UNIVERSITY OF CALIFORNIA

Los Angeles

Directed Testing of Event-Driven and Parallel Programs

A dissertation submitted in partial satisfaction of
the requirements for the degree
Doctor of Philosophy in Computer Science

by

Mohammad Mahdi Eslamimehr

2014

The dissertation of Mohammad Mahdi Eslamimehr is approved.

Todd Millstein

Glenn Reinman

William Kaiser

Jens Palsberg, Committee Chair

University of California, Los Angeles

2014

To my precious and brilliant wife Sara ...

Without her help and support it simply never would have been.

TABLE OF CONTENTS

List Of Figures

Acknowledgment

Thank you, Jens Palsberg, for being such an extraordinary adviser. I joined your lab in 2008 without much experience in research, academic writing, and deep knowledge of compilers. I became part of your lab and spent 5 years learning from you every single day. I am now graduating feeling like a researcher and better person, and for all these I am humbly indebted to you.

I would like to express my gratitude to my committee members, Professor Todd Millstein, Professor Glenn Reinman, and Professor William Kaiser, for their insightful guidance in my oral qualifying exam.

Foremost, I would like to thank my parents and my brother who always had faith in me, and their supports helped me focus on my studies.
I would also like to thank my friends Ali Sajjadi, Riyaz Haque, John Bender, and Hamid Mirebrahim who have helped and encouraged me to move forward.

Lastly, I would like to thank my wife, Sara for her support and encouragement throughout my studies.

VITA

2006	B.Sc. Computer Engineering, Sharif University, Tehran, Iran.
2007	Software Intern, Conformiq Qtronic, Helsinki, Finland.
2007-2008	Software Engineer at Ericsson AB. Stockholm, Sweden.
2008	M.Sc. Computer Science, Linkoping University, Linkoping, Sweden.
2008-Present	Research Assistant, Complier Construction Lab, UCLA, USA.
2010	Software Engineer, Samsung Electronics US RD Center, San Jose, USA.
2011	National Science Foundation (NSF)/ Stanford Research Institute (SRI) School of Formal Techniques, Palo Alto, US.
2011	Senior Software Engineer, Utopia Compression RD Department, Los Angeles, USA.
2012	Software Intern, Joseph Fourier University, Grenoble, France.
2012	Software Intern, CNRS (French National Center for Scientific Research), Grenoble, France.

Publication

Mahdi Eslamimehr, Jens Palsberg, Race Directed Scheduling of Concurrent Programs, PPOPP 2014.

Mahdi Eslamimehr, Jens Palsberg, Testing versus static analysis of maximum stack size, COMPSAC 2013.

ABSTRACT OF THE DISSERTATION

Directed Testing of Event-Driven and Parallel Programs

by

Mohammad Mahdi Eslamimehr

Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2014
Professor Jens Palsberg, Chair

Detecting computational states of a program, where safety requirements have been violated, is the main task of a software tester. We focus on three critical safety requirements. First, finding maximum stack usage in event-based systems, in order to avoid stack overflow. Second and third, absence of data race and deadlock in parallel programs, respectively. We will present how particular states of computation, where the above mentioned requirement are violated, is reached. Directed testing has shown considerable success in both academy and industry. However, applying directed testing's core form on programming paradigms, with a more complicated control flow is not nearly as successful as on sequential programs. The goal of this dissertation is to address how we can enhance directed testing to perform well with event-driven and parallel programs.

For event-driven software we present a new approach, termed event-based directed testing. Our approach combines aspects of random testing and directed testing to generate challenging event sequences, for testing event-driven software.

Our experiments show, we achieve significantly improved branch coverage and larger maximum stack sizes.

For parallel programs, we also present a new dynamic technique to detect data races and deadlocks. Our technique combines previous work on concolic execution with a new constraint-based approach to drive an execution towards a concurrency bug candidate. Our technique has found almost twice as many real concurrency bugs as the four previous techniques combined.

CHAPTER 1

Introduction

As software systems become more complicated, checking their correctness becomes a tedious task. Software safety is a requirement which is a concern for all industries, but few address it correctly. We focus on three important safety requirements. First, focus on finding maximum stack usage in event-based systems to avoid stack overflow, and second, absence of data race, and third, absence of deadlock in parallel programs. The goal of this dissertation is to show how to *reach* to particular states during computation where a safety requirement is violated. *Thus, we will show we can enhance directed testing to work well with event-driven and parallel programs.*

1.1. Analysis of Maximum Stack Size

Testing event-driven programming has found pervasive acceptance, from high-performance servers to embedded systems, as an efficient method for interacting with a complex world. However, loose coupling of event handlers obscures control flow and makes dependencies hard to detect, leading to subtle bugs. Event-driven software on resource-constrained devices has the additional challenge that if swamped with events; the software may run out of memory. Thus, estimates of the maximum stack size can be of paramount importance [1].

For example, a poor estimate led to software failure and closure of a German railway station in 1995. Specifically, the designers had estimated that 3,500 bytes of stack space would be sufficient but actually 4,000 bytes were needed. As a result, the railroad station's computer experienced stack overflow and failed [2].

Intuitively, the maximum stack size during a run is the high water mark or the peak value of the stack pointer. We focus on a much-studied question about stack space for event-driven software:

Q: what is the maximum stack size across all inputs?

A programmer can use the answer to ensure that sufficient stack memory is available for a particular application. Additionally, the programmer can use the smallest or cheapest memory unit that has sufficient capacity and thereby help control size and cost. This is welcome for many event-driven applications that run in embedded systems for which physical size and hardware cost are major concerns.

Like most other interesting questions about programs, the above question is undecidable. Ideally, we would answer the above question by running the program on all inputs, possibly indefinitely in case of nontermination. Each run has a maximum stack size and we can then take the maximum across all runs to get the answer to the question. The result is the *true* maximum stack size.

The above question can be answered approximately by testing (running the program) and by static analysis (analyzing the program text). A testing approach underestimates the true answer

by finding the maximum stack size for some runs on some inputs. A static analysis overestimates the ideal answer by working with conservative abstractions of program constructs and values. In slogan form, we have the following relationships for maximum stack size:

$$tested \leq true \leq static$$

How close are *tested* and *static*? In some situations no nontrivial sound static analysis exists, and we have only the trivial sound static analysis that says that the stack is unbounded. A typical such scenario is an embedded system for which some of the event-driven software is written in assembly code. The assembly code usually contains instructions that add or subtract from the stack pointer, to enable the stack to shrink or grow. Can current nontrivial sound static analyses handle such instructions? The answer is *Yes* if the instructions add or subtract *constants*=; while the answer is *No* if the instructions add or subtract the contents of a *register*. If no nontrivial sound static analysis exists, then a programmer must use the best testing approach, and perhaps take a chance with an unsound static analysis. Such techniques are inherently unsafe and a standard engineering solution is to over-provision: if the testing approach estimates the maximum stack size to be n, then go with memory of size 2n, for example, though even 2n may be insufficient.

If a sound static analysis exists, then we can use it to safely allocate the estimated amount of memory and be sure that no stack overflow will occur. Ideally we can find an *optimal* static analysis that always produces the true maximum stack size. However, static analysis must terminate, including for nonterminating programs, so usually static analysis is forced to be conservative and nonoptimal. For maximum stack size of event-driven software, the state-of-the-art static analysis was presented in [3], [4] (See also [5], [6], [4]) and has been implemented in multiple tools. In this paper we address the following question.

Q: how good is the state-of-the-art static analysis of maximum stack size?

We use testing to answer the above question. We have done an experiment with the state-of-the-art testing approach [7] (see also [6]) on benchmarks that are event-driven assembly code programs. In those benchmarks, all arithmetic on the stack pointer either adds or subtracts constants, according to our manual inspection, so the static analysis is sound, we believe. We found a big gap between the estimates: the testing approach achieves a maximum stack size that on average is only 67 percent of that achieved by static analysis. Our benchmark suite consists of software for sensor nodes and proved to be a major challenge for the testing approach. For a different benchmark suite, Regehr [7] found that testing and static analysis are much closer.

Our experiment raises a classical question that arises for a variety of problems that can be addressed with both testing and static analysis. *Is the gap mostly due to weak testing or overly conservative static analysis?* The answer is that better testing is possible and that the static analysis is near optimal for our benchmarks. We make those points by presenting two new testing approaches that almost match the static analysis. The first approach is called DTall and achieves a maximum stack size that on average is within 99 percent of that achieved by static analysis. The second approach is called VICE and achieves a maximum stack size that on average is within 94 percent of that achieved by static analysis. VICE is two orders of magnitude faster than DTall. Our results show that the state-of-the-art static analysis produces excellent estimates of maximum stack size.

1.2. Data Race Detection

Concurrent programming with shared memory offers both the benefit of efficient execution and the pitfall of data races. Efficiency can be achieved when we let multiple processors run in parallel and exchange data via the shared memory. A data race arises when two processes simultaneously access a shared memory location and at least one of the two accesses is a write operation. Data races often result in hard-to-detects bugs and usually the programmers of concurrent software should try to avoid data races.

One reason for why data races are problematic can be found in a seminal paper by Adve, Hill, Miller, and Netzer [8]. Their observation is that on suitable hardware, every execution of a data-race-free program is *sequentially consistent*. Sequential consistency was introduced by Lamport in 1979 and means that "the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program" [9]. Sequential consistency provides a useful memory model that simplifies the task of producing correct concurrent programs. If programmers can avoid data races, they can use sequential consistency as their memory model.

Researchers have developed many techniques to help programmers detect data races. Some of those techniques require program annotations that typically must be supplied by a programmer; examples include [10], [11]. 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 23 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 race-detection techniques into three categories: static, dynamic, and hybrid. A static technique examines the text of a program without running it; a dynamic technique runs a program, possibly multiple times, and gathers information during those executions; and a hybrid technique does both.

The advantage of a static technique is that if it is sound, then it will report every possible race, though it may also report false positives. We will show via experiments that the best existing static technique reports a large number of false positives that would be daunting to examine by hand. For our benchmarks, the Chord tool reports total 127136 data races. So, current static techniques are of little use to working programmers.

The advantage of a dynamic technique is that it reports only real races. For example, for our benchmarks, the FastTrack, Goldilocks, CalFuzzer, and Pacer tools together report total 304 data races. So, current dynamic techniques give programmers valuable help, yet our experiments show that they leave many races to be discovered.

The advantage of a hybrid technique is that it may be able to combine the best of both worlds, static and dynamic. The best existing hybrid technique appears to a technique by O'Callahan and Choi [12] that we call Hybrid, which for our benchmarks report a total 405 data races. This technique may produce both false positives and false negatives, yet the tool provides programmers with output of a fairly manageable size.

In this study we focus on dynamic techniques. We will present a dynamic technique that reports significantly more real races than the previous techniques.

The main shortcoming of the existing dynamic techniques is that when they search for an execution that lead to a real race, they often come up empty handed. We present a novel approach to execution search that gives much better results. The central concept in our approach is the standard notion of *schedule*, which is a sequence of events that must be executed in order.

The Challenge. Search for an execution that leads to a real race.

Our Results. We present *race directed scheduling* that for given a race candidate searches for an input and a schedule that lead to the race. The search iterates a combination of concolic execution and schedule improvement.

We have implemented race directed scheduling in a tool that does race detection for Java programs. As requested by the program chair, we use a pseudonym for our tool's name; we will refer to our tool as *Racageddon* in this dissertation.

We use an existing hybrid technique to produce a manageable number of race candidates.

For our benchmarks, our tool found 72 real races that were missed by the best existing dynamic techniques. Among the 304 real races found by the existing dynamic techniques, our technique found 272 of them. Our tool is fully automatic and its user needs no expertise on data races. Once our tool reports a race, it can replay the execution that leads to the race.

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

- An effective and easy-to-use tool for dynamic race detection and
- A large-scale experimental comparison of seven race detectors.

1.3. Deadlock Detection

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 [13]. 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 of those techniques require program annotations that typically must be supplied by a programmer; examples include [14] [15] [16] [17] [18] [19] [20] [21] [22]. Other techniques work with unannotated programs and thus they are easier to use. In this section 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. The best static tool is Chord [23] [24] 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 [17], IBM ConTest [25] [26], Jcarder [27] and Java HotSpot [28], 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 section 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. If that executable schedule is more than a million step of computation, then we refer to the deadlock as a *rare* deadlock. We will show how to do a better search for executable schedules and how to find rare deadlocks.

The challenge. Help programmers find rare deadlocks.

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 *ConLock* 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 rare deadlocks: it found 33 deadlocks that happened after more than one million computation steps, including 28 new deadlocks. Our tool is fully automatic and its user needs no expertise on deadlocks. Once our tool reports a deadlock, it can replay the execution that leads to the deadlock.

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

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

1.4. Contributions

The event-Based directed testing algorithm has been implemented in VICE, a tool that automatically and accurately finds maximum stack usage of Virgil programs. The research contribution in VICE is "Testing Versus Static Analysis of Maximum Stack Size" [30].

The race directed scheduling technique has been implemented in Racageddon, a tool that automatically find data races in Java programs. Research contribution in Racageddon is "Race directed Scheduling of Concurrent Programs" [31].

1.5. Outline

In chapter 2 we discuss the core form of directed testing and its tools along with their strengths and limitations. We end the chapter with directed testing's shortcomings and strengths. In chapter three we introduce event-based directed testing, and its application on finding maximum stack size. In chapter four we introduce race directed scheduling, a technique to find data races automatically. In chapter five we show how we could enhance directed testing to find rare deadlocks in Java programs.

CHAPTER 2

State-of-Art Testing of Sequential Programs

Software Testing has been experiencing its best practice for years. However, except few large corporations such as Microsoft (since 79% of developers also write test suites [32]) adoption of testing in industry is still poor. Automatic Testing and in particular automatic test input generation has received increased attention in both academy and industry. The main reason is most software suffers from low pressure, low quality, or outdated test suits, due to high cost of software testing [33]. In these situations automatic testing of software can provide extreme value.

2.1. Background

Despite progresses in *Static Analysis*, *Symbolic Execution*, and *Model Checking*, *Software Testing*, is still the predominant technique to ensure software reliability. However, testing still accounts for 50-80% of overall cost of software development [34].

Many algorithms have been introduced to improve software testing, yet in practice; it is challenging, expensive, and rarely performed properly. In fact, to test a program test engineers develop a test harness to simulate the behavior of the program's environment. More development is also needed to verify the functional correctness of a program. For example, adding assertion codes to check the program's output. Hand-written tests are error-prone, expensive, and not exhaustive. Consequently, many errors that should have been reported during early stages of testing remain hidden until software deployment.

Recent achievements in amplifying the power of computers, has recalled for automated testing. *Random Testing* is a proven technique for finding programs' bugs. It can automatically generate many random test cases, and execute them to explore different paths of the program to find errors. Studies show random testing is more effective in contrast with manual testing [32]. It is an appealing technique, since it is fast, scalable, inexpensive, and has no space overhead. Nonetheless, random testing cannot confirm the correctness of a program, and suffers from poor code coverage. Therefore, it can test only a limited portion of all program's testable paths, for example, the probability of reaching to the $then$ branch in the "$if(x == 5)\ then$ ..." is $\frac{1}{2^{32}}$ if x is a 4 bytes integer program input.

Exhaustive Enumeration can also generate huge amount of test cases automatically, but many of them are meaningless, and cannot even take the program beyond the initialization phase. Further studies introduced *Constraint/Specification Based Exhaustive Enumeration*. It can generate all valid inputs that may satisfy a program's constraints. Even though it generates data inputs selectively, yet many equivalent and redundant inputs test the same behavior of the program. For instance, assume a small program with four inputs all of which appear in at least one conditional. The program needs to exhaustively execute $(\mathbf{2^{32}})^{\mathbf{4}} = \mathbf{2^{128}}$ combinations to cover all valid input. This is extremely time-consuming even for a program with a limited number of inputs. In fact, McMinn [35] showed testing any reasonably-sized program with exhaustive enumeration is infeasible.

