening, Ganesh Narayanaswamy, and Subodh Sharma. Precise predictive analysis for discovering communication deadlocks in MPI programs. In *Proceedings of FM'14, Formal Methods*, pages 263–278, 2014.

[26] Apache Software Foundation. Derby. `http://db.apache.org/derby`.

[27] P. Godefroid. Model checking for programming languages using Verisoft. In *Proceedings of POPL'97, 24th Annual ACM Symposium on Principles of Programming Languages*, pages 174–186, 1997.

[28] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *Proceedings of PLDI'05, ACM SIGPLAN Conference*

on Programming Language Design and Implementation, pages 213–223, 2005.

[29] Klaus Havelund. Using runtime analysis to guide model checking of Java programs. In *Proceedings of SPIN'00, Model Checking Software, International SPIN Workshop*, pages 245–264. Springer-Verlag, 2000.

[30] Klaus Havelund and Thomas Pressburger. Model checking Java programs using Java pathfinder. *Software Tools for Technology Transfer*, 2(4):366–381, 2000.

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

[32] David Hovemeyer and William Pugh. Finding concurrency bugs in Java. In *Proceedings of the PODC Workshop on Concurrency and Synchronization in Java Programs*, 25–26 2004.

[33] Chun-Hung Hsiao, Jie Yu, Satish Narayanasamy, Ziyun Kong, Cristiano L. Pereira, Gilles A. Pokam, Peter M. Chen, and Jason Flinn. Race detection for event-driven mobile applications. In *Proceedings of PLDI'14, ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[34] Jeff Huang, Charles Zhang, and Julian Dolby. CLAP: Recording local executions to reproduce concurrency failures. In *Proceedings of PLDI'13, ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2013.

[35] P. Joshi, M. Naik, K. Sen, and D. Gay. An effective dynamic analysis for detecting generalized deadlocks. In *ACM FSE'10, Symposium on the Foundations of Software Engineering*, 2010.

[36] P. Joshi, C. S. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *Proceedings of PLDI'09, ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 110–120, June 2009.

[37] Vineet Kahlon, Franjo Ivancic, and Aarti Gupta. Reasoning about threads communicating via locks. In *Proceedings of CAV'05, International Conference on Computer Aided Verification*, pages 505–518, 2005.

[38] Vineet Kahlon and Chao Wang. Universal causality graphs: A precise happens-before model for detecting bugs in concurrent programs. In *Proceedings of CAV'10, International Conference on Computer Aided Verification*, pages 434–449, 2010.

[39] T. Li, C. S. Ellis, A. R. Lebeck, and D. J. Sorin. A dynamic deadlock detection mechanism using speculative execution. In *Proceedings of the USENIX Technical Conference*, 2005.

[40] Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. Race detection for android applications. In *Proceedings of PLDI'14, ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[41] Rupak Majumdar and Ru-Gang Xu. Directed test generation using symbolic grammars. In *Proceedings of the twenty-second IEEE/ACM International Conference on Automated Software Engineering*, 2007.

[42] Daniel Marino, Christian Hammer, Julian Dolby, Mandana Vaziri, Frank Tip, and Jan Vitek. Detecting deadlock in programs with data-centric synchronization. In *ICSE'13, International Conference on Software Engineering*, 2013.

[43] S. Masticola. *Static Detection of Deadlocks in Polynomial Time*. PhD thesis, Rutgers University, 1993.

[44] S. Masticola and B. Ryder. A model of Ada programs for static deadlock detection in polynomial time. In *Proceedings of ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 97–107, 1991.

[45] M. Naik, C.-S. Park, and D. Gay. Effective static deadlock detection. In *ICSE'09, Eighteenth International Conference on Software Engineering*, pages 386–396, 2009.

[46] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for Java. In *Proceedings of PLDI'06, ACM Conference on Programming Language Design and Implementation*, 2006.

[47] Elissa Newman, Aaron Greenhouse, and William Scherlis. Annotation-based diagrams for shared-data concurrency. In *Workshop on Concurrency Issues in UML*, 2001.

[48] Oracle. Java hotspot vm options. `http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html`.

[49] Oracle. The Java tutorials; deadlock. `http://docs.oracle.com/javase/tutorial/essential/concurrency/deadlock.html`.

[50] Oracle. JDK, 1.4.2. `http://www.oracle.com/technetwork/java/javase/index-jsp-138567.html`.

[51] Veselin Raychev, Martin Vechev, and Manu Sridharan. Effective race detection for event-driven programs. In *Proceedings of OOPSLA'13, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 151–166, 2013.

[52] Mahmoud Said, Chao Wang, Zijiang Yang, and Karem A. Sakallah. Generating data race witnesses by an SMT-based analysis. In *NASA Formal Methods*, pages 313–327, 2011.

[53] Malavika Samak and Murali Krishna Ramanathan. Trace driven dynamic deadlock detection and reproduction. In *Proceedings of PPOPP'14, ACM Annual Symposium on Principles and Practice of Parallel Programming*, 2014.

[54] Cesar Sanchez, Henny B. Sipma, Zohar Manna, and Christopher D. Gill. Efficient distributed deadlock avoidance with liveness guarantees. In *Proceedings of EMSOFT'06, International Conference on Embedded Software*. Springer-Verlag (*LNCS*), 2006.

[55] Koushik Sen. Concolic testing. In *Proceedings of the twenty-second IEEE/ACM International Conference on Automated Software Engineering*, pages 571–572, 2007.

[56] Koushik Sen and Gul Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *Proc. 18th International Conference on Computer Aided Verification*, pages 419–423, 2006.

[57] Traian Florin Serbanuta, Feng Chen, and Grigore Rosu. Maximal causal models for multithreaded systems. Technical report, University of Illinois at

Urbana-Champaign. Available from ideals.illinois.edu.

[58] F. Sorrentino, A. Farzan, and P. Madhusudan. PENELOPE: Weaving threads to expose atomicity violations. In *ACM FSE'10, Symposium on the Foundations of Software Engineering*, pages 37–46, 2010.

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

[60] Lin Tan, Yuanyuan Zhou, and Yoann Padioleau. acomment: Mining annotations from comments and code to detect interrupt related concurrency bugs. In *ICSE'11, International Conference on Software Engineering*, 2011.

[61] William Thies and Michael Ernst. Static deadlock detection for java libraries. In *Proceedings of ECOOP'05, European Conference on Object-Oriented Programming*, pages 602–629. Springer-Verlag (*LNCS*), 2005.

[62] Raja Vallé-Rai, Etienne Gagnon, Laurie Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java bytecode using the soot framework: Is it feasible? In *Proceedings of CC'00, International Conference on Compiler Construction*. Springer-Verlag (*LNCS*), 2000.

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

[64] C. von Praun and T. R. Gross. Object race detection. In *OOPSLA, ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 70–82, 2001.

[65] Chao Wang and Kevin Hoang. Precisely deciding control state reachability in concurrent traces with limited observability. In *Proceedings of VMCAI'14, Verification, Model Checking, and Abstract Interpretation*, pages 376–394. Springer-Verlag (*LNCS*), 2014.

[66] Yin Wang, Terence Kelly, Manjunath Kudlur, Stephane Lafortune, and Scott Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *Proceedings of OSDI'08, 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008.