University of Nebraska - Lincoln
DigitalCommons@University of Nebraska - Lincoln

Computer Science and Engineering: Theses, Dissertations, and Student Research | Computer Science and Engineering, Department of

Winter 12-1-2012

Dynamic Data Race Detection and Healing

Du Li
University of Nebraska-Lincoln, dawn2004@gmail.com

Follow this and additional works at: http://digitalcommons.unl.edu/computerscidiss

Part of the Programming Languages and Compilers Commons, and the Software Engineering Commons

Li, Du, "Dynamic Data Race Detection and Healing" (2012). *Computer Science and Engineering: Theses, Dissertations, and Student Research*. 52.
http://digitalcommons.unl.edu/computerscidiss/52

This Article is brought to you for free and open access by the Computer Science and Engineering, Department of at DigitalCommons@University of Nebraska - Lincoln. It has been accepted for inclusion in Computer Science and Engineering: Theses, Dissertations, and Student Research by an authorized administrator of DigitalCommons@University of Nebraska - Lincoln.

DYNAMIC DATA RACE DETECTION AND HEALING

by

Du Li

A DISSERTATION

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Doctor of Philosophy

Major: Computer Science

Under the Supervision of Professors Witawas Srisa-an and Matthew B. Dwyer

Lincoln, Nebraska

December, 2012

DYNAMIC DATA RACE DETECTION AND HEALING

Du Li, Ph.D.

University of Nebraska, 2012

Advisers: Witawas Srisa-an and Matthew B. Dwyer

Perpetual availability is an important operational goal in today's computer systems. However, achieving this goal is challenging because modern software systems contain faults that can cause them to fail. For example, multi-threading is widely used in modern software to fully utilize the computing capability of multicore processors. However, employing multi-threading can lead to concurrency faults such as deadlock and data race that are notoriously difficult to to isolate, detect, and repair.Data races, which involves two concurrent accesses to the same data where at least one is a write, are the most common concurrency faults.

As our first step, we investigate the main sources of race detection overhead and find that a large effort is spent on repeatedly monitoring operations that cannot cause data races or have already been identified as causes of races. Based on these observations, we propose two orthogonal optimizations for race detection: Stationary Object Suppression (SOS) and Loop Iteration Sampling (LIS).

SOS employs a dynamic program analysis technique to filter out Stationary Objects; which are read-only objects that can be shared by multiple threads. As such, they can never participate in data races. By eliminating monitoring operations on Stationary Objects, SOS can detect up to six times more races within an overhead budget than Pacer, a state-of-the-art sampling based race detector.

Although SOS can greatly reduce the number of objects to monitor for race detection, it repeatedly monitors identified sources of races. A further investigation shows that loops in a program substantially contribute to occurrence of such repetitive data races. We propose

a sampling based race detector, LIS, which adjusts sampling rate for data access operations within loops to be inversely proportional to number of iterations.

To achieve perpetual availability, the next step is to address these software faults as they are detected during deployment. We propose a race healing system that can automatically generate and apply repairs during program execution. The system applies a fix immediately after a race is detected to prevent the race from occurring again.

COPYRIGHT

© 2012, Du Li

ACKNOWLEDGMENTS

I am deeply grateful to my advisers, Witawas Srisa-an and Matthew Dwyer, for supporting and mentoring me.

Dr. Srisa-an has been advising me all through my PhD study. His endless passion and patience strongly encourages me to overcome obstacles over the past five years. His smile always greatly comforts me when days were hard. He has sponsored me to attend major conferences since the very beginning of my PhD study, exposing me to the community and motivating me to be part of the community. Some ideas in this dissertation were actually inspired by the interaction with other scholars at research venues.

Dr. Dwyer has educated and influenced me in many ways. He has contributed to every part of this dissertation. As a well-respected world class scholar, he deeply impressed me with his dedication and very insightful research views. Dr. Dwyer is incredibly helpful so that I feel I learn quite a bit every week I meet with him. He also provides invaluable advice for my career plan and helps me to obtain very important opportunities. It is impossible for me to get where I am without his support.

I am fortunate to have Myra Cohen and Anuj Sharma on my PhD advisory committee. They have been very supportive for my dissertation proposal, writing and oral defense. They also provided valuable comments on my proposal and dissertation. They did all they can to allow me to achieve my goal. Otherwise, it would be more challenging for me to finish my PhD study.

The atmosphere in ESQuaReD is very friendly. Everyone is connected and there are a lot collaborations going on all the time. For students, we can easily receive mentoring from more than one faculty members, leading to exciting research projects. My dissertation work is an example of such a collaboration.

I would thank my parents who are in China now. They have given me endless love since

I was born and tried to help in any way they could. Although they live thousands of miles away from me, I feel their support every day and night. I appreciate the love from my wife, Jia Pu. She takes good care of me while she is a very busy PhD student as well.

Running is my major hobby during the past five years. The Lincoln trail system provides excellent environment to develop this hobby. I would thank the trail system for more than 1000 miles running in my time at Lincoln, which is not only an exercise, but an effective way to refresh and recharge myself.

GRANT INFORMATION

This dissertation is supported in part by the National Science Foundation through awards CNS-0720757 and CCF-0912566, the National Aeronautics and Space Administration under grant number NNX08AV20A, and by the Air Force Office of Scientific Research through awards FA9550-10-1-0406 and FA9550-09-1-0687.

Contents

Contents **viii**

List of Figures **xii**

List of Tables **xiv**

1 Introduction **1**

1.1 Overview 3

1.2 Background 5

1.3 Motivation 7

1.3.1 Overhead of Race Detection 7

1.4 Opportunities for Race Healing 10

1.5 Contributions 11

1.5.1 Stationary Analysis 11

1.5.2 Loop Iteration Sampling 12

1.5.3 On-the-fly Race Healing System 12

1.6 Dissertation Statement 13

1.7 Dissertation Organization 14

2 SOS: Stationary Object Suppression **15**

2.1 Introduction 16
2.2 Approach 19
2.2.1 Challenges and Opportunities 21
2.2.2 Applying Stationary-Object Optimization 22
2.2.3 Major Concepts 22
2.2.4 Supporting Stationary Object Analysis 24
2.3 Implementation 26
2.3.1 Effects on Race Detection Coverage 28
2.3.2 Reducing Monitored Objects and Operations 32
2.4 Performance 34
2.4.1 Performance of SO 35
2.4.2 Performance of SO_n 36
2.4.3 Performance of SO_s 38
2.5 Conclusions 40

3 LIS: Loop Iteration Sampling **42**

3.1 Introduction 43
3.2 Effects of Loops on Race Detection 45
3.2.1 Monitored Operations within Loops 46
3.2.2 Detected Races within Loops 47
3.3 Approach 48
3.3.1 Loop Iteration Sampling Algorithms 49
3.3.2 Implementation 53
3.4 Performance Evaluation 55
3.4.1 Efficiency of SOS_{LIS} 56
3.4.2 Effectiveness of SOS_{LIS} 58

3.4.3 Effectiveness of SOS_{LIS} with Fixed Overhead 60
3.5 Conclusion 63

4 RaceDr: A Race Healing System 65

4.1 Introduction 65
4.2 Preliminaries 69
4.2.1 Semantics of Multithreaded Programs 69
4.2.2 Vector Clock Based Race Detection 71
4.3 Atomicity Violation Detection 72
4.3.1 Atomicity Violation Detection Algorithm 74
4.3.2 Proof of Correctness 75
4.4 Overview 78
4.4.1 Precise Race Detection 79
4.4.2 Race Healing Systems 80
4.4.3 Our Approach 81
4.5 Implementation of RaceDr 83
4.5.1 Race Detection 84
4.5.2 Race Healing 86
4.6 Evaluation of RaceDr 89
4.6.1 Evaluation Environment 90
4.6.2 Using RaceDr In-House 90
4.6.2.1 Cost of Race Healing in RaceDr 90
4.6.2.2 Performance of Repaired Races 93
4.6.2.3 RaceDr Fix Effectiveness 94
4.6.3 Using RaceDr in Deployed Environments 95
4.6.3.1 Impacts of Sampling on Detection Effectiveness 95

4.6.3.2 Impacts of Disabling Race Detection 97
4.7 Conclusion 99

5 Related work **101**

5.1 Race Detection 101
5.2 Atomicity 104
5.3 Fault Repair 105
5.4 Optimizing Runtime Monitoring 107
5.5 Stationary Fields 110

6 Conclusion and Future Work **111**

6.1 Conclusion 111
6.2 Future Work 112
6.2.1 Code Based Sampling for Race Detection 112
6.2.2 Race Repair Study 113
6.2.3 Concurrency Attack Study and Fixing 114

Bibliography **117**

List of Figures

1.1 Architecture of race detection and healing system. 4

1.2 Example of data races. 9

1.3 Bug 31018 in Apache Bugzilla 11

2.1 Percentage of objects that must be monitored in SO. 33

2.2 Percentage of operations that must be monitored in SO. 33

2.3 Overhead of SO and SO_n normalized against that of FastTrack implementation in Pacer. 36

2.4 Comparing the number of missed races in SO_n with that of SO normalizing with the number of races detected by FastTrack. 37

2.5 Comparing the effectiveness of SO_s and Pacer in detecting races within low overhead budgets. 39

2.6 Comparing race detection effectiveness between SO_s and *Pacer*. 41

3.1 Projected fractions of costs if operations in loops are not monitored. 47

3.2 Data Race Example 50

3.3 Instrumentation For Method Call 54

3.4 Overheads of SOS, $Pacer_{LIS}$ and SOS_{LIS} normalized against Pacer's. 58

3.5 Comparing the raced detection effectiveness of detectors with loop-optimization with the ones without given low overhead budgets. For each application, we also report the number of detected unique races by SOS_{LIS} above the solid black line. 61

3.6 Comparing race detection effectiveness at low overhead between $Pacer$, SOS, and SOS_{LIS}. 63

4.1 Unique races detected across 20 program runs. 69

4.2 Example of Atomicity Violation 73

4.3 Partial order decided by locks. 77

4.4 Apache Tomcat excerpt (left) and high-level race (right, in box). 82

4.5 Effects of sampling on detection coverage. 94

4.6 Race repairs with varying detection overhead. 96

4.7 Percentage of detected races after 20 runs. 98

List of Tables

2.1 Basic description of each benchmark. 31

2.2 Comparing the number of detected races and detected bytecodes that cause races (FastTrack versus our approach). 31

2.3 Anaysis result of object demographics. The percentage of SO objects over all objects reported in parentheses for each application. 33

2.4 Comparing read and write operations between FastTrack and our approach. Note that we also report the percentage of Non-Stationary Reads over Fast-Track Reads for each application in parentheses. 34

2.5 Comparing the slowdown factor of the implementation of FastTrack in Pacer ($FastTrack_{Pacer}$), SO, and SO_n over the execution time of RVM with no race detection. For example, *xalan* running on $FastTrack_{Pacer}$ is 11.6 times slower than running on RVM with no race detection. 35

3.1 Comparing the slowdown factor of the implementation of $Pacer$, SOS, $Pacer_{LIS}$, and SOS_{LIS} over the execution time of RVM with no race detection. For example, *avrora* running on $Pacer$ is 5.1 times slower than running on RVM with no race detection. 57

3.2 Comparing the number of detected unique races. 58

3.3 Comparing the average sampling rate used by each approach to maintain the budgetted overhead. 62

4.1 Basic description of each benchmark. 67

4.2 Vector clock updates for Figure 4.4. 86

4.3 Race detection and race healing slowdown factors and effects on performance due to repairs. 91

4.4 Reporting lock usage and lock acquisition characteristics. 92

4.5 Average number of detected races before a repair is applied. 97

Chapter 1

Introduction

Perpetual availability is a key requirement of today's computer systems as society's reliance on such systems is increasing. For instance, when safety critical systems such as medical devices or utility systems fail, there are dire consequences that can result in loss of life. In addition, as more organizations rely on application servers and cloud services to conduct financial transactions, support day-to-day operations, and provide valuable services to their customers, failures of these systems can result in consequences ranging from customers annoyance to heavy financial losses [Par11]. To provide perpetually available services, the fault detection and repairing processes have to be non-intrusive and automatic given the highly likely presence of faults in deployed systems. As such, non-automated and time consuming repair processes that include fault issuing, manual fault repairing and even dynamic patching cannot be used. Moreover, these typical processes often require system rebooting, which impairs system availability.

Currently, multicore processors have become the default configuration not only for high performance servers, but also for personal computers and mobile computing devices. Therefore, multithreading is an effective way to exploit computation power of multicore processors. However, writing correct concurrent programs is challenging due to subtle

concurrency faults like data races, deadlocks and atomicity violations, which represent major classes of concurrency faults in modern systems. Although static analysis and model checking can guarantee correctness of concurrent programs, they cannot scale to large complex applications in real world [HJM04, KYKS09, VJL07, NAW06, CLL+02]. Software testing is a practical way to find concurrency faults [PS08]. However, concurrency faults are sensitive to thread interleaving, exponentially increasing with the number of threads, which test suites are unable to exhaustively cover in general. As a result, concurrency faults could stay dormant in testing period and then appear during fielding [LPSZ08, KP04]. Based on this emerging computing trend, we need to have approaches and techniques that can detect and repair various types of concurrency faults in deployed systems.

One famous example of a concurrency fault is the Northeast Blackout of 2003 that caused part of the US and Canada to lose power for many hours [KP04]. That incident was initiated by a data race that prevented an alarm system from notifying power-plant operators of impending power transmission problems. The main culprit that caused the alarm system to fail was a race condition that corrupted a shared data structure. Once this data structure was corrupted, the alarm system entered a live-lock state and completely stopped processing incoming events causing massive backlogs. During their investigation, engineers discovered that the same race had manifested itself as corrupted alarm logs. However, after three million hours of operation, that race-related fault had never caused the alarm system to completely fail.

This example shows that even very well tested systems can still have data races that may produce harmful results. As such, it is critical to have race detection capability in deployed systems. Because it is quite common for data races to appear multiple times before they can actually harm the system, there are opportunities to repair these races before they become harmful.

1.1 Overview

This dissertation takes a first step toward building an automatic and transparent concurrency fault detection and healing system. The focus of our work is on **building a race detection and healing system for deployed applications**. We choose data races for several reasons. First, data races are a common class of concurrency faults, which occurs when there are two concurrent accesses to the same data, and at least one of them is a write. Second, historical reports have shown that data races can cause major failures, resulting in heavy financial losses and even fatalities [KP04, LT93]. Third, recent research shows that data race still exists in widely used commercial software such as eclipse IDE and Apache Web server [BCM10, Fou]. Fourth, it is also possible to extend data race detector to cover other interesting property violations including atomicity. As such, we build our system to satisfy the following requirements:

1. Low detection overhead. Our system yields low overhead so that it can be used in deployed software.

2. Efficient repair. The detection and healing process is on-the-fly and transparent so it can occur without interrupting execution of applications.

3. Effective repair. Our system is capable of determining if a repair can subsequently cause deadlock. If it does, the repair is not applied.

The architecture of our proposed system is shown in Figure 1.1. To clarify, $RaceDr$ is a race healing framework hosting race detectors like $Pacer$, LIS and SOS as plugin components. The system includes the *Race Detection* and *Race Healing* components. Race Detection component can support multiple race detectors. Currently, three have been implemented: *Pacer*, *SOS* and *LIS*. *Pacer* was introduced and implemented by Bond et

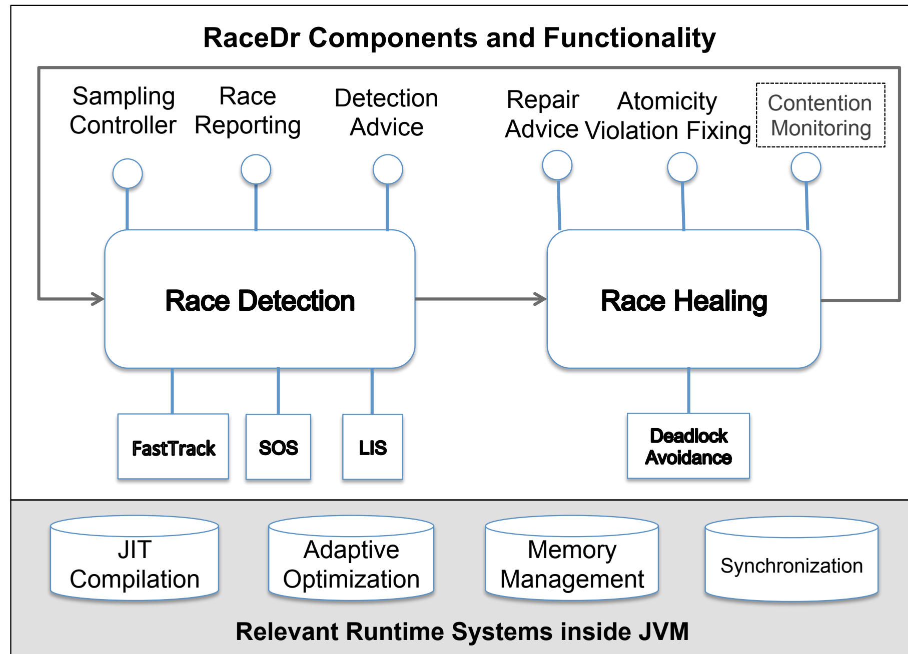

Figure 1.1: Architecture of race detection and healing system.

al. [BCM10] while *SOS* and *LIS* are proposed by us. All of these race detectors use sampling to control overhead. We extend *SOS* to detect atomicity violations that can lead to high-level data races. Atomicity violation is defined as an interruption of an atomic method execution by other thread's action. Atomicity violations can occur even if the program is data race free.

Detection Advice can help to rule out benign races from harmful races. Our proposed *RaceDr* can dynamically inject code to fix or avoid program faults. For example, *RaceDr* can automatically generate and deploy patches to fix data races detected by *Race Detection*. *Atomicity Violations Fixing* will automatically fixing high level data races on the fly; however, the fixing could introduce deadlocks. As such, RaceDr includes *Deadlock avoidance* to ensure that repairs cannot cause deadlocks. As future work, we will implement *Contention Monitoring* to help evaluate the performance of generated repairs. If a repair causes

a significant performance slowdown due to contention on that repair lock, the system can remove the repair. We built RaceDr as part of a Java Virtual Machine (JVM) to exploit the existing runtime infrastructure.

1.2 Background

Two classes of approaches have been proposed to dynamically detect data races. Vector-clock-based approaches build *Happens-Before* relationships by recording logical times of memory accesses with vector clocks [Lam78]. In this approach, there is a race if two memory accesses do not have *Happens-Before* relationship and at least one of them is a write. The approach is precise, that is, there are no false positives. However, it has high overhead due to the need to monitor all memory accesses. As a dynamic detection approach, it can only report races that have been detected under observed interleaving. This means, that other races may exist but they can only be exposed under different interleavings that have not been executed. As such, they are not reported.

Lockset-based approaches assume that every access to shared data has to be protected by locks in order to ensure data race freedom. Therefore, this approach checks locks protecting each memory access. A data race will be reported if two accesses do not share at least one lock while accessing the same memory location. This approach generally outperforms a vector-clock-based approach since it does not record the logical time of each memory access. It can also discover potential data races that do not present in observed thread interleavings. However, Lockset based approach is imprecise and can report false warnings because it does not consider some synchronizations like fork-join relationship.

Significant advances have been made to reduce overhead of dynamic race detection. For instance, $FastTrack$[FF09], as a vector-clock-based approach, achieved comparable performance as lockset-based approach while it still keeps the precision. Even with these

recent advancements, $FastTrack$ still incurs an average overhead of 850% across a range of programs. Although sampling based techniques like $Pacer$[BCM10] can successfully control the race detection overhead to be low enough for deployed systems, it is not effective because to incur no more than 86% overhead, the approach can only monitor about 3% of possible race-inducing operations.

Furthermore, data race is not only hard to detect, but also challenging to fix. Typically, repairing data race by inserting locks can introduce other faults including deadlocks [PKL+09]. Atomicity violation fixing could be even more challenging and error prone since it involves locking a code region instead of just two data access operations. Again, improper fixing could seriously reduce execution concurrency or even lead to deadlocks.

In addition, if repairs can be automatically generated, applying them transparently is still a major challenge. Recently, there have been several research efforts that have been introduced to repair concurrency faults. AFix [JSZ+11] is a tool to automatically fix atomicity violations. It takes a fault report as its input, then automatically generates patches to fix the violations. It also uses static and dynamic program analysis to ensure the fix is deadlock free. Note that AFix automates the manual patch generation procedure; however, it needs recompilation to repair faults, causing the system to be unavailable during the recompilation and restart period. LOOM provides a framework to apply hot patches manually, but it still require patches generated by developers, which could be time consuming and error prone as we discussed [WCY10]. To the best of our knowledge, there is no existing work that can automatically generate and deploy repairs for concurrency faults.

1.3 Motivation

Next we use a motivating example to show why the overhead of race detection is so high and the opportunities to improve its efficiency. We also study the real world examples to show the property of data races and atomicity violations, leading to a chance to automatically fix them on the fly.

1.3.1 Overhead of Race Detection

To demonstrate the inefficiency of existing race detectors, consider an example shown in Figure 3.2. The program has two instances of class $Datum$, s and n, which hold a value $flag$ that is operated by three method calls: $inc()$, $clear()$ and $isReady()$. Thread $RacerA$ and thread $RacerB$ are currently running. The instance of $RacerB$ loops to check if s is ready. If s is ready, it increases the $flag$ value of the n. The instance of $RacerA$ clears the $flag$ value of the n first, then performs the same task as $RacerB$.

Checking for data races with $FastTrack$ involves monitoring all read and write operations. For example, each call to $inc()$ involves one read from and one write to $flag$. Including the writes that are performed during initialization of the two $Datum$ field involves 2,000,005 writes and 4,000,000 reads. However, it is unnecessary to monitor all these 6,000,005 operations for two reasons. First, the four writes(two writes to fields s and n in instance of $Example$, and two writes to filed $flag$ in object s and n) during the initialization cannot get involved in races since the objects are not shared at that point. Second, field s is read-only after initialization, that is, the object that s references cannot involve races, so the 2,000,000 reads to field s do not need to be monitored. As such, around one third (2,000,004) of operations can avoid monitoring to reduce the overhead, without affecting detection effectiveness.

In Figure 3.2, there are two distinct races: $race1$ and $race2$. The sources of $race1$

consist of code at line 16 and code at line 26. Sources of $race2$ consists of code at line 13 and code at line 26. $Pacer$, a sampling based race detector based on FastTrack, applies sampling uniformly. As such, when we configure $Pacer$ to operate at a high sampling rate (e.g., 80%), $race2$ could be detected up to one million times. Any detection after the initial detection is redundant because the sources have already been identified. When we configure $Pacer$ to operate at a small sampling rate (e.g., 3%) to reduce overhead (86%), the race detection rate is also very low. Consequently, there is a good chance that $Pacer$, operating at that sampling rate, would miss $race1$. On the contrary, because the statements that cause $race2$ are executed frequently, $race2$ has a much higher probability than $race1$ of being detected.

```
1 public class Example {
2   Datum s = new Datum();
3   Datum n = new Datum();
4
5   public static void main(String[] args){
6     new RacerA().start();
7     new RacerB().start();
8   }
9 }
10
11 class RacerA implements Runnable extends Example{
12   public void run() {
13     n.clear(); //race2
14     for (int i=0; i<1000000; i++){
15       if (s.isReady()) {
16         n.inc(); //race1
17       }
18     }
19   }
20 }
21
22 class RacerB implements Runnable extends Example{
23   public void run() {
24     for (int i=0; i<1000000; i++){
25       if (s.isReady()) {
26           n.inc(); //race1, race2
27       }
28     }
29   }
30 }
31
32 class Datum {
33   int flag = 1;
34   public void inc() { flag++; }
35   public void clear() { flag=0; }
36   public boolean isReady() { flag>0; }
37 }
```

Figure 1.2: Example of data races.

Based on the observation stated above, we identify two possible race detection optimizations that can significantly reduce detection overhead but without significantly affect detection rate.

- **Optimization 1.** Existing race detection approaches monitor unnecessary operations, contributing to excessive overhead. The overhead can be reduced by identifying unnecessary operations and then avoiding monitoring them.
- **Optimization 2.** Current sampling based race detection approaches detect redundant data races mainly due to loop code. The redundancy can be removed to reduce overhead.

1.4 Opportunities for Race Healing

The following example is related to a fault report from Apache Tomcat fault repository as shown in Figure 1.3. One responsibility of method `SystemLogHandler` in the `util.log` class is to capture and redirect output and error messages to the corresponding application's log file. The captured messages are maintained on a globally accessible stack called `reuse`. A call to `isEmpty()` and a call to `pop()` can race – the former reads data storing the size of the stack and the latter can write that data. However, an observable error only occurs when `pop()` is invoked when `reuse` is empty. Any typical dynamic and precise race detector would be able to detect the first instance that a read by `isEmpty()` happens concurrently with a write by `pop()` or vice versa.

```
if (!reuse.isEmpty()) {
  log = (CaptureLog)reuse.pop();
} else {
  log = new CaptureLog();
}
```

Figure 1.3: Bug 31018 in Apache Bugzilla

The example demonstrates a key insight: data races occur frequently in deployed concurrent software but often their effects on system behavior are too subtle to be noticed and would not be classified as a failure. Therefore, we can take advantage of the time window to dynamically fix data races before they cause dire consequences.

1.5 Contributions

My work to date has made the following contributions to the area of dynamic race detection and repair. They are described in the next three subsections.