Significant problems in random testing and exhaustive enumeration motivated studies for *Symbolic Execution*. Instead of generating concrete data inputs, like integer, symbolic execution

produces symbols that represent values. The program runs normally except variables may have a value in a symbolic form. A *Constraint Solver* or a *Theorem Prover* is used later to solve constraints which are collected through the execution, and later generates test cases. Symbolic execution provides better code coverage and avoids redundant test cases. However, current constraint solvers are not powerful enough to solve complex constraints, not even normal arithmetic formulas like$(\boldsymbol{y} > \boldsymbol{x^3}\ \%\ \mathbf{21})\ .\mathbf{5}\ \neq \mathbf{40}$. Given that, in large or complex program, collecting constraints become intractable, and solving them would be computationally expensive and time consuming.

2.2. Directed Testing Era

Directed Testing (A.K.A *Concolic Testing*: the term was suggested by Koushik Sen as a combination of *CONCrete* and *symbOLIC* execution) has received a great attention recently. It addresses the mentioned challenges associated with random testing and symbolic execution, and opened new windows for providing efficient and automated test generation tools.

Directed testing enhances symbolic execution by running a program symbolically and concretely at the same time. A key feature in directed testing is whenever it cannot solve a constraint, symbolic values in the constraint are replaced by random concrete values. This will prevent discontinuation of symbolic execution, when the constraint solver encounters an undecidable constraint.

To the best of our knowledge combining concrete and symbolic execution (with user feedback) was initially suggested by Larson and Austin [36]. In their method, software testers provide concrete values instead of a random input generator or a constraint solver. Moreover, only solvable constraints would be collected and later will be handed over to the solver. This lowers the path coverage, but decreases the computational cost.

Later, Godfroid et al. pioneered the first directed testing tool called DART (Directed Automated Random Testing) [37]. In fact DART combined three algorithms: (1) Automated extraction of the program interface from source code , (2) Automated generation of a test driver to produce random test inputs, and (3) Directed generation of test inputs. DART's experimental results suggest is has low space overhead, and improves code coverage. However, it can only handle constraints with integer types, so to find a new path, DART negates the last encountered constraint. DART, also cannot collect and solve constraints generated by a program e.g. dynamic data structure and pointer operations.

CUTE and jCUTE [38], and CREST [39] are further concolic tools introduced by Koushik Sen. Unlike DART, CUTE cannot automatically extract the program's interface, while it is equipped with a more efficient constraint solver that could handle complex data structure with pointers and dynamic data types. CUTE does not generate test inputs selectively; inputs are generated randomly and purified through rounds of executions. This lowers the coverage in practice.

Further studies to improve the code coverage in concolic tools led to Hybrid Concolic Testing [40], and CESE (Concolic Execution with Selective Enumeration). Hybrid Concolic Testing combines random and concolic testing to expand the depth and width of the program state space exploration. It first tests the program randomly to improve the code coverage. Once random testing no longer has success in exploring new paths, concolic testing takes control of execution from the current state of the program. Consequently, Hybrid Concolic Testing uses random testing to reach deep states of the program with less execution, and uses concolic execution to explore new paths.

CESE is an automatic test input generator that interleaves selective enumeration with directed symbolic test generation. The algorithm initially converts the program input grammar to symbolic grammar. It furthermore, uses enumerative techniques to exhaustively enumerate all valid inputs which is accepted by the symbolic grammar. Finally, CESE uses a symbolic test generator to produce these enumerated symbolic inputs and runs them. Consequently, each CESE's execution round is faster than symbolic execution.

Recently, LIME concolic tool [41] is developed in Helsinki University. The main improvements in LIME over existing concolic tool like jCUTE are the following:

1. The use of bitvector SMT solver Boolector [42] makes the symbolic execution more precise as integers are not considered unbounded.
2. The twin class hierarchy instrumentation approach of LIME allows core classes to be instrumented.
3. LIME architecture supports distributed testing.

2.3. Directed Testing's Limitations and Promises

Applying directed testing on classic sequential programs has showed great success in both research and practice (For additional information see C. Pasareanu survey [43]). The idea has become popular enough to motivate its applications in various areas: database [44], web application servers [45] and clients [46], mobile sensor network [47], network card device driver [48]. However, in most of these works researchers exploited the directed testing core form and utilize it in a new algorithm to find bugs in the corresponding applications. The reason is using classical directed testing on more complicated program paradigms is not as successful as using it on sequential programs. Our experiments, confirms the usage directed testing's core form in event-driven and parallel programs, not to be as successful as it was promised.

In event-driven software, our evaluations showed traditional directed testing is only as good as genetic algorithm [7] in terms of code coverage, and it could explore 60% of benchmark's code in average. Directed testing also showed poor performance in computing program's functional requirements such finding maximum stack usage. We note that the directed testing approach achieves a maximum stack size that is only 67 percent of that achieved by static analysis.

K.Sen et al. [38] reported the same unfortunate experience in using traditional directed testing on parallel programs. Beside, our experiments also approve that traditional directed testing could only detect limited number of concurrency bugs like data races and deadlocks.

Our preliminary experiments show a gap to explore for enhancing directed testing. On the other hand other studies suggested that directed testing can be exploited to different programming paradigms. In the next three chapters we explain in detail how we can improve core form of directed testing to efficiently and accurately handle problems we defined in the previous chapter.

CHAPTER 3

Directed Testing of Event-Based Software

For event-driven software on resource-constrained devices, estimates of the maximum stack size can be of paramount importance. For example, a poor estimate led to software failure and closure of a German railway station in 1995. Static analysis may produce a safe estimate but how good is it? In this paper we use testing to evaluate the state-of-the-art static analysis of maximum stack size for event-driven assembly code. First we note that the state-of-the-art testing approach achieves a maximum stack size that is only 67 percent of that achieved by static analysis. Then we present better testing approaches and use them to demonstrate that the static analysis is near optimal for our benchmarks. Our first testing approach achieves a maximum stack size that on average is within 99 percent of that achieved by static analysis, while the second approach achieves 94 percent and is two orders of magnitude faster. Our results show that the state-of-the-art static analysis produces excellent estimates of maximum stack size.

3.1. Event Sequence in Event-Driven Software

The classical notion of a program first consumes an input, then computes, and finally produces an output. In contrast, an event-based program receives its input via events *during* the program execution. The task of the event-based program is to process those events.

For example, our benchmarks run on sensor nodes (Berkeley Motes) and receive events that are generated by devices that are connected to the CPU. Among those devices are a timer, an analog-to-digital converter (ADC), a universal synchronous asynchronous receiver/transmitter (USART) Atmel-usart10], and a serial peripheral interface bus (SPI) [49]. The sensor node can use the timer to wake itself up periodically, use the ADC to convert sensor data to digital form, use the USART for serial communication with terminals, and use the SPI to communicate on a synchronous serial data link with external devices in master or slave mode.

Event-based programs such as sensor-network software are usually designed to run indefinitely (or until the battery dies). Thus, events can keep coming. Notice though that a finite test run consumes only a finite number of events.

Each event consists of a name and a value. The name specifies the source of the event and also the event handler that will process the event. The value is input to the program.

From the program's viewpoint, consecutive events have a *wait time* between them. This wait time can be completely arbitrary and depend on uncoordinated devices beyond the programs control. However, for a particular run we can record both the events and the wait times. Or, for the purpose of planning a test run, we can first *generate* an event sequence and then use that to test the program.

In this chapter, we represent an event sequence as a sequence of triples:

$$(event\ name, event\ value, wait\ time)$$

The idea is to wait the number of milliseconds specified by *wait time* and then fire an event called *event name* and paired with *event value*.

For example, here is our representation of an event sequence with four events:

$$[(\text{main}, 673{,}100), (\text{m\_intr}, -8634756{,}200), (\text{main}, -991{,}400), (\text{m\_intr}, 34{,}800)]$$

The first event $(\text{main}, 673{,}100)$ will occur after 100 milliseconds, the second event $(\text{m\_intr}, -8634756{,}200)$ will occur after 300 milliseconds, the third event $(\text{main}, -991{,}400)$ will occur after 700 milliseconds, and the fourth event $(\text{m\_intr}, 34{,}800)$ will occur after 1,500 milliseconds.

Let us return to the evaluation of the state-of-the-art static analysis of maximum stack size. Our goal is to find an event sequence that achieves a large maximum stack size. For creating a suite of candidate event sequences, a designer must decide on the number of event sequences, the number of events in each event sequence, the event names, the event values, and the wait times.

3.2. Seven Testing Approaches

We now present seven testing approaches that all automatically test event-driven software without a human in the loop. Testing approaches 1-4 are from previous work, while 5-7 are new.

How to determine the number of events in each event sequence. Our benchmarks work with 2--5 event handlers. For simplicity we want every event sequence for every benchmark to have the same number of events. We determined the number of events via the following preliminary experiment that anyone can repeat for any benchmark suite. First we noted that the number of events for our benchmark suite should be at least 5 such that we can hope to exercise every handler during a single run. Second we observed that more events may exercise longer program paths. The question is: when does an increase of the number of events begin to produce diminishing returns? We use testing approach 1 (see below for details) to run experiments with different numbers of events in each event sequence. We doubled the number of events, doubled it again, and so on, until we saw no major improvement in maximum stack size. We found that 40 events in each event sequence appears to be a good number for our benchmarks so all our experiments use event sequences with 40 events.

Now we must generate event sequences that each contains 40 event names, 40 event values, and 40 wait times.

How to determine samples of wait times. Four of the testing approaches use *samples* of the wait times. We chose to fix *three* different samples and use them across all those four testing approaches. The number *three* is somewhat arbitrary; we wanted a number greater than one to give diversity in the experiments yet small enough that our experiments could finish in a reasonable time. We determined the three particular samples via the following preliminary experiment that anyone can repeat for any benchmark suite. For each benchmark we ran each event handler in isolation to determine the worst-case time to execute any handler alone (in any of the benchmarks). That worst-case time is the longest time any single handler may be able to block other handlers from running. Once we had that number, we divided the time interval from 0 to that number into three equally sized intervals. Finally, from each of those three intervals we sampled a wait time using a uniform distribution.

We will use the numbering (1-7) of the approaches throughout the paper. Those seven approaches span a wide variety of techniques that one might try. Ultimately, testing approach 7 is the best we are able to do given a reasonable amount of time. Testing approaches 4-6 can be understood as restrictions of testing approach 7.

Testing approach 1 is a form of random testing that tries 3,000 event sequences based on randomly chosen samples of event names and event values, and the three particular samples of wait times that we found as discussed above. The number 3,000 is somewhat arbitrary; we

wanted a number that was large enough to produce good results yet small enough that our experiments could finish in a reasonable time.

We compare seven testing approaches:

Approach	#	event names	event values	wait times
	1	Sample	Sample	Sample
	2	GA	GA	Sample
	3	All	Sample	All
	4	Sample	DT	All
VICE:	5	SA-Tree	DT	Sample
	6	All	DT	Sample
DTall:	7	All	DT	All

The experiments justified the use of three wait times because, somewhat surprisingly, we encountered some cases where a *longer* wait time leads to a *larger* stack size. This phenomenon stems from situations such as the following. Suppose we have reached a state *S* of the computation where a run of the handler for event *B* would reach a maximally large stack. Suppose also that in state *S*, events *A* and *B* have fired and the handlers for *A* and *B* are enabled. The hardware *arbits deterministically* which handler will run; and let us assume that the hardware chooses *A*. So, *B* will run later; possibly in a state with a smaller stack than state *S* so the run of *B* will fail to reach a maximally large stack. Can we get the hardware to choose *B* instead of *A*? One potential answer is: increase the wait time such that *A* isn't enabled in state *S*. Hence, a longer wait time has the potential to produce a larger stack size. Testing approach 1 is our base line; the other six approaches do better.

Testing approach 2 is a genetic algorithm (GA) [7] that uses 20 generations of each 50 event sequences, for each of the three chosen samples of wait times. We chose 20 generations and 50 event sequences because the total number of runs would be $20 \times 50 \times 3 = 3{,}000$, which matches the number of runs with testing approach 1. The first generation has a randomly chosen sample of event names and event values. Each later generation hopes to improve on the previous one by swapping and mutating the event names and event values. Specifically we map a generation to a new generation in the following way. We first do 50 swaps of subsequences of length 25 among the event sequences. We then mutate one event in each event sequence; each mutation replaces the event name with a randomly chosen event name, and it replaces the event value with a randomly chosen event value. The fitness function is the maximum stack size observed during a run.

Testing approach 3 is similar to testing approach 1 in that it samples the event values, but also goes much further in that it tries all combinations of i) all sequences (of length 40) of event names, and ii) all integer wait times in a wide interval. The interval of wait times is handler specific and defined as follows. The lower bound of the interval is 8 milliseconds; we found that going lower often caused testing to run out memory. The upper bound of the interval is the worst-case time to execute the handler for the previous event in isolation. Note that if the upper bound is high, trying all integer wait times in the interval may lead to a lengthy testing effort. In such a case, we recommend the use of a large number of samples drawn from a uniform distribution across the interval.

Our preliminary experiment, mentioned in Section 1, tried testing approaches 1 and 2. When we found that the results from those approaches are suboptimal, we tried the much slower testing

approach 3 which gave just a small improvement. We concluded that we need a better approach to generate event values.

Testing approaches 4--7 all use directed testing (DT) to generate event values. Directed testing [37] is based on concolic execution [50], which is a technique related to model checking [51], theorem proving [52], symbolic execution [53], and run-time monitoring and testing [11]. The idea of directed testing is to execute the code with concrete and symbolic values simultaneously, and to use the result to generate new inputs for another execution. The term concolic combines the words "concrete" and "symbolic". In each round, the symbolic part of an execution collects constraints from each condition on the control-flow. Those constraints represent the executed control-flow path and they have *the concrete input to the run* as one of the possible solutions. We can now easily construct constraints for a different potential control-flow path by taking a prefix of the collected constraints and *negating* the last constraint from the prefix. Concolic execution will submit those new constraints to a constraint solver, and if they are solvable, the concolic execution will use the solution as concrete input to a new round of execution. In the first round, the input is chosen randomly. Experience shows that concolic execution achieves better branch coverage with fewer test cases than testing with random inputs.

Testing approach 4 samples the event names, does DT to determine event values, and tries all integer wait times in a wide interval. In essence, testing approach 4 is standard DT applied to many combinations of event names and wait times. This gives a significant improvement over testing approach 3, yet falls well short of the results from static analysis. We conclude that we must do better to generate challenging event names.

Testing approach 5 is the one we call VICE (Virgil Integrated Concolic Engine). Compared to testing approach 4, VICE handles event names more accurately and wait times less accurately. Specifically, VICE uses a novel technique called SA-Tree to generate event names, uses DT to determine event values and tries three samples of wait times. VICE is the fastest of the seven approaches and gives a good trade-off between testing time and quality of the results.

Testing approach 6 does more than testing approach 5 by trying *all* sequences (of length 40) of event names, in addition to use DT to determine event values and to try three samples of wait times. However, the exhaustive coverage of the sequences of event names cannot improve on VICE because SA-Tree generates all event sequences that matter. We have included testing approach 6 in our experiments to demonstrate the large impact SA-Tree has on static analysis time.

Testing approach 7 is the one we call DTall and is both the slowest and the closest to optimal. DTall uses DT to determine event values and it tries all combinations of i) all sequences (of length 40) of event names, and ii) all integer wait times in a wide interval. DTall comes close to the results from static analysis and demonstrates that the best known static analysis is near optimal for our benchmarks.

3.3. VICE Example

Overview. We now explain the initial portion of a run of VICE on the example program in Figure3.1, which is a simplified version of one of our benchmarks. The program has four if-statements and two event handlers: main and m_intr. Our description of the example run is high level and ignores some details. VICE proceeds in phases that each consists of multiple rounds. We will explain just one phase with five rounds. During a phase, the event names stay unchanged in each round; the example uses the sequence of event names: (main, m_intr, main, m_intr). So, all event sequences in the example will have length four.

