A Comprehensive Framework to Replicate Process-Level Concurrency Faults

Supat Rattanasuksun
University of Nebraska-Lincoln, supat609@gmail.com

Follow this and additional works at: http://digitalcommons.unl.edu/computerscidiss
Part of the Computer Engineering Commons, and the Computer Sciences Commons

http://digitalcommons.unl.edu/computerscidiss/160

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.
A COMPREHENSIVE FRAMEWORK TO REPLICATE PROCESS-LEVEL CONCURRENCY FAULTS

by

Supat Rattanasuksun

A DISSERTATION

Presented to the Faculty of The Graduate College at the University of Nebraska In Partial Fulfillment of Requirements For the Degree of Doctor of Philosophy

Major: Computer Science

Under the Supervision of Witawas Srisa-an and Gregg Rothermel

Lincoln, Nebraska

November, 2018
Concurrency faults are one of the most damaging types of faults that can affect the dependability of today’s computer systems. Currently, concurrency faults such as process-level races, order violations, and atomicity violations represent the largest class of faults that has been reported to various Linux bug repositories. Clearly, existing approaches for testing such faults during software development processes are not adequate as these faults escape in-house testing efforts and are discovered during deployment and must be debugged.

The main reason concurrency faults are hard to test is because the conditions that allow these to occur can be difficult to replicate, causing them to appear non-deterministically. Once these faults have been discovered during deployment and reported back to engineers, they are still very challenging to reproduce for the same reason. Furthermore, since concurrency faults can be complex, it is difficult for users to diagnose faults correctly. This can lead to bug reports that do not contain sufficient information or are totally incorrect.

The goal of this dissertation is to make the process of reproducing concurrency faults more effective and efficient. Effectiveness means that we can reproduce faults more deterministically, and engineers can continue to debug applications in spite of incomplete reports. Efficiency means that using our proposed approaches, engineers take less time to perform the debugging process. This includes less time to develop
detectors, less time to identify applications that can instigate reported faults, and less time to run the applications to reproduce reported faults. The results of our empirical evaluations reveal that the proposed systems collectively allow concurrency fault reproduction to be more effective, efficient, and accurate.
ACKNOWLEDGMENTS

The completion of this dissertation would not be possible without the financial support from Bangkok University. I am proud and honored to be working at this institution. Specially, I would like to thank BU Presidents, Petch Osathanugrah, Former BU President Dr. Thanu Kulachol, Former BU President Dr. Mathana Santiwat, Former BU Vice President Assistant Professor Somchit Likhittaworn, BU Executive Vice President for Academic Affairs Associate Professor Dr. Tiparatana Wongcharoenm, BU Vice President for Academic Affairs Associate Professor Dr. Aunya Singsangob, Vice President for Administrative Affairs Dr. Sountaree Rattapasakorn, all of the BU Executives and Deans from School of Science and Information Technology for giving me this opportunity allowing me to join this doctoral program.

To complete this dissertation, most importantly, I would like to thank my advisors, Associate Professor Dr. Witawas Srisa-an and Professor Dr. Gregg Rothermel for patiently guiding, teaching, and training me to do the researches academically and professionally. They always support me in every step in the past years. Thanks for my supervisory committee Assistant Professor Dr. ThanhVu H. Nguyen and Associate Professor Dr. Peter Angeletti. I also want to thanks for the best mentor, advisor, and good friend, Assistant Professor Dr. TingTing Yu. Also, I would like to thanks to Professor Dr. Matt Dwyer for accepting me to join the UNL years ago and introducing me to the static analysis.

I’m also grateful to all of Professors and friends from ESQuaReD Lab, my friends from Nonsee, my first programming teacher Assistant Professor Dr. Kittpat Save-tratanakaree, Former UMT Executives for the scholarship and my ELI teachers at UMT, my friends and colleagues from BU, my friends from TANU, and to my friends from Nebraska, from Montana, from Thailand, and from many countries. Thanks
to my buddy friends Thammasak Thianniwet and Zhiqiang Li. Specially thanks to Professor Dr. Henry Joel at the UMT for your software engineering guidance and belief in me, my former advisor and first machine learning teacher Professor Dr. David Optiz, and the former chairperson of CS Department at the UMT Emeritus Professor Dr. Alden Wright. Also thanks to my dearest friend, Ken.

Finally, thanks to my aunts, my uncles, my cousins, and all of my relatives for all types of supports. And for the most important persons in my life, I want to thank my parents, my sister, and my brother. They have been supporting me in the past years and always be there for me. I love you all very much. My dad once told me "Study as much as you can and you want." and I did as you said, Pa. I miss you and I’m going back home soon.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>vi</td>
<td>Contents</td>
<td>vi</td>
</tr>
<tr>
<td>xi</td>
<td>List of Figures</td>
<td>xi</td>
</tr>
<tr>
<td>xiii</td>
<td>List of Tables</td>
<td>xiii</td>
</tr>
<tr>
<td>1</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1</td>
<td>Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>Approaches</td>
<td>6</td>
</tr>
<tr>
<td>1.3</td>
<td>Contributions</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>General Background</td>
<td>9</td>
</tr>
<tr>
<td>2.1</td>
<td>Definitions</td>
<td>9</td>
</tr>
<tr>
<td>2.1.1</td>
<td>Race</td>
<td>9</td>
</tr>
<tr>
<td>2.1.2</td>
<td>Atomicity Violation</td>
<td>10</td>
</tr>
<tr>
<td>2.1.3</td>
<td>Contention</td>
<td>10</td>
</tr>
<tr>
<td>2.1.4</td>
<td>Deadlock</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>Distinction between Race and Atomicity Violations</td>
<td>11</td>
</tr>
<tr>
<td>2.3</td>
<td>Reproducing Races and Atomicity Violations at the Process Level</td>
<td>12</td>
</tr>
<tr>
<td>2.4</td>
<td>Race and Atomicity Violation Detection Techniques</td>
<td>13</td>
</tr>
</tbody>
</table>
3 Exploring Real-World Executable Concurrency Faults 15
3.1 Application Selection Process ............................................. 15
3.2 Selected Applications .......................................................... 17
  3.2.1 Under-constrained Process Interaction Faults ...................... 18
  3.2.2 Over-constrained Process Interaction Faults ...................... 21

4 RCRF: A Reproduction Framework for Use in Debugging Races,
Deadlocks, and Excessive Synchronization Delays 23
4.1 Introduction ........................................................................... 23
4.2 Background .......................................................................... 24
  4.2.1 Motivation ......................................................................... 24
  4.2.2 Achieving Observability and Controllability ...................... 26
4.3 Introducing RCRF ................................................................. 28
  4.3.1 Identifying Relevant System Events ................................... 32
  4.3.2 Mapping System-level Events to Source Code ................... 35
  4.3.3 PuD Modification ............................................................... 35
  4.3.4 Fault Reproduction ........................................................... 38
  4.3.5 Fault Validation ............................................................... 38
4.4 Empirical Study ................................................................. 40
  4.4.1 Objects of Analysis ............................................................ 41
  4.4.2 Variables and Measures .................................................... 41
  4.4.3 Study Methodology ........................................................... 42
  4.4.4 Threats to Validity ............................................................ 43
  4.4.5 Results ............................................................................... 44
4.5 Discussion ............................................................................. 47
  4.5.1 Additional Analysis ........................................................... 47
5.6 Work Related to RDSC ............................................. 78
5.7 Conclusion .......................................................... 80

6 CFI: Concurrency Faults Inspector ................................... 81
6.1 Introduction .......................................................... 81
6.2 Background .......................................................... 84
   6.2.1 Analyzing Incomplete Bug Reports ......................... 84
   6.2.2 Converting A Happen-Before Graph to Feature Matrix . 88
6.3 Introducing Concurrency Fault Inspector (CFI) ................. 90
   6.3.1 COLLECTOR: Identify Race-Inducing Applications and Relevant System Events ............................................. 94
   6.3.2 SYNTHESIZER: Generating the VIs and Synthesizing Datasets for Classifier .................................................. 95
   6.3.2.1 VIRTUAL INTERLEAVINGS GENERATOR (VIG) .... 96
   6.3.2.2 DATASET SYNTHESIZER (DS) .................. 97
   6.3.3 TRAINER: Training and Testing the Classifier, then Predicting the Results of VIs ........................................ 100
   6.3.4 VERIFIER: Confirming the Presence of Faults ............ 103
6.4 A Case Study ......................................................... 104
   6.4.1 Objects of Analysis ............................................. 104
   6.4.2 Variables and Measures ....................................... 106
   6.4.3 Study Methodology ............................................. 106
   6.4.4 Threats to Validity ............................................. 107
   6.4.5 Results .......................................................... 107
6.5 Discussion ............................................................ 112
6.6 Related Work ....................................................... 113
6.7 Conclusion .......................................................... 114

7 Conclusions and Future Work ................................. 116

Bibliography ......................................................... 118
List of Figures

2.1 A typical file access with proper synchronization and an atomicity violation 12

4.1 Workflow in RCRF .......................... 29
4.2 Sample Scribe output .......................... 32
4.3 Mapping a block of system calls to a line of source code in UPDATEDB 34
4.4 A modified UPDATEDB source file with injected sleep calls 39

5.1 The control flow graph of a function "Average" .......................... 62
5.2 Sequential flow code to calculate the hypotenuse in the Pythagorean theorem 65
5.3 Using RDSC to detect atomicity violations in system call logs 67

6.1 Bug report from MySQL number 36537 .......................... 86
6.2 Bug report from SLIDEMATCH number 641521 .......................... 87
6.3 Relationship between nodes in a Happen-before graph .......................... 89
6.4 Labeling the feature from the happen-before graph .......................... 89
6.5 Initializing the feature matrix .......................... 90
6.6 The feature matrix represents the happen-before graph .......................... 91
6.7 Workflow in CFI .......................... 92
6.8 Workflow in The COLLECTOR .......................... 95
6.9 Workflow in The SYNTHESIZER .......................... 96
6.10 Examples of Virtual interleavings (VIs) ........................................ 96
6.11 Example of a dataset from running every interleaving between LOCATE
and UPDATEDB ........................................................................... 98
6.12 Transforming the feature matrix and evaluating the output from the oracle
into a line of dataset .................................................................... 99
6.13 Workflow in TRAINER ............................................................... 100
6.14 Workflow in The VERIFIER ..................................................... 103
6.15 The example of virtual interleavings generated from LOCATE-UPDATEDB 112
6.16 The log file from executing the interleavings from LOCATE-UPDATEDB . . 113
List of Tables