1.5.1 Stationary Analysis

As stated in Optimization 1, a large number of objects cannot be involved in data races. These objects are *stationary objects*. **A stationary object is an object that is written during the initialization period and after the object has been read, there are no more write operations.** The notion of stationary objects is extended from the definition of stationary fields [UL08]. A race occurs when two accesses to a shared data are not properly synchronized and at least one access is a write [BCM10, FF09]. A stationary object cannot get involved in races for two reasons. First, a stationary object is not shared in initialization

phrase(Only the creator thread can access it.) Second, there are no writes after a stationary object is shared by definition.

1.5.2 Loop Iteration Sampling

Fundamentally, stationary analysis aims to increase density of data races over objects by reducing monitoring on a class of objects that cannot involve races. Based on Optimization 2, we propose a sampling technique that aims to increase density of data races over executed code, which is orthogonal to stationary analysis. The main idea is to differentiate the monitoring efforts between operations that reside in loop regions and those that do not. The goal is to reduce the overhead for monitoring repetitive operations in loops. The example in Figure 3.2 reveals a glaring inefficiency of techniques that uniformly sample monitoring operations over time. This finding motivates us to build a sampling based race detector that differentiates sampling loop code and non-loop code. It monitors iterations of loops in a program and decreases the sampling rate for code in a loop when iterations for that loop increases. More details about this approach, named *LIS*, will be explained in Section 3.

1.5.3 On-the-fly Race Healing System

As discussed, mission critical systems cannot afford downtime. The Just-In-Time (JIT) compilation infrastructure provides the facility to transparently correct faulty code, like code involving races. It is natural that we fix the data races on the fly by regenerating racy code with proper synchronization based on JIT compilers. Therefore, the program can keep running while the data races are eliminated. Moreover, as the system keeps fixing data races, the system will have fewer and fewer races and become data race free eventually, which leads to the opportunity that the data race detector can be shut down after all or most data races are fixed. We propose an online race healing system, *RaceDr*, to transparently fix

data races. However, data race freedom cannot ensure atomicity of the program [LPSZ08], so we propose an approach to automatically fix atomicity violations based on *RaceDr*.

1.6 Dissertation Statement

As stated, data races are currently the most common concurrent fault. However, existing data race detection techniques are often too expensive for field use. Furthermore, existing fault healing techniques cannot repair races immediately, transparently, and non-intrusively after they are detected. We propose an efficient, effective and transparent framework, including two orthogonal race detection optimizations and an one-the-fly healing system, to detect and repair data races and atomicity violations for deployed software.

The dissertation makes the following contributions:

1. We build a dynamic program analysis technique called *Stationary Analysis*, to identify objects that cannot participate races. When the analysis is used, we can enhance race detection rates by up to six times while maintaining the same overhead as that of a sampling based system without stationary analysis.

2. We extend an existing sampling based race detector to differentiate sampling rates between code within loops and code outside loops. This approach reduces the race detection overhead by reducing redundant monitoring operations (i.e., monitoring operations that have already been identified as racy), but without significantly sacrifice race detection coverage.

3. We propose a race healing system that works with our race detectors to automatically fix races as soon as they are detected.

4. We extend our race detector to detect atomicity violations. Then we utilize our healing system to also fix these detected atomicity violations.

1.7 Dissertation Organization

This dissertation is organized as follows. In Chapter 2 and Chapter 3, we introduce our two dynamic data race detectors, *SOS* and *LIS*, including the approaches and evaluation results. Chapter 4 focuses on our data race healing system, *RaceDr*. We will explain the prototype we have built and the future work. In Chapter 5, we discuss related work.

Chapter 2

SOS: Stationary Object Suppression *

Data races are subtle and difficult to detect errors that arise during concurrent program execution. Traditional testing techniques often fail to find these errors, but recent research has shown that targeted dynamic analysis techniques can be developed to detect races that occur during a program execution. State-of-the-art techniques have shown that precise race detection (i.e., where no false race reports are generated) can be achieved with comparable runtime overhead to the best known dynamic techniques. Unfortunately, even with these recent advances the overhead of race detection remains very high—commonly incurring an 8 times slow down in program execution [FF09].

In this work, we incorporate an optimization technique based on the observation that many thread-shared objects are written early in their lifetimes and then become read-only for the remainder of their lifetime; these are known as *stationary objects*. Races cannot occur on accesses to stationary object since all accesses are reads, and therefore, our proposed approach does not monitor those accesses. The main contribution of our work is concentrating the monitoring effort on objects that can participate in races. Our experimental result shows that our proposed system only incurs an an average overhead of 45%

*Some of the material in this chapter appeared in [LSaD11].

of that of *FastTrack*, a low-overhead dynamic race detector. We then compared the effectiveness of our approach to detect races in deployed environments with that of *Pacer*, a sampling based race detector based on *FastTrack*. We found that our approach can detect 5 times more races than *Pacer* when we budget 50% for runtime overhead.

2.1 Introduction

Modern microprocessors provide multiple processing cores per chip. As such, a natural way for developers to achieve higher performance is to employ thread-level parallelism in their applications. However, writing correct concurrent programs can be challenging, especially achieving proper synchronization of access to shared resources. Improper synchronization can lead to runtime errors such as deadlocks and data races, which are difficult to detect, isolate, and correct. As an example, data races are often sensitive to execution interleaving, and therefore, may only occur infrequently and intermittently. Furthermore, many of these races do not always produce incorrect results, making detecting their presence difficult. Thus, we have seen many instances in which these races stay dormant during the testing period and then manifest themselves during deployed system operation [Apa, Can, Lia].

Detecting data races can lead to improved dependability of multithreaded applications. However, doing so can prove to be difficult. Approaches that rely on static program analysis can scale well but tend to result in conservative approximation of potential sources of races [AFF06, NAW06, VJL07]. On the other hand, dynamic detection techniques that use vector clocks to track the ordering of accesses to individual object fields are precise, but incur very high runtime overhead [BCM10, PS03, PS07, FF09]. Dynamic detection techniques based on recording information at the class level rather than the object level, such as lockset approaches, can reduce runtime overhead, but they can also issue many false race reports [EQT07, SBN+97].

FastTrack, a dynamic race detection system introduced by Flanagan and Freund, can reduce the overhead of vector clock based race detection to be about the same as that of lockset based race detection, but without compromising the precision provided by the use of vector clocks [FF09]. The key idea in FastTrack is to replace a large percentage of full vector clock operations with a more time and space efficient operation to track access ordering. According to their reported results, even with this optimization FastTrack still slows applications down by eight times on average, making it infeasible for use in many testing and system deployment contexts.

To reduce the runtime overhead of dynamic analyses, a number of researchers have explored the use of sampling techniques, e.g., [AVY08, BLH10, DDE08]. For race detection, the *Pacer* system adds sampling to the FastTrack algorithm [BCM10]. Sampling is effective in reducing runtime overhead, but there is a corresponding reduction in data race detection. As an example, when the sampling rate of Pacer is set to 1 and 3%, the execution overheads are on average 52 to 86%, respectively. At 86% overhead, the average race detection rate is 30% (ranging from 0.9% to 82.7%). When the sampling rate is set to 100% (i.e., Pacer detects all possible dynamic races), the average execution slowdown is a factor of 12.

State-of-the-art dynamic race detection techniques such as FastTrack and Pacer are too expensive to be "turned on" all the time in deployed systems. Moreover, when sampling is used to drive overhead down to a tolerable level, e.g., less than 100%, the number of detected data race drops significantly. In this work, we introduce an optimization approach that eliminates the need to monitor a large number of objects, and shared accesses to fields of those objects, for race detection.

The proposed optimization is based on the notion of *stationary field* introduced by Unkel and Lam [UL08]. A stationary field is a field that has been written during the initialization period and once the field has been read, there are no more write operations

afterward. We extend the definition of "stationary" to define *stationary objects*. The key insight that motivates our work is that a race occurs when two accesses to a shared variable are not correctly synchronized, and *at least one access is a write* [BCM10, Lam78]. Because stationary objects are not written after escaping, they cannot be involved in races. We define a dynamic analysis to determine whether an object is stationary and couple it with the FastTrack algorithm where processing of read operations is disable when an object is stationary.

We implemented this analysis on top of the Pacer's code base in Jikes RVM and explored its effectiveness in reducing the race detection cost of FastTrack algorithm implemented as part of Pacer. (Note that in the remaining of this chapter, our evaluation of FastTrack is based on the Pacer's implementation of the algorithm.) We also evaluated its ability to increase the detection of races by Pacer at low sampling rates. In addition, we implemented an additional *LiteRace*-like optimization [MMN09] that disables stationary object analysis on "hot" methods and explore its effects on overhead and race detection effectiveness. We show, through an experimental evaluation using 6 multithreaded Java benchmarks, that optimizing dynamic data race detection in this way is effective in reducing the average overhead of FastTrack by 55% and increasing the effectiveness of race detection with 50% overhead budget by nearly a factor of 6 over Pacer.

The key contributions of the work lie in: (1) the design of a light-weight dynamic analysis that can eliminate the need to monitor a large number of read accesses for precise data race detection, (2) an evaluation of the potential overhead reduction that can be achieved by coupling this analysis with state-of-the-art race detection approaches, and (3) an evaluation of the potential improvement in the detection of data races that can be achieved at very low overhead rates by coupling the analysis with Pacer.

2.2 Approach

In Java, a programmer can declare an object field to be *final*, if the field is read-only after it has been initialized by the constructor [UL08]. The use of this feature allows the Java compiler to both enforce the read-only property and exploit that property for optimization. Unfortunately, programmers, in practice, rarely declare fields to be final, even though the access pattern of many fields meet the requirements for final fields. Furthermore, the definition of a final field comes with some rather restrictive conditions; these conditions are designed to make the validation of field finality by the compiler simple. One effect of these conditions is that certain fields that share the essential characteristics of final fields cannot be declared final; for example, fields whose initialization logic is within the constructor but involves complex control flow (which can make validation of field finality complex) or whose initialization logic is performed outside of the constructor. While Java itself cannot capture the read-only nature of such fields, work by Unkel and Lam provides a solution [Unk09, UL08].

In their work, Unkel and Lam introduced the concept of *stationary fields* as a generalization of final fields. The definition of stationary fields extends that of final field by allowing a stationary field to have multiple writes across multiple methods as long as these write operations happen before all read operations. A final field, on the other hand, is written only once during execution [GJSB05].

They then proposed an automatic inference approach to statically detect these stationary fields. During initialization, their algorithm monitors read and write operations to each field. It then analyzes whether a field is written after initialization; if it is, that particular field becomes *non-stationary*. To reduce the analysis overhead, they make a "simplifying assumption" that an object initialization occurs before that object's reference is stored into any object [UL08]. After the reference is stored, the object is referred to as *lost*. In their

implementation, lost objects are identified by monitoring `putfield` bytecode [Unk09]. In terms of the relationship between lost objects and method-escaping objects, lost objects are only a subset of method-escaping objects because the definition of lost does not account for any reference returns at the end of a method call. Furthermore, a subset of lost objects may be thread-escaping in multi-threaded applications.

The result of their empirical study using 26 benchmark programs showed that stationary fields are prevalent in Java programs. The percentage of stationary reference-typed fields ranges from 44 to 59% when both applications and libraries are considered. For primitive-typed fields, the percentage of stationary-field is greater than 30% in each of the evaluated benchmarks. Two key insights from this study are applicable to dynamic race detection:

1. There is no need to monitor for races before an object is lost because the object is still thread local. Note that Unkel and Lam use to notion of lost to indicate the end of initialization. However, from data race detection point of view, lost also indicates that an object may no longer be thread-local and can participate in races.

2. Races cannot occur on stationary fields because they are not written after they have become lost. As a reminder, in order for data race to occur, at least one concurrent access must be a write. Stationary fields are read-only.

These insights tell us that *monitoring only non-stationary fields should be sufficient for race detection*. The potential impact is significant reduction in the number of fields or objects (if we can identify objects with only stationary fields) that must be monitored for data races. We also anticipate that fewer monitored objects would lead to fewer monitored read and write operations, resulting in significantly lower dynamic race detection overhead. Next, we describe how this insight can be applied to existing dynamic race detection approaches such as FastTrack.

2.2.1 Challenges and Opportunities

In principle, this sounds straightforward, but several challenges must be overcome to realize this optimization in the context of modern data race detection algorithms.

First, we must enable and disable the processing of read operations on a per object basis. Doing this efficiently is not possible using an instrumentation approach; FastTrack is implemented through instrumentation. We use per object meta-data encoded in the VM's object structure to control the processing of the race detection algorithms.

Second, we must detect when an object's field will only be read during the portion of the object's lifetime when it is thread shared. We could use a static analysis, such as Unkel and Lam's stationary field analysis [UL08], but instead, we use an efficient dynamic analysis to determine object fields that are read-only when shared. This analysis sets the per object meta-data that controls data race algorithm processing.

Third, since we use a dynamic analysis to determine read-only object fields we must account for the fact that such an analysis cannot know the *future* access pattern on the object. We achieve this by designing an *optimistic* analysis that assumes each object is read-only once it escapes its creating thread and then reclassifies an object once a write is observed. While efficient, the weakness of this approach is that it creates a window during program trace within which data races may not be detected. We discuss the impact of this on data race detection in Section 2.3.1.

As we demonstrate, this optimization strategy can work with essentially any object-based data race detection approaches. It can be applied to significantly reduce the overhead of data race detection. It can also be coupled with existing sampling-based data race detection techniques to significantly improve the number of data races that can be detected when monitoring at very overheads, e.g., below 100%. We explore this in detail in Section 2.4.

2.2.2 Applying Stationary-Object Optimization

The goal of our work is to only apply race detection monitoring on objects that can potentially suffer from data races and no monitoring at all on objects that cannot suffer from races. Our main insight is that races can only occur on non-stationary fields and therefore, monitoring should only be turned on for objects with such fields. Later in this section, we will describe our approach to accomplish this goal. However, we first would like to describe three concepts that are essential for this work. First, we briefly summarize FastTrack, an optimized vector clock-based race detection technique that has demonstrated much lower overhead than other vector clock-based approaches. Second, we define *stationary object*, which is a notion we extend from the previously described stationary field. Third, we describe the three categories of objects that are used as part of the stationary object analysis and how the category information is recorded. In the last part of this section, we describe our approach to integrate dynamic analysis of stationary objects with FastTrack and discuss the shortcomings and benefits of the proposed integration.

2.2.3 Major Concepts

FastTrack. In vector clock-based race detection mechanisms, each vector clock (VC) and VC operation incur $O(n)$ time and space overheads, where n is the number of threads, respectively (see Section 5 for more information about vector clock-based race detection). In FastTrack, the write vector clock is replaced with an *epoch*—a lightweight representation for recording the last write performed on a field that contains the single clock value, `c`, at which the write occurred and the thread that performed the write, `t`. Updating an epoch and comparing an epoch to general vector clock are $O(1)$-time operations. In addition, FastTrack can use epochs to replace vector clocks in most of the read operations. As such, FastTrack is an order of magnitude faster than a traditional vector clock-based race

detector [FF09] and 2.3 times faster than DJIT+, a VC-based race detector for C++ [FF09, PS07].

Stationary Object. Work by Unkel and Lam [UL08, Unk09] introduced stationary fields as a generalized final fields. In our work, we extend the definition of the term "stationary" to objects. That is, a *stationary object* is an object that contains *only stationary fields*. If there is at least one field in the object that is not stationary, the object is considered as a *non-stationary object*. This new definition allows us to control the race detection monitoring at the object-level instead of the field-level. While the object-level monitoring is a coarser monitoring granularity (e.g., if an object has multiple fields but only one field is non-stationary, any read/write access to this object must still be monitored for races), it eases the implementation of the mechanism to enable or disable the monitoring process.

Categorizing Objects. Because our approach only monitors objects for races when they have become non-stationary, we first need to record each object's status. An object's status can be either *initial*, *lost*, or *non-stationary*; initial and lost objects are interpreted as being stationary. Every object is set to initial upon creation. Once the reference to an object is assigned to a field in another object, its status changes to lost. Note that this is the definition of lost introduced by Unkel and Lam [UL08]. If a field in a lost object is written, then its object's status changes to non-stationary. Previous work has shown that there are several ways to record object meta-data. We develop our proposed technique inside a JVM so our approach embeds meta-data in the object's structure; an approach similar to that used in QVM and other work to maintain runtime information as part of dynamic analysis [AVY08, JBM04].

2.2.4 Supporting Stationary Object Analysis

Dynamic data race detection involves monitoring operations related to locking as well as reads and writes of memory locations from different threads.

Monitoring Lock Usage. Our approach monitors lock usage information, such as `monitor_enter` and `monitor_exit` operations, inside the JVM [BCM10, XSaJ08]. The captured information is then used to perform race detection and manage vector clocks in a manner that is identical to existing race detection algorithms. That is, the vector clock for a lock is updated when a thread acquires the lock, and the clock for a thread is incremented when the thread releases the lock.

Conceptually, if a safe determination of which objects are stationary were performed, then lock operations on those objects would not need to be performed. This might be done, for example, via a static analysis that determines that all instances of a class are stationary. We use a dynamic analysis that optimistically determines whether objects are stationary and then reverts to full data race processing when it is determined that an object is non-stationary. Consequently, we prefer a safe treatment of lock operations and monitor them fully for race detection. Since lock operations are relatively rare (3.2% in the FastTrack study), the performance penalty for this decision is modest.

Monitoring Reads and Writes. We implemented our mechanism as read- and write-barriers. The stationary object analysis code as well as race detection code is injected into the processing of various bytecodes such as `putfield` and `getfield`, which perform write and read operations, respectively, on objects.

Algorithm 1 describes the process to monitor for races after a read operation. In the FastTrack algorithm, a read operation requires a check with previous writes on that variable for races; the operation is simply shown as function `checkRaceWrite-Read` in the algorithm (line 2). Next, FastTrack updates the read component of the vector clock of that

```
Input: objRef, objField

1:  if objRef.status == nonStationary {
2:      checkRaceWrite-Read(objRef.objField, currentThread)
3:      updateReadVC(objRef.objField, currentThread)
4:  }
```

Algorithm 1: ReadMonitor()

particular thread. The operation is shown as function `updateReadVC` (line 3) [FF09]. Note that these basic VC operations are clearly described in [FF09, BCM10]. Because our technique only monitors for races when an object becomes non-stationary, it must check the object's status (the code for this check is highlighted in gray) for each read operation. If the status is non-stationary, race detection code is executed.

```
Input: objRef, objField, primitiveValue

1:  if objRef.status == lost
2:       objRef.status = nonStationary
3:  if objRef.status == nonStationary {
4:      checkRaceWrite-Read(objRef.objField, currentThread)
5:      checkRaceRead-Write(objRef.objField, currentThread)
6:      updateWriteVC(objRef.objField, currentThread)
7:  }
```

Algorithm 2: PrimitiveWriteMonitor()

Because our algorithm keeps track of lost objects, our analysis needs to distinguish between writes of primitive data and writes of reference data. Algorithm 2 describes the steps in our proposed algorithm to monitor for races after primitive-typed data, *primitiveValue*, is written to a field, *objField*, of an object, *objRef*. Again, the code to support stationary object analysis is highlighted in gray. The unhighlighted lines show the generic race detection operations that must be performed after a write.

First, our analysis checks the status of the written-to object. If it is lost, then the write operation causes the object's status to change to non-stationary (line 1 and 2). When the

status is non-stationary (line 3), the race detection monitoring code (line 4 to 6) is executed.

When reference-type data is written to a field in an object, additional analysis to track lost objects is needed. In Algorithm 3, lines 3 to 9 are exactly the same as those in Algorithm 2. However, lines 1 and 2 are needed to change the object's status of *referenceValue* from initial to lost because the write operation makes the object pointed to by *referenceValue* accessible from *objRef*.

Input: *objRef*, *objField*, *referenceValue*

```
1: if referenceValue.status == initial
2:     referenceValue.status = lost
3: if objRef.status == lost
4:     objRef.status = nonStationary
5: if objRef.status == nonStationary {
6:   checkRaceWrite-Read(objRef.objField, currentThread)
7:   checkRaceRead-Write(objRef.objField, currentThread)
8:   updateWriteVC(objRef.objField, currentThread)
9: }
```

Algorithm 3: ReferenceWriteMonitor()

2.3 Implementation

We implemented our stationary-object analysis on top of Pacer [BCM10], a sampling-based race detection technique based on FastTrack [FF09]. Pacer implements the FastTrack algorithm inside Jikes RVM 3.1.0, a high performance meta-circular Java Virtual Machine (JVM) [Jik11]. In addition to implementing the FastTrack algorithm, Pacer is also capable of performing sampling in order to control the runtime overhead of race detection. We will also use this sampling feature in our performance evaluation. Next, we describe some key components in our implementation.

Metadata. To record the status of an object (i.e., initial, lost, non-stationary), we used two bits from the object's header field in RVM. Using these two bits does not incur additional storage space, but requires masking and unmasking operations. We also explored an alternative approach to add one extra word per object to record status information. This approach eliminates the masking and unmasking operations, but increases space overhead by one word per object, which can affect heap usage and GC performance. Our investigation revealed that the bit stealing design performs slightly better than the latter approach.

Instrumentation. To detect status transitions, we instrumented all write operations at runtime. We took advantage of the existing write barrier infrastructure in Jikes RVM. There are two kinds of write barriers, primitive type and reference type. Primitive write barriers capture writes to primitive-typed fields. On the other hand, reference write barriers capture writes to reference-typed fields. Since the mechanism to process objects and arrays is similar, we do not differentiate between objects and arrays in our discussion. We also modified the read-barrier mechanism in Pacer to check an object's status before executing the race detection code.

As shown in Algorithm 3, there are two kinds of status transitions in stationary-object analysis that must be detected at runtime:

Transition 1: *initial* → *lost*

Transition 2: *lost* → *non-stationary*

The reference-type write barriers have been implemented to detect both transitions. Once a reference is written to a host object in the heap, the referenced object is marked as lost. The status of the host object also changes to non-stationary if and only if the current status of the host object is lost. On the other hand, the primitive-type write barrier only detects *Transition 2*. Once a field is written, the host object is marked as non-stationary if and only if its current status is lost. The write-barrier code executes right before the actual

write happens.

There are two compilers in Jikes RVM: *baseline* and *optimizing*. Every method is initially compiled by the baseline compiler prior to its first execution. Later, if the method becomes "hot," the optimizing compiler recompiles the method with more optimizations. Our system can be configured to apply instrumentation related to stationary-object analysis to the baseline, the optimizing, or both compilers. Disabling instrumentation in the optimizing compiler allows us to realize a form of optimized data race detection that mimics the intuition of LiteRace [MMN09] — races in frequently executed code are rare, whereas races in cold code are more prevalent.

Enabling/Disabling Race Detection. Pacer's implementation of FastTrack adapts all read/write operations to update and compare vector clocks. We built our stationary-object analysis on top of Pacer. In this implementation, each time a read or write operation occurs, our system checks the status of the host object. If its status is initial or lost, our system does not execute the race detection code. Otherwise, it performs race detection. Our implementation can also work with Pacer's existing sampling feature, which can be enabled or disabled at the end of each garbage collection cycle. Next, we describe the possible impacts of our stationary-object analysis and report our experimental result that quantify these impacts.

2.3.1 Effects on Race Detection Coverage

As shown in the three algorithms, our approach incurs small monitoring overhead to perform status check for each of initial or lost object and full race detection overhead for non-stationary objects. This is much different than FastTrack in which race detection analysis and vector clock updates are performed on all objects.

While this can result in a significant saving, it does come at a cost of *missing a par-*

ticular type of race that occurs at a particular time. Next, we describe the ability of our approach to detect Write-Write, Read-Write, and Write-Read races and explain why our approach misses a particular kind of race. We also report the result of comparing the effectiveness of our race detection against that of FastTrack.

Detecting Write-Write Races. One key step of stationary object analysis is a detecting write operation on a lost object. When that occurs, the object's status changes to non-stationary and from this point onward, race detection is enabled on this object. As such, our technique can detect any write-write race that occurs in a particular program execution.

Detecting Write-Read Races. When a read operation is performed on a non-stationary object, the first step of race detection is to check if the read races with prior writes. Since we turn on race detection when write operations are performed on lost and non-stationary objects, our system already maintains sufficient information on these objects to detect races. As such, our technique can detect any write-read race that occurs in a particular program execution.

Detecting Read-Write Races. When a write operation is performed on a lost object, a check with prior reads is performed. However, because we do not monitor read operations for lost objects, our system does not have sufficient information to detect the first instance of read-write race when an object is in lost state. As such, this type of race goes undetected in our system. However, if the same type of race occurs on the same object later on, our system would be able to detect it because by then the object has already become non-stationary, an object state in which full race detection monitoring is enabled. Next, we quantify the occurrences of undetected races due to our approach.

Quantifying Undetected Races. We conducted an experiment to quantify the number of races that goes undetected in our approach when compared to those detected by the FastTrack implementation in Pacer.

Table 4.1 describes a set of six benchmarks, which are commonly used by researchers to evaluate Java systems. Three are from the DaCapo 2006 benchmark suite (*eclipse*, *hsqldb*, and *xalan*) with two additional benchmarks from the DaCapo-9.12-bach suite (*avrora* and *sunflow*) [BGH+06]. The three from the 2006 suite were used in the evaluation of Pacer. Note that some benchmarks in the 2009 suite overlap with those in the 2006 suite. In addition, not all benchmarks in the 2009 suite can run on our version of Jikes RVM with Pacer.

We also attempted to obtain the source code of *pseudojbb2000*, which was also used to evaluate Pacer. Unfortunately, we were not able to do so; as such, we used a newer version called *pseudojbb2005*, which is SPECjbb2005 that has been modified to generate predictable workload in each run [Sta05].