Round one. In the first round, the event sequence is random so we might begin with this event sequence:

$$[(\text{main}, 673), (\text{m\_intr}, -8634756), (\text{main}, -991), (\text{m\_intr}, 34)]$$

(We don't list or discuss the wait times in this section.) The concolic execution will fire the first event and now let us say that before main calls transmitValue in line 09, the execution fires the second event and interrupts main. We now have two event handlers on the stack. Next m_intr calls transmitValue in line 24 and we collect the constraint

$$y = a$$

that relates the actual parameter (line 24) to the formal parameter (line 12). We use y to denote a symbolic variable related to the program variable y, and similarly for a and a. In the body of transmitValue in line 16, let us assume that the condition atomic_swap(sending,true) returns false. We collect constraints from the conditions of the if-statements provided that they are arithmetic or logical equations. So we don't collect any constraints from the if-statement in line 16, while we do collect the constraint

$$a > 2000$$

from the if-statement in line 17 because (a>2000) failed: a has the value -8634756 so the execution doesn't take the branch that requires a > 2000. Now the second event handler terminates and we return to the first event handler. That event handler eventually calls transmitValue in line 09 and we collect the constraint

$$x = a$$

In the body of transmitValue we collect the same constraints as before and again the execution doesn't take the branch that requires a > 2000 because a has the value 673. Now the first event handler terminates. Later the execution fires the third and fourth events, and we can see that no new branches will be executed while handling those events.

During the first round of concolic execution, the maximum stack size occurred when we had two event handlers on the stack and m_intr called transmitValue which, in turn, called atomic_swap. The execution took the same branch each time in lines 16 and 17, while it never reached line 27 or 29.

After completion of the first round, we solve the three collected constraints above, pick a solution at random, and use it to help generate another event sequence. We use the four event-handler names from before and pair each of them up with values from the picked solution to the constraints. For example, we may get the event sequence:

$$[(\text{main}, 2833), (\text{m\_intr}, 4756), (\text{main}, 77733), (\text{m\_intr}, 6500)]$$

Round two. In the second round of concolic execution, let us assume that the firing of events proceeds like in the first round. The execution will four times reach line 17 and find each time that the condition a > 2000 is satisfied. So, the execution will exercise a new branch and eventually call checks in line 18 and from the call collect the constraint

$$a = s \ \wedge b = t$$

In the body of checks we will in each of the four cases find that s is different from 5000 so also in this round the execution doesn't reach line 29. Along the way, we collect the constraint

$$s = 5000$$

In line 27.

During the second round of concolic execution, the maximum stack size occurred when the stack contained two event handlers and stack frames for transmitValue and checks. That maximum stack size is similar to the maximum stack size encountered in the first round.

After completion of the second round, we find that the above constraints have a unique solution ($y = x = a = s = 5000$) that we use to help generate another event sequence. And again, we use

```
00 program TestProgram {
01          entrypoint main = TestMe.main;
02           entrypoint timer_comp = testMe.m_intr;
03          }
04
05          component TestMe {
06                      field sending:bool = false;
07                       method main(x:int):void {
08                                  computeValue();
09                                  transmitValue(x);
00                       }
11                      method computeValue():void { ... }
12                      method transmitValue(a:int):void {
13                      local buffer:int, b:int;
14                      b = rand(100);
15                      local bufferSize:int = (a+b) * 256;
16                      if (atomic_swap(sending,true)) return;
17                                   if (a > 2000) {
18                                  buffer = checks(a,b);
19                                  sending = false;
20                                  return;
21                      }
22          }
23          method m_intr(y:int):void {
24                      transmitValue(y);
25           }
26           method checks(s:int, t:int):int {
27                      if (s==5000) {
28                                  t=square(s);
29                                  if (s<-5) return square(-s);
30                                   else return 0;
31                      }
32           return 1;
33          }
34          method square(root:int):int { ... }
35          method rand(seed:int):int { ... }
36          method atomic_swap(cur:bool,status:bool)
37                       :bool { ... }
38 }
```

Figure 0.1 Virgil Example Program.

the four event-handler names from before and pair each of them up with values from the solution to the constraints. For example, we may get the event sequence

$$[(\text{main}, 5000), (\text{m\_intr}, 5000), (\text{main}, 5000), (\text{m\_intr}, 5000)]$$

Round three. In the third round of concolic execution, let us assume that the firing of events proceeds like in the second round. The execution will four times reach line 27 and find each time that the condition s==5000 is satisfied. So, the execution will exercise a new branch and eventually call square from which we collect the constraint:

$$s = root$$

Then the execution will reach line 29 and find that the condition s<-5 isn't satisfied. By the way, notice that the chance of reaching line 28 with event sequences generated randomly or by genetic algorithms is vanishingly small. We will collect the constraint

$$s < -5$$

During the third round of concolic execution, the maximum stack size occurred when the stack contained two event handlers and stack frames for the methods transmitValue, checks, and square, which is the highest so far.

Rounds four and five. After completion of the third round, we find that the collected constraints are unsolvable (because we have both $s == 5000$ and $s < -5$). We then repeatedly remove the last added constraint until we find that the remaining constraints are solvable, and then we proceed as before. We note that the third round has already achieved as much as one can do for the example program. VICE continues with a fourth and a fifth round until it notices that in two consecutive rounds, no improvements were achieved for the maximum stack size. At that point, the phase of the concolic execution terminates.

3.4. VICE Description

VICE uses six data types and six tools, see Figure 3.2.

Types:	VirgilProgram	=	see http://compilers.cs.ucla.edu/virgil
	machineCode	=	AVR assembly code
	eventSequence	=	$(\text{identifier} \times \text{int} \times \text{int})\text{list}$
	constraint	=	a Virgil arithmetic or logical expression
	nameSequence	=	$(\text{identifier})\text{list}$
	prefixTree	=	a prefix-tree of elements of nameSequence
Tools:	concolic	:	$(\text{VirgilProgram} \times \text{eventSequence}) \rightarrow (\text{constraint} \times \text{float})$
	compiler	:	$\text{VirgilProgram} \rightarrow \text{machineCode}$
	avrora	:	$\text{VirgilProgram} \times \text{eventSequence} \rightarrow \text{int}$
	SA-Tree-Gen	:	$\text{VirgilProgram} \rightarrow \text{prefixTree}$
	random	:	$\text{nameSequence} \times \text{int} \rightarrow \text{eventSequence}$
	generator	:	$(\text{nameSequence} \times \text{int} \times \text{constraint}) \rightarrow (\text{eventSequence})$

Figure 0.2 VICE Data types and tools.

Types. Each program that we test is a VirgilProgram, that is, a program in the Virgil programming language [54], which is an object-oriented language for resource-constrained devices. Virgil is a full-fledged language with classes, objects, loops, recursion, etc.

We compile Virgil programs to machineCode, that is, AVR assembly code. The key input to each execution is an eventSequence, which is a list of triples, where each triple consists of an event name (an identifier), an event value (an int), and a wait time (an int that measures

milliseconds). Each of the constraint is a Virgil arithmetic or logical expression. For our benchmarks, we found no need to use other forms of constraints; arithmetic or logical constraints are sufficient for our testing approaches to almost match the static analysis. We leave to future work to investigate whether other benchmarks require use of other forms of constraints to almost match the static analysis.

A prefixTree is a prefix-tree of sequences of event names.

Tools. The tool concolic is a concolic execution engine that executes a Virgil program while firing events from an event sequence, with the specified wait time between consecutive events. The result of a run of concolic is a constraint and the branch coverage that was recorded. We implemented concolic on top of an existing Virgil interpreter. The concolic execution engine works with *concolic values*, that is, a pair of a concrete value and a constraint.

The tool compiler is an open-source Virgil compiler [54] that generates AVR assembly code.

The tool avrora is an open-source simulator for AVR assembly code [55] that executes an AVR assembly code program while firing events from an event sequence, with the specified wait time between consecutive events. The result of a run of \avrora is the maximum stack size that was recorded. A run of avrora is deterministic, hence reproducible. Specifically, avrora measures time in terms of machine cycles and we use the wait times to determine the exact machine cycle at which to fire an event. Additionally, \avrora implements all aspects of the hardware, including the "breaking of a tie" that happens when two events have fired and both handlers are enabled. So, any two runs of avrora on a benchmark and an event sequence always proceed in exactly the same way.

The tool SA-Tree-Gen applies a static analysis to a Virgil program [54]. The static analysis determines conservatively, for each program point, which event handlers are enabled. The result of a run of SA-Tree-Gen is a prefixTree called the SA-Tree that represents the static information as a collection of sequences of event names. According to the static analysis, each sequence of event names can be the basis for an event sequence for which each event will be handled. The SA-Tree avoids names of events that have no chance of being handled because the corresponding event handler is disabled. We can compare the generated SA-Tree with a *full* prefix-tree that represents all possible sequences of event names (up to a given length). For each of our benchmarks, the SA-Tree is a much pruned version of the full tree. Testing approach 6 explores the full prefix-tree.

The tool random takes a nameSequence and a wait time as input and produces an event sequence based on the input nameSequence, with event values generated according to an exponential distribution, and with each wait time equal to the input wait time. The tool generator takes a nameSequence, a wait time, and a constraint, and generates an event sequence. The generator uses the open-source constraint solver Choco [56] [57]to solve the constraint. Notice that we generate event sequences based on source-level information and use them to test code at the assembly level.

Approach. Figure 3 gives pseudo-code for VICE, while Figure 4 illustrates how VICE works. The input to VICE is a Virgil program and a wait time. VICE proceeds in phases that each consists of multiple rounds. Each phase focuses on one nameSequence in the SA-Tree for the Virgil program. In each phase, VICE iterates until two consecutive rounds found no improvement to the maximum stack size or the branch coverage. In each round VICE updates the variable *noChange* to count how many recent rounds had no change. The condition $noChange < 2$ tells when to terminate a phase. The variable *maxStack* contains the maximum

stack size found so far, the variable *branchCoverage* contains the branch coverage found so far, and the variable *seq* holds the current event sequence, which is based on the chosen nameSequence and the input wait time, and which initially has event values chosen randomly.

```
Input:    VirgilProgram p, int waitTime
Output:   int /* the maximum stack size */
Local:    prefixTree tree = SA − Tree − Gen(p)
          machineCode code = compiler(p)
          int maxStack = 0
Method: for each nameSequence ns ∈ tree do {
            int noChange = 0
            float branchCoverage = 0
            eventSequence seq = random(ns, waitT ime)
            while (noChange < 2) {
               int ms = avrora(code, seq)
               (constraint float) (c, bc) = concolic(p, seq)
               seq = generator(ns, waitT ime, c)
               if ((ms > maxStack) _ (bc > branchCoverage))
               then {maxStack = ms; branchCoverage = bc; noChange = 0 }
               else { noChange = noChange + 1 }
            }
        }
        return maxStack
```

Figure 0.3 VICE Algorithm.

We compile each Virgil benchmark program to AVR assembly code. In each round, the algorithm executes both avrora on the assembly code and concolic on the Virgil program to get a new maximum stack size, a new constraint, and a new measure of the branch coverage.

The generator uses a constraint solver to find new event values for an event sequence that otherwise has the same event names and wait times as all other event sequences in the current phase.

A worse alternative. VICE measures maximum stack size at the assembly level in every round of concolic execution. We have experimented with an alternative approach that measures maximum stack size at the *source* level, and only after a completed run measures the maximum stack size at the assembly level for the most challenging event sequence. The alternative approach is faster because it uses the assembly-level simulator just once. However, the results are considerably worse because the source-level stack-size estimates are imprecise.

3.5. Experimental Results

We compare a static analysis and the seven testing approaches listed in Section 3.3. We wrote all the implementations in Java and ran them on Sun Java2 SDK 1.5 on a 2.8 GHz iMac. Most of the runs used less than 60 MB.

We implemented the genetic algorithm on top of the Java Genetic Algorithm Library (JGAL) from http://jgal.sourceforge.net.

For testing approaches 1, 2, 5, 6, our samples of the wait times are 153 ms, 327 ms, and 594 ms.

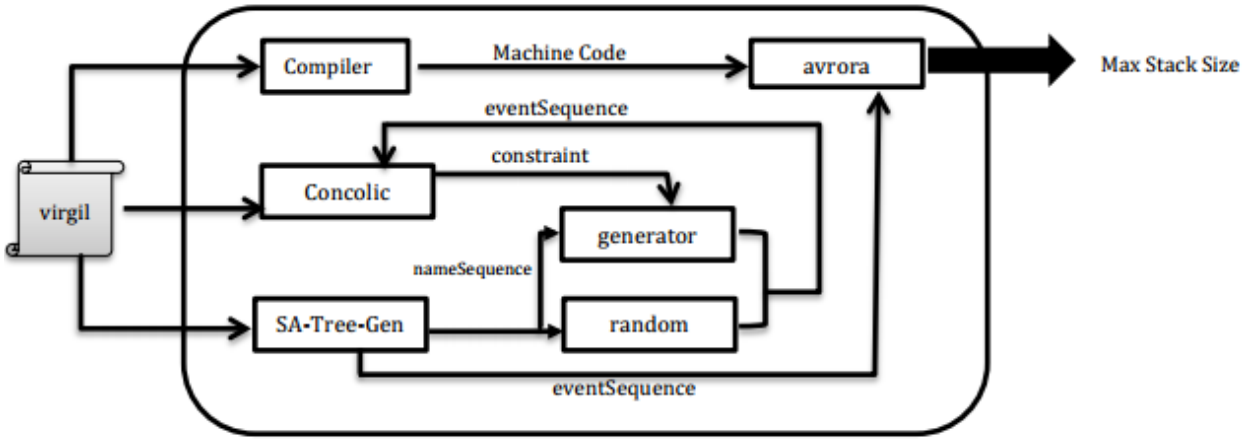

Figure 0.4 Illustration of how VICE works.

For each testing approach we find the maximum stack size of a program in the same way: we first compile the program and then use Avrora to run the assembly code and return the maximum stack size.

3.5.1. Benchmarks

The following table shows some statistics about our seven benchmarks, including the number of lines of Virgil code and also the number of lines of code after translation to C, which is a step on the way in the translation to AVR assembly code. The table also shows the number of event handlers.

Benchmark	LOC (Virgil)	LOC (C)	no. of handlers
TestCon1	329	461	4
TestCon2	347	528	3
StackTest1	293	513	2
StackTest2	251	483	2
TestUSART	1,226	1,737	5
TestSPI	859	1,109	3
TestADC	605	1,055	4

We use four microbenchmarks and three benchmarks that test device drivers for Berkeley Motes. We designed the microbenchmarks testCon1 and testCon2 to test VICE's power to explore different execution paths. These programs have many complex numerical expressions and nested conditional statements and loops. TestCon1 has four event handlers, all without parameters, more than 300 LOC and its nesting depth of control structures is 11. TestCon2 has 3

event handlers each of which has 8 formal parameters, almost 350 LOC, and 37 complex numerical expressions.

The microbenchmark StackTest1 is a more complete version of the example program in Figure 3.1 and includes nested function calls, unreachable code, and atomic structures. StackTest2 consists of nested functions of depth 23.

The TestUSART benchmark tests the operation of the USART driver; the TestSPI benchmark tests the operation of the SPI driver; and the TestADC benchmark tests the operation of the ADC driver.

Previous work [3] has shown that even for programs with a bounded stack, the maximum stack size can grow exponentially in the number of event handlers. The number of handlers in our benchmarks, namely 2--5, is typical of event-driven AVR applications that we have found.

In summary, our benchmarks are nontrivial and turn out to be a major challenge for the previous-best testing approaches.

3.5.2. Measurements

Figure 3.5 shows the maximum stack sizes found by the seven testing approaches (numbered 1--7) and by a static analysis of maximum stack size (labeled SA) that comes with the Avrora distribution. Note that the static analysis guarantees an upper bound on the stack size for every benchmark. This implies that even if each device that generates events should malfunction and generate an event every millisecond, we can rest assured that the stack is bounded by the value given by the static analysis.

Figure 3.6 shows the timings of the testing runs and the timings of running the static analysis. All time measurements are in minutes and are averages of 10 runs after some warm-up runs to fill the caches.

Figure 3.7 shows the branch coverage that each testing approach achieved.

3.5.3. Assessments

In Figure 3.5 the last line gives a geometric mean for each testing approach. The mean is taken over the fractions of the maximum stack size found by the testing approach and the maximum stack size found by static analysis. For example, for testing approach 1, we take the geometric mean of these fractions:

Similarly, in Figure 3.6 the last line gives a geometric mean for each testing approach; the denominator is the execution time of testing approach 7 (which is DTall). In Figure 3.7 the last line gives a geometric mean for each testing approach.