3.1 Basic Characteristics of Object Programs .................................. 17

4.1 Empirical Results ............................................................... 45
4.2 Optimization of Numbers of System Calls ................................. 48
4.3 Interleavings That Can Cause Reported Races ............................. 49
4.4 Additional Fault Types Found ............................................... 49
4.5 Results from RacePro .......................................................... 51
4.6 Numbers of Targeted Races and Incidental Races ....................... 53

5.1 Gen() and Kill() for Each Line of Source Code ......................... 63
5.2 Entry() and Exit() for Each Line of Source Code ....................... 63
5.3 Gen() and Kill() for Each Line in a Sequential Program .............. 65
5.4 Entry() and Exit() for Each Line in a Sequential Program .......... 65
5.5 Entry() and Exit() Focusing on the Variable "c" Only .................. 66
5.6 Entry() and Exit() Focusing on the Variable "Data" Only ............... 66
5.7 Comparing Dynamically Detected Faults Between RCRF and RDSC .... 73
5.8 Comparing Unique Faults Between RCRF and RDSC .................... 74
5.9 List of Oracle Numbers and Names Used in RCRF and RDSC .......... 76
5.10 Results Comparing Oracles Used Between RCRF and RDSC .......... 77
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 Comparing the Results of Classification Algorithms</td>
<td>101</td>
</tr>
<tr>
<td>6.2 Predicting the VIs of LOCATE-BACKUP using classifiers from different algorithms</td>
<td>102</td>
</tr>
<tr>
<td>6.3 Universe of Applications</td>
<td>105</td>
</tr>
<tr>
<td>6.4 The applications after refinement</td>
<td>105</td>
</tr>
<tr>
<td>6.5 Effectiveness of CFI</td>
<td>108</td>
</tr>
<tr>
<td>6.6 Prediction Accuracy of CFI</td>
<td>109</td>
</tr>
<tr>
<td>6.7 Efficiency of CFI</td>
<td>110</td>
</tr>
<tr>
<td>6.8 Efficiency of CFI</td>
<td>111</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Motivation

Concurrency faults are difficult to detect and debug because they are sensitive to execution interleavings. *Process-level races* and *atomicity violations*, where multiple processes access operating system resources without proper synchronization and protection, are particularly challenging to handle. A process-level race occurs when multiple processes access a system-wide shared resource (e.g., file, device, hardware register) without proper synchronization. Atomicity violations occur when there is execution interference by two or more processes or threads on a block of statements that must be executed atomically. A study of real world concurrency fault characteristics [1] revealed that atomicity violations are the most common causes of concurrency faults. Prior work also found that *under-constrained process interactions* can lead to both issues [2].

There are existing race and atomicity violation detectors that have been designed for a specific language or to operate at the thread level. However, they may not be able to detect and reproduce such faults due to heterogeneous environments. This is
because races and atomicity violations at the process level can occur across different applications written in different languages and accessing diverse resources. In addition, detection and reproduction of races or atomicity violations require observability of shared memory accesses and synchronization operations. Furthermore, process-level races and atomicity violations can occur in kernel services (e.g., system call routines and device drivers) that are not observable by software engineers. Finally, these faults can occur at times well past system deployment due to changes in system configurations. Because the scenarios in which process-level races and atomicity violations can occur are complex, these faults often elude in-house testing efforts and remain undiscovered until after system deployment. For example, up to 73% of all races reported after deployment in various Linux distributions are process-level races [3].

To avoid these faults (in this proposal, we simply refer to process-level races and atomicity violations as “faults”), developers often use various synchronization primitives (e.g., semaphores, spin-locks, barriers) to synchronize accesses to shared resources. In effect, these mechanisms allow only one process to access a shared resource at a time, and thus, prevent such faults from occurring. However, serializing accesses to shared resources can also result in over-constrained process interactions [2], leading to two additional problems. First, adding synchronization primitives can lead to deadlocks or livelocks [4]. Second, serially accessing shared resources can lead to excessive lock contention, resulting in long delays for processes that try to access a shared resource while it is being accessed by another process. A prior study has shown that lock contention can slow down an access to a shared resource by between 10 times (on a single CPU system) and 50 times (on a multiprocessor system) [5]. While such contention may not produce incorrect results, these delays can lead to various types of Quality of Experience (QoE) faults ranging from minor annoyances such as long
delays in shutting a system down [6,7,8,9,10,11] to intermittent system reboots due to timing violations [12,13,14,15]).

When faults due to over- or under-constrained process interactions are discovered, they are typically reported to engineers. Examples of bug reports describing such faults can be found in various bug repositories including those for Linux distributions such as Red Hat, Ubuntu and Debian. In most cases, bug reports include only general descriptions of symptoms, while in other cases, bug reports are more detailed and provide inputs that cause the faults. Engineers use this information to identify potential sources of faults (e.g., file systems, network stacks, or device drivers) and attempt to reproduce them as part of the debugging process. Unfortunately, reported information alone is often not sufficient to reproduce faults. For example, we have encountered cases in which 50,000 execution instances cannot reproduce a reported fault due to the small execution interleaving windows that must occur in order for the faults to manifest themselves.

In practice, when attempting to reproduce faults, engineers often use sleep or sched_yield calls to pause the execution of pairs of processes [16,17], to amplify the chances of enforcing execution windows that lead to those faults. Inserting yield points at the source code level, however, is an error-prone approach when dealing with such faults. Shared resource usage can be implicit; that is, an application may make a request to use a shared resource indirectly by invoking a system call that invokes another system call to access the resource. Therefore, engineers may miss the opportunity to observe the behavior of an implicit system call if they fail to insert a yield point prior to the event that can result in that call. While a record and replay approach [3] has been used to detect concurrency faults such as races and atomicity violations, engineers may fail to observe problems due to replay divergence—a mismatch between the actual actions recorded in a trace and the modified actions
intended to enforce execution windows. Replay divergence occurs when engineers directly modify a trace of a program’s execution and cannot replay it after the modifications [3].

It is worth noting that over the past decade, much research has been conducted on techniques that facilitate detection of concurrency faults, and especially races (e.g., [18,19,20,21,22,23,24,25]). These techniques can dynamically observe events of interest and then analyze information collected from those observations to determine whether races occur. However, most existing techniques focus on multi-threaded applications within single processes, and have not been adapted to handle process-level race conditions such as races caused by improper shared resource accesses among multiple processes, software signals and hardware interrupts.

Synchronization related faults can also involve more than two pairs of shared resource accesses. In such cases, to localize faults based on a reported symptom, engineers need to first identify applications that can access the same shared resources. Often, a bug report includes the application that encountered the fault but not the other applications that access the same resources, causing concurrency faults to occur. (We will tackle this challenge in our third approach) In the case in which engineers are somehow able to identify these applications, they then need to identify code locations that share a resource and then observe the behavior of each pair-wise access to the shared resource. (A pair-wise access is the minimum requirement for exposing a race or atomicity violation [1].) If engineers fail to exercise any pair-wise accesses, they may be unable to reproduce the reported races, atomicity violations, or resource contention, and also unable to identify other locations that can possibly cause additional faults on the same shared resource.

To effectively and efficiently reproduce faults, it follows that engineers need a bug reproduction framework that allows them to effectively debug faults by observing system-
level events such as system calls and signals that occur during program execution, without encountering replay divergence. This will allow them to uncover both explicit and implicit system calls that might be relevant to the faults they are trying to reproduce. They also need an approach for automatically mapping system-level events to invocation points in source code and inserting yield points prior to and after those invocation points. They then need to be able to guide the interleaving execution toward reported faults.

Once a bug reproduction system is in place, the second challenge in race reproduction is for engineers to develop detectors that can detect the presence of targeted faults when trying to debug two or more applications. The current approach is to develop oracles; this can be laborious as the nature of these faults may receive engineers to develop a customized detector for each type of race or atomicity violation. In our empirical work, to reproduce races in 20 pairs of applications, we needed to develop 16 detectors. As developers try to debug more races, the collection of detectors would undoubtedly grow larger. To simplify this process, it would be useful to develop a generic detector that can address different classes of races.

A third challenge in race or atomicity violation reproduction is that a bug report may not provide sufficient information to identify the pair of applications that can race. Instead, the report may provide the race symptoms experienced by an application but fail to identify the other application(s) that could race with it. To reproduce such races, it is critical for engineers to identify applications that can possibly race with the program under debug (PuD). Thus, we need a system that can help an engineer identify applications that can be used to instigate a reported fault within a universe of applications that are accessible by the engineer.
1.2 Approaches

To effectively reproduce concurrency faults, we introduce three approaches to address the aforementioned challenges. To address the first challenge, we introduce a reproduction framework for use in debugging races, deadlocks, and excessive synchronization delays (RCRF), a race reproduction framework to assist with debugging races, deadlocks, and excessive synchronization delays. RCRF is a partially automated framework that identify accesses to the same shared resources. Our approach uses Linear Temporal Logic (LTL) as oracles to determine the faults. To enable observability and controllability, the framework uses static analysis tools to inject delays to create interleavings between the processes that are likely to result in races.

To address the second challenge, we extend the framework by statically analyzing applications containing previously reported faults. We found that atomicity violations occur frequently in these applications. It is worth noting that our observation also corresponds to a prior result reported by Lu et al. [1]. As such, we extend RCRF to also incorporate an atomicity detection algorithm based on reaching definition (RD) analysis in addition to LTL. We refer to this algorithm as "reaching definition on system call logs" (RDSC). The system call logs are generated by running applications. Because many types of races are the results of atomicity violations, we anticipate that RDSC can also help reduce the developer’s efforts in constructing customized detectors to meet the specifications of particular reported races. This can further simplify the race reproduction process.

To address the third challenge, we leverage our insights into various types of process-level races so that we can classify various patterns of races that can help us determine whether races can occur between a pair of applications. Based on this insight, we propose our third approach; which helps software engineers identify applications
that can interact with each other. Our approach leverages existing machine learning algorithms to construct classifiers that can determine whether an arbitrary application can race with a PuD based on resource usage analysis and features.

1.3 Contributions

The intended contributions for this proposed works are as follows:

1. We present RCRF, a reproduction framework for use in debugging races, deadlocks, and excessive synchronization delays. RCRF can help software engineers reproduce reported process-level races, deadlocks, and quality of experience (QoE) faults that occur because of excessive delays due to contention, enabling them to potentially debug these faults. RCRF performs a hybrid analysis by leveraging existing static program analysis tools, dynamic kernel event reporting tools, and yield points to provide the observability and controllability needed to reproduce these faults. We conducted an empirical study to evaluate RCRF; our results show that RCRF can be effective for reproducing our targeted synchronization related faults. The detail of this framework is proposed in Chapter 4.