We ran each benchmark 50 times and identified races that have been detected in these 50 runs. This is necessary because races occur non-deterministically, and therefore, a large number of runs are needed to detect most of the possible races. We also chose 50 runs to replicate the experiment used to evaluate the performance of Pacer.

Detected races are reported in Columns 2 and 3 in Table 2.2. Note that we used the same methodology as that used by the authors of Pacer, in which we report races that occur over 25 times. Our approach misses 5 out of 116 races detected by FastTrack. Four of the missed races are in *eclipse* and one is in *xalan*; no races are missed in the other four benchmarks.

From the programmer's point of view, one important source of information that is used to fix races is the actual source of the reported data races. This information is presented in a race report as bytecode indices within methods. In some applications, we observed that a source can participate in many races. Columns 4 and 5 in Table 2.2 report identified sources. As shown in the table, *eclipse* is the only benchmark for which our approach misses some of the sources (3). Note that while our approach did not detect one race in

Benchmark	Description	Number of Threads
avrora	Discrete event simulator of a sensor network (DaCapo 2009).	27
eclipse	Executes some of non-GUI jdt performance tests for Eclipse (DaCapo 2006).	16
hsqldb	Execute a number of transactions against a model of a banking application (DaCapo 2006).	402
pseudojbb	A program emulating 3-tier system (a modified SPECjbb2005).	17
sunflow	A multi-threaded global illumination rendering system (DaCapo 2009).	5
xalan	Transforms XML documents into HTML (DaCapo 2006).	9

Table 2.1: Basic description of each benchmark.

xalan, it could identify all sources of data races.

When one is willing to sacrifice some race detection for a reduction in the overhead of runtime monitoring, as is done in Pacer, the fact that our optimization is lossy has less impact. In that setting, one must study the cost-benefit of the technique, i.e., how fault detection varies with overhead, which is what we do discuss in Section 2.4.

	Detected Races		Identified Sources	
Benchmark	FastTrack	SO	FastTrack	SO
avrora	12	12	10	10
eclipse	**27**	**23**	**32**	**29**
hsqldb	23	23	28	28
pseudojbb	22	22	21	21
sunflow	13	13	20	20
xalan	**19**	**18**	**36**	**36**

Table 2.2: Comparing the number of detected races and detected bytecodes that cause races (FastTrack versus our approach).

2.3.2 Reducing Monitored Objects and Operations

Table 2.3 reports the numbers of objects that an implementation of FastTrack monitors ("All Objects"), the objects that remain stationary throughout the program execution ("Stationary Objects"), and the objects that are determined to be non-stationary ("Non-Stationary Objects"). The percentage of all objects that are monitored when using the *Stationary-object Optimization* (*SO*) are reported in Figure 2.1; these are the non-stationary objects. There are four applications that must monitor fewer than 30% of the total number of objects with *pseudojbb* monitoring less than 2%. The two remaining have significantly more non-stationary objects with *avrora* having the just under 88%.

While a reduction in the number of objects that requires monitoring can provide a sense of the potential savings in race detection cost, the reduction in the number of monitored operations ultimately determines the overhead. Table 2.4 shows the counts of read and write operations for an implementation of FastTrack and for the non-stationary objects using our SO. As expected, the reductions due to the SO lie primarily in the eliminated monitoring of read operations on stationary objects. We also see a slight reduction in the number of monitored write operations. We can achieve this reduction because SO does not monitor any write to an object that is still in the initial state.

Figure 2.2 plots the percentage of FastTrack operations that require monitoring under SO. There are four applications that must monitor fewer than 70% of monitored operations in an implementation of FastTrack with *hsqldb* monitoring about 40%. Out of these four, three (pseudojbb, sunflow, and xalan) also have the largest percentage of stationary objects. However, for *eclipse* the SO eliminates the need to monitor nearly 90% of the objects, yet it still needs to monitor 71% of read/write operations. This result shows that read operations are not distributed evenly among objects. In fact, a large number of objects that we stop monitoring may not have many read/write operations. Next, we evaluate the impacts of the

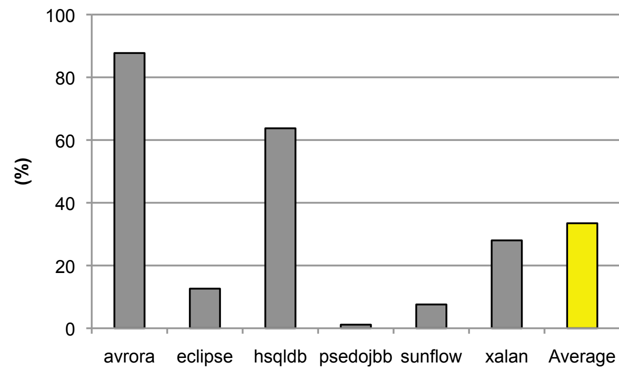

Figure 2.1: Percentage of objects that must be monitored in SO.

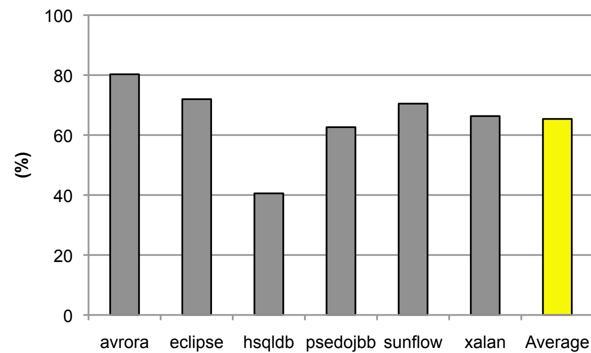

Figure 2.2: Percentage of operations that must be monitored in SO.

reduction in monitored operations on runtime performance.

Benchmark	All Objects	Stationary Objects	Non-Stationary Objects
avrora	301796	37013 (12.26%)	264783
eclipse	1416852	1237967 (87.37%)	178885
hsqldb	134959	48923 (36.25%)	86036
pseudojbb	8246421	8152388 (98.86%)	94033
sunflow	2286384	2113066 (92.42%)	173318
xalan	135852	97791 (71.98%)	38061

Table 2.3: Anaysis result of object demographics. The percentage of SO objects over all objects reported in parentheses for each application.

Benchmark	FastTrack Reads ($\times 10^6$)	Non-Stationary Reads ($\times 10^6$)	FastTrack Writes ($\times 10^6$)	Non-Stationary Writes ($\times 10^6$)
avrora	25.85	20.37 (78.78%)	2.01	2.00
eclipse	63.41	40.65 (64.09%)	18.13	18.04
hsqldb	12.99	4.17 (32.11%)	1.90	1.87
pseudojbb	4.08	2.06 (50.51%)	1.34	1.33
sunflow	9.76	6.80 (69.67%)	0.270	0.269
xalan	8.86	5.40 (61.03%)	1.47	1.45

Table 2.4: Comparing read and write operations between FastTrack and our approach. Note that we also report the percentage of Non-Stationary Reads over FastTrack Reads for each application in parentheses.

2.4 Performance

In this section, we evaluate the runtime performance of SO against that of the implementation of FastTrack in Pacer. As mentioned previously in Section 2.3, our approach can perform instrumentation in the baseline compiler, optimizing compiler, and both. As such, we present two versions of SO: the first version performs instrumentation in both compilers (SO) and the second version performs instrumentation only in the baseline compiler (SO_n). That is, there is no instrumentation to support stationary-object analysis when methods become "hot" and have been recompiled by the optimizing compiler. We then evaluate the performance of these two versions and compare the race detection effectiveness of SO_n against that of SO.

Lastly, we introduce SO_s, which is based on SO_n and incorporates the sampling feature of Pacer. One important objective of Pacer is to perform low-overhead race detection in deployed systems. As shown in their paper, Pacer is able to detect races while maintaining 86% runtime overhead. However, the sampling rate required to achieve this low overhead is 3% of all read/write operations which limits race detection effectiveness. We compare the race detection effectiveness of SO_s against that of Pacer by maintaining fixed runtime overheads of 50%, 70%, 85% and 100%. We also discuss why applying the stationary-

object optimization can increase the effectiveness of sampling-based race detection systems such as Pacer.

In terms of experimental methodology, we executed each of the 6 benchmarks 10 times for each of the data race detection systems. The average performance is reported.

2.4.1 Performance of SO

Tables 2.3 and 2.4 show that the proposed SO can substantially reduce the numbers of objects and operations that must be monitored, respectively. The reduced monitoring efforts result in execution overhead reductions over that of FastTrack as reported in Table 2.5.

On average, the overhead of SO is 72% of that of FastTrack (see gray bars in Figure 3.4). The lowest overhead of 60% of FastTrack's is achieved in *hsqldb*. We also see that in *sunflow* the overhead is as high as 81% of FastTrack's. In all, the overhead ranges from 60% to 82% of FastTrack's. Also notice that the reduction percentage for each benchmark correlates well with the number of operations that has been reduced by SO; that is, the overhead reduction percentage is close to the operation reduction percentage.

Benchmark	Slowdown Factor (×)		
	$FastTrack_{Pacer}$	SO	SO_n
avrora	5.1	4.5	4.2
eclipse	21.0	16.8	8.2
hsqldb	12.2	7.3	3.8
pseudojbb	7.7	5.1	4.3
sunflow	29.2	24.1	13.1
xalan	11.6	7.1	2.6
average	13.8	10.3	5.9

Table 2.5: Comparing the slowdown factor of the implementation of FastTrack in Pacer ($FastTrack_{Pacer}$), SO, and SO_n over the execution time of RVM with no race detection. For example, *xalan* running on $FastTrack_{Pacer}$ is 11.6 times slower than running on RVM with no race detection.

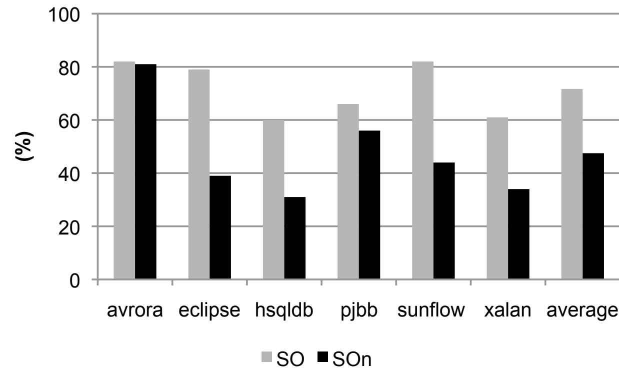

Figure 2.3: Overhead of SO and SO_n normalized against that of FastTrack implementation in Pacer.

2.4.2 Performance of SO_n

In this configuration, we further reduce the number of monitored operations by not having the optimizing compiler perform any instrumentation to support stationary-object analysis or race detection. There are two main insights for exploring this configuration.

1. Based on our race reports, we noticed that all types of races occur repeatedly; many races occur several thousand times in a run. As such, it is possible that monitoring only a small window of operations would be sufficient to uncover most if not all types of races.

2. Work by Marino et al. suggests that "data races are likely to occur when a thread is executing a cold (infrequently accessed) region in the program" [MMN09]. Their experimental result indicated that if they use this observation to guide a sampling-based race detector, it can detect over 70% of data races.

Our approach of only enabling instrumentation in the baseline compiler exploits these two insights by (i) creating a monitoring window based on the time a method executes as unoptimized object code, and (ii) cold regions of the code would be unlikely to go through

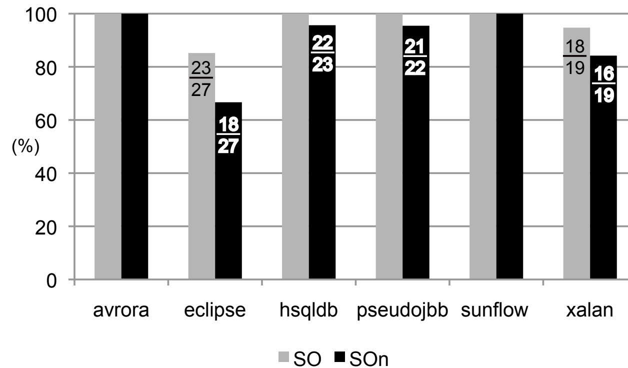

Figure 2.4: Comparing the number of missed races in SO_n with that of SO normalizing with the number of races detected by FastTrack.

the optimizing compiler. As such, when races occur in a cold method, our system can detect them. Figure 2.4 compares the race detection effectiveness of SO_n to those of SO and an implementation of FastTrack. Notice that when our approaches miss detecting races, we report the numbers of detected races over the number of all detected races by the Pacer's implementation of FastTrack. For example, SO_n detects 18 races while FastTrack detects 27 races or $\frac{18}{27}$ in *eclipse*.

As shown in the figure, SO_n still detects over 90% of races in four out of six applications. In the worst performing application, *eclipse*, SO_n still detects 67% of all races detected by the Pacer's implementation of FastTrack. It also detects over 84% of races in *xalan*. Based on this result, this approach still provides good race detection coverage especially in a low-overhead race detection system that are expected to miss some races (e.g., sampling based race detection).

Figure 3.4 reports the average overhead of SO_n normalized against that of the implementation of FastTrack in Pacer (see black bars). The average overhead is 46% of that of FastTrack implementation. The major reason for the smaller overhead is due to the number of monitored operations that have not been injected by the optimizing compiler. With

SO, the average reduction in the number of operations is 30%. SO_n can further increase this reduction percentage to 55%. That is, SO_n monitors less than half of the operations monitored by FastTrack while still detecting 90% of the races.

2.4.3 Performance of SO_s

One interesting feature of Pacer is its ability to guarantee proportionality when sampling is used. As shown by Pacer, sampling can significantly reduce the runtime overhead to a point that race detection is possible in deployed systems [BCM10]. Because we implement our dynamic stationary-object analysis in Pacer, we can also utilize this feature as a way to reduce overhead of SO_n by controlling the sampling rate.

In this section, we report the result of our experiment to evaluate the suitability of using SO_n in deployed environments. Our methodology is applying Pacer's sampling mechanism to SO_n—we call the resulting technique SO_s. We then observe the number of detected races for a particular overhead budget (i.e., 50%, 70%, 85%, and 100%). We then compare the number of detected races for each benchmark for SO_s and Pacer and report the results in Figure 3.5.

It is worth keeping in mind that the overhead of the sampling mechanism in Pacer is around 30%; it is impossible to reduce monitoring overhead below that threshold. Because SO_s also incurs additional overhead for mechanisms to support stationary-object analysis, the overhead of SO_s without turning on race-detection is already at 35%. As such, we set our lowest budgeted overhead to be at 50%.

As shown in the figure, SO_s can detect more races than Pacer in five out of six benchmarks when the overhead is set to be below 100%. For *pseudojbb*, all races originated from one commonly used class. As such, both Pacer and SO_s can detect over 75% of the races with low sampling rates. Also notice that SO_s can detect nearly the same number of races

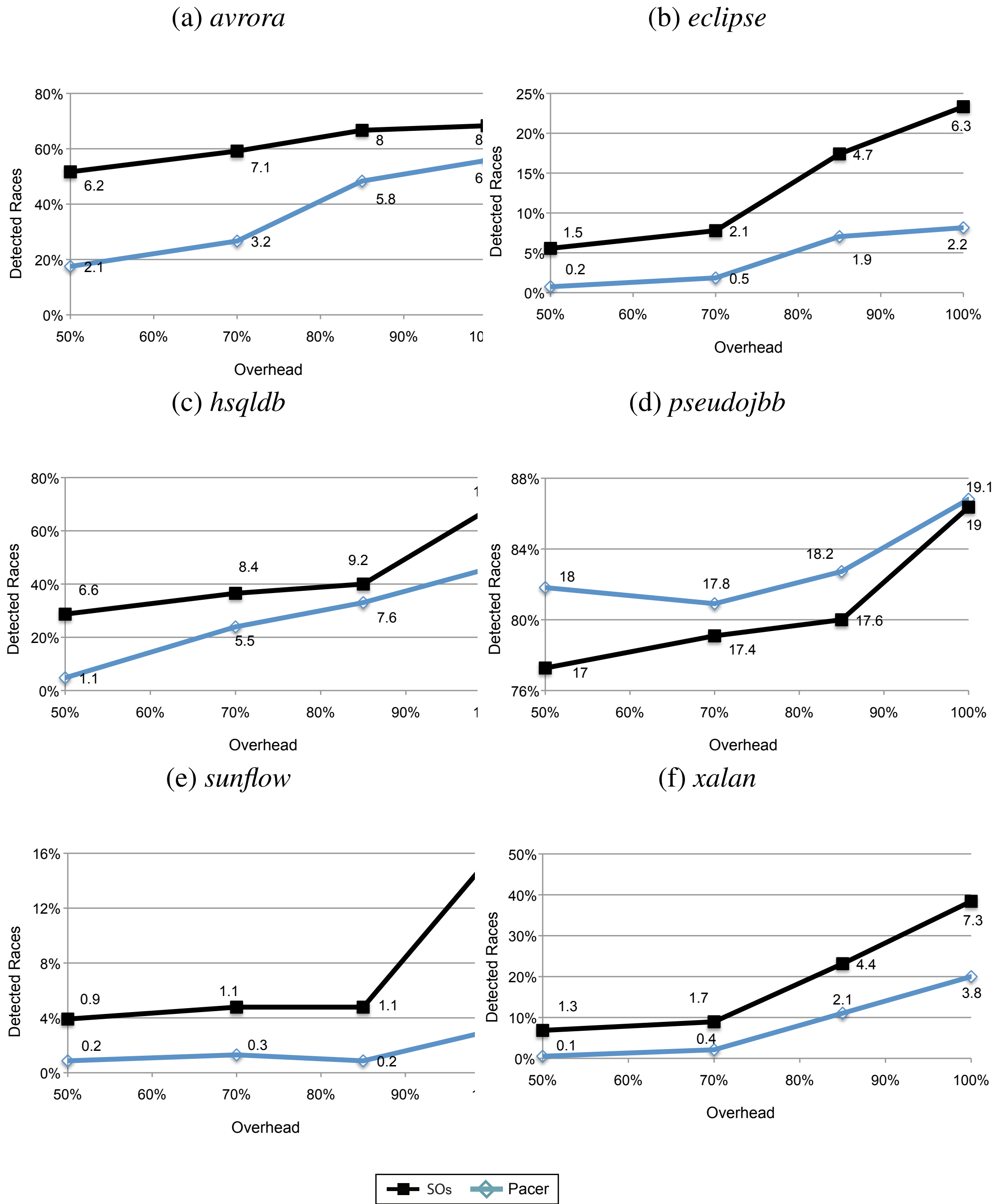

Figure 2.5: Comparing the effectiveness of SO_s and Pacer in detecting races within low overhead budgets.

as that of Pacer in *pseudojbb* (difference of 1 fewer race or less).

Another interesting benchmark is *sunflow*. Pacer cannot consistently detect races until the overhead budget is set to 500% (not shown in the graph). At that overhead, it can detect 1.7 races on average while SO_s can detect 10.7 races on average. One key characteristic that makes SO_s more successful is the reduced set of objects, and read and write operations, that must be sampled. Sunflow is by far the most expensive application for Pacer to execute. It incurs 29 times slowdown when compared to RVM without race detection (see Table 2.5). As such, it can only sample at a very low rate to maintain 100% or less overhead. On the other hand, when the operations that cannot participate in races have been culled by our approach, a low sampling rate has a much better chance of catching race inducing operations. As an example, Pacer can only sample about 3% of the operations in *sunflow* to maintain 85% overhead. On the other hand, a sampling rate in SO that can maintain the same overhead is as effective as a sampling rate of 9% in Pacer. That is, in order for Pacer to detect the same number of races as that of SO, it needs to sample at 9%.

Another interesting point is that SO_s is more effective at the lowest overhead budget. As shown in Figure 3.6, SO_s can detect on average a factor of six more races than *Pacer* when the overhead budget is 50%. As the overhead budget increases, this factor becomes smaller. At 100% overhead, SO_s detects more races than *Pacer* by a factor of 2. If we increase the budgeted overhead to 500% or more, the numbers of detected races between the two approaches begin to converge.

2.5 Conclusions

Data races are subtle and difficult to detect errors that arise during concurrent program execution. As of now, state-of-the-art techniques have shown that precise race detection can be achieved, but at a cost of 8 times slow down in program execution. In this work, we

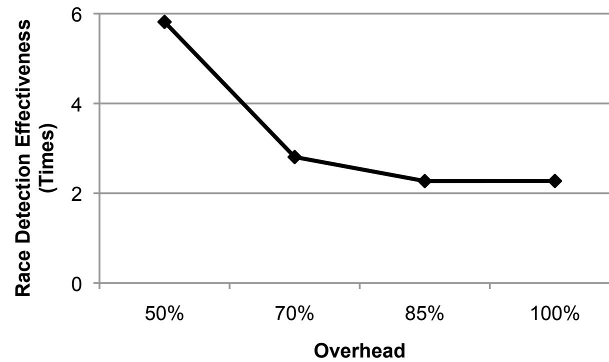

Figure 2.6: Comparing race detection effectiveness between SO_s and *Pacer*.

incorporate an optimization technique based on the *stationary objects* notion. Stationary objects are only written to early in their lifetimes and then become read-only afterward. As such these objects can never participate in data races since they are never written. Based on this insight, we develop a methodology that concentrates the monitoring effort on objects that can participate in races. Our experimental result shows that our proposed system only incurs an average overhead of 45% of that of an implementation of *FastTrack*. Furthermore, when our approach is applied under tight overhead budgets (less than 100%), it can detect up to a factor of 6 times more races than Pacer, a sampling-based race detector based on FastTrack algorithm.

Chapter 3

LIS: Loop Iteration Sampling

Currently, state-of-the-art precise race detectors are too expensive to be "turned on" all the time. As such, recently introduced approaches use sampling to bring the cost of race detection down to a feasible level. In doing so, their effectiveness in detecting races suffers because with sampling, monitoring is only operational for a fraction of the program execution, thereby, reducing opportunities to observe the occurrence of races.

Furthermore, our study reveals that the small fraction of the execution that a detector spends on monitoring is often of limited value due to repetitive detection of the same races. Most programs consist of loop regions. Because these detectors sample uniformly with respect to time, executions in loops mean that loop regions are sampled more often than non-loop regions. Non-uniform sampling increases detection overhead but without increasing detection effectiveness.

In this work, we introduce *Loop Iteration Sampling* or *LIS*, a new detection optimization that spends less time to monitor repetitive operations that occur inside loop regions. The algorithm employs a differential sampling approach to significantly and continuously reduce the monitoring efforts in long running loops. When we apply LIS to two existing sampling-based race detectors, $Pacer$ and SOS, we see a 52% and 33% reduction in over-

head, respectively. LIS also increases the numbers of detected races by a factor of seven for $Pacer$ and a factor of two for SOS, when we restrict the race detection overhead to less than 100%.

3.1 Introduction

As the speed of microprocessors tails off, exploiting the availability of multiple processing cores per chip is becoming an increasingly popular way for developers to achieve higher performance. A natural way to do this is to shift from writing purely sequential applications to employing thread-level parallelism in their applications. Writing correct concurrent programs can, however, be challenging, because improper synchronization of access to shared resources can lead to runtime errors such as deadlocks and data races, which are difficult to detect, isolate, and correct. This is because concurrency faults are sensitive to execution interleavings. As such, they often appear unpredictably and intermittently.

Dynamic race detection provides a way for programmers to detect races as they occur during execution of a program. Currently, dynamic detection techniques that use vector clocks [BCM10, FF09, PS03, PS07] to track the ordering of accesses to individual object fields provide precise detection, but incur very high runtime overhead. Work by Li et al. attempts to reduce the overhead by applying stationary analysis, which identifies objects that are read-only and cannot participate in races [LSaD11]. The evaluation of this technique shows that stationary analysis can reduce the number of objects that must be monitored by an average of 65% across six benchmarks. However, the analysis can only reduce the number of operations that must be monitored by an average of 35%.

As part of this work, we conduct an investigation which reveals that many of these access operations occur within loops. As such, they are executed repeatedly as part of a program execution. We also discover that when a source of data race exists in a loop, it is

detected again and again so monitoring these operations does not necessarily reveal new sources of races. This insight provides an opportunity to reduce the race detection overhead without degrading race detection effectiveness by appropriately reducing monitoring efforts within loops.

Furthermore, our investigation also reveals that monitoring in loops can also reduce the benefit of using sampling as a way to reduce race detection overhead [BCM10, LSaD11]. Existing sampling-based approaches do not sample uniformly across a program. For example, *Pacer*, a sampling based race detector, employs garbage collection events to enable and disable race monitoring. Conceptually, their approach seeks to uniformly sample time slices during which they monitor for data races. However, when a program has long running loops, Pacer ends up monitoring more within those loops, because the program spends more time in those loops. The net result is that sampling is non-uniform across the program locations that might participate in a data race. This leads to increased detection overhead, but without a corresponding increase in detection effectiveness. In the absence of information about where races might be located in a program, a sampling approach for runtime monitoring should uniformly distribute sampling across the entire program to balance cost and effectiveness.

Our work advances the state-of-the-art in race detection in two ways. First, we introduce a new race detection optimization technique that significantly reduces the monitoring efforts in loop regions, and therefore significantly reduces race detection overhead. We achieve this reduction by applying a *loop iteration sampling* (LIS) that reduces the monitoring effort in a loop based on the execution frequency of that loop. We implemented our optimization technique in Jikes RVM and evaluated its effectiveness. The experimental results reveal that it can reduce race detection overhead by an average of 54%, when compare to Pacer, while maintaining nearly the same race detection effectiveness as that of a detector without LIS.