Figure 3.8 shows a plot of the mean percentages in Figures 3.5 and 3.6; note that the x-axis uses a log-scale.

Benchmark	1	2	3	4	5	6	7	SA
TestCon1	318	441	417	455	505	506	511	516
TestCon2	366	612	798	703	846	866	882	894
StackTest1	421	353	318	619	703	749	958	979
StackTest2	353	324	390	420	564	564	564	566
TestUSART	459	481	472	525	664	664	664	665
TestSPI	490	350	481	490	518	522	529	533
TestADC	247	306	283	302	306	306	308	310
% of SA	62	67	71	81	94	95	99	100

Figure 0.5 Maximum stack sizes in bytes. The last line gives a geometric mean.

Benchmark	1	2	3	4	5	6	7	SA
TestCon1	7.21	8.83	281	38	1.53	16	439	0.10
TestCon2	10.11	3.11	173	29	2.48	45	381	0.12
StackTest1	12.92	2.55	179	23	0.56	26	307	0.05
StackTest2	2.85	2.29	165	72	4.13	56	266	0.05
TestUSART	7.44	3.05	204	43	1.18	16	452	0.32
TestSPI	3.99	3.11	197	26	0.79	9	393	0.15
TestADC	3.01	1.37	289	33	0.45	6	444	0.13
% of (7)	1.6	0.4	55	9	0.3	5	100	0.11

Benchmark	1	2	3	4	5	6	7
TestCon1	23	56	61	72	92	93	94
TestCon2	21	60	78	78	89	89	90
StackTest1	26	40	73	80	64	71	73
StackTest2	20	43	69	81	99	99	99
TestUSART	23	58	66	85	96	96	96
TestSPI	32	56	71	75	67	73	75
TestADC	22	62	69	78	98	98	98
% of (7)	24	53	69	78	85	88	89

Figure 0.6 Branch coverage in percent. The last line gives a geometric mean.

Testing approaches 1-3. Testing approach 1 uses a total of 3,000 random event sequences and the result is a stack-size-fraction mean of 62%. Testing approach 2 uses a genetic algorithm to improve the choice of event names, and that improves the stack-size-fraction mean to 67%. Testing approach 3 goes further by trying all combinations of event names and all integer wait times within a wide interval; the stack-size-fraction mean goes up to 71%. Note that testing approach 2 is almost two orders of magnitude faster than testing approach 3. Note also that in some cases testing approach 3 gives *worse* results than testing approach 2 because of poorer samples of the event values. Notice finally that the genetic algorithm in most cases is faster than random testing. The reason is that the procedure for generating random event sequences is quite slow, while one of the main ways the genetic algorithm produces new event sequences is to swap subsequences from existing event sequences.

Testing approaches 4-7. Testing approach 4 samples the event names and tries all integer wait times within a wide interval; the result is a stack-size-fraction mean of 81%. Thus, testing approach 4 dominates testing approaches 1--3 so we conclude that the use of directed testing to determine event values is essential to get good results. Testing approach 5 is the VICE approach, which, in sharp contrast to testing approach 4, samples the wait times but uses our SA-Tree technique to generate event names. VICE is 30x faster than testing approach 4 and yet it produces a better stack-size-fraction mean, namely 94%. Note also that VICE is within 3x of the running time of the static analysis. Testing approach 6 tries all combinations of event names and samples the wait times. The result is marginally better than VICE, namely 95%, but more than an order of magnitude slower. Finally, testing approach 7 is the DTall approach which tries all combinations of event names and all integer wait times within a wide interval. DTall achieves a result of 99%, though at the expense of the longest execution time of all the approaches. We conclude that testing *can* almost match the static analysis, which shows that the static analysis is about as good as it can be. We also conclude that VICE gives an excellent trade-off between

precision and execution times; it is faster than all the other testing approaches and it is outperformed only by two much slower approaches.

Number of event sequences. VICE achieves its results with significantly fewer event sequences than random testing and the genetic algorithm. For four benchmarks, the difference is 2X, while for three benchmarks, the difference is 10X.

Branch coverage. Figure 3.7 shows that VICE and DTall produce excellent branch coverage numbers. Notice that the previous best testing-approach (approach 2) achieved a much lower branch coverage (53 percent) than VICE (85 percent) and DTall (89 percent). The wide spread of coverage numbers support that the benchmarks are nontrivial: we can find event sequences that lead most branches to go either way and yet only the best testing approaches achieve that.

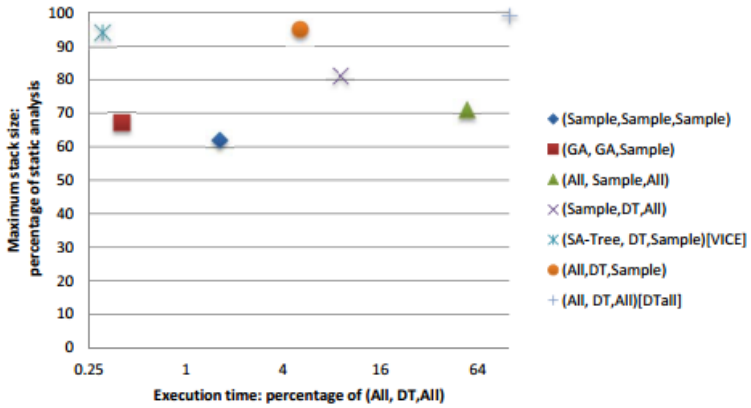

Figure 0.7 Comparison of seven testing approaches.

CHAPTER 4

Race Directed Scheduling of Concurrent Programs

Detection of data races in Java programs remains a difficult problem. The best static techniques produce many false positives, and also the best dynamic techniques leave room for improvement. We present a new technique called race directed scheduling that for a given race candidate searches for an input and a schedule that lead to the race. The technique is implemented in a tool namely *Racageddon*. The search iterates a combination of concolic execution and schedule improvement, and turns out to find useful inputs and schedules efficiently. We use an existing technique to produce a manageable number of race candidates. Our experiments on 23 Java programs found 72 real races that were missed by the best existing dynamic techniques. Among those 72 races, 31 races were found with schedules that have between 1 million and 108 million events, which suggest that they are rare and hard-to-find races.

4.1. Two Techniques from Previous Works

Racageddon uses two techniques from previous work [12] [58]. In both cases, *Racageddon* uses those techniques as "black boxes", that is, as unmodified components for which we rely only on their input-output behavior. We implemented both techniques ourselves after a careful study of the seminal papers [12] [58].

Generation of race candidates. We use a hybrid race detector by O'Callahan and Choi [12] that we call Hybrid. Hybrid combines lockset-based detection and happens-before-based detection into a single efficient technique that can produce both false positives and false negatives. We view the output of Hybrid as *race candidates* that deserve further attention. Hybrid provides a rather small number of race candidates, namely a total of 405 for our benchmarks of more than 4.5 million lines of code. Those 405 race candidates are an excellent starting point for our search for real races.

Schedule improvement. We use an approach to schedule improvement by Said, Wang, Yang, and Sakallah [58]. Their method maps a schedule to a permutation of the schedule. The idea is that a user supplies both a schedule that represents a trace of a program execution and also a race candidate, and then in return gets a schedule that has a better chance to lead to the race. The method has "memory": it takes advantage of the schedules that have been submitted in all previous calls. Together, all those schedules provide a wealth of information about happens-before relationships in a specific program. The method uses an SMT-solver and is highly efficient, even for the schedules of lengths beyond length 10^8 that we encountered in our experiments.

4.2. Race Directed Scheduling

We now present our approach to data race detection. We will use pseudo-code to describe both our approach and the data types that we use.

4.2.1. Data Types

We begin with a description of six data types that we use in *Racageddon*.

Program	=	a Java 6 program
Input	=	input to a Java 6 program
Event	=	threadID × statementLable
EventPair	=	Event × Event
Race	=	EventPair × Input × Schedule

Racageddon 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.

When a program execution executes a particular statement in a particular thread, we refer to that as an *event* that has type Event. In the context of race detection, the key data type is EventPair that we use to describe two events that may form a race.

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

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

4.2.2. Two Tools

Let us describe the interfaces to the two off-the-shelf tools from Section 2 in terms of the data types listed above.

hybrid	Program → (EventPair set)
improve	(Schedule × EventPair) →(Schedule⊕ {none})

Here hybrid stands for O'Callahan and Choi's technique, while improve stands for Said, Wang, Yang, and Sakallah's technique. Notice that hybrid maps a Java program to a set of event pairs, that is, a set of race candidates. Notice also that improve maps a schedule to a better schedule or else to none if no better schedule was found. Notice finally that we leave implicit that improve has "memory" and takes advantage of the schedules that have been submitted in all previous calls.

4.2.3. Concolic Execution

Let us describe the interfaces to the two off-the-shelf tools from Section 3.2 in terms of the data types listed above. *Racageddon* uses concolic execution as one of its components. We will summarize the idea of concolic execution and we will introduce a slight generalization of the approach that we use in *Racageddon*.

Concolic execution [39] [59] [40] [37] [38] [60], executes code with concrete and symbolic values simultaneously and uses the result to generate inputs for another execution. The term "concolic" combines the words "concrete" and "symbolic". Each execution collects constraints from the symbolic values and the conditions in the control-flow. Those constraints represent the executed control-flow path and they have *the concrete input to the run* as solution.

Suppose we want to execute a particular event, that is, a particular statement in a particular thread. We can execute a sequence of concolic runs that successively get closer and closer to execute the desired event. The idea is to do a minor modification of the constraints collected from conditions of branches. Imagine that a prefix of the concolic run made progress towards the desired event but at a particular branch B went off in nonpromising direction. We take the constraints from the prefix plus the *negation* of B. The solution to those constraints is an input that will steer the next concolic execution a little closer to the desired event by going off in the promising direction at branch B.

Experience shows that concolic execution achieves better branch coverage with fewer test cases than testing with random inputs. In the first round of concolic execution, the input is chosen randomly.

We can generalize the standard approach to pursue execution of an entire schedule, that is, an event sequence. For example, suppose we want execution of the schedule (e_1, e_2, e_3). Some rounds of concolic execution may lead to execution of e_1. We can refer to those rounds together as a super-round. Now we can use the constraints that lead to execution of $e_1$and continue with a second super-round that leads to execution of first $e_1$and later e_2. Finally, we can do a third super-round and achieve execution of the entire schedule.

The above method generalizes easily to schedules of any length. If we manage to execute an entire given schedule, we continue to explore additional schedules that have the given schedule as prefix.

We describe our interface to concolic execution in the following way.

Concolic = (Program × Schedule) → ((Race set) × Schedule)

The input to concolic is a program and a schedule, and concolic will execute one super-round per element in the schedule. A run of concolic has two outputs. The first output is a set of all races that were found by any of the individual concolic executions. The second output is a schedule that represents the trace of final concolic execution, irrespectively of whether the given schedule was executed. We emphasize that each call to concolic may do many concolic executions, hence have many opportunities to collect races.

4.2.4. Helper Functions

We use three helper functions:

Informally, present checks that the two elements of an event pair occur consecutively in a schedule. Additionally, swap makes a change to each element $((e', e''), v, s)$ of a race set, namely to swap e' and e'' both in the first component of the triple and also where they first occur consecutively in s. Finally, $\uplus$ does something akin to a union of two race sets, namely to do the union based only on the event pair of each race. We will maintain the invariant that for a given $c \in$ EventPair, a race set contains at most one race of the form (c, v, s) . The idea of $X \uplus Y$ is that if X contains a race of the form (c, v', s') , and Y contains a race of the form (c, v'', s''), then $X \uplus Y$ will, somewhat arbitrarily, contain the first race (c, v', s') (and leave out (c, v'', s'')).

Formally,

$$\text{present}\big((e', e''), (e_1, \dots, e_n)\big) = f(x) = \begin{cases} true, & if\ \exists\, i : e' = e_i \wedge e'' = e_{i+1} \\ false, & otherwise \end{cases}$$

$$\text{swap}(X) = \{((e'', e'), v, (e_1, \dots, e_{i-1}, e_{i+1}, e_i, e_{i+2}, \dots, e_n)) \mid \big((e', e''), v, (e_1, \dots, e_n)\big) \in X \wedge i \in 1..(n-1) \text{ is the smallest index such that: } e' = e_i \wedge e'' = e_{i+1}\}$$

For every $X \in$ (Race set) we assume that if $(c', v', s') \in X$ and $(c'', v'', s'') \in X$ and $c' = c''$, then $v' = v''$and $s' = s''$. The following definition of $\uplus$ maintains this property.

$$X \uplus Y = X \cup \{((e'.e''), v, s) \in Y \mid \forall (e_x, v_x, s_x) \in X : e_x \neq (e', e'')\}$$

4.2.5. Racageddon Overview

Racageddon iterates a combination of concolic execution and schedule improvement. We begin with a run of hybrid to produce candidate races and then we do two phases of search for races. In the Phase 1 we do a separate search for each of the candidate races. In the Phase 2 we do a search based on the races found in Phase 2. For our benchmarks, our experiments with *Racageddon* found 291 real races in Phase 1 and 53 additional real races in Phase 2.

In Phase 1 we interleave calls to concolic and improve. The idea is to turn the search for a race into a search for a schedule that leads to the race. Each call to concolic will produce a more promising schedule, after which a call to improve will further improve that schedule. In more detail, each call to concolic 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 race. The call to improve will permute some events in the schedule to make the next concolic run have a better chance to succeed.

In Phase 2 we consider each race found in Phase 1 and do a swap of the two racing events in the schedule that lead to the race. The "swapped" schedule leads to a race of the same two events, which in itself provides nothing new. The interesting aspect of the "swapped" schedule is that a concolic execution will continue after the race and may proceed in a different way than the execution in Phase 1. Our experience is that those continued executions may find races that Phase 1 missed. Once Phase 2 finds a new race, we also do a swap of the schedule that led to that new race.

4.2.6. *Racageddon* Pseudo-code

Figure 1 shows pseudo-code for *Racageddon*. We will now go over the pseudo-code in detail. We hope our pseudo-code and explanation will enable a better understanding of the approach and enable practitioners to implement *Racageddon* easily.

The input to the *Racageddon* procedure is a program while the output is a set of races. The first four lines of *Racageddon* declares these four variables: (1) a set of race candidates, called *candidates*, that we initialize by a call to hybrid, (2) a set of races, called races, that initially is the empty set and that we eventually return as the result of the procedure, (3) a set of races, called r, that we use to hold intermediate results, and (4) a schedule, called *trace*, that holds each trace produced by concolic.

Phase 1 consists of a for-each-loop that tries each of the event pairs in the set of candidates. For each event pair we use a while-loop to do iterations that each does one call to improve and one call to concolic. We use the integer variable i to count the number of iterations and we bound i by 1000 to ensure that the search terminates, even if unsuccessful. In practice, the highest number of calls to improve and concolic we did for any of our benchmarks was 197. So, none of our experiments exercised the condition $i \leq 1000$. We initialize *trace* to empty schedule, denoted by ϵ, such that the initial call to improve can work correctly; that call will return ϵ.

The while-loop uses a Boolean-variable *done* to keep track of whether the search for a particular candidate can be terminated before i reaches 1000. We have two reasons for

terminating the search early, which we done by setting *done* to true. If the candidate pair c is present in the *trace* executed by concolic, as found by the call present($e, trace$), then we can declare success and terminate the search. If the call to improve($e, trace$) returns none, then the search has stalled, and we abandon the search. While abandoning a search may seem sad, our experiments do it in some cases. One of the reasons may be that the race candidate actually isn't a real race!