2. We develop RDSC by extending RCRF to include a generic detector based on reaching definitions analysis (RD) to detect races and atomicity violations. RD is a data flow analysis on the source code of a program to track the values of variables flowing on the execution paths. Compilers use RD to check for uninitialized, and unused variables. We apply RD to the trace of dynamic kernel event logs of concurrency processes. The interleavings between processes may violate atomicity. The definitions obtained with RD can help us find the
origins of races. We present the algorithm, empirical study, and results of this work in Chapter 5.

3. For the last part of this work, we develop CFI, a concurrency fault classification to analyze patterns of real-world process-level races and atomicity violations. Because these faults occur at deployment across shared resources, it is possible for any arbitrary pair of applications to race. We envision that our proposed classification framework can help engineers reproduce faults when the bug report does not provide sufficient details to do so. Our system identifies potential faults that can occur between a universe of applications that can interact with our PuD on the same system, and therefore, increase the effectiveness of RCRF. We present the algorithm, empirical study, and results of this work in Chapter 6.
Chapter 2

General Background

In this section, we provide relevant background information related to races and atomicity violations and their reproduction. However, we first define common types of faults that occurs due to improper concurrency management.

2.1 Definitions

2.1.1 Race

We define a process-level race as a race that occurs when (1) two processes access a shared resource, and (2) they could have accessed the shared resource in an order different from the original order. This definition of a race is broader than the standard definition [17,26]. Our definition also includes order violations, in which the desired order of shared resource accesses is reversed and the accesses may or may not be protected by a common lock [1,27]. An order violation is a necessary but not sufficient condition for a race.
2.1.2 Atomicity Violation

Atomicity is a property that governs execution of a block of code \((B)\) consisting of a sequence of instructions in a program. Atomicity is preserved if the outcome of concurrently executing \(B\) is equivalent to the outcome of serially executing \(B\). If this property is not preserved, atomicity is violated \([4,28]\).

2.1.3 Contention

We define synchronization contention as an event that occurs when one or more processes try to access a shared resource that is already being accessed by a process, and that particular process holds the exclusive right to that resource at that time. As such, other processes attempting to access the resource would either be suspended until the resource is available or would repeatedly try to access the resource ("spin") until they are successful. A synchronization delay refers to the amount of time that a process must wait, either through suspension or spinning, to access the currently claimed shared resource \([4]\). With contention, there are no detectable output faults. However, delays may result in functional faults that can cause a system to prematurely terminate \([12]\) or non-functional faults that can degrade performance \([29,30]\). We refer to this type of fault as a Quality of Experience (QoE) fault.

2.1.4 Deadlock

Our definition of deadlock follows the common definition: deadlock occurs when two or more processes are each waiting for the other to release a shared resource \([4]\).
2.2 Distinction between Race and Atomicity Violations

In the context of our work, we focus on detecting races and atomicity violations that occur on resources shared by multiple processes. Some of these resources include data structures maintained by the operating system such as file and signal data structures which can be accessed via system calls. To identify potential sources of races, we need to first model system calls that can access resources of interest. Identifying a resource of interest is based on a fault description such as these that can be found in a bug repository. We then consider each system call that can access a particular resource of interest as a basic operation that can cause races.

With respect to atomicity violations, we consider a sequence of system calls that must be performed atomically to preserve correctness. As an example, a typical file access requires a number of system calls (shown in Figure 2.1) that must be performed atomically to preserve correctness. An atomicity violation can occur if another process (P1) can modify the intermediate state of the file data structure while a process (P0) is trying to access the file. This can result in an unstable intermediate state. On the other hand, if we are trying to identify the source of a race that can result in corruption of an output file, we may focus only on the write system call as it is the only one that performs the write to the output file.
Properly atomic process

P0
open
fstatat64
read
close
open
fstatat64
read
close
.
.
.

P1
open
fstatat64
read
close
open
fstatat64
read
.
.
.

Atomicty violation

P0
open
write
open
fstatat64
read
close
stat64
open
fstatat64
read
close
open
close
fstatat64
.
.

Figure 2.1: A typical file access with proper synchronization and an atomicity violation

2.3 Reproducing Races and Atomicity Violations at the Process Level

In practice, when attempting to reproduce faults, engineers often use sleep or sched_yield calls to pause the execution of pairs of processes [16, 17], to increase the chances of enforcing execution windows that lead to those faults. Inserting yield points at the source code level, however, is an error-prone approach when dealing with such faults. Shared resource usage can be implicit; that is, an application may make a request to use a shared resource indirectly by invoking a system call that invokes another system call to access the resource. Engineers may miss the opportunity to observe the behavior of an implicit system call if they fail to insert a yield point prior to the event that can result in that call. While a record and replay approach [3] has been used to detect process-level races, engineers may fail to observe problems due to replay divergence – a mismatch between the actual actions recorded in a trace and the modified actions intended to enforce execution windows. Replay divergence occurs
when engineers directly modify a trace of a program’s execution and cannot replay it after the modifications [3].

2.4 Race and Atomicity Violation Detection

Techniques

It is worth noting that over the past decade, much research has been conducted on techniques that facilitate race and atomicity violation detection (e.g., [18, 19, 20, 21, 22, 23, 24, 25, 28, 28, 31, 32, 33, 34, 35]). These techniques can dynamically observe events of interest and then analyze information collected from those observations to determine whether races occur. However, most existing techniques focus on multi-threaded applications within single processes, and have not been adapted to handle process-level race conditions such as races caused by improper shared resource accesses among multiple processes, software signals and hardware interrupts.

Synchronization related faults can also involve more than two pairs of shared resource accesses. In such cases, to localize races based on a reported symptom, engineers need to identify locations that share the same resource and then observe the behavior of each pair-wise access to the shared resource. (A pair-wise access is the minimum requirement for exposing a race [1].) If engineers fail to exercise any pair-wise accesses, they may not only be unable to reproduce the reported races or contention, but also unable to identify other locations that can possibly cause additional faults on the same shared resource.

It follows that engineers need a bug reproduction framework that allows them to effectively debug faults by observing system-level events such as system calls and signals that occur during program execution, without encountering replay divergence.
This will allow them to uncover both explicit and implicit system calls that might be relevant to the faults they are trying to reproduce. They also need an approach for automatically mapping system-level events to invocation points in source code and inserting yield points prior to and after these invocation points. They then need to be able to guide the interleaving execution toward reported faults.
Chapter 3

Exploring Real-World Executable

Concurrency Faults

This chapter describes the process we took to explore real-world concurrency faults and collect real-world executable applications to be used to evaluate our proposed systems.

3.1 Application Selection Process

To gain a better understanding of real-world concurrency faults and collect experimental objects that can be used in our studies, we needed to locate applications that meet the following three criteria:

1. There must be known faults in these applications and basic descriptions of those faults.

2. The faults must be within the scope of the classes of faults and the types of shared resources we want to target.
3. We must be able to execute the applications and modify them according to our experimental processes.

To select applications we turned to several bug report repositories, including those for Red Hat, Ubuntu, Debian, and GNU, and searched them using the keywords “race”, “concurrency”, “file”, “signal”, “socket”, “contention”, “deadlock”, “priority inversion”, “quality of service”, and “glitches”. With respect to collecting objects with QoE faults, we focused on multimedia applications as they are likely to suffer from QoE faults such as runtime glitches, long start-up or shut-down delays, or excessive delays between playing media files. We considered only bug reports related to code written in C, C++, or shell scripting language.

For each case considered, we analyzed the bug reports to determine what pairs of programs were involved in causing the reported faults. We then narrowed our focus further to bug reports that involve resources that we want to focus on. These resources include files or network sockets that are shared among multiple processes. We did this because in practice, detecting races on a particular type of resource requires that we be able to observe accesses to that resource type. In this work, we focus only on types of shared resources in which we can commonly find faults. However, we design our proposed solutions as frameworks so they can also be extended to generically cover other types of shared resources.

Finally, we replicated the environments needed to potentially reproduce these faults by creating shell scripts and processes that can possibly exploit these faults (e.g., if a fault is suspected to be due to a signal, we create a program that sends the signal based on our configured frequency).
### Table 3.1: Basic Characteristics of Object Programs

<table>
<thead>
<tr>
<th>Program 1</th>
<th>Program Pair</th>
<th>Program 2</th>
<th>LoC</th>
<th>Type of Races</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPDATEDB (S)</td>
<td>14,405</td>
<td>UPDATEDB (S)</td>
<td>14,405</td>
<td>Improper Synchronization on share resources</td>
</tr>
<tr>
<td>LOCATE (C)</td>
<td>14,643</td>
<td>UPDATEDB (S)</td>
<td>14,405</td>
<td>Improper Synchronization on share resources</td>
</tr>
<tr>
<td>BASH (S)</td>
<td>144,536</td>
<td>BASH (S)</td>
<td>144,536</td>
<td>Improper Synchronization on share resources</td>
</tr>
<tr>
<td>MV (C)</td>
<td>47,023</td>
<td>RM (C)</td>
<td>45,850</td>
<td>Improper Synchronization on share resources</td>
</tr>
<tr>
<td>RM(FTS) (C)</td>
<td>69,524</td>
<td>SYMLINK (S)</td>
<td>3</td>
<td>TOCTTOU and improper link resolution before file access</td>
</tr>
<tr>
<td>MKDIR (C)</td>
<td>29,095</td>
<td>CLEANER (C)</td>
<td>13</td>
<td>TOCTTOU and improper link resolution before file access</td>
</tr>
<tr>
<td>MKNOD (C)</td>
<td>29,307</td>
<td>CLEANER (C)</td>
<td>13</td>
<td>TOCTTOU and improper link resolution before file access</td>
</tr>
<tr>
<td>MKFIFO (C)</td>
<td>28,972</td>
<td>CLEANER (C)</td>
<td>13</td>
<td>TOCTTOU and improper link resolution before file access</td>
</tr>
<tr>
<td>MKDIR2 (C)</td>
<td>35,970</td>
<td>CLEANER (C)</td>
<td>13</td>
<td>TOCTTOU and improper link resolution before file access</td>
</tr>
<tr>
<td>MKNOD2 (C)</td>
<td>36,201</td>
<td>CLEANER (C)</td>
<td>13</td>
<td>TOCTTOU and improper link resolution before file access</td>
</tr>
<tr>
<td>MKFIFO2 (C)</td>
<td>35,828</td>
<td>CLEANER (C)</td>
<td>13</td>
<td>TOCTTOU and improper link resolution before file access</td>
</tr>
<tr>
<td>LN (C)</td>
<td>56,978</td>
<td>CLEANER (C)</td>
<td>13</td>
<td>TOCTTOU and improper link resolution before file access</td>
</tr>
<tr>
<td>FIND (C)</td>
<td>22,929</td>
<td>CLEANER2 (C)</td>
<td>8</td>
<td>TOCTTOU and improper resolution before file access</td>
</tr>
<tr>
<td>CSPLIT (C)</td>
<td>29,905</td>
<td>MSIGSIXER (C)</td>
<td>46</td>
<td>Signal Handler Race Condition</td>
</tr>
<tr>
<td>PXZ (C)</td>
<td>407</td>
<td>STEALER (S)</td>
<td>6</td>
<td>Bypass a restriction or privilege escalation</td>
</tr>
<tr>
<td>LOGROTATE (C)</td>
<td>3,722</td>
<td>STEALER2 (S)</td>
<td>4</td>
<td>Bypass a restriction or privilege escalation</td>
</tr>
<tr>
<td>PS (C)</td>
<td>110,761</td>
<td>GREP_SCRIPT (S)</td>
<td>5</td>
<td>Inevitable race on a shared pipe between processes</td>
</tr>
<tr>
<td>MAR_HIGH (C)</td>
<td>183</td>
<td>MAR_LOW (C)</td>
<td>183</td>
<td>Priority inversion</td>
</tr>
<tr>
<td>MIXXX (CPP)</td>
<td>350,730</td>
<td>MIXXX (CPP)</td>
<td>350,730</td>
<td>Poor QoE on Media Player</td>
</tr>
<tr>
<td>MULTITHREAD (C)</td>
<td>1,141,835</td>
<td>MULTITHREAD (C)</td>
<td>1,141,835</td>
<td>Deadlock from two mutex variables in GLIBC</td>
</tr>
</tbody>
</table>