Second, when our approach is used in combination with temporal sampling, it automatically calculates an LIS rate for each loop that aims to spread monitoring uniformly across the program and thereby reduce overhead while yielding good race detection effectiveness. We then evaluate the effectiveness of LIS under four race detection overhead budgets: 50%, 70%, 85%, and 10%. Our evaluation reveals that LIS allows more races to be detected while maintaining the same overhead as those of SOS's and Pacer's. This is because LIS allows a higher temporal sampling rate to be used given a specific overhead budget for a given application.

The remainder of this chapter is organized as follows. Section 3.2 reports the results of our experiment to investigate the effect of loops on race detection performance. Section 3.3 describes our analysis and the implementation of our technique in Jikes RVM. Section 3.4 reports the results of our experimental evaluation of LIS. We compare our results against the most cost-effective sampling based race detection techniques available and show that adding LIS improves those techniques significantly. Section 5 compares our work to other existing work in this area. Section 3.5 discusses future work that can be done to further improve the detection effectiveness of the proposed analysis.

3.2 Effects of Loops on Race Detection

We conducted a set of experiments on the same multithreaded benchmarkas as we used in Chapter 2 to evaluate the effects of memory access operations within loops on race detection overhead. The term loops refers to for-loop, while-loop, and iterators. We modified the Jikes RVM to detect loop execution and then report the numbers of monitored operations that are outside of loop regions. The race detectors used in our experiment are Pacer [BCM10] and SOS [LSaD11]. Pacer was configured to monitor every operation (100% sampling rate).

3.2.1 Monitored Operations within Loops

An underlying principle that motivates the use of cache to speed up memory access time is the 90/10 rule-of-thumb. It states that a program spends 90% of its execution time in only 10% of the code. As such this rule implies that a major part of a program execution is spent in loops.

We hypothesize that such execution behavior also provides an opportunity to reduce the cost of race detection. This is because if race detection related operations reside in loops, they would be accessed repeatedly. The result of our prior work on stationary analysis also hints that such repeated accesses occur in most programs. When we applied stationary analysis, we see an average of 65% reduction in the number of objects [LSaD11]. However, in terms of monitored operations, stationary analysis could only achieve less than 35% reduction (see the last gray bar in Figure 3.1). This result implies that objects that should be monitored for races are accessed repeatedly.

To verify our hypothesis, we conduct an experiment to eliminate monitored operations that reside in loops. We report the results in Figure 3.1. Note that these potential savings are over-approximated because in practice, we cannot eliminate all operations that must be monitored in loops. However, the results provide upper-bounds on the attainable savings.

The results indicate that four out of six applications can significantly benefit from LIS. Three applications show that 60% to 70% of monitored operations reside in loop regions (eclipse, hsqldb, and sunflow). We also see that 50% of operations in xalan are in loop regions. The exceptions are pseudojbb and avrora, which shows little potential benefit from applying LIS. On average, LIS yields higher overhead saving than stationary optimization, but by only a few percent. However, in some applications, loop iteration sampling is more than twice as effective in reducing the number of monitored operations than stationary object suppression. From the results, we conclude that reducing monitored operations in

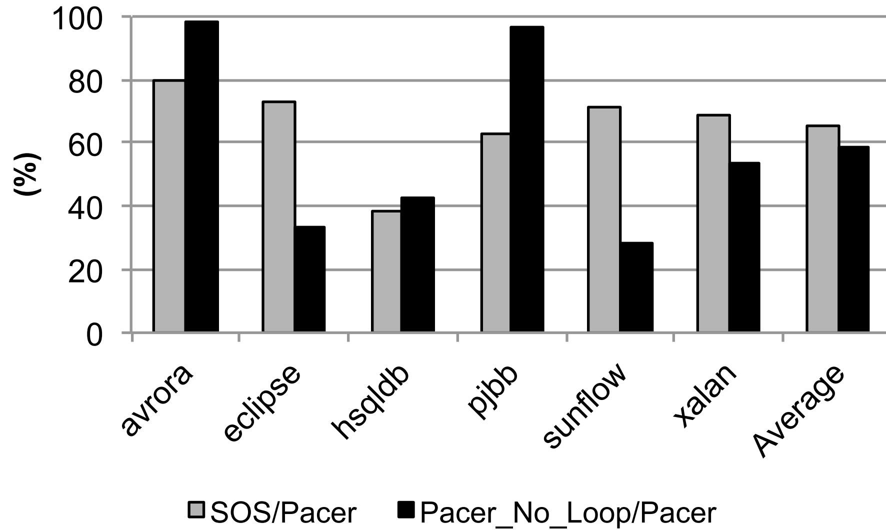

Figure 3.1: Projected fractions of costs if operations in loops are not monitored.

loop regions has the potential to significantly reduce the overhead of race detection in some applications.

3.2.2 Detected Races within Loops

Our previous study also indicated that the statements involved in a data race are often detected repeatedly during a program run. For example, when Pacer is used without sampling, it can detect 12 unique sources of races in avrora. However, during execution, those twelve sources have been dynamically detected over 417,000 times. This averages to nearly 35,000 detected dynamic races per unique race. In the other applications, we see the ratio between dynamic races and unique races to be 599.37 for eclipse, 19.30 for hsqldb, over two million for pseudojbb2005, 3.31 for sunflow, and 1.63 for xalan.

Next, we investigate the number of dynamic races that are detected in loops. We found that in avrora, eclipse, and pseudojbb2005, over 90% of dynamic races are detected in loops. The other three applications indicates that less than 17% of dynamic races are de-

tected in loops. This observation provides even more reasons to apply LIS. First, in applications where most dynamic races are detected in loops, we should be able to reduce monitoring in loop regions but still be able to detect most of unique races. To illustrate this point, we use avrora as an example. As reported earlier, most races are detected in loops. Furthermore, each unique race is dynamically detected over 35,000 times. If we are to reduce the monitoring in loops by a factor of 100, each unique race can still be detected as many as 350 times. Second, for applications that do not have many races in the loop, there is no need to spend a lot of monitoring efforts in loop regions. As such, these applications are also candidates for LIS.

In the next section, we provide the details of our approach to reduce the monitoring overhead in loop regions. We do so by applying a scaling factor to control the sampling rate of monitoring in loop regions. We then describe our implementation of the LIS in Jikes RVM which is designed to integrate with Pacer and SOS.

3.3 Approach

As stated previously, state of the art dynamic race detectors [BCM10, MMN09, FF09] are still too expensive to be used in deployed systems. As such, sampling has been used on these detectors as a way to reduce overhead. Recent work by Li et al. applies stationary analysis to reduce overhead by an average of 35% [LSaD11]. However, as shown in Section 3.2, there might be an opportunity to significantly reduce the race detection overhead by customizing the sampling of monitoring operations in loop regions. In this section, we describe our approach to perform sampling to reduce race detection cost. We refer to our proposed loop optimization technique as *Loop Iteration Sampling*.

The main idea is to differentiate the monitoring efforts between operations that reside in loop regions and those that do not. The goal is to reduce the overhead for monitoring

repetitive operations in loops. As shown before, current approaches can spend up to 80% of their monitoring effort on operations in loops. However, that effort does not always yield increased race detection. Next, we describe our approach to differentiate monitoring in loop and non-loop regions.

3.3.1 Loop Iteration Sampling Algorithms

To understand how loops cause redundant data access monitoring, consider the simple example in Figure 3.2. The program has a class $Example$ containing a static field d. Two instances of threads $loopRacer$ and $regularRacer$ concurrently access field $Example.d$. There are two distinct data races between write operation at line 22 and read operations at line 12 and 14, denoted as $race1$ (race between lines 12 and 22) and $race2$ (races between lines 14 and 22), respectively.

Race detectors such as *FastTrack* or *Pacer* (with 100% sampling rate) would monitor all data access operations and detect one instance of $race1$ and one million instances of $race2$. Clearly, there is no need to identify the source of a data race (e.g. $race2$) one million times.

When we configure Pacer to operate at a small sampling rate (e.g., 3%) it samples operations to monitor uniformly over the execution time. However, the 3% sampling rate, which can cause up to 86% overhead [BCM10], is very low. Consequently, there is a good chance that $Pacer$, operating at that sampling rate, would miss $race1$. On the other hand, because the statements that cause $race2$ are executed frequently, $race2$ has a much higher probability than $race1$ of being detected.

The example reveals a glaring inefficiency of techniques that uniformly sample monitoring operations over time. To overcome this inefficiency, we propose a Loop Iteration Sampling technique that samples the number of operations to be monitored inside loop regions. The goal of the proposed technique is to reduce overhead without degrading de-

```
 1 public class Example {
 2   static Datum d = new Datum();
 3
 4   public static void main(String[] args){
 5     new LoopRacer().start();
 6     new RegularRacer().start();
 7   }
 8 }
 9
10 class loopRacer {
11   public void run() {
12     Datum d1 = Example.d; //race1
13     for (int i=0; i<1000000; i++){
14       Datum d2 = Example.d; //race2
15     }
16   }
17 }
18
19 class regularRacer {
20   public void run() {
21     while (true) {
22       Example.d = new Datum(); //race1, race2
23     }
24   }
25 }
```

Figure 3.2: Data Race Example

tection effectiveness. Next, we describe LIS.

If a heap access bytecode is currently executing in a loop, which is denoted by *op*, our algorithm records the current number of loop iterations that have bee executed, denoted as S. Let the sampling rate of operations outside of loops be α. The sampling rate of *op* is calculated as follows:

$$SamplingRate(op) = \frac{\alpha}{10^{\lceil \lg(S) \rceil}}$$

To illustrate the potential benefits of this scheme, reconsider the example from Figure 3.2. The read operation at line 12 has nearly one million times higher probability to be sampled than the read operation in line 14. As explained below, with LIS the probabilities

that $race1$ and $race2$ would be detected become much closer – since the sampling rate of line 14 within the loop converges to $\alpha/10^6$. This means that most loop iterations are not monitored, resulting in overhead reduction. As for write operation at line 22, the sampling rate will eventually decrease to a very low level. For instance, the sampling rate will be $0.000001 * \alpha$ after one million iterations, which means that the race detection overhead within that loop would approach the minimum. This characteristic can be quite beneficial to long running server applications as the detection overhead continues to decrease as the program continues to run.

Next we explain Algorithm 5 and Algorithm 6. To do so, we need to provide some information about the sampling mechanism in Pacer, which we refer to as "global" sampling mechanism because it controls sampling for the whole program. Currently, the global mechanism is regulated by garbage collection invocations. For example, if the sampling rate is 10%, monitoring takes place every 10th garbage collection invocation. In this approach, there is a centralized Boolean variable, *samplingStatus*, that records whether we should monitor operations at this time. In this example, the variable is only set to 1 every 10th garbage collection cycle.

Our LIS approach extends the mechanism used in Pacer to control monitoring in loops. Because, we adjust the sampling rates based on the number of iterations, we create a centralized Boolean array to record whether loops should be monitored at this time. We refer to this Boolean array as *loopSampling*. To illustrate the relationship between LIS and the global sampling mechanism in Pacer, we provide a simple example. Let us assume that the sampling rate is 10% and the garbage collector has just been invoked. However, this is the 9th invocation and not the 10th invocation so the sampling mechanism is disabled. This means that *samplingStatus* and every element in *loopSampling* are set to false. This also means that from this point on to the next garbage collection invocation, there will not be any race detection monitoring.

The way we organize *loopSampling* is that each element stores information for a set of loops whose iteration counts have the same logarithm when truncated to an integer. For example, if we have two loop regions, both have iterated through 8 times, these two loops belong to the same group, which is 10 iterations or less. Any memory read and write operation in these two loops, would check whether monitoring is enabled by accessing a *loopSampling[1]*. Variable *range* in Algorithm 5 computes the log-based index into the *loopSampling* array.

Continuing the example, when the garbage collector is invoked again it is the 10th invocation so temporal sampling is on. The mechanism sets *samplingStatus* to true and then applies Algorithm 6 to set each element of *loopSampling*. The way we set each element to true or false is based on the number of sampled iterations (*sampledOps*) for each group and the total number of iterations (*totalOps*). Note that *sampleOps* and *totalOps* are integer arrays with the same dimension as *samplingStatus*. The values of *totalOps* and *sampledOps* are updated in Algorithm 5 and used in Algorithm 6 to calculate the current sampling rate (*currentRate* in Algorithm 6) and the rate that should be sampled within a particular group of loops (*targetRate* in Algorithm 6). As shown in Algorithm 6, we periodically adjust the sampling rate by multiplying a weight that is obtained by dividing the target sampling rate with actual sampling rate.

We illustrate the process to adjust the sampling rate in Algorithm 6 below. First, assume the target sampling rate for loops with fewer than 10 iterations is 10%. Assume also that there is an data access operation within a loop, which has executed eight iterations. The operation will check *loopSampling[1]* to see if monitoring should be performed. Second, assume the sampling rate for code outside loop is 100%, then *loopSampling[1]* = .1 * (1 / .1). Third, assume that out of the eight iterations (*totalOps[1]==8*), two iterations have been sampled (*sampledOps[1]==2*). This means the actual sampling rate is 25% (2/8). Because we set the target sampling rate to 10% in the previous step, the current rate of 25%

is too high. Consequently, we reduce the target sampling rate to 4% $(.1 * (.1/.25))$ in order drive the current sampling rate towards the target of 10%.

```
{Input: data access operation, op}

if SamplingStatus(op) == true then
    checkRace(op, currentThread)
    updateVectorClock(op, currentThread)
```

Algorithm 4: Data Race Monitoring

```
{Input: data access operation, op}
{Output: whether monitor or not for this execution}

loopSampling[0..n] = {true, false};
sampledOps[0..n]
totalOps[0..n]

if op is within loops then
    range ← ⌈lg(op.iterations)⌉
    totalOps[i] ++
    if loopSampling[range] then
        sampledOps[i] ++
        return true
else
    return false
```

Algorithm 5: Sampling Status

3.3.2 Implementation

We incorporate LIS into two existing race detectors that have been implemented in Jikes RVM, $Pacer$ and SOS (see Chapter 2 for more information). $Pacer$ is based on the $FastTrack$ but it incorporates a sampling mechanism that can be used to control overhead. SOS is then based on Pacer so it also has sampling feature. Next,we describe some key components in our implementation.

Loop identification. The first major challenge to deploy LIS is to dynamically identify loops. The dynamic compilers in Jikes RVM identifies basic blocks inside loops as part

{Input: α is the sampling rate for code outside loops}
{Output: $loopSampling[0..n]$ containing sampling decisions for different sampling rate}
for $i = 1 \rightarrow n$ **do**
  $targetRate[i] \leftarrow \frac{\alpha}{10^i}$
  $result = random(0, 1)$ {Generate a random float number within [0,1]}
  if $sampledOps[i] \neq 0$ **then**
    $currentRate = \frac{sampledOps[i]}{totalOps[i]}$
    $targetRate[i] = \frac{targetRate[i]^2}{currentRate}$
  if $result \leq targertRate[i]$ **then**
    $loopSampling[i] =$ **true**
  else
    $loopSampling[i] =$ **false**

Algorithm 6: Sampling Decision

of the compilation process. We exploit this information and perform instrumentation of intermediate representations to record loop iterations. For data race detection, we only instrument heap access bytecodes such as $getfield$, $putfield$, $getstatic$, $putstatic$, $aaload$, $aastore$, etc. However, it is quite complex to deal with a method call in loop. This is because a method could also be called by a different calling context. This problem becomes even more complex for method calls in nested loops.

To deal with such issues, we create a runtime loop checker. The checker sets a counter for each thread to record the loop depth of method calls. Every time the program enters a method call in loops, the counter is increased. When the program exits from a method call in a loop or loops, the counter is decreased. As such, when the counter value is non-zero, the program is executing in loop(s), shown in Figure 3.3.

```
1 currentThread.loopDepth++:
2 //method call within a loop
3 foo();
4 getCurrentThread().loopDepth--:
```

Figure 3.3: Instrumentation For Method Call

Sampling decision. As mentioned above, we count iterations for each bytecode, so it is straightforward to apply the algorithm in Section 3.3.1 to dynamically calculate sampling rate for bytecodes inside loops.

Further optimization. There are two compilers in Jikes RVM: *baseline* and *optimizing* compilers. Every method is initially compiled by the baseline compiler prior to its first execution. Later on, if the method becomes “hot”, the optimizing compiler recompiles the method with more optimizations. As reported by $LiteRace$[MMN09], data races tend to occur in ”cold” region. The insight motivates us to further enhance data race coverage by disabling sampling in $baseline$ compiler. With this approach, more monitoring is done inside the baseline compiler.

3.4 Performance Evaluation

In this section, we evaluate the performance of LIS. Our experimental methodology is to apply LIS to enhance two existing data race detection techniques: $Pacer$ and SOS. Note that SOS also uses the sampling mechanism of $Pacer$. As such, the mechanism to control sampling in $Pacer$, which we refer to as *global* sampling is also used to control sampling rate in SOS. We then create two new detectors: $Pacer + LIS$, or $Pacer_{LIS}$, and $SOS + LIS$, or SOS_{LIS}. We then study both the efficiency of the effectiveness of the two new detectors relative to $Pacer$ and SOS without further optimization under two operating scenarios: no sampling and controlled overheads through sampling.

The way we configure $Pacer_{LIS}$ and SOS_{LIS} is as follows. For the case that sampling is not used, we set the global sampling rate to 100%. Setting it to 100%, in effect, disables the periodic sampling component of Pacer. Next, we enable the sampling component of LIS. The sampling rate is only applied to loop regions to control the monitoring efforts. Initially, the LIS rate is the same as global sampling rate (100% in this case). However,

as the loop continues to run, the LIS rate is periodically recalculated and decreased. In this example, our detectors monitor all operations in non-loop regions and then only some operations as controlled by the LIS rate in loop regions.

Second, we assess the potential for LIS to achieve low overhead race detection at a level that might be considered for use in deployed settings. To do this we fix the overhead of each analysis and then study the race detection effectiveness. In all of the analyses we work with the mechanism for controlling overhead is through the reduction of the global sampling rate, however, sampling rate and and overhead are not directly proportional. Complicating matters further the relationship between sampling rate and overhead varies with each of the four analyses we study. For example, Bond et al. have shown that Pacer is able to detect races while maintaining 86% runtime overhead and to achieve this requires a sampling rate of 3%. We explore four fixed sampling overhead rates 50%, 70%, 85% and 100% that allow us to explore the *least expensive*, and least effective, instances of the analyses. For each analysis, across each program, we computed the sampling rate required to achieve those overheads.

In terms of experimental methodology, we executed each of the 6 benchmarks 10 times using each of the race detectors. The data we report is computed in terms of averages across those 10 runs.

3.4.1 Efficiency of SOS_{LIS}

Figure 3.1 shows that the LIS has the potential to reduce the number of operations that must be monitored – specifically the operations occurring within loops. The reduced monitoring effort may result in execution overhead reductions relative to the detectors that do not employ LIS as reported in Table 3.1.

On average, the overhead of SOS_{LIS} is 48.5% of that of Pacer (see the last black bar

in Figure 3.4). It is also interesting that in *sunflow* the overhead of SOS is 81% of Pacer's but the overhead of SOS_{LIS} is only 23% of Pacer's. In all, the overheads of SOS_{LIS} range from 23% to 91% of Pacer's. As shown in Figure 3.1, avrora does not have many operations in loops, and therefore, does not benefit from applying LIS. This is the main reason why LIS is only able to reduce the overhead of avrora by 9%.

In addition, we also see more uniform overheads with SOS_{LIS}. As an example, when sunflow is executed on SOS, the slowdown is 23.9 times. This is an improvement from the slowdown experienced by Pacer but it is still quite large. Across all applications, the slowdown factors of SOS range from 4.2 to 23.9 times. In contrast, the slowdown factors of SOS_{LIS} range from 3.5 to 7.1 times. This implies that SOS_{LIS} yields more consistent and predictable performance.

When only LIS is used with Pacer, its average overhead is 65% of Pacer's. On the other hand, when SOS is used with Pacer, its average overhead is 72% of Pacer's. This indicates that if only one optimization technique is to be used, LIS is more effective in reducing overhead than SOS.

	Slowdown Factor (×)			
Benchmark	$Pacer$	SOS	$Pacer_{LIS}$	SOS_{LIS}
avrora	5.1	4.2	5.2	4.7
eclipse	21.0	16.6	7.6	7.1
hsqldb	12.2	7.3	7.8	3.5
pseudojbb	7.7	5.1	8.1	5.2
sunflow	29.2	23.9	8.4	6.7
xalan	11.6	7.1	6.3	5.5
average	14.5	10.7	7.2	5.4

Table 3.1: Comparing the slowdown factor of the implementation of $Pacer$, SOS, $Pacer_{LIS}$, and SOS_{LIS} over the execution time of RVM with no race detection. For example, *avrora* running on $Pacer$ is 5.1 times slower than running on RVM with no race detection.

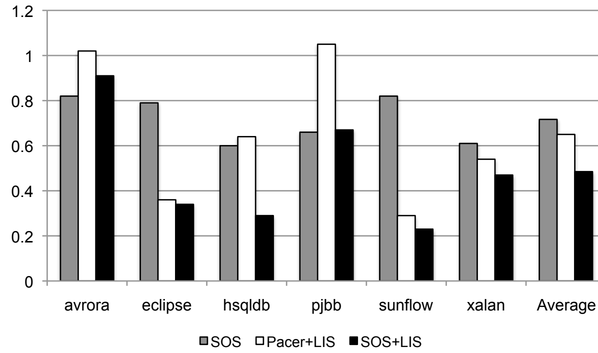

Figure 3.4: Overheads of SOS, $Pacer_{LIS}$ and SOS_{LIS} normalized against Pacer's.

3.4.2 Effectiveness of SOS_{LIS}

Typically, when we apply optimization techniques such as SOS or LIS, one possible negative impact is a reduction the race detection effectiveness. That is, the system may miss some races. In SOS, the first instance of race that involve write after read is not detected. In LIS, because we monitor fewer operations in loops, it can also miss races. Table 3.2 reports the race detection effectiveness.

Benchmark	$Pacer$	SOS	$Pacer_{LIS}$	SOS_{LIS}
avrora	12	12	10	10
eclipse	**27**	**23**	**20**	**18**
hsqldb	23	23	23	22
pseudojbb	22	22	21	21
sunflow	13	13	13	13
xalan	19	18	19	18

Table 3.2: Comparing the number of detected unique races.

As shown in the table, SOS_{LIS} can detect all unique races in one application, all but one unique races in three applications (hsqldb, pseudojbb, and xalan), which accounts for missing at most 5% of all detected races (xalan). In avrora, it misses two races or 16.67%. It misses nine races or 33% in eclipse. By analyzing the runtime data (which is not shown),

we discover a very unique situation with eclipse. In other applications, the percentage of reduction in the number of detected dynamic races range from 0% (in xalan) to 19% (in pseudojbb) when SOS is used. However, we see a reduction is 60% in eclipse. (With Pacer, there are 16,191 detected dynamic races in eclipse. After SOS is applied, the number of detected dynamic races reduces to 6,444, which is a 60% reduction).

Further investigation reveals that eclipse has a few races that cannot be detected by SOS due to the its limitations. This explains the 60% reduction in the number of detected dynamic races. With SOS_{LIS}, the number of detected dynamic races is further reduced by another 99%, when compared to SOS (As reported earlier, there are 6,444 instances of detected dynamic races. After LIS is applied to SOS, the number reduces to only 54). The result of the analysis indicates that (1) eclipse has many unique races that cannot be detected by SOS; and (2) in addition, it also has many unique races that reside in loop regions but they are in code locations that are not frequently monitored. As such, when we reduce the monitoring efforts in loops, these races go undetected. Despite all these missed races, SOS_{LIS} still detects 66% of unique races detected by $Pacer$ while yielding only 35% of Pacer's overhead.

In summary, we can achieve the greatest reduction in most applications when both SOS and LIS are used together. We also notice that the reduction percentage for each benchmark correlates well with the number of operations that have been eliminated by loop sampling; that is, the overhead reduction percentages are similar to the operation reduction percentages. In terms of race detection effectiveness, SOS_{LIS} can detect most races detectable by $Pacer$. As such, we conclude that we can achieve significant overhead reduction without sacrificing significant race detection effectiveness.

3.4.3 Effectiveness of SOS_{LIS} with Fixed Overhead

In this section, we report the result of our experiment to evaluate the suitability of using SOS_{LIS} in deployed environments. As explained above, our evaluation approach is to apply Pacer's sampling mechanism to SOS_{LIS} and observe the number of detected races for a particular overhead budget (i.e., 50%, 70%, 85%, and 100%). We compare the percentage of unique detected races for each benchmark, for each of the four detectors at each overhead budget. We report the results in Figure 3.5. To aid in understanding the data in these figures, we also report the average number of unique races detected by SOS_{LIS} as labels above the lines representing SOS_{LIS}.

It is worth keeping in mind that the overhead of the sampling mechanism in Pacer is around 30%; therefore, it is impossible to reduce monitoring overhead below that threshold. Because LIS also incurs additional overhead for mechanisms to support analysis. As such, we set our lowest budgeted overhead to be at 50%.

As shown in the figure, SOS_{LIS} can detect more races than any other approaches in five out of six benchmarks when the overhead is set to be below 100%. The only exception is avrora, in which LIS is not effective in reducing the number of operations. As such, LIS is also not effective in reducing the overall detection overhead. In fact, SOS is more effective at reducing the number of monitored operations in avrora. For *pseudojbb*, all races originated from one commonly used class. As such, all configurations can detect over 75% of the races with low sampling rates.

Another interesting benchmark is *sunflow*. Pacer and SOS cannot consistently detect races when the overhead is 70% and below. At 100%, SOS can detect races more consistently. Throughout the range of budgets, SOS_{LIS} can consistently detect races. The detected races range from 2 at 50% overhead budget to 8 at 100% overhead budget.