Notice how each iteration of the while-loop begins with trace, improves it to a schedule s (unless improve returns none), which then after execution of concolic turns into a new value for *trace*.

```
(Race set) Racageddon (Program p) {
   EventPair set) candidates = hybrid(p)
   (Race set) races = ∅
   (Race set) r
   Schedule trace

   /* Phase 1: try the candidates */
   for each EventPair c 2 candidates do {
      boolean done = false
      int i = 0
      traces = ϵ
      while (! done) ∧ (i ≤ 1000){
         case improve(c, trace) of
            Schedule s : {
                        (r, trace) = concolic(p, s)
                        races = races ⊎ r
                        done = present(c, trace)
                  }
                  none: {done = true}
         }
         i = i + 1
         }
}

/* Phase 2: try swaps of the races */
(Race set) workset = swap(races)
for each Race (c, v, s) ∈ workset do {
                  (r, trace) = concolic(p, s)
                  races = races ⊎ r
                        workset = workset ⊎ swap(r)
                  }
return races
}
```

Figure 0.1 Racageddon Algorithm.

Phase 2 is a *workset* algorithm that uses the variable *workset* that holds a set of races. Initially *workset* is the set of races found in Phase 1, but swapped, in the sense that we now want to search for the "swapped" race. The main part of Phase 2 is a for-each-loop that iterates over the elements of *workset*. We use an advanced for-each-loop that works correctly even if elements are added to *workset* during a run of the for-each-loop. Here, "works correctly" means that the for-each-loop does one iteration per element of workset, even if an element is added to *workset* multiple times or added after the execution of the for-each-loop begins.

For each element of *workset*, Phase 2 makes one call to concolic and collects any races that may be found. For each new race found in Phase 2, we add the race to *workset* such that we eventually can say that we tried the "swapped" version of every race that we found.

4.2.7. Example

We now present an example in which we walk through a run of \Racageddon\on this program with three shared variables and two threads:

$$x, y, z \text{ are shared variable}$$
$$z \text{ has an initial value received } from \text{ user input}$$

Thread 1:	Thread 2:
$l_1: x = 6$	$l_4: x = 2$
$l_2: if(z > 4)$	$l_5: if(z\text{^}2 + 5 < x\text{^}2)$
$l_3: \quad y = 5$	$l_6: \quad y = 3$

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

The call to hybrid produces two race candidates:

$$candidates = \{(e_1, e_4), (e_1, e_5)\}$$

Now we begin Phase 1 of *Racageddon*. Suppose the for-each loop first considers the candidate(e_1, e_4).

Now we run the first iteration of the while-loop. Initially *trace* is the empty schedule so improve returns the empty schedule. Now we run concolic on the empty schedule. Suppose that the initial random input, which becomes the values of the shared variable z, is 0. Nondeterminism can lead to several traces; suppose we get

$$trace = e_1, e_2, e_4, e_5$$

Notice here that we don't get to $e_3$because the condition in $e_2$fails due to $0 < 4$, and we don't get to e_6 because the condition in e_5 fails due to $z^2 + 5 = 5$ and $x^2 = 4$ and $5 > 4$.

Now we run the second iteration of the while-loop. First we run improve on (e_1, e_4) and *trace*:

$$trace = e_1, e_4, e_2, e_5$$

Now we run concolic on trace, and like above, let us suppose the initial random input leads to $z = 0$. The execution of concolic finds the race for which we are searching, so we can add that race to *races*:

$$races = \{((e_1, e_4), 0, (e_1, e_4, e_2, e_5))\}$$

Like above, we don't get to execute e_3 or e_6; the conditions in e_2 and e_5 fails for the same reasons as above.

Next the for-each-loop in Phase 1 considers the candidate (e_1, e_5).

Now we run the first iteration of the while-loop. Let us assume that this iteration proceeds like the first iteration for (e_1, e_4) so we get:

$$trace = e_1, e_4, e_2, e_5$$

Now we run the second iteration of the while-loop. First we run improve on , (e_1, e_5) and *trace*, which produces this permutation of *trace*:

$$trace = e_4, e_5, e_1, e_2$$

Notice that even though e_5 and e_1 occur consecutively, we won't terminate the search because we are looking for (e_1, e_5). Now we run concolic on *trace*, and which leads to an execution with this trace:

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

for which z had the initial value 10. (We skip the constraints and merely note that they have solution 10, among other solutions.) Note that *trace* contains e_3 because the condition in e_2 succeeds due to 10>4.

Now we run the third iteration of the while-loop. First we run improve on , (e_1, e_5) and *trace*, which produces this permutation of *trace*:

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

Next, the execution of concolic finds the race for which we are searching, so we can add that race to *races*:

$$races = \{((e_1, e_4), 0, (e_1, e_4, e_2, e_5)),$$
$$((e_1, e_5), 10, (e_4, e_1, e_5, e_2, e_3))\}$$

We don't get to execute e_6 because the condition in $e_5$fails due to $z^2 + 5 = 105$ and $x^2 = 36$ and 105>36.

Now the for-each-loop has processed both elements of the set *candidates*, so we are done with Phase 1 and can move on to Phase 2. Notice that we successfully found both candidate races to be real races.

In Phase 2 we consider swapped versions of the two races found in Phase 1:

$$workset = \{((e_1, e_4), 0, (e_1, e_4, e_2, e_5)),$$
$$((e_1, e_5), 10, (e_4, e_1, e_5, e_2, e_3))\}$$

Let us here focus on the run with the schedule (e_4, e_1, e_2, e_5). The call to concolic eventually executes $(e_4, e_1, e_2, e_5, e_3)$ and collect these constraints:

$$x = 6 \wedge z > 4 \wedge x^2 + 5 < x^2$$

that have solution $z = 5$. The next concolic execution therefore executes $(e_4, e_1, e_2, e_5, e_3, e_6)$, which contains the race (e_3, e_6). We add that race to *races*:

$$races = \{((e_1, e_4), 0, (e_1, e_4, e_2, e_5)),$$
$$\big((e_1, e_5), 10, (e_4, e_1, e_5, e_2, e_3)\big)$$
$$(e_3, e_6), 5, (e_4, e_1, e_2, e_5, e_3, e_6)\}$$

In summary, hybrid produced two candidates races, Phase 1 found both candidates to be real races, and Phase 2 found one additional race.

4.3. Experimental Results

We ran all our experiments on a Linux CentOs machine with two 2.4 GHz Xeon quad core processors and 32 GB RAM.

Name	LOC	# threads	Brief description
Sor	1270	5	A successive order-relaxation benchmark
TSP	713	10	Traveling Salesman Problem solver
Hedc	30K	10	A web-crawler application kernel
Elevator	2840	5	A real-time discrete event simulator
ArrayList	5866	26	ArrayList from java.util
TreeSet	7532	21	TreeSet from java.util
HashSet	7086	21	HashSet from java.util
Vector	709	10	Vector from java.util
RayTracer	1942	5	Measures the performance of a 3D raytracer
MolDyn	1351	5	N-Body code modeling dynamic
MonteCarlo	3619	4	A financial simulator, using Monte Carlo techniques to price products
Derby	1.6M	64	Apache RDBMS
Colt	110K	11	Open Source Libraries for High Performance Scientific and Technical Computing
ChordTest	62	11	Mini-benchmark; comes with the Chord race detector
Avrora	140K	6	AVR microcontroller simulator
Tomcat	535K	16	Tomcat Apache web application server
Batic	354K	5	Produces a number of Scalable Vector Graphics (SVG) images based on Apache Batic
Eclipse	1.2M	16	Non-GUI Eclipse IDE
FOP	21K	8	XSL-FO to PDF converter
H2	20K	16	Executes a JDBCbench-like in-memory benchmark
PMD	81K	4	Java Static Analyzer
Sunflow	108K	16	Tool for rendering image with raytracer
Xalan	355K	9	XML to HTML transformer
TOTAL	4587K		

Figure 0.2 Benchmarks.

4.3.1. Benchmarks

Figure 4.2 lists our 23 benchmarks which we have collected from seven sources:

- From ETH Zurich: Sor, TSP, Hedc, Elevator.
- From java.util, Oracle's JDK 1.1: ArrayList, TreeSet, HashSet, Vector.
- From Java Grande: RayTracer, MolDyn, MonteCarlo.
- From the Apache Software Foundation: Derby.
- From European Organization for Nuclear Research (CERN): Colt.
- From the Chord distribution: ChordTest.
- From DaCapo [106]: 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 3 small (less than 1K LOC) benchmarks.

Figure 4.2 also lists the high watermark of how many threads each benchmark runs.

4.3.2. Race Detectors

We compare *Racageddon* with one static race detector, namely Chord [24], one hybrid race detector, namely the one that we call Hybrid [12], and four dynamic race detectors, namely FastTrack [61], Goldilocks [62], CalFuzzer [60], and Pacer [63]. Additionally we compare with a combined dynamic technique that we call FGCP.

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

We discussed Hybrid in Section 2.

FastTrack, Goldilocks, CalFuzzer, Pacer, and Racageddon are all dynamic techniques that report only real races.

FastTrack and Goldilocks are based on the observation that a race happens if two accesses to a memory location (of which at least one access is a write) are not ordered by the happens-before relation. FastTrack uses a clever representation of the happens-before relation to achieve constant-time overhead for almost all monitored operations. Goldilocks uses a lockset-based algorithm to improve the precision of the computation of the happens-before relation.

CalFuzzer performs random testing by choosing thread schedules at random and stopping a thread when it is about to execute a statement in a candidate race pair. Like *Racageddon*, CalFuzzer uses Hybrid to generate race candidates.

Pacer is a sampling-based data race detector that detects any race at a rate equal to the sampling rate. In our experiments, the sampling race was 100% and for each benchmark we used 100 trials.

We use FGCP to stand for the union of FastTrack, Goldilocks, CalFuzzer, and Pacer in following sense. We can implement FGCP as a tool that for a given benchmark starts runs of FastTrack, Goldilocks, CalFuzzer, and Pacer in four separate threads, and if any one of them reports a race, then FGCP reports a race.

4.3.3. How we handle Reflection

Many of the benchmarks use reflection, yet each of the race detectors listed above either doesn't support reflection or supports reflection poorly. We overcome this problem with the help of the tool chain TamiFlex [64].

The core of the problem is that all the race detectors do either a static analysis or some form of ahead-of-time instrumentation. Reflection tends to make static analysis unsound and to 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 runs deviates from the recorded runs, TamiFlex issues a warning.

We have combined each of the race detectors with TamiFlex and we have run all our experiments without warnings. As a result, the race detectors all handle reflection correctly and in the same way.

The webpage https://code.google.com/p/tamiflex/wiki/DaCapoAndSoot gives a good example of how to combine TamiFlex with a different tool.

	Number of races found		
Name	Total=phase1+ phase 2		
Sor	3	2	1
TSP	2	2	0
Hedc	11	9	2
Elevator	8	5	3
ArrayList	7	7	0
TreeSet	3	3	0
HashSet	8	7	1
Vector	4	4	0
RayTracer	4	3	1
MolDyn	6	4	2
MonteCarlo	3	2	1
Derby	18	15	3
Colt	10	7	3
ChordTest	2	2	0
Avrora	13	12	1
Tomcat	21	19	2
Batic	29	23	6
Eclipse	51	46	5
FOP	18	16	2
H2	39	30	9
PMD	13	12	1
Sunflow	30	22	8
Xalan	41	39	2
TOTAL	344	291	53

Figure 0.3 Races found by Racageddon.

4.3.4. Measurements

Figure 3 shows the numbers of races found in 23 benchmarks by *Racageddon*, including whether the races were found in Phase 1 or in Phase 2.

Figure 4 shows, for each benchmark, the number of schedules tried by *Racageddon* and the longest schedule that found a race.

Figure 5 shows the numbers of races found in 23 benchmarks by 7 techniques.

Figure 6 shows the time each of the runs took in minutes and seconds, and it shows the geometrical mean for each technique.

Some of the executions of Goldilocks crashed, which we indicate in Figure 5 and Figure 6 with "-". If we compare Figure 5 and Figure 6 we see that for ArrayList and Batic, we list that

Goldilocks reported races while we list no execution times. The reason is that for ArrayList and Batic, our runs of Goldilocks crashed, yet the execution log contained some races that we report in Figure 5.

Figure 7 shows, for each benchmark, the lengths of the 72 schedules that lead to races found only by *Racageddon*.

Figure 8shows, for each benchmark, how many of the races found by Hybrid are actually real races, as found by the combination of FGCP and *Racageddon*.

Name	Schedule	longest schedule that found a race
Sor	14	6,803
TSP	8	6,047
Hedc	28	249,268
Elevator	28	9,005
ArrayList	47	132,990
TreeSet	17	110,087
HashSet	38	139,553
Vector	40	6,308
RayTracer	9	71,084
MolDyn	188	4,680
MonteCarlo	24	12,061
Derby	105	108,302,900
Colt	63	948,033
ChordTest	2	505
Avrora	23	702,961
Tomcat	197	1,284,917
Batic	39	1,407,554
Eclipse	53	102,879,384
FOP	41	153,074
H2	35	297,655
PMD	48	310,049
Sunflow	37	1,624,320
Xalan	56	2,907,450

Figure 0.4 Schedules tries by Racageddon.

4.3.5. Evaluation

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

Racageddon. We can see in Figure 3 that *Racageddon* found a total of 344 real races, including 291 races found in Phase 1 and 53 races found in Phase 2. The split between Phase 1 and Phase 2 demonstrates a subtlety of race directed scheduling: even when we have a schedule that finds a race, a swap of the race pair can lead to other races.

Number of schedules. We can see in Figure 4 that the number of schedules tried by *Racageddon* is rather modest and appears to be no worse than the product of a small constant and the number of race candidates. Note that in *Racageddon*, some runs of concolic finds multiple races. We can also see in Figure 4 that the longest schedules that found races can have lengths that are more than 100,000,000. This shows that the improve method scales to long schedules.

Racageddon versus other Dynamic Techniques. We can see in Figure 5 that *Racageddon* finds the most races (344) of all the dynamic techniques. Among those 344 races, 72 races were found only by *Racageddon* and are entirely novel to this paper, while 272 were also found by FGCP. Dually, 32 races were found only by FGCP. In summary, we have that the combination of FGCP and *Racageddon* found 376 races in the 23 benchmarks.

Found only by FGCP:	32
Found by both:	272
Found only by :	72
Total:	376

FastTrack versus Pacer. Pacer is based on FastTrack and as expected, every race found by FastTrack is also found by Pacer. Pacer finds many more races (286) than FastTrack (79) so our experiments confirm that Pacer is a highly successful extension of FastTrack.

FGCP details. The combined dynamic technique FGCP found 304 races. Pacer was the biggest contributor to that collection of 304 races. Among those 304 races, Pacer found 286, some of which were also found by Goldilocks and CalFuzzer. The remaining 304-286=18 races were found Goldilocks (10 races) and CalFuzzer (8 races). In more detail, Goldilocks found additional races in Avrora (1), Batic (3), FOP (2), SunFlow (2), and Xalan (2) (and CalFuzzer found none of those 10 races). CalFuzzer found additional races in TreeSet (1), HashSet (1), Derby (1), Eclipse (4), and H2 (1) (and GoldiLocks found none of those 8 races). We conclude that Goldilocks, CalFuzzer, and Pacer are all worthwhile techniques that each finds races that the other techniques don't find. As a combined dynamic technique FGCP is highly powerful.

	Static	Hybrid	Dynamic							
benchmarks	Chord	Hybrid	FastTrack	Goldilocks	CalFuzzer	Pacer	FGCP	Racageddon		
								total	new	FGCP
Sor	3	8	0	0	0	3	3	3	3	0
TSP	17	3	1	1	0	1	1	2	1	1
Hedc	143	5	3	1	1	11	11	11	4	7
Elevator	54	13	1	-	0	4	4	8	4	4
ArrayList	8	14	0	1	5	6	6	7	1	6
TreeSet	11	13	0	-	6	8	9	3	0	3
HashSet	0	11	0	-	8	7	8	8	0	8
Vector	17	9	0	-	5	5	5	4	0	4
RayTracer	159	2	1	1	1	3	3	4	1	3
MolDyn	92	43	0	1	2	5	5	6	1	5
MonteCarlo	101	5	0	0	1	2	2	3	1	2
Derby	1110	21	1	-	2	14	15	18	4	14
Colt	549	13	0	0	3	7	7	10	3	7
ChordTest	2	2	1	1	2	2	2	2	0	2
Avrora	1887	9	3	3	6	11	12	13	1	12
Tomcat	110061	52	12	11	11	20	20	21	3	18
Batic	970	12	9	10	9	32	35	29	7	22
Eclipse	9401	77	14	-	13	39	43	51	8	43
FOP	34	21	5	5	8	13	15	18	3	15
H2	869	19	5	-	9	25	26	39	13	26
PMD	292	14	9	8	4	13	13	13	0	13
Sunflow	353	16	8	11	9	19	21	30	11	19
Xalan	1003	23	6	9	10	36	38	41	3	38
TOTAL	127136	405	79	63	115	286	304	344	72	272

Figure 0.5 The numbers of races found in 23 benchmarks by 7 techniques.