### 3.2 Selected Applications

The pairs of programs that meet our three selection criteria are shown in Table 3.1. In the table, the character(s) in parentheses after the application name indicate(s) whether the application is a C program (C), a shell script (S), or a C++ Program (CPP). We also report other basic characteristics of the objects including lines of code (LoC) and the types of faults that have been reported. As we run each pair
of programs, we also needed to log OS related runtime events that can be used to detect faults. As such, we created an experimental environment that supports multiple kernels and can work with different tools to monitor runtime behaviors. For example, we used Scribe to monitor runtime events for the first 17 pairs of applications. We then used Strace for the last three pairs of applications. We also used the Gentoo 2.6.35 kernel to run 18 pairs of apps, the Ubuntu 3.0.101-rt130 real-time kernel to run MAR_HIGH and MAR_LOW, and Ubuntu 3.0.0-12 to run the pair of MIXXX application. Each pair of applications has particular behaviors when they interact with each other, as we describe next.

3.2.1 Under-constrained Process Interaction Faults

A pair of UPDATEDB processes from version 4.1.20 [36] can race with each other while performing database updates. This classic race occurs when one of the two UPDATEDB processes, without proper synchronization of file accesses, naively modifies the database while the other is using it.

locate [37] is an application from version 4.1.20 of the GNU Find Utilities. This GNU package provides a modular directory and file search system. LOCATE is used to locate all types of files in the system based on a database created by UPDATEDB (also from version 4.1.20 of the GNU Find Utilities). A bug report indicates that these two can race, causing the database file to be prematurely deleted by UPDATEDB.

bash (Bourne Again Shell version 3.0) [38] is an sh-compatible shell for Unix operating systems. Like other shells, BASH works as a command manager. It receives command lines from a user through standard input devices such as the keyboard or mouse, sends these commands to the operating system, and interacts with the user by displaying results on an output device such as the monitor. By default, BASH can
save a history file as a log for the system or the developer to use. The reported race in this program occurs when two instances of BASH interact, causing the history file to be corrupted.

MV and RM are applications from version 6.9 of the GNU Core Utilities [39]. MV is a basic Unix command used to move files from a source location to a destination location. RM is a basic Unix command that deletes files. Typically, to perform atomic file replacement using a shell script, RM, LN, and MV are used in sequence. However, an implementation change in MV in coreutils-6.9-16.fc8 creates a race in a commonly used atomic rename script. When a race occurs, the link previously created is removed before the file is moved, causing the source file for the MV operation to be missing.

RM (or remove [40]) is a command from GNU Coreutils 8.4, designed to remove files and directories recursively. Using the option -r or -R, the RM command can hierarchically delete a file from a given location in the root directory. The race that can occur with this program is classified as a TOCTTOU race. When the race occurs, the directory traversing status will be NOFOLLOW, meaning that there are no symbolic links or symlinks underneath the root directory and the deletion can proceed. This race allows attackers to create a SYMLINK, causing the deletion of the symlink target. We created a shell script, SYMLINK, to create a symbolic link inside an existing directory and act like an attacker in this scenario.

MKDIR, MKNOD, and MKFIFO [41] are basic commands from GNU Core Utils 5.2.1. MKDIR is used to create a directory or a structure of directories. This command provides an option -m to specify the priority of directories. These two steps should be atomic, but instead there is a gap between them that can allow another application to interfere between the two steps and modify the resource. In this particular scenario, CLEANER, a simple shell script, can interfere and remove the newly created directory that MKDIR tries to set a priority on. There are also other commands that can suffer
from a similar atomicity violation. **mknod** works as a device special file creator with the same option `-m`. It has the same defect as **mkdir**. **mkfifo** is used to create a FIFO pipeline, again with the option `-m`. It can also suffer from the same atomicity violation.

Commands **mkdir2** [42], **mknod2**, and **mkfifo2** also suffer from the same atomicity violation previously reported. However, these implementations are from a later package, GNU Core Utils 5.97.

**ln** [43] is used to create a link between two files. This is an application from GNU Core Utils 5.94. A race can occur in this command as a file can be removed before a new symlink is created. This is also a classic TOCTTOU race.

**find** [44] is a utility application from GNU FindUtils 4.1.20. It can be used to locate files. A race can occur on the internal file structure as another process (**cleaner2** in this case) can simultaneously modify the structure causing **find** to fail.

**csplit** [45] is an application from GNU Core Utils 5.2.0. A bug report indicates that an interrupt signal, **SIGINT**, can create a race on file deletion, leaving a temporary file to remain in the system instead of being deleted. To create this race, we created **mSigShooter**, a customized application that instigates an Interrupt Signal (**SIGINT**) to cause races and terminate a **csplit** process. **mSigShooter** can instigate multiple interrupt signals, but in this case only two instances of **SIGINT** are needed.

**pxz** or Parallel XZ [46] is a C utility that uses the LZMA compression algorithm to compress files using multiple processing cores [47]. The reported race is that an intermediate file created as part of the compression process can be seen by another process after creation but just prior to applying the correct permission setting. To reproduce this race we created **stealer**, a customized shell script that exploits the vulnerability (which involves a race between **pxz** and **stealer**) by attempting to steal the information **pxz** is working on.
LOGROTATE [48] is an administration tool used to manage the log files from a system. It rotates, compresses, and mails those system logs based on the configuration, which can be handled daily, weekly, monthly, or manually. The race occurs after a log file has been created but before the access mode (chmod) can be set. This small window allows other applications to see the content of the log file. This is a priority escalation problem.

PS is a common Linux command that produces a snapshot of the current process or the current status of processes running on an operating system. GREP SCRIPT is a shell script that executes the GREP command, which is a GNU search tool, allowing a user to search each line of input data for specific string patterns (e.g., keywords). This particular race occurs when the user calls PS on the command line and sends the result via a pipe to GREP to find a specific word [49] (e.g., ps auxw | grep foo). The output of this command should be a list of existing processes that contain foo in their names. In theory, when PS runs, it should not see GREP since it should not be running until the data is sent through the pipe for grep to process. However, the command-line interface creates a pipe, then executes both PS and GREP. Running GREP too early is the reason PS can see it and list it along with the other processes.

3.2.2 Over-constrained Process Interaction Faults

MARS PATHFINDER [12] is a recreation of a priority inversion event that occurred on the Mars Pathfinder in 1997. When a high priority process tries to access the information bus while a low priority process is using it, the lock used to protect the bus prevents the high priority process from accessing it. However, if a mid priority process wants to execute, it would then preempt the low priority process while still holding the lock on the information bus. This can cause a long wait time for the
high priority process. If this wait time becomes too long, the system will reboot as it suspects something is wrong with the high-priority task. We classify this as a QoE problem due to the long delay due to lock contention.

**Mixxx 1.8.0** [50] is a C++ open source DJ sound mixer software application. Its design is based on the object-oriented programming paradigm. Each component works simultaneously and users can interact with the application even while it performs mixing tasks. The application can process various types of media file formats including compressed files, and handles data stored directly on local storage or streamed over the network. Based on its design, the application’s components often share both input and output resources that can lead to concurrency faults. From the bug report [50], Mixxx can intermittently freeze, have audio glitches, or become silent while loading a new media file (FLAC).

The **GLIBC library** has been reported to cause multiple processes or threads to *deadlock*. To reproduce this fault, we created a multithreaded program using the library from Glibc 2.17, which has at least four instances of reported deadlocks [51,52,53,54]. For example, a report indicated that two threads sharing two mutex variables can deadlock on each other [51]. So we created a program that has two threads sharing two mutex variables.

Next we describe the two proposed systems, *RCRF* and *RDSC*, that have been designed to reproduce process-level races and atomicity violations. The empirical evaluations of these two systems are conducted using all pairs of applications or a subset of these pairs.
Chapter 4

RCRF: A Reproduction Framework for Use in Debugging Races, Deadlocks, and Excessive Synchronization Delays

4.1 Introduction

In this chapter, we present RCRF, a reproduction framework for use in debugging races, deadlocks, and excessive synchronization delays, an automated framework that helps software engineers automatically reproduce reported synchronization-related faults.¹ RCRF performs a hybrid analysis by leveraging existing static program analysis tools and kernel event reporting tools including STrace [56], DTrace [57], iNotify [58], and Scribe [59]. These tools provide the observability needed to identify

¹This chapter describes work published in the paper “RRF: A Race Reproduction Framework for Use in Debugging Process-Level Races” [55].
explicit and implicit accesses to shared resources and automatically map them to application events. To reproduce a fault that occurs due to synchronization problems among multiple processes, RCRF executes the instrumented processes and observes the runtime behaviors of all pair-wise accesses of shared resources. RCRF also provides a repository in which engineers can store specified correctness properties that can be used to validate whether a pair of resource access operations experience races or contention and locate where they occur in the programs.