One key insight is that sunflow is by far the most expensive application for Pacer and

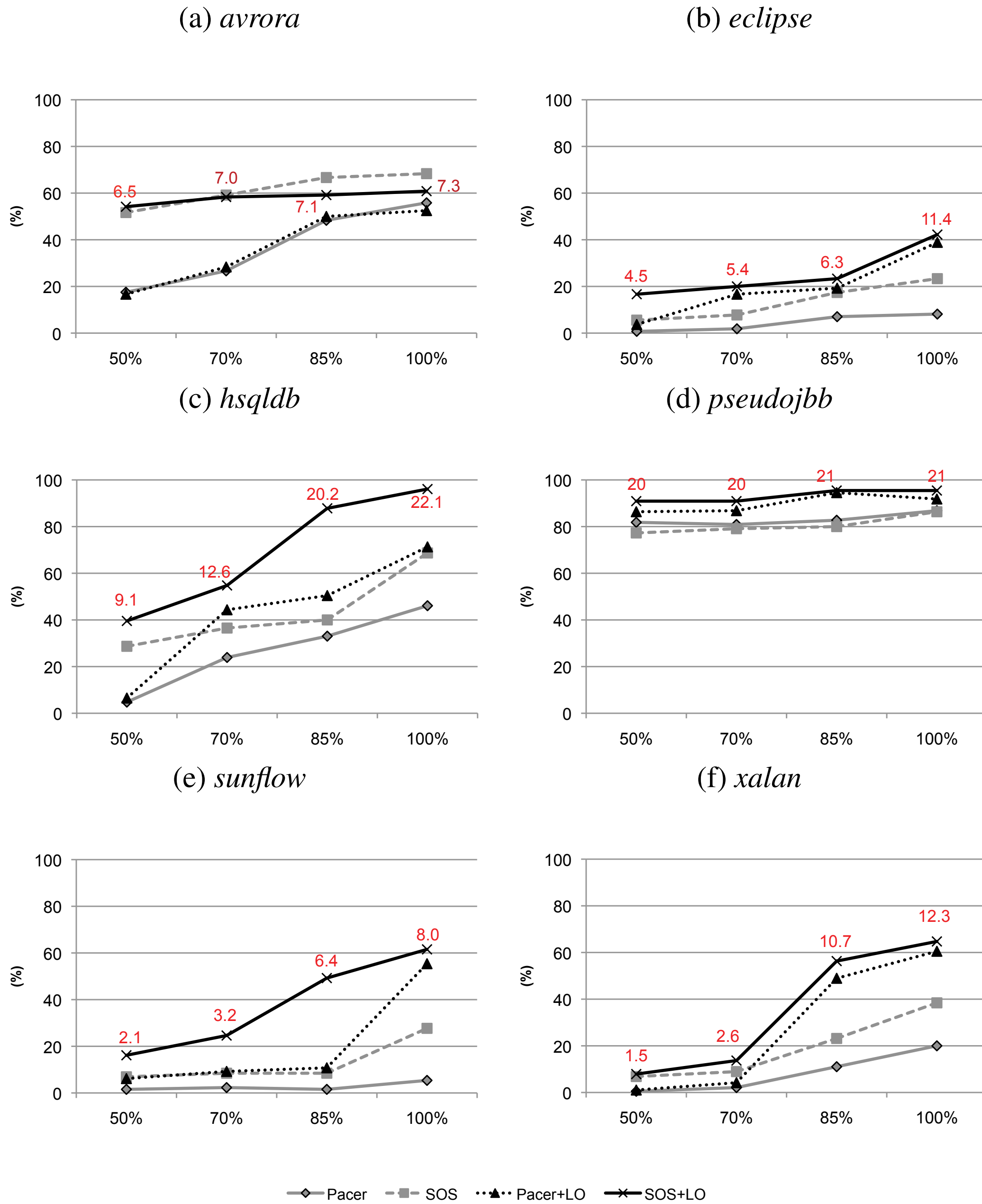

Figure 3.5: Comparing the raced detection effectiveness of detectors with loop-optimization with the ones without given low overhead budgets. For each application, we also report the number of detected unique races by SOS_{LIS} above the solid black line.

SOS to execute. It incurs 29 and 24 times slowdown, respectively when compared to RVM without race detection (see Table 3.1). As such, they can only sample at low rates to maintain 100% or less overhead. Low sampling rates mean that they would miss a lot of races. Furthermore, there is no consistency in the number of detected races across runs for $Pacer$; some runs would report no races while others report a few races. As such, the average number of detected races across ten run is only 0.7.

Benchmark	Sampling Rate (%) at an Overhead Budget of 50%	70%	85%	100%
$Pacer$	0.9	1.5	2.8	5.1
SOS	3.6	7.9	11	14.2
$Pacer_{LIS}$	5.4	9.8	15.3	18.2
SOS_{LIS}	10.1	12.3	21	31.6
Average	**5.00**	7.88	12.52	**17.28**

Table 3.3: Comparing the average sampling rate used by each approach to maintain the budgetted overhead.

The results also reveal that $Pacer$ can only use an average of 5% sampling rate to maintain less than 100% overhead (see Table 3.3). On the other hand, SOS_{LIS} can use up to 31% sampling rate to maintain the same overhead. This is because SOS_{LIS} incurs low detection overhead. This characteristic also makes the approach more effective at detecting races when the budget is very low. As shown in Figure 3.6, SOS_{LIS} can detect on average a factor of 10 more races than $Pacer$ when the overhead budget is 50%. As the overhead budget increases, the factor becomes smaller because the numbers of detected races begin to saturate; since there are a fixed number of races after some time all of the detectable races will be reported. At 100% overhead, SOS_{LIS} detects four times more races than *Pacer*. It is also worth noting that when we compare the numbers of detected races between SOS and SOS_{LIS} across different allowable overhead budgets, SOS_{LIS} consistently detect more

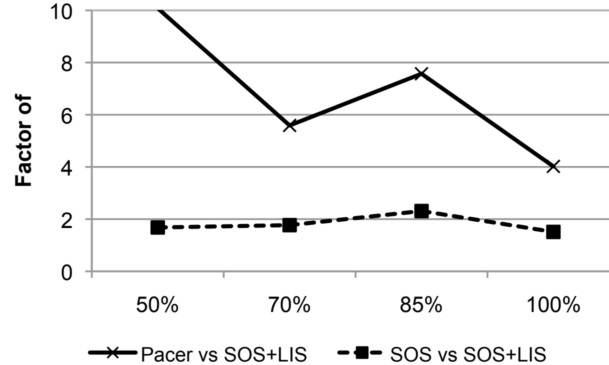

Figure 3.6: Comparing race detection effectiveness at low overhead between $Pacer$, SOS, and SOS_{LIS}.

races than SOS by a factor of 1.5 to 2.3.

In summary, these characteristics make SOS_{LIS} more effective at detection race when the overhead budget is small. They also open up an opportunity for the approach to be feasibly used to detect races in deployed systems.

3.5 Conclusion

In our preliminary study, we have shown that a large amount of race detection overhead has to do with the monitoring efforts in loop regions. Furthermore, when temporal sampling is used to control the detection overhead, a large portion of sampled operations lay within loops. As such, the sampling appears to be non-uniform with respect to the code regions, and data races that exist in loops are detected repeatedly.

Repeated detection of the same race causes unnecessary overhead and does not increase the detection effectiveness. In this work, we propose an optimization technique called Loop Iteration Sampling, which reduces the monitoring effort when a program is executing in loops. The non-loop regions are unaffected. We have implemented our proposed optimization on two race detectors available in Jikes RVM, Pacer and SOS. We then evaluate

the effectiveness of LIS using six Java multithreaded benchmarks that contain data races. The result of our evaluation shows that for programs that perform significant iterative computation, the optimization can reduce race monitoring overhead by as much as 80%. On average, the approach reduces the overhead of race detection by 53% when we compare to Pacer and 30% when compare to SOS. We also find that the approach provides much more consistence performance over both Pacer and SOS.

Next, we evaluated the effectiveness and efficiency of LIS to detect races under tight overhead budgets. LIS is on average a factor of 7 more effective than Pacer and a factor of 2 more effective than SOS. This is because LIS is more efficient than SOS or Pacer alone so given an overhead budget, SOS with LIS can set the temporal sampling rate to 31% while Pacer and SOS can only set the rate to 5.1% and 14.2%, respectively.

Chapter 4

RaceDr: A Race Healing System

In this chapter, we present *RaceDr*, which is a system that builds on state-of-the-art race detection techniques and couples them with on-the-fly generation of race repairs. RaceDr is implemented for Java programs within the JVM making the technique transparent and automatic. We evaluate RaceDr on a collection of multi-threaded Java applications and find that it incurs a modest increase in overhead relative to existing precise race detectors. Moreover, the fixes RaceDr produces are deadlock-free and can be applied when RaceDr is disabled – rendering the benefits of the fixes with low overhead (15%).

4.1 Introduction

Self-healing systems [Har10, KLT+07] have been introduced to enhance the dependability of these systems. The basic idea is to create ways for systems to (i) dynamically detect problems and identify the sources of those problems, (ii) reason about the resolution of those problems, and then (iii) apply software fixes on-the-fly as the system continues to operate. To date, several research efforts have built adaptive systems using different approaches including dynamic detection of faults and regeneration of code with fixes,

e.g., [ZM04, JSZ+11]. However, only a few efforts have targeted self-healing of *concurrency faults*, such as data races and atomicity violations, in deployed software [KLT+07, Zen03].

One aspect of data races that makes them attractive as a target for self-healing is that, in general, *data races occur frequently in concurrent software but often their effects on system behavior are too subtle to be noticed and would not be classified as a failure* [KP04]. We seek to exploit this observation by developing *RaceDr*, a self-healing system for Java applications that is capable of *detecting both low-level and high-level data races early during system execution and modifying the software to avoid future occurrences of races that might cause serious system failures.* Our proposed system is more complete, efficient, and effective than existing healing systems for races and atomicity violations [KLT+07, JSZ+11, WCY10] because:

1. RaceDr can detect data races precisely and efficiently. Our proposed system utilizes state-of-the-art sampling-based and highly optimized race detection algorithms that can be tuned to work well in different application scenarios (e.g., during testing or under deployment).

2. RaceDr can repair races on-the-fly. The repair process happens in real-time and is automatic and transparent to the software developer. Existing approaches often require developers to use extra tools, such as bytecode instrumentors. Developers also need to stop the application, analyze execution reports to generate repairs, apply the repairs, and then relaunch the application. As such, those repair processes are invasive and cumbersome.

3. RaceDr does not introduce new faults such as deadlocks in the application. Unlike previous work that relies on timeout to break deadlocks caused by the generated

Prog.	Source	#Thr [A]	#Class [B]	#Meth. [C]	#Dyn. Races [D]	#Race Src. [E]
avrora	DaCapo 2009	27	397	1802	493138	32
eclipse	DaCapo 2006	16	1230	9580	514376	46
hsqldb	DaCapo 2006	402	113	1012	511	35
pseudojbb	SPECjbb 2005	17	261	952	$6.95e^7$	148
sunflow	DaCapo 2009	5	121	986	447	24
xalan	DaCapo 2006	9	360	2203	81	26
Total Races					$7.05e^7$	311

Table 4.1: Basic description of each benchmark.

repairs, our system analyzes whether repairs can cause deadlock. If so, then the system breaks the deadlock and then removes the repairs.

In this chapter, we study the application of RaceDr to a collection of six multi-threaded programs. Their characteristics are shown in Table 4.1. Columns A-C list the number of threads, loaded classes, and compiled methods. Column's D and E show, respectively, the number of dynamic race reports generated for these applications using state-of-the-art precise race detection algorithms and the number of unique race *source*s, i.e., pairs of read and write statements in the programs.

In addition to presenting RaceDr, in this work, we explore two scenarios for its use : (a) during testing and debugging and (b) during deployment.

RaceDr can serve as a drop-in replacement for a dynamic race or atomicity detector during testing and debugging. Using a typical race detection system can result in a large number of dynamic race reports, up to millions of such reports as shown in column D in the table. Most of this information is redundant, since a single race source may be reported many times. RaceDr includes a dynamic race repair component, but it incurs

modest additional overhead relative to the best precise detectors. RaceDr goes further than just detecting races by fixing races when they are first detected, thereby suppressing much of the dynamic race report information without sacrificing reports of the unique race sources – the root causes of the fault. Moreover, RaceDr can issue, as part of its report, a patch which encodes the race fixes. In this way, developers can see both the report of the races and the fixes that were generated. These fixes can be applied for use in deployed system operation, if the developer approves of them, or they can serve as useful guidance in helping the developer produce their own fixes.

RaceDr is implemented within a JVM and thus, it can also be used during deployment to enhance dependability of concurrent software. We use a highly optimized sampling-based race detection technique, which allows us to control overhead while still detecting significant numbers of races. Furthermore, as RaceDr performs race repair an application might reach a state in which very few new races are being detected. In each of the applications we studied, we saw that the number of unique race sources detected drops off very quickly with the number of program runs (see Figure 4.1). This phenomenon allows a user to define a threshold below which RaceDr is disabled. At that point the repaired applications can continue to run with just the additional overhead of locking that results from the race fixes – in the applications we studied this overhead is 15% on average.

The rest of this chapter is organized as follows. Section 4.2 formalizes the semantics of concurrent program traces. Section 4.3 presents the atomicity violation detection algorithm and correctness proof. Section 4.4 provides an overview of the current state of the art in race detection and race healing systems, and presents the key features of RaceDr that distinguish it. Section 4.5 describes our implementation of RaceDr inside a Java Virtual Machine (JVM). Section 4.6 reports the result of our experimental evaluation of RaceDr in both the testing/debugging and deployed scenarios. We conclude with future research directions in Section 4.7.

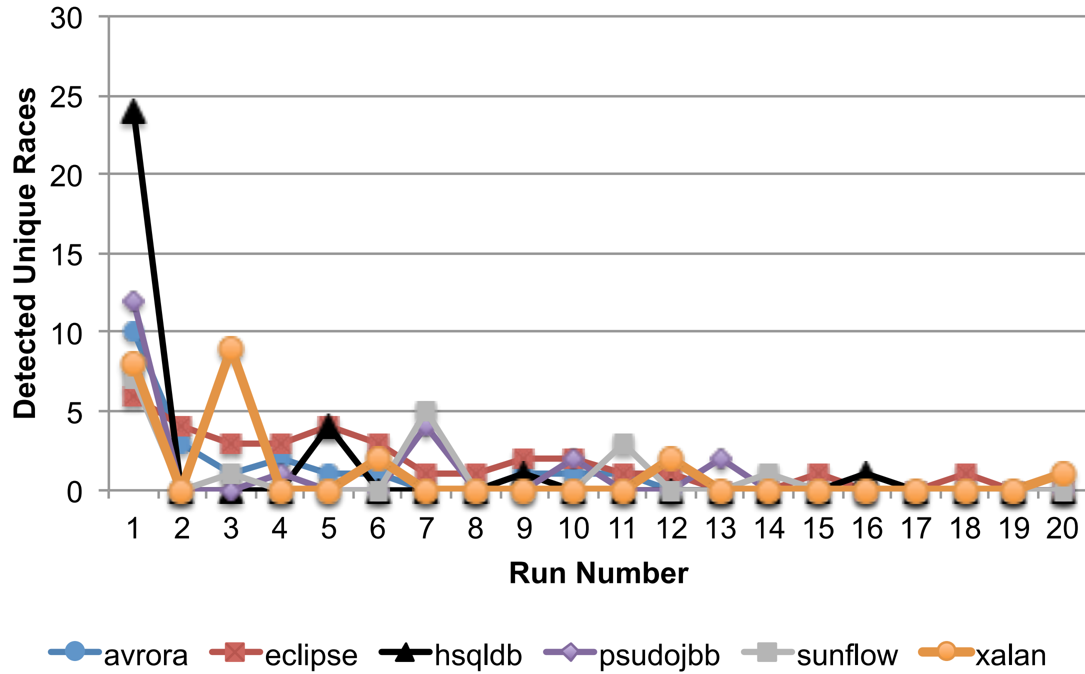

Figure 4.1: Unique races detected across 20 program runs.

4.2 Preliminaries

In this section, we formalize the semantics of multi-threaded programs.

4.2.1 Semantics of Multithreaded Programs

A multithreaded program consists of a set of concurrently executing threads denoted as $t \in Tid$, which conduct operations to access variables $x \in Var$ and locks $m \in Lock$. Operations that a thread can perform are listed as follows:

- $rd(t, x)$: thread t reads from variable x.
- $wr(t, x)$: thread t writes to variable x.
- $acq(t, m)$: thread t acquires lock m.

- $rel(t, m)$: thread t release lock m.
- $fork(t, u)$: thread t forks a new thread, u.
- $join(t, u)$: thread t blocks until thread u terminates.
- $enter(t)$: thread t enters an atomic section.
- $exit(t)$: thread t exit from an atomic section.

The rd and wr are data access operations. The acq, rel, $fork$ and $join$ are synchronization operations. The $enter$ and $exit$ are notions that represent a thread enter or leave a predefined atomic method. A $trace\ \alpha$ captures an execution of a program by listing a sequence of operations performed by the threads. The *happens-before* relation, denoted as $\rightarrow_{hb}$, is the smallest transitively-closed closure over operations in a trace[Lam78]. Operation a happens before operation b in trace α, denoted as $a \rightarrow_{\alpha} b$, if a occurs before b and one of the following conditions holds:

- a and b are performed by the same thread.
- a is a release of lock m and b is an acquire of lock m.
- a is a fork of a new thread u by a thread t, $t \neq u$, and u is the thread that performs b.
- a is performed by thread u, and b is a join that blocks a thread t, $t \neq u$, until u terminates.

We define *logic order* between two operations a and b, denoted as $LogicOrder(a, b)$ in trace α as follows:

- a *happens before* b.
- a *happens after* b if and only if b *happens before* a.

- Otherwise, a and b are *concurrent*.

Operations a and b are *conflict* if they access the same variable and at least one of them is a write. A trace has a race condition if it has two operations a and b that satisfy the following two conditions:

- a and b are concurrent.
- a and b are conflict.

The definition of *happens-before relation* involves only two operations. Next, we lift the definition to specify the logic order between a method and an operation.

4.2.2 Vector Clock Based Race Detection

Vector clocks(VC) are an widely used race detection data structure, which precisely tracks the happens-before relationship between operations. VC based race detection algorithms perform dynamic analysis on all synchronization, read, and write operations. They detect concurrent variable accesses, and if one is a write, they report a data race. Typically,VC base race detection algorithm stores a vector clock for each synchronization object, each variable read, and each variable write. A vector clock is indexed by thread identifier: C[1::n]. For each synchronization object o, the analysis maintains a vector clock Co that maps every thread t to a clock value c. VC updates as follows:

- For $acq(t, m)$: $C_t \leftarrow C_t \bigcup C_m$.
- For $rel(t, m)$: $C_m \leftarrow C_t$, $C_t[t] \leftarrow C_t[t] + 1$.

For example, if thread t acquires lock m, the join of t and m's vector clocks into t's vector clock by updating each element $C_t[i]$ to $max(C_t[i]; C_m[i])$. When a thread t releases

a lock m, the analysis copies the contents of ts vector clock to m's VC. It then increments the t entry in t's VC.

4.3 Atomicity Violation Detection

In Chapter 2-3, we explored approaches to efficiently detect data races caused by unexpected thread interference. However the absence of data races is not sufficient to ensure the correctness of a program. Figure 4.2 shows a known atomicity violation in Java $StringBuffer$ class. Between method calls to $length()$ and $getChars()$, another thread can acquire the lock and modify field $length$, therefore, method $append$ can encounter a fault due to accessing a pale value of field $length$. This example illustrates that simply synchronizing methods $length()$ and $getChars()$ cannot ensure the correctness although it does guarantee data race freedom.

```
1 public final class StringBuffer{
2     ...
3     int length;
4     public synchronized StringBuffer append(StringBuffer sb) {
5         int len = sb.length();
6         //Another thread could call method setLength()
7         //to change the value of  field length.
8         sb.getChars(0, len, value, count);
9         ...
10    }
11
12    public synchronized int length() {...}
13    public synchronized void setLength(...) {...}
14    public synchronized void getChars(...) {...}
15    ...
16 }
```

Figure 4.2: Example of Atomicity Violation

The fault in above example is defined as *atomicity violation*, namely high level data races[AHB03]. Atomicity is a common property for multi-threaded programs. It places stronger restrictions on thread interleaving than race conditions do: A code block (usually a method) is atomic if for any arbitrarily interleaved program execution, there is a serial execution of the atomic code block, which means no thread interleaving, with the equivalent overall effect.

Besides the fault shown in Figure 4.2, recent research has found that some subtle defects due to atomicity violations even in well tested systems, even in well-tested libraries [FQ03]. Havelund reports finding similar errors in NASAs Remote Agent spacecraft controller [AHB03], and Burrows and Leino [BL04] and von Praun and Gross [vPG03]

have detected the same type of defects in Java applications.

Atomicity provides a strong guarantee on noninterference between threads. This guarantee can safely reduce complicated problem of reasoning or verify about behavior of multi-threaded program to a less challenging problem of reasoning about behavior of sequential programs. The reduction substantially facilitates standard techniques such as program analysis, software testing and verification.

In summary, atomicity is a fundamental and widely applicable correctness property of multi-threaded programs. It helps to detect more program faults caused by undesirable thread interference that cannot be found by race detectors. In addition, fixing atomicity violations is more challenging than fixing data races since only synchronizing shared data accesses is not enough.

In this Section, we propose a solution to detect and fix atomicity violations in Java programs. We extend SOS race detector to cover high level data races as well. We propose an approach to generate and deploy repairs to the violations as soon as detected. Meanwhile, we also ensure our repair will not introduce new faults like deadlocks.

4.3.1 Atomicity Violation Detection Algorithm

Atomicity violation occurs if two data operations in an execution of an atomic method have different logic order with an operation executed by another thread when accessing the same data and one of these operations is write. For instance, in Figure 4.2, operation *length()* at line 5 happens before *setLength()* at line6 while *setLength()* happens before *getChars(...)*, therefore, the atomicity of method *append()* is violated. We formalize the definition of atomicity violation as follows.

Definition Atomicity Violation. Suppose t and $t\prime$ are two threads, x is an variable, $op_1(t, x)$ and $op_2(t, x)$ are two data operations and $op_1(t, x)$ happens before $op_2(t, x)$, $op_3(t\prime, x)$ is

another data operation, $op_1(t, x)$ and $op_2(t, x)$ belongs to method m, method m has an atomicity violation caused by $op_1(t, x)$, $op_2(t, x)$ and $op_3(t\prime, x)$ if one of the following conditions holds:

- $op_1(t, x)$ and $op_3(t\prime, x)$ are concurrent.
- $op_2(t, x)$ and $op_3(t\prime, x)$ are concurrent.
- $op_1(t, x)$ happens before $op_3(t\prime, x)$ and $op_3(t\prime, x)$ happens before $op_2(t, x)$.

There are two classes of atomicity violations based on the definition. One involves data races while the other is data race free. If there is a data race between two operations and one of operation is in an atomic method, then the method must have an atomicity violation. Furthermore, if there is no data races, a program can still has atomicity violations as showed in Figure 4.2.

We extend SOS data race detection algorithm to capture atomicity violations. For atomicity violations involving data races, SOS can directly detect them; for those without data races, we reduce them to data races by changing the instrumentation rules in SOS. The intuition behind this instrumentation change is to ignore lock acquire/release operations if this lock has been released in the same atomic method. For instance, the lock acquire and release operations at line 8 in Figure 4.2 won't be reflected in VC in order to reduce the atomicity violation in method $append(...)$ to a data race between methods $setLength(...)$ and $getChar(...)$.

4.3.2 Proof of Correctness

Next, we prove the correctness of above atomicity violation detection algorithm.

Lemma 4.3.1. *Suppose* $acq(t, m) \rightarrow_{hb} acq(t\prime, m)$ *in trace* α, $rel(t, m)$ *is the operation that releases the lock acquired by operation* $acq(t, m)$, $rel(t\prime, m)$ *is the operation that releases*

```
enter(t):
t.is_Atomic++;

exit(t):
t.is_Atomic- -;
if (t.is_Atomic == 0) then
   t.released_locks.clear();

acq(t, m):
if ((is_Atomic == 0) || (!t.released_locks.contains(m))) then
   update VC;

rel(t, m):
if ((is_Atomic == 0) || (!t.released_locks.contains(m))) then
   update VC;
   t.released_locks.add(m);
```

Algorithm 7: Instrumentation for reducing atomicity violations to data races

```
// Operation op1 belongs to method m1.
// Operation op2 belongs to method m2.

if (op1 races with op2) then
   if (m1 is atomic) then
      m1 has an atomicity violation.
   if (m2 is atomic) then
      m2 has an atomicity violation.
```

Algorithm 8: Atomicity violation detection

the lock acquired by operation $acq(t\prime, m)$*,* δ = $\{$*trace between* $acq(t, m)$ *and* $rel(t, m)\}$*,* $\delta\prime$ = $\{$*trace between* $acq(t\prime, m)$ *and* $rel(t\prime, m)\}$ *then* $\exists$ *trace* β*,* $acq(t\prime, m) \rightarrow_{hb} acq(t, m)$ *in trace* β*, and* α *is identical to* β *except operations in* δ *and* $\delta\prime$*.*

Proof. This lemma is valid since locking can only form a partial order of operations. □

We use the example in Figure 4.3 to illustrate the intuitive meaning of Lemma 4.3.1. A program generates trace α, trace δ happens before trace $\delta\prime$ because thread t get lock m prior to $t\prime$. Generally, the same program can generate trace β, in which trace δ exchanges positions with trace $\delta\prime$ and the rest of δ and $\delta\prime$ are the same. The reason is that locks

in modern programming languages decide a partial order for the execution of two code regions which share the same lock. In other words, the execution order of the code regions protected by the same is reversible given no specific semantics constraints.