Chord. Chord is possibly the best current static race detector, yet our experiments strongly suggest that Chord finds a large number of false positives. We conclude that accurate static race detection continues to be an open problem.

Timings. The geometrical means of the execution times for each technique show that FastTrack and Hybrid are the fastest, while Pacer is the slowest. *Racageddon* is more than twice as fast as Pacer yet *Racageddon* finds significantly more races. Note that the timings for CalFuzzer and *Racageddon* include the time to execute Hybrid.

	Static	Hybrid	Dynamic					
benchmarks	Chord	Hybrid	FastTrack	Goldilocks	CalFuzzer	Pacer	FGCP	Racageddon
Sor	2:18	0:49	0:08	0:44	2:29	9:44	4:49	2:18
TSP	2:22	0:55	0:03	0:10	1:50	11:23	4:37	2:22
Hedc	4:07	1:00	0:08	0:25	2:01	5:00	3:08	4:07
Elevator	1:10	0:39	0:03	-	1:11	3:58	2:40	1:10
ArrayList	2:40	0:50	0:05	-	1:18	5:18	4:11	2:40
TreeSet	3:11	0:18	0:06	-	0:44	7:02	3:25	3:11
HashSet	2:58	0:21	0:06	-	0:59	4:57	2:43	2:58
Vector	0:43	0:15	0:01	-	0:38	5:05	2:52	0:43
RayTracer	1:24	0:09	0:03	0:38	0:26	4:18	2:22	1:24
MolDyn	0:38	1:42	0:02	1:08	2:49	15:36	6:45	0:38
MonteCarlo	2:31	2:02	0:04	1:16	4:01	16:31	6:58	2:31
Derby	35:09	1:26	0:13	-	1:50	11:34	5:02	35:09
Colt	4:37	0:04	0:10	0:23	0:09	4:48	2:23	4:37
ChordTest	0:05	0:01	0:01	0:02	0:05	0:54	0:10	0:05
Avrora	19:37	2:40	0:39	4:57	3:19	23:03	11:17	19:37
Tomcat	12:01	3:57	0:41	4:11	6:01	45:12	19:00	12:01
Batic	27:29	3:01	0:18	-	3:55	30:01	14:54	27:29
Eclipse	41:11	3:50	0:35	-	4:14	48:46	19:15	41:11
FOP	6:50	0:17	0:12	0:36	0:25	13:21	4:49	6:50
H2	8:38	0:31	0:09	-	0:49	18:50	7:31	8:38
PMD	15:48	0:16	0:14	1:03	0:38	17:41	7:22	15:48
Sunflow	16:00	0:41	0:23	2:01	1:06	18:17	6:03	16:00
Xalan	33:11	2:39	0:20	3:00	3:47	30:37	13:19	33:11
geom. mean	4:36	0:40	0:08	-	1:16	10:41	4:51	4:36

Figure 0.6 Timings in minutes and seconds.

Rare and frequent races. In a seminal paper, Marino, Musuvathi, and Narayanasamy [66] made a distinction between rare and frequent races:

> *"We classified as rare those racing instruction pairs that occurred fewer than 3 times for each million non-stack memory instructions executed. The rest are considered frequent."*

A related idea stems from Burckhardt, Kothari, Musuvathi, and Nagarakatte [65] who characterized the depth of a bug as the minimum number of scheduling constraints required to find that bug. In the spirit of these ideas, let us consider whether *Racageddon* finds any rare races. Figure 4 lists the longest schedule that *Racageddon* used to find a race for each benchmark. Six of those schedules have more than a million events, including one schedule with more than 100 million events. For 18 of those longest schedules, the result was that *Racageddon* found a race that FGCP didn't find. The exceptions are TSP, Elevator, Vector, MolDyn, and ChordTest, and we notice that those five benchmarks have some of the shortest "longest schedules" among the benchmarks.

Figure 7 lists the lengths of the 72 schedules that lead to races found only by *Racageddon*.

Name	Lengths
Sor	6462, 6661, 6803
TSP	5623
Hedc	57327, 224341, 236804, 249268
Elevator	6573, 7924, 8673, 8914
ArrayList	132990
TreeSet	-
HashSet	-
Vector	-
RayTracer	71084
MolDyn	4305
MonteCarlo	12061
Derby	58483566, 98555637, 105053813, 108302900
Colt	824877, 919592, 948033
ChordTest	-
Avrora	702961
Tomcat	1066481, 1169274, 1284917
Batic	182982, 323737, 760003, 1379402, 1393478, 1400516, 1407554
Eclipse	1697703, 3068331, 3429715, 16605080, 16785570, 77145639, 98049000, 102879384
FOP	134705, 150499, 153074
H2	32742, 116085, 217288, 232170, 241100, 264912, 273842, 276819, 279795, 285748, 294678, 296133, 297655
PMD	-
Sunflow	374598, 730944, 1283212, 1348185, 1478131, 1494379, 1543108, 1575594, 1608075, 1620019, 1624320
Xalan	2674854, 2849301, 2907450

Figure 0.7 The lengths of the 72 schedules that lead to races found only by Racageddon.

We can groups those lengths as follows:

lengths	#
$10^3 - 10^4$	9
$10^4 - 10^5$	4
$10^5 - 10^6$	28
$10^6 - 10^7$	22
$10^7 - 10^8$	6
$10^8 - 10^9$	3

The table shows that many of those schedules are long, hence rare. Specifically, 31 races were found with schedules that have between 1 million and 108 million events, which suggests that they are rare and hard-to-find races.

Hybrid. Both CalFuzzer and *Racageddon* use Hybrid to produce race candidates. CalFuzzer focuses solely on the race candidates, while *Racageddon* discovers additional race candidates. Overall, Hybrid is successful is producing a worthwhile starting point for those two dynamic techniques. We can see in Figure 8 that for our benchmarks, Hybrid reports 405 race candidates

of which 238 (59%) are real races. Future work may be able to show that some of the remaining 405-238=167 race candidates are real races.

	Number of races	
Name	reported	real
Sor	8	3
TSP	3	1
Hedc	5	5
Elevator	13	7
ArrayList	14	6
TreeSet	13	8
HashSet	11	7
Vector	9	4
RayTracer	2	2
MolDyn	43	5
MonteCarlo	5	3
Derby	21	17
Colt	13	9
ChordTest	2	2
Avrora	9	9
Tomcat	52	20
Batic	12	11
Eclipse	77	46
FOP	21	15
H2	19	17
PMD	14	12
Sunflow	16	6
Xalan	23	23
TOTAL	405	238

Figure 0.8 Hybrid; real is as found by FGCP and Racageddon.

4.4. Related Work

In Section 2 we discussed two techniques for race detection, namely one by O'Callahan and Choi [12] and one by Said, Wang, Yang, and Sakallah [58] that we use in *Racageddon*. In Section 4 we discussed five additional techniques, namely Chord [24], FastTrack [61], Goldilocks [62], CalFuzzer [60], and Pacer [63] that we have compared experimentally with *Racageddon*. The goal of this section is to highlight some other notable techniques and tools in the area of race detection and related areas.

Dynamic race detectors. FastTrack, Goldilocks, CalFuzzer, and Pacer were some of the best dynamic race detectors for Java until now. A predecessor of Pacer, namely LiteRace [66] was the seminal paper that showed how to do race detection in a way that samples and analyzes selected portions of a programâs execution. Prior to LiteRace, a paper by Jump, Blackburn, and McKinley [67] presented a sampling technique that they applied in the context of memory management.

Some well-known dynamic race detectors work for other languages than Java, including the seminal Eraser [68], and a tool by Sack et al. [69].

Arnold and M. Vechev and E. Yahav [70] presented the QVM run-time environment that continuously monitors an execution and potentially detects defects, including races.

Hybrid race detectors. The technique by O'Callahan and Choi [12] that we call Hybrid continues to be one of the best and most scalable hybrid techniques for race detection. Other hybrid techniques include one by von Praun and Gross [71], RaceTrack [72], and MultiRace [73]. We leave to future work to do a large-scale study of those three hybrid techniques like we did for Hybrid. In particular, future work should evaluate how well those techniques perform when we want to use their output as race candidates for other tools such as CalFuzzer and *Racageddon*.

Static race detectors. Chord remains one of the best among the scalable static race detectors to date, hence it was our choice for experimental comparison in this paper. Among the other static race detectors, some use static analysis, including Warlock [74], RacerX [15], LockSmith [75], and Relay [76], some use model checking, including an approach by Henzinger, Jhala, and Majumdar [77], and some use type systems, including an approach based on ownership by Boyapati, Lee, and Rinard [11], and approaches that capture common synchronization patterns by Freund [78] and later by Abadi, Flanagan, and Freund [10]. A related approach based on type systems by Sasturkar, Agarwal, Wang, and Stoller [79] enables specification and check of atomicity. Finally, Effinger-Dean, Boehm, Chakrabarti, and Joisha [80] presented a characterization of extended interference-free regions of C programs in which variables cannot be modified by other threads. All the static approaches may produce false positives and thus have a goal that is dual to our objective to find real races.

Other techniques. We implemented a precursor to *Racageddon* as an extension of Java PathFinder [29]. Our Java PathFinder extension is effective at exploring all execution paths yet doesn't scale up to our current benchmarks.

CHAPTER 5

Deadlock Directed Testing of Concurrent Programs,

or How to Detect Rare Deadlocks

We present a new technique to find real deadlocks in concurrent Java programs. For 4.5 million lines of Java, our technique found almost twice as many real deadlocks as four previous techniques combined. Our technique is particularly good at finding rare deadlocks: it found 33 deadlocks that happened after more than one million computation steps, including 28 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 to drive an execution towards a deadlock candidate.

5.1. Our Deadlock Detection Technique

We now present our approach to find deadlocks.

5.1.1. 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 [EslamimehrPalsberg13b] and that we illustrate in Figure 5.1. Each circle in Figure 5.1 is a schedule. The search is an alternating sequence of execute and permute steps:

$$(\text{execute}.\,\text{permute})^{i}.\,\text{execute}$$

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.

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.

The alternation of permute and execute steps is considerably 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 [31] showed how to implement *execute* via a series of concolic executions, as we will summarize below. In Section 5.3 we show how to define a *permute* function that successfully helps to find rare deadlocks.

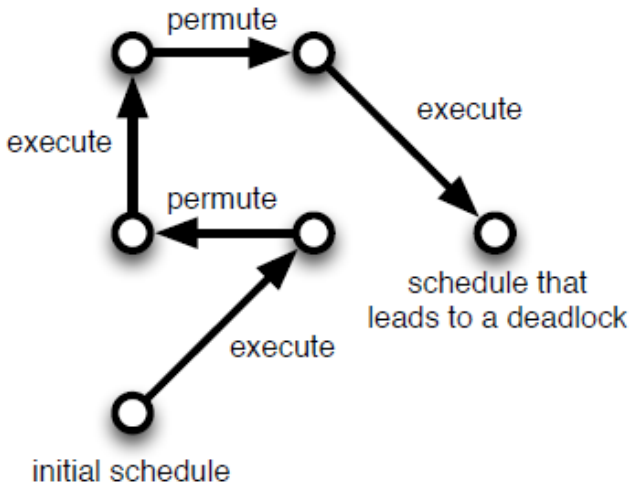

Figure 05.0.1 An illustration of the basic ConLock.

5.1.2. Data Types

We use the following eight data types in *ConLock*.

ConLock 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.

When a program execution executes a particular statement in a particular thread, 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.

Program	=	a Java 6 program
Input	=	input to a Java 6 program
Lock	=	a Java 6 object
Event	=	$\text{threadId} \times \text{statementLable}$
Schedule	=	Event sequence
Link	=	$\text{threadId} \times (\text{statementLable}) \times (\text{statementLable})$
Cycle	=	Link set
Deadlock	=	$\text{Cycle} \times \text{Input} \times \text{Schedule}$

5.1.3. Deadlock Candidates

Our technique relies on access to a set of deadlock candidates. We use Havelund's technique GoodLock [91] 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:

$$\text{GoodLock: Program} \rightarrow \text{Schedule}$$

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 [91]. 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.

5.1.4. The InitialRun Function

Here is the interface to the initialRun function:

$$\text{initialRun: Program} \rightarrow \text{Schedule}$$

A call to initialRun simply executes the program with a random input and records the schedule.

5.1.5. The Execute Function

Here is the interface to the *execute* function:

$$\text{execute: (Program} \times \text{Schedule} \times \text{Cycle)} \rightarrow$$
$$\text{((Input} \times \text{Schedule} \times \text{boolan)} \oplus \{\text{none}\})$$

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:

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

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.Here is a summary of how we implement execute. For a single event, a well-known idea is to execute a series of

```
(Deadlock set)DeadlockTool(Program p){
  (Cycle set) candidates = GoodLock(p)
  (Deadlock set) dlocks = ϕ ;

  for each Cycle c ∈ candidates do {
    boolean found = false
    boolean stalled = false
    int i = 0
    Schedule s = initialRun(p)
    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 05.0.2 ConLock Algorithm.

concolic executions [32] that eventually finds an input that lead to execution of the desired event (if possible). Eslamimehr and Palsberg [31] generalized this idea to work for a sequence of events. The idea is to execute a series of concolic executions that eventually finds an input that leads to execution of all of the events in the sequence in order. The series of concolic executions for the $(n = 1)^{th}$event in the sequence builds on what was achieved for the first *n* events. Once we have matched the entire input schedule, we continue exploration until we have executed a schedule that contains as many of the events embodied in the deadlock candidate as possible. If we cannot match the input schedule at all, then execute returns none

5.1.6. The Permute Function

We will describe the design of the permute function in the following section. Here, we merely list its interface:

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

Notice that permute maps a schedule and a deadlock candidate to a better schedule or else to none if no better schedule was found.

5.1.7. ConLock Pseudo-code

Figure 5.2 shows pseudo-code for *ConLock*, 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 *ConLock* procedure is a program while the output is a set of real deadlocks. The first two lines of *ConLock* declares these two variables: (1) a set of deadlock candidates, called *candidates*, that we initialize by a call to GoodLock, and (2) 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 holds a trace produced by an initial run. For each deadlock candidate we use a while-loop to do an alternation of calls to execute and permute, as illustrated in Figure 5.2. 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, we can take comfort in that some searches have no chance to succeed because the deadlock candidate is not a real deadlock!

5.1.8. Example

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

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.

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:
l_1: $x = 6$
l_2: $synchnorized(A)\{$
l_3: $if(y > 4)$
l_4: $synchnorized(B)\{$
}
}

Thread 2:
l_5: $x = 6$
l_6: $synchnorized(B)\{$
$l_{7:}$: $if(y^2 + 5 < x^2)$
l_8: $synchnorized(B)\{$
}
}

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*:

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

Now we do an initial run of the program. Suppose that the initial random 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:

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

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 the permute function, which we explain next.

5.2. The Design of Permute Function

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

5.2.1. Background: Dynamic Data 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 [109] 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 [58], one can phrase the problem to find an executable permutation of a trace as a constraint-solving problem, and one 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 [31], combined Said et al.'s permute function with concolic execution and thereby obtained an efficient and useful dynamic race detector. What we need now is a permute function that works well for deadlock detection.

5.2.2. Static Characterization of Potential Deadlocks

Deshmukh, Emerson, and Sankaranarayanan [95] 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

$$synchnorized(A)\{synchnorized(B)\{\dots\}\}$$

the lock-order 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 can embed their characterization of deadlocks into a permute function.

5.2.3. A Memory-les Permute Function for Deadlock

We want a permute function that works well for deadlock detection. We now describe a baseline version of such a function that we call the *memory-less* permute function. 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. In the following subsection, we present an enhanced permute function that leads us to find an additional 25 deadlocks.

Our memory-less permute function combines Deshmukh et al.'s static analysis of deadlocks [95] with aspects of Said et al.'s permute function [58] and a constraint that encodes a deadlock pattern for 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. The bulk of Said et al's paper [58] describes how to define (1.1), (1.2), and (1.3). We won't list the constraints here and instead we refer the reader to Said et al's paper [58] for details.

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.

Finally, let us explain how we represent a deadlock candidate. First we need to introduce the notation used in the constraints that we have otherwise omitted. Suppose we have a trace $t =< e_1, \dots, e_n >$. The constraints use *n* position variables $o_1, \dots, o_n$. The idea is that the value of o_1 is the position of e_1 in the permuted trace. A solution to the constraints is an injective function

$$S = \{o_1, \dots, o_n\} \rightarrow \{1, \dots, n\}$$

Now let us explain how we represent the following deadlock candidate from Section 5.2.7:

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

Let us define $e_2 = (Thread\ 1, l_2), e_4 = (Thread\ 1, l_4), e_6 = (Thread\ 2, l_6), e_8 = (Thread\ 2, l_8)$. We present *c* with the constraints

$$(o_2 < o_4\ ) \wedge (o_6 < o_8)$$

Where < is the happens-before relation. This representation generalizes in straightforward manner to other deadlock candidates.

The grand total is a constraint that consists of the constraints (1.1) and (1.2) from Said et al., all Deshmukh et al.'s 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.

5.2.4. 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 by chance may 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 [37] 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.

5.2.5. Example

Let us return to the example from Section 2.7. Here we will focus entirely on 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 listed above. Here are the constraints used by the permute function. First we list the constraints from Said et al. that we labeled (1.1), namely the constraints that preserve the happens-before relation for each thread:

$$e_1 < e_2 \wedge e_2 < e_3 \wedge e_3 < e_4 \wedge e_5 < e_6 \wedge e_6 < e_7$$

Next we list the constraints from Said et al. that we labeled (1.2), namely the constraints that ensure write-read consistency:

$$e_5 < e_7$$

Next we list Deshmukh et al.'s constraints for lock order:

$$e_2 < e_4 \wedge e_6 < e_8$$

Let us assume that the program has no aliasing; then we have no alias constraints. Finally, we have a constraint that encodes the deadlock candidate:

$$e_2 < e_8 \wedge e_6 < e_4$$

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.

5.3. Experimental Results

We implemented GoodLock as an extension of Java PathFinder [29]. We use the Lime concolic execution engine; Lime is open source, http://www.tcs.hut.fi/Software/lime. In our implementation, events are at the Java bytecode level; we use Soot [102] to instrument bytecodes. We ran all our experiments on a Linux CentOs machine with two 2.4 GHz Xeon quad core processors and 32 GB RAM.

5.3.1. Benchmarks

Figure 5.3 lists our 22 benchmarks which we have collected from six sources:

- From ETH Zurich [90]: Sor, TSP, Hedc, Elevator.
- From java.util, Oracle's JDK 1.4.2: ArrayList, TreeSet, HashSet, Vector.
- From Java Grande, [JDK1.4.2]: RayTracer, MolDyn, MonteCarlo.
- From the Apache Software Foundation [Derby]: Derby.
- From European Organization for Nuclear Research (CERN) [Colt]: Colt.
- From DaCapo [106]: 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 5.3 also lists the high watermark of how many threads each benchmark runs, and the input size in bytes for each benchmark.

5.3.2. Deadlock Detectors

We compare ConLock with one static deadlock detector, namely Chord [23], one hybrid deadlock detector that we call GoodLock [29], and four dynamic deadlock detectors, namely DeadlockFuzzer [17], IBM Contest [25], Jcarder [27], and Java HotSpot [28]. 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 the execution of a multi-threaded program, computes a lock dependency relation, and uses the transitive closure of this relation to suggest potential deadlocks.

DeadlockFuzzer, IBM Contest, Jcarder, Java HotSpot, and ConLock 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 [79].

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.

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 Performance Scientific and Technical Computing
ChordTest	62	11	74	Mini-benchmark; comes with the Chord race detector
Avrora	140K	6	88	AVR microcontroller simulator
Tomcat	535K	16	366	Tomcat Apache web application server
Batic	354K	5	206	Produces a number of Scalable Vector Graphics (SVG) images based on Apache Batic
Eclipse	1.2M	16	34	Non-GUI Eclipse IDE
FOP	21K	8	658	XSL-FO to PDF converter
H2	20K	16	116	Executes a JDBCbench-like in-memory benchmark
PMD	81K	4	24	Java Static Analyzer
Sunflow	108K	16	616	Tool for rendering image with raytracer
Xalan	355K	9	404	XML to HTML transformer

TOTAL	4587K		58	

Figure 05.0.3 Benchmarks.

5.3.3. 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 [64]. The core of the problem is that reflection is at odds with static analysis and bytecode instrumentation: reflection may 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.

5.3.4. Measurements

Figure 5.4 shows, for each benchmark, the number of calls to execute across all deadlock candidates, and the number of concolic runs across all calls to \execute. Intuitively, the first number is the number of iterations of the while-loop across all deadlock candidates; each iteration calls execute once. Each of those calls to execute tends to do a large number of concolic runs, and the second number is the grand total count of all those concolic runs.

Name	# calls to execute	# concolic runs
Sor	11	591
TSP	16	387
Hedc	29	10,550
Elevator	18	119
ArrayList	19	613
TreeSet	20	205
HashSet	17	422
Vector	21	175
RayTracer	10	86
MolDyn	9	57
MonteCarlo	30	101
Derby	14	2,349,385
Colt	16	73,129
Avrora	39	6,007,128
Tomcat	661	5,923,744
Batic	41	428,881
Eclipse	726	3,901,827
FOP	15	64,050
H2	24	17,580
PMD	12	99,105
Sunflow	19	41,051
Xalan	414	87,933

Figure 05.0.4 Dynamic counts of ConLock's search process.

Figure 5.5 shows the numbers of deadlocks found in 22 benchmarks by 7 techniques.

Figure 5.6 shows the time each of the runs took in minutes and seconds, and it shows the geometrical mean for each technique.

Figure 5.7 shows, for each benchmark, the lengths of the 146 schedules that lead to deadlocks found by ConLock. The 86 schedules highlighted with boldface font lead to deadlocks found only by ConLock.

5.3.5. Evaluation

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

Number and length of schedules. We can see in 5.4 that the number of calls to execute is rather modest: for every benchmark, it is at most twice the number of deadlock candidates. We can also see in Figure 5.4 that each call to execute does many concolic runs to match a given schedule. We can see in Figure 5.7 that those schedules can be long; the longest schedules that found deadlocks have more

than 10,000,000 events. This also shows that the \permute method scales to long schedules.

ConLock versus other Dynamic Techniques. We can see in Figure 5.5that ConLock finds the most deadlocks (146) of all the dynamic techniques. Among those 146 deadlocks, 86 deadlocks were found only by ConLock and are entirely novel to this study, while 60 were also found by DIJJ. Dually, 15 deadlocks were found only by DIJJ. In summary, we have that the combination of DIJJ and ConLock found 161 deadlocks in the 22 benchmarks.

	Static	Hybrid	Dynamic							
benchmarks	Chord	GoodLock	DeadlockFuzzer	IBM ConTest	Jcarder	Java HotSpot	DIJJ	ConLock		
								total	new	FGCP
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 05.0.5 The numbers of deadlocks found in 22 benchmarks by 7 techniques.

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

Let us consider the 15 deadlocks that DIJJ found but ConLock 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 ConLock 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 5.1, hence they may deadlock.

	Static	Hybrid	Dynamic				
benchmarks	Chord	GoodLock	DeadlockFuzzer	IBM ConTest	Jcarder	Java HotSpot	ConLock
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 05.0.6 Timings in minutes and seconds.

We conclude that ConLock 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. Now we analyze the individual contributions of the four techniques. Our first observation is, 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. Our second observation is 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. We note that ConLock also finds that particular deadlock in Elevator.

Chord. Chord is possibly the best current static deadlock detector, yet our experiments strongly 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 continues to be an open problem.

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

Name	length
Sor	**5705**
TSP	**6688**
Hedc	**1009**, 11488, **55133, 73956, 104440, 116573, 172832, 178127, 189601, 197893, 207813, 228867, 244167, 249130, 251800, 269624, 269911, 273123, 275003, 277145**
Elevator	3099, **10029, 10753, 12369, 13680**
ArrayList	15873, **19012, 19632**, 47100, 58881, **80431, 80512, 110532, 111407**
TreeSet	303, 22889, 52011, 59217, 77138
HashSet	921, 11630, 23705, 53186, 93122
Vector	**2007, 4401, 4788, 6020**
RayTracer	3981**, 8212**
MolDyn	5194
MonteCarlo	4392, **11972**
Derby	56430**, 786620, 23725394, 34440100**
Colt	**456313, 720898, 1362838**
Avrora	1802**, 23520**, 65219, 65820, 242749, **550892, 600236**
Tomcat	308, **3494**, 82442, 83710, 197126, **482100,** 871390**, 891376**, 2973632, 3976200, **6061234**, 6535192**, 7105988**, **7359792, 7367253, 8001527, 8091572, 8119634**
Batic	1997, 5481, 10781, 72918, 114666, 203675, 259178, **908327, 1034685, 1220565**
Eclipse	736, 1267, 7723, 31884, 72535, 209734, 475110, **920255, 946701,** 989271, 995021, 1537020, 1792033, **3000287, 6197522, 9801562, 11732081, 11885360, 13870290, 13992176, 15753208, 16001526, 18275300**
FOP	23991, 56028**, 119886, 130898**
H2	19763, **109587, 296001**
PMD	678, 2923, **219561, 287023**
Sunflow	15873, 67325**, 550192, 888237, 991720, 1089212**
Xalan	8805, 52249, 116023, **294027, 1080194**, 1973260, **2774053, 3207368, 3304152**

Figure 05.7 The lengths of the 146 schedules that lead to deadlocks found by ConLock. Bold font indicates new.

Rare deadlocks. Burckhardt, Kothari, Musuvathi, and Nagarakatte [65] characterized the depth of a bug as the minimum number of scheduling constraints required to find that bug. In the spirit of that idea, we will say that a deadlock is *rare* if it occurs after more than a million steps of computation. Let us consider whether ConLock finds any rare deadlocks. Figure 5.7 lists the lengths of the 146 schedules that lead to deadlocks found by ConLock. We can groups those lengths as follows:

lengths	#
$10^2 - 10^3$	5
$10^3 - 10^4$	20
$10^4 - 10^5$	39
$10^5 - 10^6$	49
$10^6 - 10^7$	24
$10^7 - 10^8$	9

The table shows that many of those schedules are long, hence the deadlocks are rare. Specifically, 33 deadlocks were found with schedules that have between 1 million and 34 million events, which suggests that they are rare and hard-to-find deadlocks.

We note that 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.

In Figure 5.7 the numbers in bold font are for schedules that lead to deadlocks found only by ConLock. Among the rare 33 deadlocks, 28 were found only by ConLock.

We conclude that ConLock does a much better job than previous work to find rare deadlocks.

5.4. Limitations

Our approach has four main limitations.

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

Second, our approach relies on GoodLock to produce deadlock candidates. In case GoodLock misses a deadlock, so will ConLock.

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 constraint solving in execute, we are at the mercy of expressions in the program text and the power of our chosen constraint solver.

Fourth, our approach has no support for native code.

5.5. Related Work

In Section 4, we discussed six techniques for deadlock detection, namely Chord [24], GoodLock [29], DeadlockFuzzer [17], IBM Contest [25], Jcarder [27], and Java HotSpot [28] and we did a large-scale experimental comparison of all six and ConLock. 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 and M. Vechev and E. Yahav [70] presented the QVM run-time environment that continuously monitors an execution and potentially detects defects, including deadlocks. Huang, Zhang, and Dolby [93] 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 [108]. All three approaches monitor executions but do nothing to drive an execution towards a deadlock.

Model checking. Demartini et al. [85] presented a translation from Java source code to Promela that enables deadlock detection via the SPIN model checker [92]. The translator predates Java 6 and would require significant extension to handle our benchmarks.

Chaki et al. [103] and Godefroid [89] presented model checkers for C that can find deadlocks. We leave to future work to try those approaches for Java.

Static deadlock detectors. Static deadlock detectors [10] 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 study.

Conclusion

To find the maximum stack size in the context of event-driven programs, our results show that the state-of-the-art static analysis produces excellent estimates of maximum stack size. Our testing approach DTall can almost match the results of the static analysis. Additionally, our approach VICE comes close and is two orders of magnitude faster than DTall. The keys to produce challenging event sequences are to use directed testing to get event values and to use our SA-Tree technique to get event names. The SA-Tree technique is an example of how static analysis can help testing be more efficient.

Our technique is useful for other languages than Virgil. The availability of a source-level interpreter greatly facilitates the collection of constraints.

VICE is a new approach to practical stress testing of event-driven software, even in situations when no nontrivial sound static analysis exists. VICE quickly generates a small number of challenging event sequences that drive the execution into "dark corners" of the software. Such event sequences may reveal faults or help confirm that the software works correctly even for corner cases. We leave to future work to investigate the bug-finding capabilities of DTall and VICE.