We conducted an empirical study to evaluate RCRF. We used 20 systems described in Chapter 3 and applied RCRF to reproduce the faults in those softwares. We studied whether our framework can reproduce the reported symptoms better than a baseline approach involving stress testing, and found that our approach was both more efficient and more effective.

Next we highlight related approaches that are used as part of our framework. We then introduce RCRF, our proposed framework to debug races, deadlocks, and QoE faults. Next, we describe the methodology used in our empirical study and report the results of our empirical evaluation and discuss the results and implications of these results on debugging practices. We then highlight work related to ours, and conclude.

4.2 Background

4.2.1 Motivation

Faults related to process-level concurrency control are caused by incorrectly constrained process interactions. Over-constraining leads to deadlock or serious synchronization contention, while under-constraining leads to the various race conditions, broadly referred to as data-races, atomicity violations, and order violations. In this work, the
over-constraining concurrency bugs we focus on involve order violations and deadlocks
and the under-constraining concurrency bugs we focus on involve synchronization
contention that can lead to QoE faults. Note that we previously defined these terms
in Chapter 2.

It is worth noting that while races, deadlocks, and contention all occur due to
attempts by multiple processes to simultaneously access a shared resource, they each
exhibit different symptoms. Because contention involves incorrect synchronization, it
is possible to extend a race detector to detect it. The design of RCRF is based on
this observation.

Next, we provide examples to illustrate how races, deadlocks, and contention occur
and why they are difficult to debug. The first issue that makes debugging process-level
races and contention difficult is the lack of execution observability. Because system-
level events including system calls and signals are typically treated as black boxes by
engineers, the sequences of events that occur inside system-level events are not seen.
As an example of a race, in the Linux kernel there was a change in the implementation
of the mv command. Previously, to accomplish its task, mv used a combination of
unlink and rename calls. The new implementation, however, just uses rename.
As such, a previously working shell script that attempts to replace a file using the
sequence of commands rm, ln, and mv no longer works correctly. In this example,
engineers must be able to observe events occurring inside the target of a system call.
Existing instrumentation tools that work at the statement level miss this particular
race because they cannot detect implicit system calls.

The second issue that renders debugging difficult is lack of execution controllability.
Typically, shared resources can be accessed at multiple locations in a pair of faulty
programs. For example, there is a scenario in which two instances of Bash shells can
race, resulting in a corrupted history file. In the Bash code, there are at least eight
locations that access the history file. These locations can potentially cause races when two instances of BASH are running simultaneously and they must be exercised in a pair-wise fashion to reproduce the race. We executed these two instances of BASH over 50,000 times, and were not able to reproduce the reported race. Clearly, an approach is needed that provides the controllability necessary to exercise interleavings to reproduce such races, and this approach must be capable of addressing situations in which multiple locations are involved in causing races.

As an example of a QoE fault, we report a symptom experienced by users of Mixxx [50] introduced in chapter 2. Users report that Mixxx can momentarily freeze while playing a song while another media file is loaded by the second player. This problem has to do with shared buffer management as the buffer can be overrun when multiple players are used. Other reported QoE faults include another media player called Rhythmbox, which stops playing when performing some other actions [60]. We have also seen reports of modem managers and network managers that can cause delays in shutting down systems [61], reports of user interfaces crashing [62], and reports of servers refusing services [63].

4.2.2 Achieving Observability and Controllability

RCRF is motivated by RACEPro—a process-level race detection technique—together with the idea of active testing. In terms of capability, RCRF can also detect and reproduce deadlocks and synchronization contention. We briefly introduce these two related techniques below.

RACEPro is an approach for reproducing races by using in-vivo execution [3]. RACEPro uses a tool, Scribe, to record system-level event traces, and thus provide observability. These event traces can be used to model system calls and identify
system-level shared resources (SRs). Specifically, RACEPRO tracks system calls that operate on shared resources (e.g., open, read, write, lstat, clone, wait, execve, and exit); these involve reads and writes on SRs. For example, the lstat system call on file $f$ reads the metadata of $f$, and the write system call on $f$ writes to both the data and metadata of $f$.

RACEPRO next analyzes systems for execution branches that can potentially lead to races, and reorders system-level events in these branches to expand the possible interleavings beyond those in the recorded trace. These reordered traces are then run on an in-vivo execution engine, thus achieving controllability. This approach has benefits: traces can be executed at near zero overhead, and access to source code is not required. The approach suffers, however, from two problems. First, reordering system calls can lead to replay divergence, causing the engine to produce an incorrect report or fail to execute. Second, the in-vivo execution engine is complex and as of this writing, is no longer being maintained. As such, it is difficult to transfer the approach to real-world usage (we discuss this further in Section 4.5.2).

Another approach for inducing races to occur is active testing. In this approach, two steps are commonly performed. First, static or dynamic analysis is used to identify code locations at which races can occur. Second, functions such as yield are used to suspend threads around code locations that may cause races. The approach attempts to expose races by pausing one thread’s execution and then checking to see whether a race actually occurs. CalFuzzer [64] implements active testing for Java to detect races occurring in shared memory, and UPC-Thrille [65] implements it for Unified Parallel C. Active testing is slower than RACEPRO due to its need to pause threads, but it does not suffer from replay divergence as it does not explicitly reorder events.

SimRacer [66] is another approach based on active testing; it exercises interleavings to detect races and deadlocks by using a virtual platform to pause and resume
binary execution. One difference between these testing approaches and RCRF is that RCRF is not a race and/or deadlock detection system; rather, it is a reproduction system designed to help engineers reproduce (and ultimately debug) reported races, deadlocks, and QoE faults. As such, RCRF need not consider all possible locations at which these faults can occur; instead, it focuses only on accesses to shared resources that have been reported to experience faults. This allows RCRF to filter out unimportant shared resources. Testing approaches, on the other hand, consider all potential sources in order to expose more faults.

### 4.3 Introducing RCRF

Figure 4.1 provides an overview of RCRF. The input to RCRF includes application configuration information such as the necessary environment variables, kernel version, and specific required libraries. We also need to include two or more programs undergoing debugging (PuDs), test inputs, bug reports, and failed output. When program inputs are provided (which is typically the case in many bug reports), RCRF can use these to exercise the program. If inputs are not provided, test case generation techniques can be used to generate inputs. The effectiveness of RCRF does depend, however, on the quality of the test inputs used.

As an example, we consider a particular bug report related to CSPLIT, a GNU Core Utils application. In this example, our environment information includes the relevant version of CoreUtils, which is 5.2.0 in this case. We also need to have the corresponding Linux kernel that supports that particular version of Core Utils, which is Gentoo 2.6.35. We also need to specify the event monitoring tool. In this case, Scribe can run on this kernel version so it was used. Next, we need to include two PuDs. In this particular example, only CSPLIT is specified in the bug report but
the reported symptom indicates that a race can occur when a signal, `SIGINT`, is sent. As such, we developed a program called `MSIGSHOOTER` that periodically sends `SIGINT` to `CSPLIT`.

Based on the reported symptom, the shared resource is a temporary file that should be removed after `CSPLIT` completes, but a race can cause it to not be removed. We ran the program and monitored the system events related to file deletion (with Scribe on) and analyzed `CSPLIT` to map the relevant system events to source code, and then injected delays. Next, we developed an internal oracle that observes when a signal is interfering with an access to the shared file. Finally, we ran the modified PuDs (CSPLIT and MSIGSHOOTER in this case) to exercise all potentially faulty interleavings; each run exercised only a pair of interleavings. We then used the oracle to determine whether the reported fault occurs. The oracle result was written into the report file. Next we describe each component of the approach in greater detail.
In Component A in Figure 4.1, inputs are used to configure and run each PuD individually. As a reproduction (not detection) framework, RCRF needs bug reports that describe the fault that must be reproduced. An engineer then specifies resources that should be observed. In some cases, bug descriptions clearly specify the resources that need to be observed; in others, engineer expertise is needed to narrow down resources to be observed. The foregoing is true, however, of all concurrency fault reproduction approaches, so we are requiring nothing beyond those.

Unlike manual approaches, however, RCRF then automatically performs analysis to identify code sections that can access the resources, and injects yield points to amplify the likelihood of reproducing those targeted faults. In this respect, RCRF is systematic, extensible, and general. Developers can perform instrumentation either at the source code level or by using other techniques or tools (e.g., PIN [67]) to perform instrumentation at the binary code level.

The output after each execution of the PuD is a log of system-level events invoked during the execution. Each event log is analyzed concurrently with the bug report to identify resources that can potentially race, deadlock, or be sources of contention by processes and system-level events that access them. The analysis result is then used to cull out events that cannot access the resources. The output of this phase is a log of events for each application that may be relevant to the reported faults.

In Component B, the outputs of the prior phase are mapped to the original source code. The result of this task is a list of potential interleaving event pairs between the two applications that can cause reported faults based on previously identified shared resources. In Component C, delay points are injected into the source code for each of the potential interleaving event pairs to amplify the chance of exposing reported faults. To reproduce faults, Component D compiles and executes the two modified PuDs (mPuDs – the original PuDs with sleep calls injected). As part of execution,
a system-level event log is generated, containing interaction information about the PuDs. Since RCRF injects `sleep` calls at the pair-wise level for each execution, a difference between the normal execution and the instrumented execution is likely to be caused by the faulty interleaving induced by the pair of events. This can help developers pinpoint the problematic pair leading to the failure.

Last, Component E uses interaction information to determine whether a fault has occurred by validating the output for correctness and analyzing the event logs against internal property-based oracles (i.e., race, deadlock, contention detectors). An execution report that includes information on whether targeted faults (e.g., race or contention) occur and the locations of event pairs that are involved in them is then generated. As such, we classify RCRF as mostly automatic.

As noted in Section 4.2, RCRF combines part of RACEPRO with active testing. Specifically, we leverage Scribe, a system-level event monitoring and logging tool, to provide observability. Scribe has been used to detect process races in deployed systems by logging system-level events in Linux; these can be processed off-line to detect races and other types of concurrency faults [59]. Performance evaluations of Scribe in server and desktop environments have shown that its average overheads are 2.5% and 15%, respectively [68]. We use Scribe to detect events due to its support for Linux and its low runtime overhead. We also show in this chapter that our framework is generalizable and flexible. We can employ the framework under different operating environments. For example, we can reproduce faults by running applications on RT-Linux operating system and use STRACE to record the kernel events. We can then apply different oracles to detect faults.