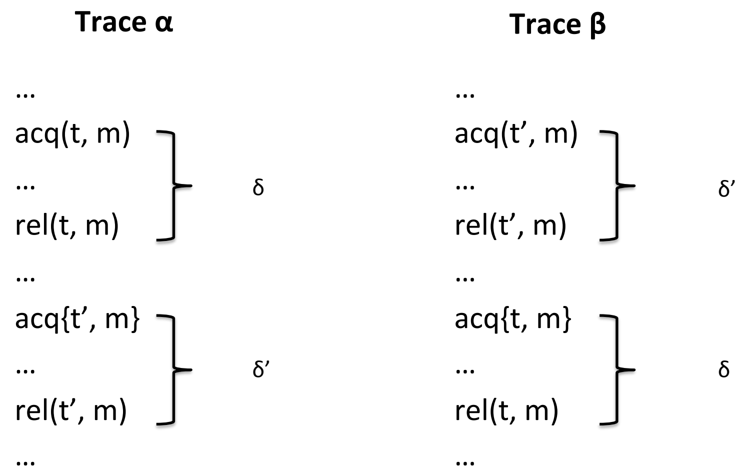

Figure 4.3: Partial order decided by locks.

Theorem 4.3.2. *If an atomic method m involves a race reported by the atomicity violation algorithm, method m has an atomicity violation.*

Proof. There are two cases:

1. Race related atomicity violations

   Suppose there is a data race involving two operations op_1, op_2 and op_1 belongs to atomic method m, then op_1 and op_2 are concurrent. Therefore, method m has atomicity violation by definition.

2. Race free atomicity violations

   Suppose the original program does not have data races, but there is a reported data

race between $op_1(t, x)$ and $op_2(t\prime, x)$ due to ignoring re-acquire a lock l, denoted as $acq(t, m)$, previously released, denoted as $rel(t, m)$ in the same atomic method.

Thread $t\prime$ must acquire the lock l, denoted as $acq(t\prime, m)$, after $rel(t, m)$ for two reasons:

- if thread $t\prime$ does not acquire lock l, there will be a race in the original program.
- if $acq(t\prime, m)$ happens before $rel(t, m)$, there will not be a reported race due to ignoring $acq(t, m)$.

So, $rel(t, m) \rightarrow_{hb} acq(t\prime, m)$.

On the other hand, there is a trace, in which $acq(t\prime, m) \rightarrow_{hb} acq(t, m)$ while $rel(t, m) \rightarrow_{hb} acq(t\prime, m)$ according to Lemma 4.3.1.

Therefore, method m has an atomicity violation caused by $rel(t, m) \rightarrow_{hb} acq(t\prime, m)$ and $acq(t\prime, m) \rightarrow_{hb} acq(t, m)$.

□

4.4 Overview

RaceDr consisting of : (a) a dynamic race detection system (which accepts parameters and advice to control its application and overhead) and (b) a race healing component which can *generate* repairs, *evaluate* repairs to determine if they are deadlock-free, and *apply* repairs to the running program. RaceDr does this transparently by leveraging a number of JVM sub-systems, such as the JIT compiler which is used to generate and apply repairs. We briefly overview existing work on race detection and healing to provide a context for understanding RaceDr in more detail.

4.4.1 Precise Race Detection

Dynamic race detection approaches, generally, rely on either lockset or vector-clock techniques. Lockset-based techniques, such as *Eraser* [SBN+97] and its descendants[CLL+02, OC03, YRC05, EQT07], generally have low runtime overhead, but they can be imprecise – they can report false races. Vector-clock-based techniques, such as *DJIT+* [PS03] and FastTrack [FF09], have the advantage of being precise – all reported races are real – but they can cause applications to slowdown by a factor of eight or more [FF09].

In RaceDr we wish to only fix real races, so we leverage precise race detectors. In recent years two developments have led to more efficient precise race detectors. Bond et al. employ temporal sampling in the *Pacer* system which allows overhead to be reduced significantly – below 100% when very low sampling rates are used [BCM10]. A sampling race-detector, by its very nature, will skip over certain read and write operations and, thus, while it may be precise it will lose some fault detection. To counter this, in our previous work we developed an optimization that identifies objects that are effectively read-only, so called *stationary objects*, and skips their analysis. This has the effect of focusing race detection on the subset of accesses that are more likely to race. As a consequence our SOS optimization results in increased fault detection, by a factor of five on average, relative to Pacer [LSaD11]. In RaceDr we use a combination of Pacer with SOS as our low-overhead precise race detector; other race detectors could be used as drop-in replacements.

To illustrate how these approaches work, consider a race that exists in a version of Apache Tomcat 6.0.24; an excerpt of the code appears on the left of Figure 4.4. The method `isEmpty()` reads the size of stack `r` and `pop()` writes the size of stack `r`. Thus if two threads, T_1 and T_2, execute the code fragment such that one is executing `isEmpty()` and the other is executing `pop()` a low-level race is reported. This race is identified just on the basis of the ordering of reads and writes on the size field of the shared stack, `r`, even if

there is no observable error.

In addition to low-level data races, this example also has a high-level data race that cannot be detected by Pacer with SOS. Assume that the methods `isEmpty()` and `pop()` are both `synchronized`. The execution shown on the right of Figure 4.4 exhibits a race where threads T_1 and T_2 concurrently execute `isEmpty()` and proceed based on a `false` result. The last thread to call `pop()` (in this trace T_1), however, is left with an empty stack.

This example shows that simply synchronizing `r.isEmpty()` and `r.pop()` is not sufficient to make this code error free. We need to ensure that these two methods are accessed atomically by each thread. Otherwise, atomicity violations that can lead to high-level data races can occur. There are several techniques proposed in the literature to detect such violations [HDVT08, FF04, WS06, PS08], but these can incur even greater slowdown than low-level race detectors (by a factor of 14 or more).

4.4.2 Race Healing Systems

A number of recent projects have explored techniques for generating or applying fixes for data races. We classify these approaches based on whether they include race detection, and if so what kind, how they generate and apply repairs, and whether they address the possibility of deadlock in repaired code. The following table summarizes our analysis of these approaches:

System	Race Detection	Repair Generation	Repair Application	Deadlock Free
AFix	-	Automatic	Off-line	Timeout
Axis	-	Automatic	Off-line	YES
LOOM	-	Manual	On-Line	-
Krena et al.	Low-Level	Automatic	On-Line	NO

AFix [JSZ+11] uses static analysis to automatically generate off-line patches that can fix single-variable high level data races that are detected by other tools. AFix uses a timeout scheme to report when fixes appear to introduce deadlocks. Axis [LZ12] improves on AFix by employing a Petri net model of resource access and constraints solving to produce fixes. This has the advantage of producing better performing fixes that are guaranteed to be deadlock-free.

LOOM [WCY10] takes a different approach. It provides a runtime mechanism to deploy execution filters to "workaround" data races in native applications. It provides an execution filter language in which users declare synchronization intent and LOOM installs and enforces those filters. Given a race report a user can create a filter that specifies how the race can be avoided, but the user must also ensure that the resulting execution avoids deadlock.

The approach of Krena et al. is closest to ours in that it detects races, albeit only low-level races, automatically generates a fix, and deploys it on-line [KLT+07]. This approach uses imprecise race detection which means that many of the generated fixes may be for false races. This can unnecessarily reduce concurrency in the fixed application. Moreover, the introduced fixes can introduce deadlocks. It also requires additional tools including bytecode instrumentor and ConTest testing framework to operate. As such, its performance is too slow to be used in deployed software.

4.4.3 Our Approach

RaceDr is realized as a customized JVM which has the benefit of allowing our technique to be more efficient than instrumentation-based solutions as well as transparent to software developers.

In RaceDr race detection algorithms can leverage information readily available inside

```
1 public static void startCapture() {
2  CaptureLog log = null;
3  if (!r.isEmpty()) {
4   log =
5     (CaptureLog)r.pop();
6  } else {
7   log =
8     new CaptureLog();
9  }
10  ...
11 }
```

```
1 assume: r.size()==1
2 (*$T_1$*)@r.isEmpty();
3 (*$T_2$*)@r.isEmpty();
4 (*$T_2$*)@r.pop();
5 (*$T_1$*)@r.pop();
6 error: (*empty stack*)
```

Figure 4.4: Apache Tomcat excerpt (left) and high-level race (right, in box).

the virtual machine (e.g., lock usage, read/write accesses). Pacer and SOS already do this for low-level races. We support the detection of a wide-range of high-level races by detecting low-level races in candidate atomic methods, i.e., those declared as public/protected, as was done by Vaziri et al. [VTD06].

An important class of high-level races can, however, occur without the presence of a low-level data race [PS08]. In Section 4.3, we describe how we modify the vector clock calculations in existing race detectors to account for the inconsistent view of atomicity across threads. This allows our race detection component to catch high-level races. The execution on the right of Figure 4.4 shows an instance of such a race that our approach can detect.

Repairs are generated by injecting new locks into the program and synchronizing racing regions using those locks. This process is implemented by customizing the JVM's JIT compilation and adaptive optimization sub-systems. Low-level race repairs wrap racing reads and writes with synchronization statements and are guaranteed to not introduce deadlocks into the program. For high-level race repairs, we insert synchronized blocks around racing regions – our current implementation synchronizes entire methods. As an example, the fix for the race in Figure 4.4 adds lock and unlock operations to protect the body of `startCapture()`. To ensure deadlock-freedom, we apply a deadlock avoid-

ance algorithm. This analysis can be done at a low cost by leveraging information about lock ownership that is maintained by the synchronization mechanism inside the JVM. If a deadlock is detected, RaceDr breaks the deadlock and then rolls back the repair.

Since repairs introduce locking, this may reduce concurrency, in our study we found that fixing all detected races in applications caused a slowdown of 15% on average. In some cases races are intended to achieve better multi-threaded performance, to support this we allow race reports to be fed to RaceDr as *detection advice* which will suppress the reporting and fixing of specified races.

The next section provides a more detailed discussion of the implementation of RaceDr and its main components.

4.5 Implementation of RaceDr

RaceDr is implemented as a new feature in *Jikes RVM version 3.1.0* [Jik11]. As such, it can take advantage of existing runtime subsystems to enable efficient and transparent repairs. Figure 1.1 illustrates the high-level organization of RaceDr and its connection with various JVM subsystems.

Leveraging JVM's Subsystems. As shown in the figure, RaceDR uses the JIT compiler infrastructure to analyze methods and dynamically and transparently insert code that controls race monitoring (e.g., manipulate and check monitoring control bits). It also utilizes the existing Adaptive Optimization Systems, which is used to recompile frequently used methods to include more optimizations, to inject and remove repairs. In addition, we also utilize the information maintained by the synchronization mechanism to identify the thread that is holding a lock. This information is used by our deadlock avoidance mechanism.

Furthermore, monitoring memory access operation can be "piggy backed" on the memory management subsystem through read and write barrier mechanisms, which supports

garbage collection. Once a race is detected, we can easily obtain information related to participating threads, methods, and code locations. As such, to use our system, there is no need to modify the source files or class files to enable monitoring. The healing process is automatic so there is no user intervention. It is also transparent, so programs run normally.

4.5.1 Race Detection

The race detection component is designed to be extensible so that it can support multiple race detection techniques. Currently, it supports Pacer and SOS. Pacer is an implementation of FastTrack with sampling support. SOS is an optimization of Pacer with dynamic stationary analysis. Because both techniques have been designed to only detect low level data races, we have modified these two algorithm to detect atomicity violations that can lead to high-level data races.

Detecting High-Level Races. Most atomicity violations involve low-level data races. As such, both Pacer and SOS can readily detected them. On the other hand, detecting atomicity violations without data races requires a change in the race detection algorithm. To understand this change, we first review the basics of vector clock based race detection – as presented in Section 2.2 of [FF09].

A vector clock $VC : TaskId \rightarrow \mathbb{N}$ records a logical time for each thread in the system. VCs are partially-ordered ($\sqsubseteq$) in a point-wise manner; $V_1 \sqsubseteq V_2 \leftrightarrow \forall t : V_1[t] \leq V_2[t]$. They also have a join operator ($\sqcup$); $\forall t : (V_1 \sqcup V_2)[t] = max(V_1[t], V_2[t])$.

For race detection, each thread t has its own local vector clock C_t whose elements record the logical time of the most recent operation by each thread. Each lock l in the program also has a vector clock L_l which are updated to reflect the thread ordering enforced by synchronized access to the lock. When thread t acquires lock l, C_t is updated to $C_t \sqcup L_l$. When thread t releases lock l, L_l is updated to C_t. For each shared variable x, in our case

fields of heap allocated data, two vector clocks R_x and W_x are maintained to record the logical time of the last read and write operations on x for each thread. A read from x by thread t is race-free as long as it follows the last write in each thread, i.e., $W_x \sqsubseteq C_t$. A write to x by thread t is race-free as long as it follows the last read and write to x in each thread, i.e., $W_x \sqsubseteq C_t \wedge R_x \sqsubseteq C_t$.

Algorithm 9 shows our modification of this basic VC-based race detection algorithm. First, we identify `public` or `protected` methods as candidate atomic methods. We track the entry and exit of such atomic methods on a per thread basis – t denotes the thread id. Our modification only performs VC updates the first time a lock is acquired/released in an atomic method. We record the set of released locks within each atomic method to enforce this. The intuition behind this modification is that the lock's VC is "stuck" at the value of the first lock release. All subsequent lock acquires within the atomic section will use the VC value for that first release. Since that VC is earlier in the partial-order than the lock VC computed by the standard algorithm, then all races reported by the basic VC algorithm will still be reported. Moreover, any additional race reports result directly from those older VC values propagating to other threads which acquired the lock and then performed read/write operations within the atomic section. This is precisely the scenario we seek to detect, thus we conclude that our extended algorithm is precise and accurate with respect to low and high-level races.

To illustrate this algorithm in action, consider the trace in Figure 4.4. Table 4.2 shows the vector clock updates at the level of granularity of a method call. The locked object is r and the shared variable is $r.size$.

The vector clock updates for the two calls to `isEmpty()` proceed as in the basic VC race detection algorithm since they are the first to appear within their thread's calls of the candidate atomic method `startCapture()`. The gray cells illustrate the VC comparisons that give rise to race reports. When T_1 executes $r.pop()$, $C_{T_1} = [1, 0]$ – from

	C_{T_1}	C_{T_1}	L_r	$R_{r.size}$	$W_{r.size}$
	$[0,0]$	$[0,0]$	$[0,0]$	$[0,0]$	$[0,0]$
$T_1@r.isEmpty()$	$[1,0]$	$[0,0]$	$[1,0]$	$[1,0]$	$[0,0]$
$T_2@r.isEmpty()$	$[1,0]$	$[1,1]$	$[1,1]$	$[1,1]$	$[0,0]$
$T_2@r.pop()$	$[1,0]$	$[1,1]$	$[1,1]$	$[1,1]$	$[1,1]$
$T_1@r.pop()$	$[1,0]$	-	-	$[1,1]$	$[1,1]$

Table 4.2: Vector clock updates for Figure 4.4.

T_1's perspective T_2 is at logical time 0. In contrast, $W_{r.size} = [1,1]$ which indicates that T_2 is at logical time 1, which is consistent with the fact that T_2's time was updated by its call to $r.pop()$ (the 4th row in the table). Thus, T_2's logical time violates the ordering constraint required for race-freedom (described above) and a race is detected. Note that a race is also detected for the write operation performed in $r.pop()$ by T_1. Since they are both nested within the same candidate atomic method, they can be addressed by the insertion of a single lock to achieve repair of both races.

Useful Features. The system also provides several features in addition to sampling. For example, the race detection component can generate reports of detected races, allowing the developer to use the information to repair races in the subsequent runs or synthesize patches for the faulty Java source files. In the case that developers create benign or intentional data races for performance reasons, detection advice can be provided to the race detector so that it ignores those races.

4.5.2 Race Healing

Our healing system transparently and automatically performs repair on-line. For low-level data races, RaceDr guarantees that our repair is deadlock-free. For high-level data races, RaceDr provides a mechanism to avoid deadlocks due to repairs.

Low-Level Race Repair. RaceDr creates a locking object, l for an object that can race,

```
1  atomicEnter(t) {
2    t.isAtomic++;
3  }
4
5  atomicExit(t) {
6    t.isAtomic--;
7    if (t.isAtomic == 0)
8      t.releasedLocks.clear();
9  }
10
11 acq(t,m) {
12   if ((t.isAtomic == 0) ||
13        !t.releasedLocks.contains(m))
14     C_t ← C_t ⊔ L_m;
15 }
16
17 rel(t,m) {
18   if ((t.isAtomic == 0) ||
19        !t.releasedLocks.contains(m))
20     L_m ← C_t;
21     t.releasedLocks.add(m);
22 }
```

Algorithm 9: Atomicity violation detector modifications.

obj. This locking object, l, stays alive in `special_objects` hashmap for the lifetime of obj. As such, the number of created locking objects is *at most equal to the number of dynamic instances of detected races* in each application.

To avoid introducing deadlocks, we adopt the classic approach to avoiding deadlock of partially ordering the locks and then acquiring locks according to that order [SG08]. In our approach, the program locks are partitioned into D – the locks defined program by the developer – and L – the locks inserted by our approach to avoid data races. Our lock insertion algorithm orders locks such that:

1. $\forall d \in D : \forall l \in L : d \prec l$, and inserts locks such that they are always acquired after developer defined locks. As such, there are no other operations between the acquisition and the data access. If the program acquires any other lock, it must be done before the acquisition of l.

2. Our approach also ensures that inserted locks are never acquired in a nested fashion – thus an ordering on the locks in L is irrelevant. Again, the algorithm acquires lock right before the access. There are no other operations between the lock object acquisition and the data access.

High-Level Race Repair. The race healing component currently utilizes a simplified Banker's algorithm to detect deadlocks due to our repairs. Typically, the Banker's algorithm handles multiple resources [SG08]. For our scenario, the calculation is done when a lock is acquired; therefore, the analysis is simplified.

The algorithm keeps track of lock usage by recording the locks that each thread is currently holding and trying to acquire. For each acquired lock, we keep the information about the thread that is holding the lock. If a deadlock is detected, the following process occurs:

1. The lock being acquired is bypassed.

2. If the lock was inserted by RaceDr, it removes the injected lock.

Timeout. Our race healing component also supports a time-out mechanism similar to that used in AFix [JSZ+11]. This solution also does not guarantee deadlock freedom but can break a deadlock once a predefined amount (e.g., 10 seconds) of time has passed. To support this mechanism, we create a special lock with programmable timeout support. If a deadlock is encountered due to our repair, our race healing component also removes the repair through a rollback mechanism.

Useful Features. Typically, the race detection component initiates the healing component whenever a race is encountered. However, the component also supports the collection of repair advice in the form of a report generated by the race detection component. This feature allows the repair process to take place separately from the race detection process.

For example, the detection can take place during debugging and the detected race report can be provided as patches for the advice mechanism to repair races prior to or during deployment.

For future work, we plan to implement a contention monitoring mechanism (shown as dotted box in Figure 1.1) to dynamically detect lock-contention due to our repairs. The goal is to use a rollback mechanism to remove particular repairs if they cause too much lock contention, resulting in significant performance degradation.

Repair Delays. There are a few situations that can delay the repair process. First, if a race is detected as part of a loop, the method continues to execute until it finishes. Races encountered in the loop would stay unrepaired until the method finishes. By leveraging the adaptive optimization system to regenerate repairs, our system can suffer from queuing delays. Consider a race detected in method `m1`. RaceDr will place `m1` with its repair advice in the adaptive optimization queue to be recompiled with repairs. While waiting in the recompilation queue, `m1` may be invoked once again and this will invoke the unrepaired copy of the method. Our evaluation (see Table 4.5) shows that the delay between detecting a race and applying a repair can vary significantly with the application.

4.6 Evaluation of RaceDr

We evaluate the effectiveness of RaceDr two usage scenarios. First, we evaluate its use *in-house*, i.e., before application deployment, to detect and fix races after which the fixes are applied to the application for deployment. Second, we evaluate the use of RaceDr in a *deployed* environment, where it can continue to find and fix defects in fielded applications. In both scenarios our primary focus is to determine: (a) How expensive is RaceDr relative to race-detection alone? and (b) How effective is RaceDr in repairing races in an application? We will exploit RaceDr's configurability to address these questions under settings that seem

appropriate for the two usage scenarios.

4.6.1 Evaluation Environment

RaceDr is implemented in Jikes RVM 3.1.0 [Jik11]. All experiments are executed on a workstation with 4 2GHz AMD Opteron dual-core processors, where each core has 16 GB of main memory. The system runs Ubuntu with Linux kernel 2.6.30.

We selected benchmarks that are commonly used by researchers to evaluate multi-threaded Java systems; Table 4.1 describes the benchmarks. Three are from the DaCapo 2006 benchmark suite (*eclipse*, *hsqldb*, and *xalan*) with two additional benchmarks from the DaCapo-9.12-bach suite (*avrora* and *sunflow*) [BGH+06]. Currently, not all benchmarks in the 2009 suite can run on RaceDr due to limitations with the Pacer race-detector, which RaceDr builds on. We also used *pseudojbb2005*, which is a version of SPECjbb2005 that has been modified to generate predictable workloads on each run [Sta05].

4.6.2 Using RaceDr In-House

In this setting, we configure the race detection component to perform full-sampling, i.e., the race detector is never turned off. This has two, related, effects: (1) the cost of race detection is high and (2) the number of races detected is also high. We believe that these are reasonable configuration options for in-house testing, especially since RaceDr effectively automates fault diagnosis and repair.

4.6.2.1 Cost of Race Healing in RaceDr

It is well-known that precise race detection comes with significant overhead. Since RaceDr includes a race detector it will also incur high-overhead, but we wish to evaluate the additional overhead that results from generating and applying race repairs.

Benchmark	Detection Only [A]	Detection and Healing [B]	Fixes Only [C]
avrora	4.76	4.83	1.07
eclipse	14.32	14.56	1.24
hsqldb	6.62	6.71	1.10
pseudojbb	4.92/4.72	5.73/5.21	1.83/1.19
sunflow	20.93	21.08	1.17
xalan	3.11	3.29	1.16
Average	9.11	9.37	1.26/1.15

Table 4.3: Race detection and race healing slowdown factors and effects on performance due to repairs.

Table 4.3 reports the slowdown factor for running just race detection (column A), running race detection and healing (column B), and running the program with just the generated race repairs (column C). Note that running with just the repairs does not require either race detection or healing to be enabled – we discuss this below. Since race detection effectiveness varies with the executed thread schedule we ran each benchmark 10 times; recall from Figure 4.1 that the number of detected unique races tapers off rapidly with additional program runs.

We configured RaceDr with the best performing precise race detector we are aware of – SOS. Li et al. [LSaD11] reported the overhead of SOS for detecting just low-level races, whereas here we report the overhead for detecting both low and high-level races. The average slowdown for this race detector alone is 9.11, whereas for Pacer the slowdown was 13.8 and for SOS detecting just low-level races the slowdown was 5.9 on the same benchmarks and execution platform [LSaD11].

When RaceDr performs race detection, race healing, and race reporting. On average the slowdown was 9.37 across our benchmarks. An additional runtime overhead of 26% is incurred by race healing and reporting. Nearly all of this additional overhead comes from race healing – the cost of race reporting is negligible.

Benchmark	Statically Injected Locks [A]	Dynamically Injected Locks [B]	Dynamic Lock Acquisitions [C]
avrora	89	261433	514362
eclipse	61	364136	739126
hsqldb	67	346	614
pseudojbb	956/42	$2.65e^7/2.58e^5$	$3.92e^8/3.51e^5$
sunflow	26	236	536
xalan	33	77	116

Table 4.4: Reporting lock usage and lock acquisition characteristics.

Discussion. By subtracting columns A from B in Table 4.3, we can compute the additional runtime overhead incurred by race healing. That additional overhead ranges from 7%, for avrora, to 81% for pseudojbb2005. Moreover, the percentage increase for pseudojbb2005 is over three times the next highest, 24% for eclipse. There are several factors that can contribute to the overhead of healing. To apply a fix, methods must be unloaded from the JVM's execution cache, recompiled, and reloaded. Depending on the number of locks injected this cost can be expensive. Another component of healing overhead is the cost to create and manage lock objects at runtime. This is mainly due to the cost to allocate lock objects, acquire and release those locks, and eventually garbage collect them.

Table 4.4 reports data on the number of locations at which locks are injected into the program (column A), the number of times those injected locks are allocated at runtime (column B), and the number of times those locks are acquired (column C). From this data it is clear that the overhead of pseudojbb2005 results from the fact that an additional 26 million locks are created and subsequently acquired at runtime. We discuss why this happens and RaceDr's support for addressing this overhead below.

Table 4.4 also highlights some of the subtleties that arise when working in a JVM/JIT environment. Theoretically, the number of statically injected locks and the number of unique races should be the same. However, our approach injects more locks than the num-

ber of unique races (see Table 4.1). Further investigation shows that due to in-lining and other compiler optimizations, a source of a race can appear in multiple compiled methods. RaceDr must inject locks in each such method.

4.6.2.2 Performance of Repaired Races

Repairs are compiled into the benchmarks by a lightweight path through RaceDr that reads the generated race reports. Since this path does not involve race detection or healing, the runtime overhead is reduced significantly. Column C of Table 4.3 reports that overhead. There are two numbers for pseudojbb: the first (1.83) includes an intentional race, which is discussed next, and the second (1.19) uses the race advice mechanism in RaceDr to avoid repairing intentional races. The overhead of repairs across all programs averages 15% when the intentional race is ignored.