In the context of concurrent *Racageddon* and *ConLock* implement a new technique that we call *directed scheduling*. We have shown how to detect data races and deadlocks by a combination of concolic execution and a novel approach to schedule permutation. Our experiments show that directed scheduling is efficient and useful.

For a large benchmark suite, our tool *Racageddon* found 72 real races that were missed by earlier techniques. Among those 72 races, more than a third namely 31 races were found with schedules that have between 1 million and 108 million events, which suggests that they are rare and hard-to-find races. Our experiments also show that a combination of Goldilocks, Calfuzzer, Pacer, and *Racageddon* finds a total of real 376 races in our benchmarks. As far as we know, this is the most comprehensive list of real races for those benchmarks that is reported in the literature. Our experiments validate Hybrid [12] as an excellent choice for producing race candidates. Across our benchmark suite, we found that Hybrid produces at most 41% false positives. .For a large benchmark suite, our tool ConLock found 86 deadlocks that were missed by earlier techniques. Among those 86 deadlocks, about a third namely 28 deadlocks were found with schedules that have more than 1 million events, which suggests that they are rare and hard-to-find deadlocks. Our technique can find rare deadlocks because the combination of concolic execution and schedule permutation helps drive an execution towards a deadlock candidate. Our experiments show that a combination of DeadlockFuzzer, IBM ConTest, and ConLock finds a total of real 161 deadlocks in our benchmarks.

Our design of *Racageddon* and *ConLock* and our experiments have shown that a combination of techniques is currently the best path to successful race detection. As far as we know, this is the most comprehensive list of real deadlocks for those benchmarks that is reported in the literature

References

[1] J. Gilson., "A New Approach to Engineering Tolerances.," The Machinery Publishing C. Ltd., 1951.

[2] K. Brunnstein, "About the “Altona railway software glitch"," The Risks Digest, 1995.

[3] D. M. R. M. T. Z. T. A. a. J. P. Krishnendu Chatterjee, "Stack size analysis of interrupt driven software," in *Special issue dedicated to Paris Kanellakis. Preliminary version in Proceedings of SAS’03, International Static Analysis Symposium, Springer-Verlag (LNCS 2694)*, San Diego, June 2003.

[4] J. R. a. A. Reid, "HOIST: a system for automatically deriving static analyzers for embedded systems," *ACM SIGARCH Computer Architecture News,* 2004.

[5] a. P. M. R. Alur, "Visibility pushdown languages," in *Proceedings of the thirty-sixth Annual ACM Symposium on Theory of Computing*, 2004.

[6] N. D. a. J. P. Dennis Brylow, "Static checking of interrupt-driven software," Proceedings of ICSE, 23rd International Conference on Software Engineering, May 2001.

[7] J. Regehr, "Random testing of interrupt-driven software," in *ACM International Conference On Embedded Software*, 2005.

[8] M. D. H. B. P. M. a. R. H. B. Sarita V. Adve, "Detecting data races on weak memory systems," in *ISCA*, 1991.

[9] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Comput,* 1979.

[10] C. F. a. S. N. F. M. Abadi, "Types for safe locking:Static race detection for Java," *ACM Transactions on Programming Languages and Systems,* 2006.

[11] S. K. a. D. M. C. Boyapati, "Korat: Automated testing based on Java predicates," in *Proc. Of International Symposium on*, 2002.

[12] R. O. a. J.-D. Choi, "Hybrid dynamic data race detection," in *PPOPP*, 2003.

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

[14] R. L. a. M. R. C. Boyapati, "Ownership types for safe programming:Preventing data races and deadlocks," in *OOPSLA, ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2002.

[15] D. E. a. K.Ashcraft, "RacerX: Effective, static detection of race conditions and deadlocks," in *SOSP, Nineteenth ACM symposium on Operating Systems Principles*, 2003.

[16] K. R. M. L. M. L. G. N. J. B. S. a. R. S. Cormac Flanagan, "Extended static checking for Java," in *In Proceedings of PLDI'02, ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2002.

[17] C. S. P. K. S. a. M. N. P. Joshi, "A randomized dynamic program analysis technique for detecting real deadlocks," in *Proceedings of PLDI'09, ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[18] C. H. J. D. M. V. F. T. a. J. V. Daniel Marino, "Detecting deadlock in programs with datacentric synchronization," in *ICSE'13, International Conference on Software Engineering*, 2013.

[19] A. G. a. W. S. Elissa Newman, "Annotation-based diagrams for shared-data concurrency," in *Workshop on Concurrency Issues in UML*, 2001.

[20] H. B. S. Z. M. a. C. D. Cesar Sanchez, "Efficient distributed deadlock avoidance with liveness guarantees," in *Proceedings of EMSOFT'06, International Conference on Embedded Software*, 2006.

[21] Y. Z. a. Y. P. Lin Tan, "acomment: Mining annotations from comments and code to detect interrupt related concurrency bugs," in *In ICSE'11, International Conference on Software Engineering*, 2011.

[22] T. K. M. K. S. L. S. M. Yin Wang, "Gadara: Dynamic deadlock avoidance for multithreaded programs," in *n Proceedings of OSDI'08, 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008.

[23] C.-S. P. a. D. G. M. Naik, "Effective static deadlock detection," in *In ICSE'09, Eighteenth International Conference on Software Engineering*, 2009.

[24] A. A. a. J. W. Mayur Naik, "Effective static race detection for java," in *Proceedings of PLDI'06, ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006.

[25] E. F. E. G. Y. N. G. R. S. U. Orit Edelstein, "Framework for testing multi-threaded Java programs," *Concurrency and Computation: Practice and Experience,* 2003.

[26] Y. N.-B. a. S. U. E. Farchi, "cross-run lock discipline checker for java," in *PADTAD*, 2005.

[27] "ENEA. Jcarder. http://www.jcarder.org.".

[28] Orale, "Java hotspot vm options," http://www.oracle.com/echnetwork/java/javase/tech/vmoptions-jsp-140102.html.

[29] K. H. a. T. Pressburger, "Model checking Java programs using Java pathfinder," *Software Tools for Technology Transfer,* 2000.

[30] J. P. Mahdi Eslamimehr, "Testing versus static analysis of maximum stack size," in *COMPSAC*, 2013.

[31] Mahdi Eslamimehr and Jens Palsberg, "Race directed scheduling of concurrent programs," in *Proceedings of PPOPP'14, ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel*

Programming, 2014.

[32] R. D. a. T. G.Venolia, "Software Development at Microsoft Observedd," Microsoft Research TR, 2005.

[33] V. J. D. D. a. D. M. Brett Daniel, "ReAssert: Suggesting Repairs for Broken Unit Tests," in *Conference on Automated Software Engineering, 2009. ASE '09. 24th IEEE/ACM International* , 2009.

[34] G. J. Myers., The Art of Software Testing, Wiley, 1979.

[35] P. McMinn, "Search-based software test data generation: a survey," *Software Testing, Verification and Reliability,* 2004.

[36] E. L. a. T. Austin, "High Coverage Detection of Input-Related Security Faults," in *12th Annual USENIX Security Symposium (SEC-2003)*, 2003.

[37] N. K. Koushik Sen Patrice Godefroid, "Dart: directed automated random testing," in *Proceedings of PLDI'05, ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[38] G. A. a. K. Sen, "Cute and jcute: Concolic unit testing and explicit path model-checking tools," in *Proc. 18th International Conference on Computer Aided Verification,*, 2006.

[39] K. J. Burnim, "Heuristics for scalable dynamic test generation," UC Berkeley tech report, 2008.

[40] R. M. a. R.-G. Xu, "Directed test generation using symbolic grammars," in *Proceedings of the twenty-second IEEE/ACM International Conference on Automated Software Engineering*, 2007.

[41] T. L. O. S. K. K. H. a. I. N. Kari K□ahk□onen, "LCT: An Open Source Concolic Testing Tool for Java Programs," in *BYTECODE*, 2011.

[42] a. A. B. R. Brummayer, in *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2009)*, 2009.

[43] W. V. Corina S. Pasareanu, "A survey of new trends in symbolic execution for software testing and analysis," *STTT 11.*

[44] S. B. J. S. W. Tanmoy Sarkar, "ConSMutate: SQL Mutants for Guiding Concolic Testing of Database Applications," in *14th International Conference on Formal Engineering Methods, ICFEM* , 2012.

[45] Y. K. Y. J. Moonzoo Kim, "Industrial Application of Concolic Testing on Embedded Software: Case Studies," in *IEEE Fifth International Conference on Software Testing, Verification and Validation*, 2012.

[46] P. J. G. k. J. a. M. D. E. A. Kiezun, "Automatic creating of SQL injection and Cross-Site scripting attacks," in *International Conference on Software Engineering*, 2009.

[47] O. L. M. h. A. C. W. S. K. a. K. W. R. Sasnauskas, "KleeNet: Discovering insidious interaction bugs in wireless sensor networks before deployment," in *ACM/IEEE International Conference on*

Information Processing in Sensor Networks, 2010.

[48] V. K. a. G. C. V. Chipounov, "S2E: A platform for in-vivo multi-path analysis of software systems," in *ASPLOS*, 2011.

[49] "Atmel. Spi deriver manufacturer datasheet. http://www.atmel.com/dyn/resources/prod documents/doc2582.pdf," September 2010.

[50] K. Sen, "Concolic testing," in *Proceedings of the twenty-second IEEE/ACM International Conference on Automated Software Engineering*, 2007.

[51] A. J. C. T. A. H. R. J. a. R. D. Beyer, "Generating test from counterexamples," in *Proceedings of ICSE*, 2004.

[52] a. N. N. RE Fikes, "Strips: A new approach to the application of theorem proving to problem solving," *Artificial Intelligence,* p. 2(3–4):189–208, Winter 1971.

[53] C. S. P. a. W. V. S. Khurshid, "Generalized symbolicexecution for model checking and testing," in *Proc. of TACAS*, 2003.

[54] B. L. Titzer, "Virgil: Objects on the head of a pin," in *Proceedings of OOPSLA'06, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, 2006.

[55] D. K. L. a. J. P. Ben L. Titzer, "Avrora: Scalable sensor etwork simulation with precise timing," in *Proceedings of IPSN'05,Fourth International Conference on Information Processing in Sensor Networks*, April 2005.

[56] "CHOCO. http://www.emn.fr/z-info/choco-solver/choco-documentation.," September 20110.

[57] F. Laburthe, "Choco: implementing a CP kernel," in *Proceedingsof CP00 Post Conference Workshop on Techniques for Implementing Constraint programming Systems (TRICS)*, September 2000.

[58] C. W. Z. Y. a. K. A. S. Mahmoud Said, "Generating data race witnesses by an smt-based analysis," NASA Formal Methods, 2011.

[59] P. T. V. G. a. D. E. Cristian Cadar, "EXE:A system for automatically generating inputs of death using symbolic execution," in *Proceedings of 13th ACM Conference on Computer and Communications Security*, 2006.

[60] K. Sen, "Effective random testing of concurrent programs," in *IEEE/ACM nternational Conference on Automated Software Engineering*, 2007.

[61] C. F. a. S. N.Freund, "Fasttrack: Efficient and precise dynamic race detection," in *Proceedings of PLDI'09, ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[62] S. Q. a. S. T. Tayfun Elmas, "Goldilocks: Efficiently computing the happens-before relation using locksets," in *FATES/RV*, 2006.

[63] K. E. C. a. K. S. M. Michael D. Bond, "Pacer: Proportional detection of data races," in *Proceedings of PLDI'10, ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2010.

[64] A. S. J. S. H. O. a. M. M. Eric Bodden, "Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders," in *ICSE, 33rd International Conference on Software Engineering*, 2011.

[65] P. K. M. M. a. S. N. Sebastian Burckhardt, "A randomized scheduler with probabilistic guarantees of finding concurrency bugs," in *ASPLOS, International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010.

[66] M. M. a. S. N. Daniel Marino, "Literace: Effective sampling for lightweight data-race detection," in *Proceedings of PLDI'09, ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[67] S. M. B. a. K. S. M. M. Jump, "Dynamic object sampling for pretenuring," in *ICMM, ACM International Symposium on Memory Management*, 2004.

[68] M. B. G. N. P. S. a. T. A. Stefan Savage, "Eraser: A dynamic data race detector for multithreaded programs," *Transactions on Computer Systems,* 1997.

[69] B. E. B. Z. M. P. P. a. J. T. P. Sack, "Accurate and efficient filtering for the Intel thread checker race detector," in *SAID 1st workshop on Architectural and System Support for Improving Software Dependability*, 2006.

[70] M. V. a. E. Y. M. Arnold, "Qvm: An efficient runtime for detecting defects in deployed systems," in *OOPSLA, ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2008.

[71] C. v. Praun, "Detecting Synchronization Defects in Multi-Threaded Object-Oriented Programs," 2004.

[72] T. R. a. W. C. Y. Yu, "RaceTrack: Efficient detection of data race conditions via adaptive tracking," in *SOSP, ACM Symposium on Operating Systems Principles*, 2005.

[73] E. P. a. A. Schuster, "MultiRace: Efficient on-the-fly data race detection in multithreaded C++ programs," *Concurrency and Computation: Practice and Experience,* 2007.

[74] N. Sterling, "WARLOCK – a static data race analysis tool," in *USENIX Winter Technical Conference*, 1993.

[75] J. S. F. a. M. H. Polyvios Pratikakis, "LockSmith: Context sensitive correlation analysis for race detection," in *PLDI, ACM Conference on Programming Language Design and Implementation*, 2006.

[76] R. J. a. S. L. J. W. Voung, "RELAY: Static race detection on millions of lines of code," in *In European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of*

Software Engineering, 2007.

[77] R. J. a. R. M. Thomas A. Henzinger, "Race checking by context inference," in *Proceedings of PLDI'04, ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2004.

[78] S. N. Freund, "Type-based race detection for Java," in *PLDI, ACM SIGPLAN 2000 Conference on Programming language design and implementation*, 2000.

[79] R. A. L. W. a. S. D. S. A. Sasturkar, "Automated type-based analysis of data races and atomicity," in *PPoPP, Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel*, 2005.

[80] H.-J. B. D. C. a. P. J. Laura Effinger-Dean, "Extended sequential reasoning for data-race-free programs," in *In ACM SIGPLANWorkshop on Memory Systems Performance and Correctness*, 2011.

[81] "Atmel. Adc deriver manufacturer datasheet. http://www.atmel.com/dyn/resources/prod documents/doc8078.pdf," September, 2010.

[82] "Atmel. Usart serial deriver manufacturer datasheet. http://www.atmel.com/dyn/resources/prod_documents/doc32006.pdf," September 2010.

[83] E. O. f. N. R. (CERN).Colt., "Colt," http://acs.lbl.gov/software/colt/.

[84] C. A. a. A. Biere, "Applying static analysis to large-scale, multithreaded java programs," in *Proceedings of ASWEC'01, 13th Australian Proceedings of ASWEC'01, 13th Australian Software Engineering Conference*, 2001.

[85] R. I. a. R. S. C. Demartini, "A deadlock detection tool for concurrent Java programs," *Software – Practice & Experience,* 1999.

[86] W. T. a. M. Ernst, "Static deadlock detection for java libraries," in *Proceedings of ECOOP'05, European Conference on Object-Oriented Programming*, 2005.

[87] A. S. Foundation, "Derby," http://db.apache.org/derby.

[88] C. F. a. P. Godefroid, "Dynamic partial-order reduction for model checking software," in *Proceedings of POPL'05,SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, 2005.

[89] P. Godefroid, "Model checking for programming languages using Versioft," In Proceedings of POPL'97, 24th Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages, 1197.

[90] C. v. P. a. T. R. Gross, "Object race detection," in *In OOPSLA, ACM Conference on Object-Oriented Programming, Systems, Languagesand Applications*, 2001.

[91] K. Havelund, "Using runtime analysis to guide model checking of Java programs," Proceedings of SPIN'00, Model Checking Software,International SPIN Workshop, 2000.

[92] G. Holzmann., "The Spin model checker," 1997.

[93] C. Z. a. J. D. Jeff Huang, "CLAP: Recording local executions to reproduce concurrency failures," in *Proceedings of PLDI'13, ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2013.

[94] A. R. a. K. W. John Regehr, "Eliminating stack overoverflow by abstract interpretation," in *Proceedings of EMSOFT'03, Third International Conference on Embedded Software*, 2003.

[95] E. A. E. a. S. S. Jyotirmoy Deshmukh, "Symbolic deadlock analysis in concurrent libraries and their clients," in *Proceedings of ASE'09, IEEE International Conference on Automated Software Engineering*, 2009.

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

[97] Oracle., " JDK, 1.4.2," http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/publications.html.

[98] M. N. K. S. a. D. G. P. Joshi, "An effective dynamic analysis for detecting generalized deadlocks," in *ACM FSE'10, Symposium on the Foundations of Software Engineering*, 2010.

[99] B. L. T. a. J. Palsberg, "Vertical object layout and compression," in *Proceedings of CASES'07, International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, Austria, September 2007.

[100] D. H. a. W. Pugh, "Finding concurrency bugs in Java," in *In Proceedings of the PODC Workshop on Concurrency and Synchronization in Java Programs*, 2004.

[101] L. a. S. D. S. R. Agarwal, "Detecting potential deadlocks with static analysis and runtime monitoring," in *PADTAD*, 2005.

[102] E. G. L. H. P. L. P. P. a. V. S. Raja Vall´e-Rai, "Optimizing Java bytecode using the soot framework: Is it feasible?," in *Proceedings of CC'00, International Conference on Compiler Construction. Springer-Verlag*, 2000.

[103] E. C. J. O. N. S. a. N. S. S. Chaki, "Concurrent software verification with states, events, and deadlocks," *Formal Aspects of Computing,* 2005.

[104] J. B. a. K. Sen, "Heuristics for scalable dynamic test generation," in *Proc. 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008.

[105] J. B. a. K. Sen., "Heuristics for scalable dynamic test generation," in *Proc. 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008.

[106] R. G. C. H. A. M. K. K. S. M. R. B. A. D. D. F. D. F. S. Z. G. M. H. A. H. M. J. H. L. I. J. E. B. M. Stephen M. Blackburn, "The DaCapo benchmarks: Java benchmarking development and analysis," in *In*

OOPSLA'06, 21st annual ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages and Applications, 2006.

[107] S. D. Stoller, "Testing concurrent Java programs using randomized scheduling," in *In Proceedings of RV'02, Workshop on Runtime Verification*, 2002.

[108] C. S. E. A. R. L. a. D. J. S. T. Li, "A dynamic deadlock detection mechanism using speculative execution," in *Proceedings of the USENIX Technical Conference*, 2005.

[109] F. C. a. G. R. Traian Florin Serbanuta, "Maximal causal models for multithreaded systems.," Technical report, University of Illinois at Urbana-Champaign. Available from ideals.illinois.edu..