Like active testing, we use process yielding to provide the controllability needed to reproduce process-level races. We inject `sleep` calls at specific code locations in the two programs to enforce a desired interleaving. RCRF is systematic in the approaches
it uses to identify relevant system-level events, map those events to application events, generate execution interleavings that should be explored, and validate whether a reported race has been reproduced. RCRF is largely automated so it can be more precise than manual approaches for generating interleavings. It is also extensible so that engineers can specify new oracles for use in detecting races, deadlocks, and contention. It can also identify interacting events that cause those faults. Next, we describe the components of RCRF.

### 4.3.1 Identifying Relevant System Events

For RCRF to work, Component A requires inputs, including a bug report, a pair of PuDs, and a configuration file. First, engineers must analyze the bug report to ascertain information relevant to revealing shared resources on which targeted faults can possibly occur. Some bug reports provide comprehensive information (e.g., inputs or execution scripts, operating environments, other running applications, and behavioral descriptions of detected faults [37, 39, 40, 45, 46]); in these cases engineers
can use the information to generate specific traces of system-level events (e.g., use the execution scripts provided). Some reports, however, are less comprehensive (e.g., providing no inputs and providing only vague behavioral descriptions of the detected faults [36,38]); in these cases engineers need to perform additional tasks, including additional test input generation and shared resource analysis, to increase the chance of exercising code sections that can potentially cause the reported faults. These activities are common to other approaches for reproducing concurrency faults, so by requiring them ourselves we are requiring nothing extra. At a minimum, we expect information regarding the incorrect symptoms experienced by users of the system (e.g., file corruption, missing signals, and socket not found errors) to be provided as part of the bug report. In some cases, a user may provide an execution script as part of a bug report that may help engineers reproduce the race [3]. In other cases, users may provide only the symptoms. In the latter case, engineers need to identify potential interleavings that might expose the reported races based on the symptom (as illustrated in the CSPLIT example). Our framework is capable of efficiently helping engineers accomplish this goal.

To further illustrate the process for identifying resources that can potentially race we provide another example, involving two applications. Suppose that process UPDATEDB can race with process LOCATE, but the bug report for this race describes only what can happen without providing execution scenarios in which the race occurs. Based on the reported symptom, which is a corrupted database file located at /usr/local/var/locatedb, an engineer could individually run the two PuDs using an existing test suite while turning on an event reporting tool such as Scribe. At the end of each run, a report of a list of system-level events is generated. For example, Figure 4.2 provides an example of Scribe output, that shows system-level events initiated by UPDATEDB. The associated command snippet is shown in Figure 4.3.
In most cases, traces generated by tools like Scribe or STrace contain extraneous information that includes other system-level events not relevant to a reported symptom. To cull out extraneous information, we rely on the analysis of the bug report. In our first example, the bug report indicates that process-level races occur on a particular database file; thus, only events that can manipulate that file are considered relevant. In the case in which a symbolic link is also made to the database file, Scribe also reports the actual file name as well as the symbolic name so we can still accurately remove extraneous information. In our example, the highlighted system calls in Figure 4.2 (fstatat64 and unlinkat) are relevant to the reported race because they access the database file; remaining system calls in the figure are not relevant.

Even when an execution script is provided as part of a bug report, the foregoing process should be applied to find possible faulty locations that may exist in the path where the reported fault exists. This is because RCRF can determine whether there are other code locations that can produce damaging faults instead of focusing only on the reported interleaving. As shown in Section 4.5, we have found examples in which a reported fault can have multiple sources within two programs. If engineers focus on a single source, they may leave other sources that result in similar faults untouched.
4.3.2 Mapping System-level Events to Source Code

Component A of RCRF produces a list of relevant system-level events for each application. These events must be mapped to the source code locations in the application that invoke their corresponding system calls. Component B constructs this mapping by combining static analysis and system-level event modeling (discussed in Section 4.2). (Note that we intend RCRF to be used by engineers who are debugging applications developed by their own organizations, so it is reasonable to assume the availability of source code.)

RCRF performs static analysis at the source code level to identify source code locations involving application events (e.g., using rm to remove the database file) and then translates each event into a sequence of system calls based on the previously constructed models. The result is a mapping of each block of system level events (e.g., system calls to write to a file and manipulate the corresponding inode structure) to the corresponding event in the source code of the PuD. Figure 4.3 illustrates how the highlighted block of system calls in Figure 4.2 is mapped to line 144 of UPDATEDB. We can perform this mapping because we have already modeled the rm operation to include a sequence of two system calls, fstatat64 and unlinkat. The output of this component is a list of locations where a fault may be initiated.

4.3.3 PuD Modification

Given a set of locations in two PuDs that can potentially race, interleavings must be enforced at these locations. This is done by using sleep calls to pause the execution of one process at a particular shared resource access point and then pause the other process at an execution point that accesses the same resource. Currently, we inject sleep calls into source code, but binary instrumentation could also be used. To
identify pause points, Component C performs pairwise analysis of relevant application events between the two processes. While a more optimized pairwise algorithm can be applied in the case of large numbers of events [36,37,38,40,46], by culling out extraneous events the number of system calls utilized can be rendered small enough to employ all combinations.

To perform pair-wise injection, the list of system calls $S_i$ from process $P_i$ and the list of system calls $S_j$ from process $P_j$ are used as inputs. System calls in $S_i$ and $S_j$ are paired if they access the same shared resource. $\sigma$ stores a list of such pairs ($\sigma_i, \sigma_j$), which is initialized to empty. Since the order of the two system calls in a pair matters, we choose to include ($\sigma_i, \sigma_j$) and ($\sigma_j, \sigma_i$) in $\sigma$. The results of this process are instrumented PuDs with sleep calls (mPuDs).

Because our implementation uses sleep calls, our framework needs to use an appropriate sleep time for each injected sleep call. To select an initial sleep time, we consider the amount of time necessary for the critical operation to be completed. We then refine this initial time through trial runs to determine an optimal sleep time for each case. In the cases we studied, we found that one second worked well in all cases. RCRF also needs to ensure that we can execute pairs of processes in specific orders. For example, we may want to ensure that a suspected system call in one process is executed before another suspected system call in another process. In this scenario, we set the sleep time of the second process to be longer than that of the first.

RCRF also injects sleep calls before and after each suspected system call to maximize the likelihood of exposing targeted faults. In this scenario, we would explore the execution interleaving that configures $P_i$ to execute before $P_j$ and then explore the interleaving that configures $P_j$ to execute before $P_i$. As such, two sleep calls (with one and two seconds of sleep time, respectively) would be injected immediately before a system call and another pair of sleep calls would be injected immediately
Algorithm 1 Sleep Injection

**Require:** a list of suspected SysCalls from $P_i$ ($S_i$),
a list of suspected SysCalls from $P_j$ ($S_j$),
source code of $P_i$ ($SRC_i$),
source code of $P_j$ ($SRC_j$),
two targeted system calls from the Pairwise Algorithm ($\sigma$), sleep time ($ST$)

1: $MP \leftarrow \emptyset$
2: for all $(\sigma_i, \sigma_j) \in \sigma$ do
3: \{ places to inject sleep before system calls \}
4: $L_{ib} = \emptyset$
5: $L_{jb} = \emptyset$
6: \{ places to inject sleep after system calls \}
7: $L_{ia} = \emptyset$
8: $L_{ja} = \emptyset$
9: if ($\sigma_i \in S_i$) then
10: $L_{ib} = \text{sleep}(ST) + \sigma_i$
11: $L_{jb} = \text{sleep}(2 \times ST) + \sigma_i$
12: $L_{ia} = \text{sleep}(ST) + \sigma_i$
13: $L_{ja} = \sigma_i + \text{sleep}(2 \times ST)$
14: else
15: $L_{ib} = \text{sleep}(2 \times ST) + \sigma_j$
16: $L_{jb} = \text{sleep}(ST) + \sigma_j$
17: $L_{ia} = \sigma_j + \text{sleep}(2 \times ST)$
18: $L_{ja} = \text{sleep}(ST) + \sigma_j$
19: end if
20: $MP \leftarrow \text{GenModPuD} (S_i, S_j, SRC_i, SRC_j, L_{ib}, L_{jb}, L_{ia}, L_{ja})$
21: \{ one sleep is injected before a system call; the other after the system call \}
22: end for
23: return a set of mPuDs $MP$

after the system call. Conditional compilation is used to specify the desired execution interleaving (e.g., $P_j$ executes before $P_i$).

Algorithm 1 presents our sleep call injection approach. For each pair of system calls $\sigma$ (Line 2), the algorithm creates four mPuDs $MP$ (Line 17). It first determines which process to execute first, by comparing the specified system call ($\sigma_i$) to system calls in $P_i$ ($S_i$). If found, $P_j$ needs to execute first and thus uses $ST$ as the sleep time (Line 8) and $P_j$ uses $2 \times ST$ as its sleep time (Line 13). If $\sigma_i$ is not found in $S_i$, it
is in $P_j$; therefore, $P_j$ needs to execute before $P_i$ (Lines 9 and 14). This information is used to inject \texttt{sleep} calls into the source files of $P_i$ and $P_j$, which are $SRC_i$ and $SRC_j$, respectively. Lines 10-11 and 15-16 specify locations at which to inject the \texttt{sleep} calls after $\sigma_i$ and $\sigma_j$. The algorithm returns the mPuDs with injected \texttt{sleep} calls. If there are multiple sources of potential faults, the same mPuDs are used and conditional compilation flags are used to enable a particular \texttt{sleep} call. Figure 4.4 illustrates a modified \texttt{UPDATEDB}.

### 4.3.4 Fault Reproduction

Next, the mPuDs must be executed to attempt to reproduce the reported symptoms. To do this, RCRF compiles each mPuD and runs them in pair-wise fashion as specified in $\sigma$. Observerability tools such as \texttt{SCRIBE} or \texttt{DTRACE} record relevant OS activities that can be used by oracles (described next) to detect whether reported faults occur and determine the causes of these faults. It is possible that even with amplification by \texttt{sleep} calls, our approach can suffer from execution non-determinism because a system call is a large block of execution steps and thus, execution interleavings between two system calls have variability. To address this we ran trials using different numbers of runs and found that we could reliably reproduce targeted faults within 10 runs.