Discussion. To understand why pseudojbb2005 incurs so much overhead, we investigated its execution behavior. Most races occur in an XML parser method. This method is called from every server thread in the benchmark and is thus, highly concurrent. To achieve high-performance, the developers of that method employ an intentional benign race, known as synchronization races in the form of a user-defined lock [NWT+07]. Unfortunately, the precise race detectors we employ identify this as a race. Because this is a commonly invoked method, over 26 million locks objects must be created to fix this race. These locks are then acquired nearly 400 million times. Therefore, the optimization that has been carefully crafted by the developers was nullified by RaceDr.

To address the presence of benign races, RaceDr provides a detection advice mechanism for developers to specify sources of races that should not be reported or fixed. We used this feature to avoid fixing the benign race in pseudojbb2005. Specifically, we informed RaceDr that it should ignore any race that occurs in the XML parser. Since the race is no longer detected, there is no race repair inserted. This results in a reduction of unique

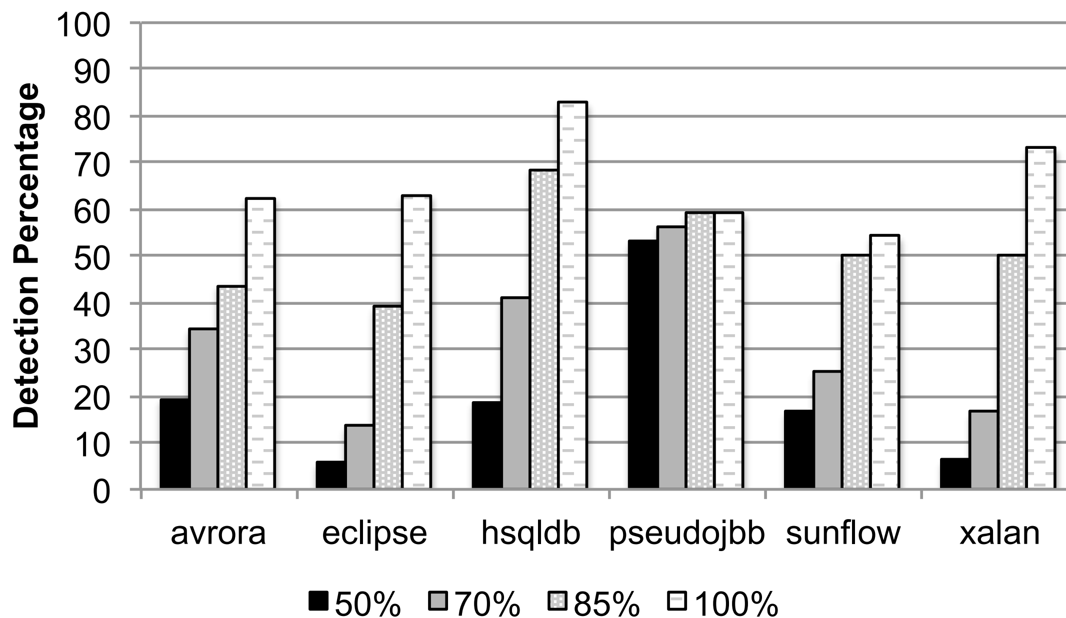

Figure 4.5: Effects of sampling on detection coverage.

races from 148 (see Column E in Table 4.1) to 32 and a 3-order of magnitude reduction in lock acquisition (see the pairs of numbers in Table 4.4). This yields reduction in the overhead of the race repairs from 83% to 19%. Also note the overhead reduction from the pairs of values in columns A and B of Table 4.3 when the benign race is suppressed. For the remainder of this chapter, we will evaluate the performance of pseudojbb2005 *without detecting and repairing benign races*.

4.6.2.3 RaceDr Fix Effectiveness

For each detected race, RaceDr will produce a potential fix. Those fixes will be analyzed to determine whether they can cause a deadlock. Across all of the experiments reported in Table 4.3 only 3 out of 318 repairs were judged as potentially deadlocking. RaceDr can be configured to deploy timed locks for those 3 races and then rollback the fixes if the lock timers expire – indicating an occurrence of deadlock.

4.6.3 Using RaceDr in Deployed Environments

As shown in the previous section, race detection comprises the dominant cost in applying RaceDr. To use RaceDr in deployed environments, we reduce this cost by applying Pacer's race detection sampling; SOS uses the same sampling mechanism as Pacer. As part of our evaluation, we investigate the impact of sampling on both the performance and race detection effectiveness of SOS extended with support for high-level races. First, we observe the race detection effectiveness when sampling is used to control the runtime overhead to be within a certain budget. Second, we evaluate the overall performance of RaceDr by exploiting the insight that we can eventually turn off race detection after a program runs multiple times.

4.6.3.1 Impacts of Sampling on Detection Effectiveness

To evaluate the effectiveness of our SOS atomicity violation detector, we perform the same experiment that was used to evaluate the effect of sampling on SOS [LSaD11]. That is, we fix the overhead of race detection to be 50%, 70%, 85% and 100% of the original benchmark execution time. We achieve these target overheads by controlling the sampling rate in SOS which limits the race detector to periodically observing small windows of execution behavior. To achieve 50% runtime overhead, the sampling rates range from 2.1% in *pseudojbb2005* to 3.8% in *hsqldb*. For 100% overhead, the sampling rates range from 14% in *pseudojbb2005* to 31% in *hsqldb*.

Figure 4.5 reports the percentages of unique race sources detected at the four target overheads. The reported values are normalized against the average numbers of detected unique races per run with full-sampling, i.e., all of the program execution is visible to the race detector (see Table 4.1, column E). The figure also illustrates that the detection coverage increases as we go from 50% to 100% overhead.

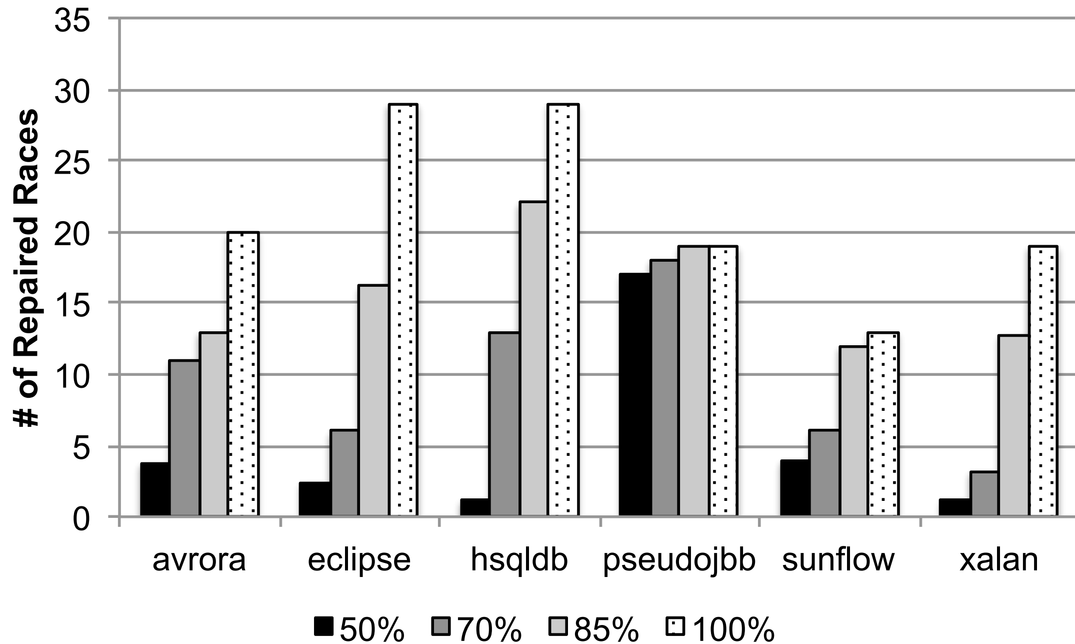

Figure 4.6: Race repairs with varying detection overhead.

Figure 4.6 reports the number of unique instances of races that have been repaired for each benchmark. Since less than 1% of the races in our study have fixes that may introduce deadlock, as expected the repaired races rises in direct proportion with the number of detected races.

Discussion. A point to take away is that RaceDr can only repair races that have been detected. Hence, its effectiveness depends on the detection coverage. Because RaceDr must first detect a race, by definition, it will miss healing that instance of the race. Additionally, there are delays in the JIT recompilation process, so more instances of the race may be missed. Furthermore, in our implementation, recompilation can only occur once the method is no longer in use. If a unique race exists in a method that is mainly a loop, multiple dynamic races may be detected before it is repaired. To quantify the effect of repair delays, we analyze race reports generated by our system with detection and healing.

Benchmark	Overhead Budget 50%	75%	85%	100%
avrora	658.75	1594.05	1662.97	2024.22
eclipse	1.44	21.31	116.67	160.58
hsqldb	2.00	8.98	22.11	17.22
pseudojbb	11.77	129.67	148.47	202.67
sunflow	2.50	1.79	1.96	6.98
xalan	2.00	2.94	1.90	1.92

Table 4.5: Average number of detected races before a repair is applied.

Table 4.5 reports the average number of detected races between the detection of each race and the application if its repair. Avrora is a sensor network simulator that has only a small number of unique races (32 in Table 4.1, column E) but can have nearly 500,000 dynamic races. This indicates that these races reside in loops. Since repairs can only be applied when the method enclosing such loops returns, it is not surprising to see long repair delays (over 2000 detected races); pseudojbb2005 exhibits a similar pattern. On the other hand, benchmarks that have relatively few dynamic races (hsqldb, sunflow, and xalan) incur much shorter delays. An interesting benchmark is eclipse. It has many dynamic races and many methods, which in this case leads to moderate repair delays.

4.6.3.2 Impacts of Disabling Race Detection

Prior work by Arnold et al. [AWR05] suggests that significant performance improvements can be achieved if the runtime information from the current execution can be used to optimize the virtual machine configuration for the next execution. This notion is directly applicable to race repair because if the repair information from the current execution can be retained for the next execution, our proposed system would be able to directly repair previously identified races.

We evaluate the benefits of retaining cross-run information using the following methodology. We tuned RaceDr to operate at 100% overhead – doubling application runtime.

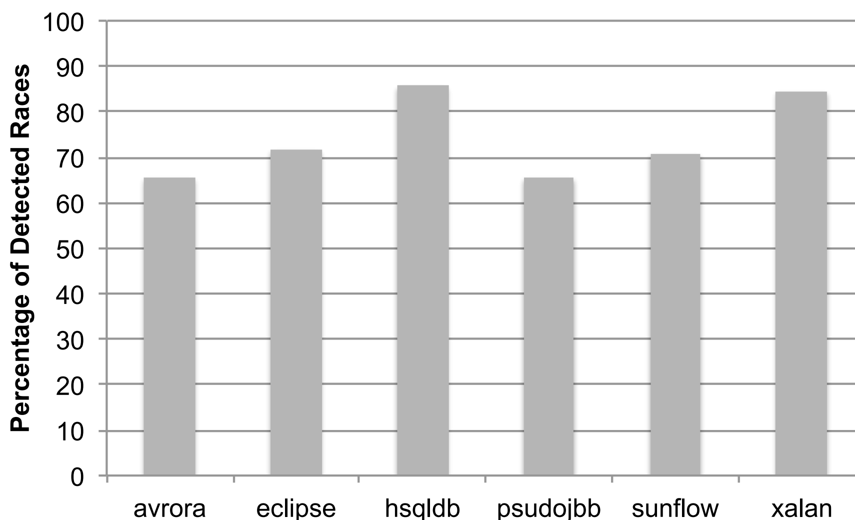

Figure 4.7: Percentage of detected races after 20 runs.

Next, we ran each benchmark 20 times. After each run the race report is recorded. This allows us to accumulate the set of race reports for a sequence of runs. Figure 4.1 reports the unique race sources detected in each run relative to the accumulated race reports from the preceding runs. It is clear from this data that for all benchmarks most of the unique races are detected within the first 10 runs.

Discussion. The effectiveness of race detection in uncovering unique race sources diminishes over time and, consequently, it makes little sense to continue to incur large race detection and healing overheads throughout an application's deployment. RaceDr allows the user to turn off off race detection, but retain the fixes previously generated. In our current implementation the user must specify what amounts to a threshold on the rate at which unique races are detected and when an application falls below that threshold, then the next execution of the application is performed without race detection and healing. We leave the exploration of more sophisticated online approaches to suppressing the operation of RaceDr components to future work.

Because RaceDr can accumulate repairs across multiple program runs, it is possible for

the system to eventually detect as many races as that of full-sampling. Figure 4.7 plots the percentage of all detected races (found at full-sampling) that are found when race detection is limited to a budget of 100% of application runtime and 20 runs are performed. Here we see that applications can detect anywhere from 65% to 85% of total number of races.

Across all benchmarks RaceDr repairs 144 races after 20 runs. This represents 73.85% of all detected unique races when the system is running with full sampling. (Our system detects 195 races when running at full sampling. This number is different than that in Table 4.1 because it does not include the benign races in pseudojbb2005.) When considering disabling of race detection and healing, we note that after 10 runs, RaceDr has already repaired 129, or 90%, of the races it will encounter in 20 runs. Selecting a threshold for disabling race detection and healing may depend on many factors, but it seems clear that there is an opportunity to exploit this phenomenon and significantly reduce runtime overhead. Recall that when those components of RaceDr are disabled the runtime overhead drops to 15%, on average, across the benchmarks (Table 4.3, column C).

4.7 Conclusion

In this work, we take a step toward building a completely transparent and automatic race healing system with *RaceDr*. Our system can detect data races precisely and efficiently through the use of a state-of-the-art race detector that can be tuned to work well in different application scenarios. It computes repairs for detected races and prior to applying those repairs, determines whether a repair can cause the system to deadlock. If it can, we can use special locks with timeouts to break the deadlock if it should occur or rollback the repairs.

Our evaluation reveals that RaceDr can effectively expose and repair races in a pre-deployment testing and debugging environment. The repair information can then be used during application deployment to provide a more robust program with modest (15%) over-

head due to additional locking. It is also possible to use RaceDr in a deployed setting. By using a low sampling rate, the overhead of RaceDr can be controlled and over a series of program runs this can result in the detection and repair of nearly as many races as would be resolved with full-blown race detection. Once the rate of newly detected races for a program drops low enough, RaceDr can be run without detection and healing which nearly eliminates the overhead incurred during subsequent program executions.

The results of our work open up several new research opportunities. For example, it is likely that the optimal point at which detection and healing should be turned off in deployed environments differs from one application to the next. As such, phase detection algorithms are needed to identify where those capabilities should be disabled and enabled. Currently our race detection uses temporal sampling, but other forms of sampling, e.g., across the methods or classes of a code base, could also be effective. Finally, detection of benign races, or even those whose locking results in high contention, could also increase the transparency and automation of RaceDr.

Chapter 5

Related work

5.1 Race Detection

Numerous recent research have been presented on race detection. *FastTrack* [FF09] and *Pacer* [BCM10] are the most closely related work ours and we have done extensive comparisons in Section 2.2.4. As such, we only compare our approach to other related race detection work in this section. Race detection, in general, can be broadly categorized into two approaches: static and dynamic race detection. Next, we describe these approaches in turn.

Static race detection Model checking has been applied to detect races for over a decade [HJM04, KYKS09]. Model checking provides precise results with no false reporting; however it can suffer from state explosion, and therefore, does not scale well.

Other static race detection techniques [VJL07, NAW06, CLL+02] can scale to large systems, but they are not precise; that is they can provide false positives. Furthermore, they often have difficulties dealing with reflection and native methods [UL08]. As such, our approach is instead based on dynamic analysis because we need to have precision in stationary object identification. Furthermore, our approach is built on top of a dynamic race

detector.

Dynamic race detection There are at least two common dynamic race detection approaches: one is based on *vector clocks* and the other is based on *locksets*. These two approaches have been used individually as well as together to detect races. Typical lockset algorithms enforce the locking discipline that every shared variable is protected by some locks and a lock is held by a thread whenever it accesses the variable. Eraser [SBN+97] is one of the representative work on race detection based on locksets. It is common for lockset algorithms to generate false positives, but they have much lower execution overhead than vector clock based approaches.

A vector clock based race detector constructs *happens-before* relationships from program statements. It pays particular attention to all accesses to shared resources and synchronization primitives such as locks. Generically, the algorithm maintains logic time vectors for all shared data and records when they were last accessed. An example of VC based algorithm is *DJIT+* which is used to detect races in concurrent C++ programs [PS03, PS07]. One of the main overhead for VC based detectors is the comparing vector clock entries. This is an $O(n)$ time operation where n represents the number of threads. As programs become more thread intensive, the overhead continues to climb.

FastTrack [FF09], LiteRace [MMN09] and Pacer [BCM10] are all based on vector clocks. As aforementioned, the main contribution of FastTrack is the reduction in the VC storage space and operation time from $O(n)$ to $O(1)$. As such, FastTrack can maintain the precision while reducing the overhead to be about the same as that of a lockset based approach.

LiteRace can also reduce overhead without compromising too much detection coverage by exploiting an insight that races are more likely to exist among infrequently executed code. Their technique gradually reduces the sampling rates as the code become "hotter" and maintain high sampling rates for "cold" code. We adapt a variation of this technique by

exploiting the compiler infrastructure in Jikes RVM to perform instrumentation in "cold" code and less instrumentation in "hot" code. Another distinction from our work is that LiteRace performs offline analysis while our approach perform analysis on-line.

Hybrid Race Detection There are also techniques that combine VC and locksets to achieve higher precision at lower cost. As an example, MultiRace combines both approaches to detect races in C++ programs [PS07]. RaceTrack uses the lockset based approach for checking locking discipline but then also applies the VC based approach to determine access orders. This approach can significantly reduced the occurrence of false race reports while maintaining 2x to 3x execution overhead [YRC05]. However, these systems do not guarantee precision; they only reduce false positives. Therefore, they are not used in our system.

Predictive Race Detection Smaragdakis et al. build a sound predictive race detector to detect some hard to find races [SES+12]. They generalize happens-before relationship to causally-precedes relation (CP) to soundly detect more races. This approach uses Fasttrack [FF09] to generate program execution traces, then it builds CP relation based on the races and detect races. The results of applying CP to real-world programs show that it can find races that are difficult to detect by happens-before based race detectors. Quantitatively they are able to predict just slightly fewer additional races as our technique. In addition to assuring soundness, CP race detection improves on other race predictors [CR07] [CSR08] [SRA06] in that its time complexity is polynomial.

Hardware Based Race Detection So far, most software based race detectors introduce significant overhead. Hardware can greatly speed up the detection, as shown by DRFx [MSM+10] and Conflict Exceptions [LCS+10]. DRFx uses a hardware buffer of memory locations accessed between fences and coherence event monitoring that checks for conflicts with addresses in the buffer. Conflict Exceptions keeps pre-assigned byte level access bits per cache line and sets aside memory space to keep access bits for out-of-cache data. How-

ever, these approaches require specialized hardware, which may not publicly be available.

5.2 Atomicity

Atomicity is a fundamental correctness property in multi-threaded programs. The research in the area dates back Lipton's reduction theory [Lip75], which defines sufficient conditions to ensure atomicity of parallel programs. This reduction theory forms the basis of a number of atomicity checking approaches including Atomizer [FF04].

Atomizer uses a dynamic analysis for detecting atomicity violations in Java programs. Atomizer is built on top of a lockset based race detector, Eraser. It uses Eraser to dynamically collect the lock information that is required for reasoning about atomicity. RaceDr also extends a race detector, SOS [LSaD11], to capture atomicity violations. The major difference is that SOS is a precise detector while lockset based approach is imprecise.

Velodrome [FFY08] improves Atomizer by ensuring completeness and soundness relative to the program execution that is observed. It builds a dependence graph for a program based on the happens-before relation over synchronization and read/write operations. Therefore, cycles in the dependence graph indicate atomicity violations in the program. Velodrome can identify the operation causing the violation, which could facilitate violation fixing. Unlike our race detection algorithms that can detect both high- and low-level data races, Velodrome cannot detect low-level data races.

Vaziri et al. [VTD06] define a notion of atomicity violations that involves associating synchronization constraints with data, which can avoid some false warnings issued by reduction theory based approaches. In follow-up work, Hammer et al. [HDVT08] propose a concurrent program correctness criterion called atomic-set serializability, which states that units of work must be serializable for each atomic data grouping. They also provide a dynamic analysis to detect violations for atomic-set serializability based on 14 program

interleaving patterns.

There is a large body of work on testing for concurrency faults. A number of techniques in this area manipulate program scheduling to better expose concurrency faults. For example, Active testing [JNPS09] randomizes the program's schedule at the application level by inserting yield points. DataCollider [EMBO10] use hardware breakpoints to detect problematic memory accesses via kernel support. There is also work that can predictively find concurrency faults based on observation of program behavior.

ConMem [ZSL10] monitors memory access information of a program and predicts possible program traces that could lead to a fault. AVIO detects atomicity violations according to program access interleaving invariants [LTQZ06]. In general, any of these techniques for boosting fault detection effectiveness can be used in concert with RaceDr. When using RaceDr for testing and debugging this is probably the preferred approach, since the program will more quickly reach the point at which new races are no longer detected.

5.3 Fault Repair

AFix [JSZ+11] is closely related to ours. AFix provides a static analysis for bug reports to automatically generate patches that can fix single-variable atomicity violations. AFix also has runtime monitoring based on timeout to avoid deadlocks. Unlike Virtual Healer, AFix cannot immediately fix bugs on-the-fly. It only provides patches that must be applied by developers. As such, AFix might need rebooting or code recompilation to work. This can greatly reduce system availability.

As a complement to AFix, LOOM can aid developers to generate ”live-workaround” to fix data races [WCY10]. It provides a language that developers can easily write a patch. LOOM also builds an update engine to install patches without interrupting the program executions. However, it still needs users to manually generate patches while *VirtualHealer*

fully automates the process of patch generation.

In addition to systems that we have discussed in this dissertation, the following systems can also perform self-healing. ClearView[PKL+09] provides a mechanism to automatically generate patches for x86 binaries at runtime. It operates at the binary level so it does not need source code and can apply patches without system reboots. However it is not design to support repairs of concurrency faults. Weimer [WNLGF09a] introduce a new approach to automatically fix faults with genetic programming. This approach is designed for testing environment since the overhead is too high to be effectively used in deployed system (i.e. 200 seconds for 63,000 line programs). Portend [KZC12] provides a system to distinguish benign races and harmful races, which could be an infrastructure of *VirtualHealer*.

PBnJ [SAM10] performs dynamic contract checking of methods. It employs SAT solver to detect contract violations based on predefined specifications. Specifications, required by PBnJ, can be difficult to write for concurrent programs due to complexities of properties. Meanwhile, read and write operations need be monitored in order to detect and fix concurrency bugs. This could be prohibitively expensive in their system.

Krena et al. proposed an on-the-fly race healing that can detect races and inject two types of instructions: scheduling and synchronization to fix them. The system was built on top of ConTest, a concurrent program testing platform so that it can perform runtime monitoring and code injection using the features provided by ConTest. The race detection is based on *Eraser* which is a lock-set based race detection approach [KLT+07].

REASSURE use rescue points to recover from software faults [PK11]. It will rollback the program execution when certain anticipated software errors occur. It keeps the software operational and returns a valid error code. However, REASSURE cannot fix errors, which is similar to LOOM.

Genetic algorithm has been employed to automatically fix software faults [LGNFW12, LGWF12, FL12, SFW10, WFLGN10, WNLGF09b], which is described as viewing pro-

grams through the lens of evolutionary biology. These approaches assume that the program functional requirements are properly encoded in the test cases. Once a program fault is located, a genetic algorithm based mechanism will evolve the program to produce program variants until one variant is found to successfully pass the functional test cases and avoid the program defects. These approaches do not require formal specifications, program annotations or special coding software. However, it is challenging to ensure the test cases completely include all functional requirements. On the other hand, the number of variants can exponentially increases for complicated program faults that involves multiple program locations.

5.4 Optimizing Runtime Monitoring

There is a long history of work on optimizing runtime monitoring. The vast majority of that work has focused on monitors aimed at checking properties of individual program states, e.g., assertions, array bounds checks [MMS98, BGS00]. A more recent trend has looked at reducing the cost of monitoring properties related to sequences of program states (or actions), e.g., data races, typestate [SY86]. Since our interest in this dissertation is in optimizing dynamic data race detection we will discuss recent approaches to reducing the overhead of state sequencing properties.

Generally speaking, there are two main approaches that have been explored to reduce monitoring overhead: using static analysis to determine instrumentation that can be safely removed, and using sampling to select a subset of instrumentation to insert in the program.

Static Analysis. Over the last several years, there have been two groups exploring the use of static analysis for reducing the cost of monitoring typestate properties — one can think of these as any property that can be described as a finite state automaton. In a pair of papers, Eric Bodden and colleagues developed a staged analysis approach [BLH08,

Bod10]. Early stages involve flow-insensitive analyses that exploit information about the program statements and data that are reference in the property, whereas later flow-sensitive analyses relate the structure of property automaton to paths in the program control flow graph to drop instrumentation. These techniques have been shown to be quite effective across a range of properties checked on the DaCapo benchmarks [BGH+06].

In [PDE10] we focused on the the general problem of monitoring properties specified as finite-state automata and the optimization of monitor instrumentation in loops. We developed conditions under which a prefix of the loop iteration space may be unrolled. The unrolled prefix is then instrumented while the remaining iterations of the loop have their instrumentation removed. This loop optimization was able to reduce the number of operations that required monitoring by several orders of magnitude across a set of DaCapo benchmarks. Unfortunately, for multi-threaded programs there is no guarantees that the behavior of other threads will occur while a loop is executing its monitored prefix, thus the technique is not directly applicable to data race detection.