### 4.3.5 Fault Validation

An important component of our framework is an \textit{oracle} that can be used to observe whether a reported fault occurs in each run. Because our targeted faults (i.e., races, deadlocks, and synchronization delays) are dynamic in nature, we employ both internal oracles [69] and output based oracles to help detect whether a targeted fault occurs and isolate the locations of its causes. The information generated by our oracles is
then used to further classify each detected fault into three possible types: **targeted**, **incidental**, and **new**. If a run is intended to exercise a particular fault (e.g., a race or a synchronization delay based on where we inject `sleep` calls into the two PuDs) described in a bug report, and that same race is detected, it is classified as targeted. If a fault is detected but it is not due to our `sleep` calls, it is either an incidental fault or a new fault. An incidental fault is a detected fault that we did not intend to exercise in a given run but would attempt to exercise later through a different set of `sleep` calls; however, it naturally occurs as part of the run. A new fault is a fault that is not related to the possible locations, determined by our analysis, at which a reported fault can occur.

We designed our internal oracles to look for property violations such as TOCTOU (time of check to time of use) violations, atomicity violations, deadlocks, excessive delays and races. Internal oracles can be implemented using various existing race and deadlock detection techniques that include vector clock [35], lockset [19], happen-before relationships [70], and linear temporal logic [71]. In the case of QoE faults, we also need to develop monitors to automatically detect symptoms such as long execution
delays and intermittent pauses. Our oracles and monitors are stored in an expandable repository so that engineers can extend our framework to detect types of faults beyond those covered in this work. At the end of the RCRF process, a report is generated to specify whether faults occur, classify their types, and provide their locations in the two PuDs. This information can help engineers localize faults.

In terms of execution complexity, there are three factors that can determine the number of runs that should be used to try to reproduce faults. The first factor is the number of system calls in the two applications that must be executed in pairwise fashion ($S_i$ and $S_j$). In our example, LOCATE and UPDATEDB have three relevant system calls each. As such, the number of pairs (interleavings) is $3 \times 3$. The second factor is the number of patterns that we want to explore for each interleaving. Currently, we have four patterns: $P_i$ before $P_j$ before the system call, $P_j$ before $P_i$ before the system call, $P_i$ before the system call and $P_j$ after the system call, and $P_j$ before the system call and $P_i$ after the system call. As such, we need to multiply each interleaving by four. The third factor is the number of runs that we need to overcome non-determinism ($r$). As such, the number of runs we need to reproduce a reported fault between two applications is $\text{sizeof}(S_i) \times \text{sizeof}(S_j) \times 4r$.

4.4 Empirical Study

RCRF provides an automated process for reproducing specific classes of faults due to under- and over-constrained process interactions. Ultimately we are concerned with whether engineers who use RCRF can localize and correct these types of faults more quickly than with current approaches. Studies of humans, however, are expensive, and before embarking on these it makes sense to study whether the approach can indeed
reproduce targeted faults reliably, and in a reasonable amount of time. We therefore designed an empirical study considering the following two research questions:

**RQ1:** Given faults that fall within the classes of faults RCRF targets, how effective is RCRF at reproducing these?

**RQ2:** Given faults that fall within the classes of faults RCRF targets, how costly is it to run RCRF on these?

### 4.4.1 Objects of Analysis

To answer our research questions we needed to locate instances of systems in which faults had been reported, that are within the scope of the classes of faults RCRF is designed to target. To do so, we used the applications described in Chapter 3 as our objects of analysis. We use all 20 pairs of apps in the evaluation of RCRF.

### 4.4.2 Variables and Measures

**Independent Variable.** We wish to assess the effectiveness and cost of RCRF. To assess effectiveness (RQ1), we wished to compare RCRF to a baseline technique, and an initial candidate for this was RACEPRO. However, we discovered that there are several limitations of RACEPRO that prevent it from running on many of our object program pairs. For example, it does not handle deadlocks and delays due to contention. We also experienced many instances of replay divergence when we used it to detect and replay races. This rendered it unsuitable for use as a baseline technique. We provide further discussion of the limitations we encountered, along with the data we were able to obtain, in Section 4.5.2.

As an alternative baseline technique for use in this study, we compare RCRF to a *stress-testing* approach typical of those commonly used in practice to detect these
types of faults. The stress-testing approach (STA) that we utilize tries to reproduce reported faults by randomly launching the pairs of programs that are known to have faults a large number of times. Note that this is a simple process that operates primarily based on the known symptoms. Our proposed approach is similar to this approach in that it is also driven by the reported symptoms but without needing to employ complex processes of recording events and then replaying them.

We evaluate STA and RCRF with the RCRF observation tool turned on in order to gather execution information that would be needed by engineers to begin to understand the causes of a reported fault, if it occurs. We also use the same oracle in both cases.

To provide a fair comparison of the approaches, for each pair of object programs, we first executed RCRF on the pair to completion, measuring the amount of time required. We then executed the STA approach for exactly that amount of time. This lets us assess the relative effectiveness of the approaches when they are given the same amounts of time.

**Dependent Variables.** To assess effectiveness (RQ1), we measure the number of test runs needed by RCRF and STA to detect each targeted fault, in the cases in which the fault was revealed. To assess the cost of RCRF (RQ2), we measure the wall clock time required for a complete run of the approach. In this case, there is no baseline approach to compare RCRF against.

### 4.4.3 Study Methodology

To automate RCRF we minimized the involvement of the user by encoding all information necessary for a run into a configuration file. This information includes the project name, shared resource name, first process number, second process number, location of the source code for the first and second processes, and log files. When
RCRF begins, the full list of system calls is filtered to exclude system calls from processes not being targeted. Then, a second filter returns just the system calls that use the shared resources. After RCRF cleans up the log, it generates mPuDs by pairing up the system calls. By calculating and injecting `sleep` calls into the source code (before or after) that map to the system calls, each mPuD can represent a set of interleaving patterns. RCRF then executes the pair of mPuDs, and passes the output of each through the validation component to determine whether the targeted fault has occurred. We repeat the foregoing process ten times for each case, because even when using forced interleavings, non-determinism can still occur.

### 4.4.4 Threats to Validity

The primary threat to external validity in this study involves the object programs utilized. We have studied just 20 pairs of faulty programs. However, the programs are real and the faults are real and non-trivial to reproduce. A second threat involves the representativeness of our oracles. These threats can be addressed only through further studies.

The primary threat to internal validity involves potential errors in the implementations of RCRF and the infrastructure used to run RCRF and STA. To limit these we extensively validated all of our tool components and scripts.

The primary threat to construct validity relates to the fact that we study effectiveness and cost measures relative to applications of RCRF, but do not yet assess whether the approach helps engineers localize and correct targeted faults more quickly than current approaches. A second threat involves the dynamic and non-deterministic nature of the fault classes we are targeting. When we inject `sleep` calls into the two PuDs and run the programs, the goal is to exercise a fault that can occur around those
two \texttt{sleep} calls. However, other faults can also naturally occur during that run, so the data we use to answer RQ1 must distinguish between faults we intend to exercise (\textit{targeted faults}) and faults that unexpectedly occur outside of the injected \texttt{sleep} calls (\textit{incidental faults}). We address this threat by analyzing the reported results to classify every detected fault into the three types previously introduced in Section 4.3: targeted, incidental, and new. This classification allows us to measure the effectiveness of RCRF more precisely.

### 4.4.5 Results

Table 4.1 provides data for both of our research questions. Column C indicates the total execution time required by RCRF in minutes. Columns D and G show the total numbers of runs for RCRF and STA, respectively. Columns E and H show the total numbers of runs in which races were detected for RCRF and STA, respectively. Columns F and I show the percentages of runs in which races were detected relative to runs in which they were not, for RCRF and STA, respectively. The first 17 rows represent pairs of programs that can suffer from various forms of targeted process-level races. The last three rows represent pairs of programs that suffer from QoE faults and deadlocks.

We turn first to RQ1, considering the comparative effectiveness of RCRF and STA for all pairs. Recall that the two techniques were executed for the same amounts of time (the amount of time required by RCRF). In that amount of time, STA conducted between 1.29 times (\texttt{mixxx-mixxx}) and 35.84 times (\texttt{ps-grep}) more runs than those of RCRF. (Because we utilized monitoring on both approaches, the differences in numbers of runs are due to the fact that RCRF injects \texttt{sleep} calls, whereas STA does not.) Nevertheless, these additional runs did not allow STA to outperform RCRF. In
Table 4.1: Empirical Results

<table>
<thead>
<tr>
<th>PUD1 (A)</th>
<th>PUD2 (B)</th>
<th>Total Time (Minutes) (C)</th>
<th>Total Runs (D)</th>
<th>RCRF Revealed Faults (E)</th>
<th>STA Revealed Faults (H)</th>
<th>% (F)</th>
<th>% (I)</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPDATEDB</td>
<td>UPDATEDB</td>
<td>18</td>
<td>360</td>
<td>110</td>
<td>938</td>
<td>68</td>
<td>7.24</td>
</tr>
<tr>
<td>LOCATE</td>
<td>UPDATEDB</td>
<td>13</td>
<td>360</td>
<td>20</td>
<td>1,286</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>BASH</td>
<td>BASH</td>
<td>613</td>
<td>7,840</td>
<td>1,308</td>
<td>56,570</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MV</td>
<td>RM</td>
<td>40</td>
<td>640</td>
<td>60</td>
<td>4,595</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RM(FTS)</td>
<td>SYMLINK</td>
<td>11</td>
<td>240</td>
<td>7</td>
<td>2,717</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MKDIR</td>
<td>CLEANER</td>
<td>1.56</td>
<td>40</td>
<td>10</td>
<td>424</td>
<td>1</td>
<td>0.25</td>
</tr>
<tr>
<td>MKNOD</td>
<td>CLEANER</td>
<td>1.82</td>
<td>40</td>
<td>11</td>
<td>90</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MKFIFO</td>
<td>CLEANER</td>
<td>1.6</td>
<td>40</td>
<td>12</td>
<td>429</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MKDIR2</td>
<td>CLEANER</td>
<td>1.62</td>
<td>40</td>
<td>10</td>
<td>440</td>
<td>1</td>
<td>0.22</td>
</tr>
<tr>
<td>MKNOD2</td>
<td>CLEANER</td>
<td>1.62</td>
<td>40</td>
<td>12</td>
<td>334</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MKFIFO2</td>
<td>CLEANER</td>
<td>1.67</td>
<td>40</td>
<td>10</td>
<td>346</td>
<td>3</td>
<td>0.86</td>
</tr>
<tr>
<td>LN</td>
<td>CLEANER</td>
<td>7.98</td>
<td>240</td>
<td>40</td>
<td>3,786</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>FIND</td>
<td>CLEANER2</td>
<td>5.6</td>
<td>160</td>
<td>30</td>
<td>283</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CSPLFT</td>
<td>MSIGSHOOTER</td>
<td>11</td>
<td>160</td>
<td>60</td>
<td>2,502</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PXZ</td>
<td>STEALER</td>
<td>20</td>
<td>320</td>
<td>20</td>
<td>613</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LOGROTATE</td>
<td>STEALER2</td>
<td>110.36</td>
<td>3,080</td>
<td>336</td>
<td>33,558</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PS</td>
<td>GREP_SCRIPT</td>
<td>60.16</td>
<td>80</td>
<td>18</td>
<td>2,867</td>
<td>797</td>
<td>27.79</td>
</tr>
<tr>
<td>MARS HIGH</td>
<td>MARS LOW</td>
<td>11.81</td>
<td>160</td>
<td>24</td>
<td>726</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MIXXX</td>
<td>MIXXX</td>
<td>2,154.16</td>
<td>550</td>
<td>347</td>
<td>709</td>
<td>393</td>
<td>55</td>
</tr>
<tr>
<td>MULTITHREAD</td>
<td>MULTITHREAD</td>
<td>164.98</td>
<td>660</td>
<td>111</td>
<td>2,466</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