Sampling. There has been a large body of work on using sampling techniques to, for example, choose a subset of assertions to enable in a program execution. Sampling can significantly reduce the overhead of runtime monitoring, but that benefit comes with a reduction in fault detection. Sampling sequencing properties is much more subtle than sampling assertions. To preserve the precision of runtime monitoring, one must sample sets of instrumentation that when enabled ensure that no false error reports will be issued.

One straightforward way to do this is to sample the set of objects that are monitored as is done in QVM [AVY08]. QVM uses a randomized sampling approach, similar in spirit to Pacer. QVM is a customized VM that uses a bit in the object header to efficiently control instrumentation that bit is set, using a biased coin flip, when the object is allocated.

Several researchers have exploited the structure of the property to identify subsets of instrumentation that preserve some fault detection, but offer potential overhead reduction.

This general approach is taken, in very different ways, in [BHL+07] and [DDE08]. Since data races have a very simple structure there is little opportunity for applying this strategy.

Most work on optimizing the monitoring of sequencing properties performs the optimization either offline or at object creation time. Pacer is slightly different in that it monitors some operations throughout an object's lifetime and only samples a subset of remaining operations; it controls sampling at garbage collection time. Work by Xian et al. [XSaJ08] also uses sampling to control the overhead of monitoring lock usage in large Java server applications. In this approach, sampling is controlled by thread creations; that is, more information is monitored during the initialization and then less as the number of threads stabilizes.

In general, for any sampling approaches one must consider the population being sampled and the relationship between that population and the properties being analyzed. In some cases, the population can be reduced with little loss in detecting property violations. This is the case for our SOS optimization – we eliminate stationary objects from the population of objects on which reads and writes are sampled.

When using temporal sampling, as is done in Pacer and QVM, the regions within which sampling is performed may vary widely in the size of the population of reads/writes that might be sampled. These techniques are insensitive to this variation in population density and this can result in unnecessarily high monitoring overhead. Our approach attempts to adjust the sampling rate within loops so that the population density within sampled temporal regions more closely matches the overall sampling rate. In doing this, we achieve lower overhead while sacrificing little in the way of race detection.

5.5 Stationary Fields

As we discussed, Unkel and Lam [UL08] presents the notion of *stationary fields* to describe fields on which all writes happen before reads, creating an opportunity to further optimize programs. Our proposed approach is based on this notion with a broader scope to include objects in addition to fields. Rogers et al. [RZKW08] extends Java language to express a field as stationary when some constraints are satisfied. With this extension, JVM can get the information about stationary fields for runtime optimization. They use class loading as a case study. The evaluation result shows it can achieve 1.67% to 3.90% speedup in their experimental subjects.

Bronson et al. [BKO09] use stationary field information to optimize strongly isolated *Software Transactional Memory* (*STM*). They use static analysis to identify stationary fields in a program, they then use the result to help improve SMT performance by eliminating memory barriers on stationary fields. Loginov et al. [LYC+08] treats stationary fields in the same way as final fields in the verification of dereference safety in Java programs. To the best of our knowledge, there is no existing work on race detection optimization with stationary field information.

Chapter 6

Conclusion and Future Work

This chapter summarizes my dissertation work and provides directions for future work. For future work, we propose a road-map to extend our current work, which includes an approach to optimize race detection, a technique to improve the online adjustment of RaceDr and a study on the quality of repairs generated by RaceDr and developers.

6.1 Conclusion

This dissertation presents research on automatic race detection and healing since data races are the most common concurrency faults and the root cause of many system failures. Our approaches optimize sampling based dynamic race detection by increasing density of data races over sampling space. We achieve this goal based on the insight that many operations access stationary objects, which cannot participate data races. Therefore, our detector does not need to monitor these operations. The evaluation results shows that our optimization can detect up to six times more races than existing with the same overhead budget in comparison to Pacer, a state-of-the-art race detector.

The goal for race detection is to find code locations that can cause data races. Therefore,

detecting one instance of a race is enough to locate the sources of that race. However, our evaluation found that even after filtering out stationary objects, race detectors still spend substantial amount of time on detecting the same races over and over again, causing some races to be needlessly detected up to several millions times. An in-depth investigation reveals that a majority of repetitive races occur in loops. Based on this insight, we built a sampling based race detector, LIS, that can adaptively adjust sampling rates according the number of iterations in the loop. By applying LIS to Pacer and SOS, we can get 52% and 33% overhead reduction, respectively.

Lastly, we propose RaceDr, a framework to transparently repair data races on-the-fly. RaceDr advances existing race repair systems as follows. First, it includes a high performance race detector, extended from SOS, that can cover both low level and high level data races. The overhead of the race detector can be controlled by adjusting sampling rate in order to use in deployed systems with tight performance constraints. Second, RaceDr ensure the correctness of repairs by not introducing deadlocks due to the repair code. Third, the repairs are automatically generated and applied so there is no need to manually perform fault repairs. Fourth, the repairs are immediately applied without interrupting the execution of the system.

6.2 Future Work

6.2.1 Code Based Sampling for Race Detection

Sampling is an effective way to reduce overhead of dynamic race detection, but existing sampling based race detection has the following limitations:

1. Current approaches, such as Pacer, use uniform sampling based on time. However, the goal of race detection is to identify code locations that cause races instead of the times

when races occur. As such, when a program is executed in loops, the temporal sampling approaches would repeatedly monitor the same code region, as identified by our investigation. Spatial sampling based on code region could make race detection more efficient.

2. Current sampling based race detectors are not cognizant of program semantics, therefore, they assume that all program code has equal probability to participate races. In fact, semantic information could greatly enhance efficiency of race detection. Program loop information can help reduce repetitive detected races as shown in Chapter 3. However, LIS just takes the first step to make use of semantics information in race detection, and there are two additional directions worth pursuing.

First, type information can guide race detectors to adjust sampling rate. For instance, a symptom of races is that a type of class has many accesses to shared data but without sufficient synchronizations. As such, this type of classes should employ high sampling rate. In addition, combination of both static and dynamic analyses can work together to reduce overhead without sacrificing precision.

6.2.2 Race Repair Study

RaceDr automatically generates repairs for low level and high level races based predefined algorithms. It is interesting to compare RaceDr's repairs with hand-crafted repairs. We plan to investigate the official repairs released by developers that can provide some insights for RaceDr to improve its race healing algorithm. We can conduct user study of race repair among different level of developers ranging from beginning programmers, graduate students, and experienced engineers to evaluate the benefits of RaceDr for different group of users. The study will answer the following questions:

- Do hand-crafted repairs introduce fewer deadlocks than RaceDr?

- What is the performance implication if applying repairs generated by RaceDr versus manually generated repairs?

As our experiment shows, RaceDr occasionally generates repairs that can cause deadlocks. There are two possible reasons for these deadlocks. First, some deadlocks occur due to the program semantics, so it is very difficult to avoid such deadlocks without non-trivial changes to the original program. Second, some deadlocks are caused by the current limitations of our system (e.g., lack of global program information, naive lock inserting algorithm). For the latter case, we expect hand-crafted repairs should provide deadlock free solutions. Therefore, it is very interesting to investigate the reasons for RaceDr to generate repairs causing deadlocks. The results can lead to improvements of our fixing mechanism.

It is known that locks can lead to performance bottlenecks. RaceDr causes about 15% performance overhead. We plan to compare the overhead of RaceDr with that of manually generated repairs to determine which approach is more efficient in terms of introducing lower runtime overhead. We believe that studying these research questions will open up opportunities to further improve RaceDr.

6.2.3 Concurrency Attack Study and Fixing

Recent investigations of real-world data races and atomicity violations have shown that they can lead security vulnerabilities [YCSS12, PMBM08]. For instance, a directory service script in Mac OS X 10.3.9 has an atomicity violation that can result in authentication vulnerability when the following actions occur. First, it writes the private and public keys to a temporary file; then it reads the keys and put them into the database. Because the two-step process is not atomic, attackers can exploit the violation by replacing the temporary file with their own public and private keys before the temporary file is re-read, causing the attackers keys to be inserted to the database instead of the actual keys. As multi-core

systems are becoming the mainstream computer platform, this class of vulnerabilities is expected to rapidly increase. However, to the best of our knowledge, there is no systematic and comprehensive study on security vulnerabilities due to data race. As such, a careful and comprehensive empirical study can characterize possible security defects caused by data races and suggest potential solutions to identify and address these defects.

To study these vulnerabilities, we hypothesize that races that can cause security threats have characteristics that are different from those that only impair program correctness. The insights gained from our study can suggest potential opportunities to expose and eliminate them. More specifically, our study will answer the following research questions:

- What percentage of races are able to compromise system security? Are they only a small portion or a majority of the races in a system?
- Which class of races tend to cause security issues, low-level data races or atomicity violations? In the case of atomicity violations, does the duration of the atomicity violation affect exploitability, and can attackers affect that duration?
- What is the vulnerability manifestation pattern? How many threads participate? How many variables are involved?
- Can the traditional race fixing solutions like inserting synchronization operations eliminate vulnerabilities?
- Can the proposed security patches in bug repositories really fix the vulnerabilities? Can they introduce new defects for security, correctness or performance?

Based on our investigations of above questions, we will conduct experiments to determine if the knowledge obtained from the study can be helpful in finding and addressing race-related vulnerabilities.

Because RaceDr can fix generic races and atomicity violations, it also has the potential to fix vulnerability due to data races that can lead to concurrency attacks. The key is to find out the particular features of races that can impair the system security. We strongly believe that knowledge about real world race vulnerabilities will be valuable for guiding research towards the detection of, and defense against, this class of increasingly prevalent attacks. We are going to extend RaceDr to prevent concurrency attacks. Since security attacks are usually time critical, the feature of on-the-fly fixing of RaceDr can fit in this situation very well.

Bibliography

[AFF06] Martin Abadi, Cormac Flanagan, and Stephen N. Freund. Types for Safe Locking: Static Race Detection for Java. *ACM Transactions on Programing Languages and Systems*, 28:207–255, March 2006.

[AHB03] Cyrille Artho, Klaus Havelund, and Armin Biere. High-level data races. In *NDDL/VVEIS*, pages 82–93, 2003.

[Apa] Apache Software Foundation. Datarace on org.apache.catalina.loader.webappclassloader. https://issues.apache.org/bugzilla/show_bug.cgi?id=37458.

[AVY08] Matthew Arnold, Martin Vechev, and Eran Yahav. QVM: An Efficient Runtime for Detecting Defects in Deployed Systems. In *Proceedings of the Conference on Object-Oriented Programming Systems Languages and Applications*, pages 143–162, 2008.

[AWR05] Matthew Arnold, Adam Welc, and V. T. Rajan. Improving Virtual Machine Performance Using a Cross-run Profile Repository. In *OOPSLA*, pages 297–311, San Diego, CA, USA, 2005.

[BCM10] Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. PACER: Proportional Detection of Data Races. In *PLDI*, pages 255–268, Toronto, Ontario, Canada, June 2010.

[BGH+06] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot, B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA*, pages 169–190, Portland, Oregon, USA, 2006.

[BGS00] Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. ABCD: Eliminating Array Bounds Checks on Demand. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 321–333, 2000.

[BHL+07] Eric Bodden, Laurie J. Hendren, Patrick Lam, Ondrej Lhoták, and Nomair A. Naeem. Collaborative Runtime Verification with Tracematches. In *Works. on Runtime Verif.*, pages 22–37, March 2007.

[BKO09] Nathan G. Bronson, Christos Kozyrakis, and Kunle Olukotun. Feedback-Directed Barrier Optimization in a Strongly Isolated STM. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’09, pages 213–225, New York, NY, USA, 2009. ACM.

[BL04] Michael Burrows and K. Rustan M. Leino. Finding stale-value errors in concurrent programs: Research articles. *Concurr. Comput. : Pract. Exper.*, 16(12):1161–1172, October 2004.

[BLH08] Eric Bodden, Patrick Lam, and Laurie Hendren. Finding Programming Errors Earlier by Evaluating Runtime Monitors Ahead-of-Time. In *International Symposium on Foundation of Software Engineering*, pages 36–47, New York, NY, USA, 2008.

[BLH10] Eric Bodden, Patrick Lam, and Laurie J. Hendren. Clara: A framework for partially evaluating finite-state runtime monitors ahead of time. In *RV*, pages 183–197, Malta, November 2010.

[Bod10] Eric Bodden. Efficient Hybrid Typestate Analysis by Determining Continuation-Equivalent States. In *Int'l. Conf. on Soft. Eng.*, 2010.

[Can] Canonical Ltd. Launchpad: data-races-implementation sql crash. https://bugs.launchpad.net/f-4d-cb/+bug/516622.

[CLL+02] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs. In *PLDI*, pages 258–269, Berlin, Germany, June 2002.

[CR07] Feng Chen and Grigore Roşu. Parametric and sliced causality. In *Proceedings of the 19th international conference on Computer aided verification*, CAV'07, pages 240–253, Berlin, Heidelberg, 2007. Springer-Verlag.

[CSR08] Feng Chen, Traian Florin Serbanuta, and Grigore Rosu. jpredictor: a predictive runtime analysis tool for java. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 221–230. ACM, 2008.

[DDE08] Matthew B. Dwyer, Madeline Diep, and Sebastian Elbaum. Reducing the cost of path property monitoring through sampling. In *Int'l. Conf. on Aut. Soft. Eng.*, pages 228–237, 2008.

[EMBO10] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective data-race detection for the kernel. In *OSDI*, pages 1–16, Vancouver, BC, Canada, October 2010.

[EQT07] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: A Race and Transaction-Aware Java Runtime. In *PLDI*, pages 245–255, San Diego, California, USA, June 2007.

[FF04] Cormac Flanagan and Stephen N Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL*, pages 256–267, Venice, Italy, Januaray 2004.

[FF09] Cormac Flanagan and Stephen N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *PLDI*, pages 121–133, Dublin, Ireland, June 2009.

[FFY08] Cormac Flanagan, Stephen N. Freund, and Jaeheon Yi. Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs. In *PLDI*, pages 293–303, Tucson, AZ, USA, 2008.

[FL12] Stephanie Forrest and Claire LeGoues. Evolutionary software repair. In *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference companion*, GECCO Companion '12, pages 1345–1348, New York, NY, USA, 2012. ACM.

[Fou] Apache Software Foundation. Apache tomcat. http://tomcat.apache.org.

[FQ03] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 338–349. ACM, 2003.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[Har10] Mark Harman. Automated Patching Techniques: The Fix is in: Technical Perspective. *Commun. ACM*, 53:108–108, May 2010.

[HDVT08] Christian Hammer, Julian Dolby, Mandana Vaziri, and Frank Tip. Dynamic Detection of Atomic-Set-Serializability Violations. In *ICSE*, pages 231–240, Leipzig, Germany, May 2008.

[HJM04] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Race Checking by Context Inference. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '04, pages 1–13. ACM, 2004.

[JBM04] Maria Jump, Stephen M. Blackburn, and Kathryn S. McKinley. Dynamic object sampling for pretenuring. In *Proceedings of the International*

Symposium on Memory Management, pages 152–162, Vancouver, BC, Canada, 2004.

[Jik11] Jikes RVM. Jikes research virtual machine. http://jikesrvm.org, 2011.

[JNPS09] Pallavi Joshi, Mayur Naik, Chang-Seo Park, and Koushik Sen. Calfuzzer: An extensible active testing framework for concurrent programs. In *CAV*, pages 675–681, Grenoble, France, June 2009.

[JSZ+11] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. Automated atomicity-violation fixing. In *PLDI*, pages 389–400, San Jose, California, June 2011.

[KLT+07] Bohuslav Krena, Zdenek Letko, Rachel Tzoref, Shmuel Ur, and Tomáš Vojnar. Healing data races on-the-fly. In *Proceedings of the 2007 ACM workshop on Parallel and distributed systems: testing and debugging*, PADTAD '07, pages 54–64, London, United Kingdom, 2007.

[KP04] SecurityFocus Kevin Poulsen. Tracking the blackout bug. http://www.securityfocus.com/news/8412, 2004.

[KYKS09] KyungHee Kim, Tuba Yavuz-Kahveci, and Beverly A. Sanders. Precise Data Race Detection in a Relaxed Memory Model Using Heuristic-Based Model Checking. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 495–499. IEEE Computer Society, 2009.

[KZC12] Baris Kasikci, Cristian Zamfir, and George Candea. Data races vs. data race bugs: telling the difference with portend. In *Proceedings of the seventeenth international conference on Architectural Support for*

Programming Languages and Operating Systems, ASPLOS '12, pages 185–198. ACM, 2012.

[Lam78] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21:558–565, July 1978.

[LCS+10] Brandon Lucia, Luis Ceze, Karin Strauss, Shaz Qadeer, and Hans-J. Boehm. Conflict exceptions: simplifying concurrent language semantics with precise hardware exceptions for data-races. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 210–221, New York, NY, USA, 2010. ACM.

[LGNFW12] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Trans. Softw. Eng.*, 38(1):54–72, January 2012.

[LGWF12] Claire Le Goues, Westley Weimer, and Stephanie Forrest. Representations and operators for improving evolutionary software repair. In *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference*, GECCO '12, pages 959–966, New York, NY, USA, 2012. ACM.

[Lia] Liang T. Chen. The Challenge of Race Conditions in Parallel Programming. http://developers.sun.com/solaris/articles/raceconditions.html.

[Lip75] Richard J. Lipton. Reduction: A Method of Proving Properties of Parallel Programs. *Commun. ACM*, 18(12), December 1975.

[LPSZ08] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ASPLOS*, pages 329–339, Seattle, WA, USA, March 2008.

[LSaD11] Du Li, Witawas Srisa-an, and Matthew B. Dwyer. SOS: saving time in dynamic race detection with stationary analysis. In *OOPSLA*, pages 35–50, Portland, OR, October 2011.

[LT93] N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, July 1993.

[LTQZ06] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. Avio: detecting atomicity violations via access interleaving invariants. In *ASPLOS*, pages 37–48, San Jose, California, March 2006.

[LYC+08] Alexey Loginov, Eran Yahav, Satish Chandra, Stephen Fink, Noam Rinetzky, and Mangala Nanda. Verifying Dereference Safety via Expanding-Scope Analysis. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '08, pages 213–224, New York, NY, USA, 2008. ACM.

[LZ12] Peng Liu and Charles Zhang. Axis: Automatically fixing atomicity violations through solving control constraints. In *ICSE*, pages 299–309, Zurich, Switzerland, June 2012.

[MMN09] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. LiteRace: Effective Sampling for Lightweight Data-Race Detection. In *PLDI*, pages 134–143, Dublin, Ireland, June 2009.

[MMS98] S. P. Midkiff, J. E. Moreira, and M. Snir. Optimizing Array Reference Checking in Java Programs. *IBM Systems Journal*, 37(3):409–453, 1998.

[MSM+10] Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. Drfx: a simple and efficient memory model for concurrent programming languages. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 351–362, New York, NY, USA, 2010. ACM.

[NAW06] Mayur Naik, Alex Aiken, and John Whaley. Effective Static Race Detection for Java. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 308–319, Ottawa, Ontario, Canada, June 2006.

[NWT+07] Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. Automatically classifying benign and harmful data races using replay analysis. In *PLDI*, pages 22–31, San Diego, California, June 2007.

[OC03] Robert O'Callahan and Jong-Deok Choi. Hybrid Dynamic Data Race Detection. In *PPoPP*, pages 167–178, San Diego, California, USA, 2003.

[Par11] Ben Parr. Learning from amazon's cloud collapse. http://www.cnn.com/2011/TECH/web/04/22/amazon.cloud.mashable/index.html, 2011.

[PDE10] R. Purandare, M.B. Dwyer, and S. Elbaum. Monitor Optimization via Stutter-Equivalent Loop Transformation. In *Proceedings of the Conference on Object-Oriented Programming Systems Languages and Applications*, pages 270–285, 2010.

[PK11] Georgios Portokalidis and Angelos D. Keromytis. Reassure: a self-contained mechanism for healing software using rescue points. In *Proceedings of the 6th International conference on Advances in information and computer security*, IWSEC'11, pages 16–32, Berlin, Heidelberg, 2011. Springer-Verlag.

[PKL+09] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *SOSP*, pages 87–102, Big Sky, Montana, USA, October 2009.

[PMBM08] Roberto Paleari, Davide Marrone, Danilo Bruschi, and Mattia Monga. On race vulnerabilities in web applications. In *Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '08, pages 126–142, Berlin, Heidelberg, 2008. Springer-Verlag.

[PS03] Eli Pozniansky and Assaf Schuster. Efficient On-the-Fly Data Race Detection in Multithreaded C++ Programs. In *PPoPP*, pages 179–190, San Diego, California, USA, 2003.

[PS07] Eli Pozniansky and Assaf Schuster. MultiRace: Efficient on-the-fly Data Race Detection in Multithreaded C++ programs: Research Articles. *Concurrency and Computation: Practice and Experience - Parallel and Distributed Systems: Testing and Debugging (PADTAD)*, 19:327–340, March 2007.

[PS08] Chang-Seo Park and Koushik Sen. Randomized Active Atomicity Violation Detection in Concurrent Programs. In *FSE*, pages 135–145, Atlanta, Georgia, November 2008.

[RZKW08] Ian Rogers, Jisheng Zhao, Chris Kirkham, and Ian Watson. Constraint Based Optimization of Stationary Fields. In *Proceedings of the International Symposium on Principles and Practice of Programming in Java*, PPPJ '08, pages 95–104, New York, NY, USA, 2008. ACM.

[SAM10] Hesam Samimi, Ei Darli Aung, and Todd Millstein. Falling back on executable specifications. In *ECOOP*, pages 552–576, June 2010.

[SBN+97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15:391–411, November 1997.

[SES+12] Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. Sound Predictive Race Detection in Polynomial Time. In *POPL*, pages 387–400, Philadelphia, PA, January 2012.

[SFW10] Eric Schulte, Stephanie Forrest, and Westley Weimer. Automated program repair through the evolution of assembly code. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 313–316, New York, NY, USA, 2010. ACM.

[SG08] Silberschatz and Galvin. *Operating System Concepts, 8th Edition*. Addison Wesley, 2008.

[SRA06] Koushik Sen, Grigore Rosu, and Gul Agha. Online efficient predictive safety analysis of multithreaded programs. *Int. J. Softw. Tools Technol. Transf.*, 8:248–260, June 2006.

[Sta05] Standard Performance Evaluation Corporation. SPECjbb2005. On-Line Documentation, 2005. http://www.spec.org/jbb2005.

[SY86] R E Strom and S Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, 1986.

[UL08] Christopher Unkel and Monica S. Lam. Automatic Inference of Stationary Fields: A Generalization of Java's Final Fields. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 183–195, San Francisco, California, USA, January 2008.

[Unk09] Christopher Unkel. *Stationary Fields in Object-Oriented Programs*. PhD thesis, Stanford University, Stanford, CA, USA, 2009.

[VJL07] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. RELAY: Static Race Detection on Millions of Lines of Code. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The foundations of Software Engineering*, pages 205–214, Dubrovnik, Croatia, 2007.

[vPG03] Christoph von Praun and Thomas R. Gross. Static conflict analysis for multi-threaded object-oriented programs. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 115–128, New York, NY, USA, 2003. ACM.

[VTD06] Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL*, pages 334–345, Charleston, South Carolina, January 2006.

[WCY10] Jingyue Wu, Heming Cui, and Junfeng Yang. Bypassing races in live applications with execution filters. In *OSDI*, pages 1–13, Vancouver, BC, Canada, October 2010.

[WFLGN10] Westley Weimer, Stephanie Forrest, Claire Le Goues, and ThanhVu Nguyen. Automatic program repair with evolutionary computation. *Commun. ACM*, 53(5):109–116, May 2010.

[WNLGF09a] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *ICSE*, pages 364–374, Vancouver, Canada, May 2009.

[WNLGF09b] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE ’09, pages 364–374, Washington, DC, USA, 2009. IEEE Computer Society.

[WS06] Liqiang Wang and Scott D. Stoller. Runtime Analysis of Atomicity for Multithreaded Programs. *IEEE Trans. Softw. Eng.*, 32(2):93–110, February 2006.

[XSaJ08] Feng Xian, Witawas Srisa-an, and Hong Jiang. Contention-Aware Scheduler: Unlocking Execution Parallelism in Multithreaded Java Programs. In *Proceedings of the Conference on Object-Oriented*

Programming Systems Languages and Applications, pages 163–180, Nashville, TN, USA, October 2008.

[YCSS12] Junfeng Yang, Ang Cui, Sal Stolfo, and Simha Sethumadhavan. Concurrency attacks. In *Proceedings of the 4th USENIX conference on Hot Topics in Parallelism*, HotPar'12, pages 15–15, Berkeley, CA, USA, 2012. USENIX Association.

[YRC05] Yuan Yu, Tom Rodeheffer, and Wei Chen. RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. In *SOSP*, pages 221–234, Brighton, United Kingdom, 2005.

[Zen03] Fancong Zeng. Deadlock resolution via exceptions for dependable java applications. In *DSN*, pages 731–740, San Francisco, CA, June 2003.

[ZM04] Fancong Zeng and Richard P. Martin. Ghost locks: Deadlock prevention for Java. *Mid-Atlantic Student Workshop on Programming Languages and Systems*, pages 1–6, April 2004.

[ZSL10] Wei Zhang, Chong Sun, and Shan Lu. Conmem: detecting severe concurrency bugs through an effect-oriented approach. In *ASPLOS*, pages 179–192, Pittsburgh, Pennsylvania, USA, March 2010.