fact, STA was unable to reproduce reported races at all on 14 of the 20 application pairs; RCRF, in contrast, was able to reproduce reported faults on all 20 pairs. For UPDATEDB-UPDATEDB, MKDIR-CLEANER, MKDIR2-CLEANER and MKFIFO2-CLEANER, STA was able to reproduce reported races, but the numbers and percentages of runs in which it did so were smaller than those in which RCRF succeeded. However, for PS-GREP, STA caught the race on 27.79 percent of all runs while RCRF caught it on only 22.5 percent of all runs. These percentages seem close, but the number of total runs are much different across the techniques: RCRF employs only 80 runs whereas STA employs 2,867 runs.

Note that it is generally true that the number of runs exercised by STA is greater than that of RCRF. In most cases, all runs can complete successfully. One exception is in the case of MIXXX-MIXXX, which is the most complex application among our pairs of applications. We identified 100 interleavings that can potentially result in
the reported fault. Based on our methodology, we run mixxx-mixxx 1,000 times (we ran each of the 100 interleavings 10 times). Out of these 1,000 runs, only 550 of them were able to complete successfully. The other 450 runs terminated prematurely. Out of these 550 completed runs, we were able to reproduce 347 faults or 63.09% of the completed runs were successful in reproducing faults.

Turning to RQ2, in 15 of 20 cases, RCRF required 40 minutes or less to execute. In four cases (BASH-BASH), (LOGROTATE-STEALER2), (MIXXX-MIXXX) and (MULTITHREAD-MULTITHREAD), however, it required 613 minutes (over 10 hours), 110.36 minutes (almost 2 hours), 2,154.16 minutes (almost 36 hours), and 164.98 (about 2.74 hours) respectively. Especially for (BASH-BASH), the execution time of RCRF is dictated primarily by the number of executions it needs to perform, and this in turn is dictated by the number of pair-wise interleavings that must be targeted. Table 4.2 (rightmost column) shows the numbers of interleavings required, and indicates the much larger number needed in the case of BASH-BASH and others (Except mixxx-mixxx). Still, 40 minutes of machine time is arguably affordable, given the difficulty of revealing races and the fact that STA could not do so in this case. However, in the case of mixxx-mixxx, there are more factors causing QoE faults. The contention events generated by applying RCRF tend to create longer QoE faults than those experienced by a normal run without RCRF. This shows the capability of RCRF to reproduce the reported symptoms due to synchronization delays.

PS-GREP is a special pair among the others. With STA, the race between of these applications on the share resource or pipeline communication channels at the process level always happens. However, RCRF is able to generate different interleavings and causes two patterns to avoid the race.
4.5 Discussion

The results in Column D, E, and F from Table 4.1 show that RCRF was able to reproduce reported faults on all 20 object program pairs. On the 18 pairs of programs other than ps-grep_script and mixxx-mixxx, for which both RCRF and STA both reproduced reported faults, RCRF reproduced the faults more times than STA. Larger numbers of fault-revealing interleavings may help engineers localize the causes of faults more efficiently. On most of the other programs, RCRF reproduced faults that STA, run an equivalent amount of time, could not. We also noticed that for mixxx-mixxx and PS-GREP_SCRIPT, the numbers of detected faults by STA, reported in Column H, are higher than the numbers of faults detected by RCRF, reported in Column E. Further investigation revealed that STA repeatedly detected the same races. Overall, these results suggest that RCRF is more effective than stress testing for reproducing difficult cases of process-level races, deadlock, and excessive delays.

It is also worth noting that RCRF possesses one characteristic that STA does not: it provides additional information to test engineers that STA cannot, in the form of execution window amplification. Typically, it is challenging to reproduce concurrency faults that occur only in very small execution windows. This is why an enormous number of runs is often needed to expose an instance of an elusive concurrency fault. By providing a systematic way to amplify execution windows, RCRF should have a better chance of reproducing such elusive faults than STA.

4.5.1 Additional Analysis

Each PuD that we studied invokes a large number of system-level events during its execution. Many of these events, however, are not relevant to reported faults. RCRF
Table 4.2: Optimization of Numbers of System Calls

<table>
<thead>
<tr>
<th>PuD1 (A)</th>
<th>PuD2 (B)</th>
<th>System Calls Using Share Resources Before Opt. (C)</th>
<th>After Opt. PuD1 (D)</th>
<th>PuD2 (E)</th>
<th>Pair-wise Interleave (F)</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPDATEDB</td>
<td>UPDATEDB</td>
<td>5,232</td>
<td>3</td>
<td>3</td>
<td>36</td>
</tr>
<tr>
<td>LOCATE</td>
<td>UPDATEDB</td>
<td>2,818</td>
<td>3</td>
<td>3</td>
<td>36</td>
</tr>
<tr>
<td>BASH</td>
<td>BASH</td>
<td>8,518</td>
<td>6</td>
<td>8</td>
<td>781</td>
</tr>
<tr>
<td>MV</td>
<td>RM</td>
<td>391</td>
<td>8</td>
<td>2</td>
<td>64</td>
</tr>
<tr>
<td>RM(FTS)</td>
<td>SYMLINK</td>
<td>1,343</td>
<td>3</td>
<td>1</td>
<td>24</td>
</tr>
<tr>
<td>MKDIR</td>
<td>CLEANER</td>
<td>2,848</td>
<td>2</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>MKNOD</td>
<td>CLEANER</td>
<td>2,871</td>
<td>2</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>MKIFO</td>
<td>CLEANER</td>
<td>2,848</td>
<td>2</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>MKNOD2</td>
<td>CLEANER</td>
<td>2,874</td>
<td>2</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>MKIFO2</td>
<td>CLEANER</td>
<td>2,847</td>
<td>2</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>LN</td>
<td>CLEANER</td>
<td>3,893</td>
<td>2</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>FIND</td>
<td>CLEANER2</td>
<td>10,820</td>
<td>4</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>CSPLIT</td>
<td>MSIGSHOOTER</td>
<td>462</td>
<td>4</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>PXZ</td>
<td>STEALER</td>
<td>4,975</td>
<td>5</td>
<td>1</td>
<td>32</td>
</tr>
<tr>
<td>LOGROTTATE</td>
<td>STEALER2</td>
<td>30,755</td>
<td>77</td>
<td>1</td>
<td>308</td>
</tr>
<tr>
<td>PS</td>
<td>GREP_SCRIPT</td>
<td>1,027</td>
<td>2</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>MARS_HIGH</td>
<td>MARS_LOW</td>
<td>246</td>
<td>2</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>MIXXX</td>
<td>MIXXX</td>
<td>675,720</td>
<td>7</td>
<td>7</td>
<td>100</td>
</tr>
<tr>
<td>MULTITHREAD</td>
<td>MULTITHREAD</td>
<td>1,027</td>
<td>12</td>
<td>12</td>
<td>66</td>
</tr>
</tbody>
</table>

culls out these extra events, and to see to what extent it succeeds in this, we gathered data on the numbers of system calls using shared resources before and after this culling. Table 4.2 presents the results, and shows that culling reduces the number of system calls to be considered by several orders of magnitude (Columns D and E). This also reduces the number of interleavings that must be exercised as shown in Column F.

Recall that RCRF calculates the pair-wise interleavings that can access shared resources in each application. For example, CSPLIT has four possible locations and MSIGSHOOTER has one. As such, in this case RCRF produces four pairs of interleavings. We can also view each pair of interleavings as a potential source of faults. The third column in Table 4.3 reports the numbers of potential sources of faults observed in each pair of programs, and the fourth column reports the number of these potential sources that produced the reported fault. On all 20 of our program pairs, multiple
Table 4.3: Interleavings That Can Cause Reported Races

<table>
<thead>
<tr>
<th>PuD1</th>
<th>PuD2</th>
<th># of Potential Sources of Races</th>
<th># of Sources That Failed</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPDATEDB</td>
<td>UPDATEDB</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>LOCATE</td>
<td>UPDATEDB</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>BASH</td>
<td>BASH</td>
<td>196</td>
<td>79</td>
</tr>
<tr>
<td>MV</td>
<td>RM</td>
<td>16</td>
<td>5</td>
</tr>
<tr>
<td>RM(FTS)</td>
<td>SYMLINK</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>MKDIR</td>
<td>CLEANER</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>MKNOD</td>
<td>CLEANER</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>MKFIFO</td>
<td>CLEANER</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>MKDIR2</td>
<td>CLEANER</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>MKNOD2</td>
<td>CLEANER</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>MKFIFO2</td>
<td>CLEANER</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>LN</td>
<td>CLEANER</td>
<td>24</td>
<td>2</td>
</tr>
<tr>
<td>FIND</td>
<td>CLEANER2</td>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td>CSPLIT</td>
<td>MSGSHOOTER</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>PXZ</td>
<td>STEALER</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>LOGROTATE</td>
<td>STEALER2</td>
<td>308</td>
<td>5</td>
</tr>
<tr>
<td>PS</td>
<td>GREP_SCRIPT</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>MARS_HIGH</td>
<td>MARS_LOW</td>
<td>16</td>
<td>4</td>
</tr>
<tr>
<td>MIXXX</td>
<td>MIXXX</td>
<td>100</td>
<td>54</td>
</tr>
<tr>
<td>MULTITHREAD</td>
<td>MULTITHREAD</td>
<td>60</td>
<td>11</td>
</tr>
</tbody>
</table>

potential sources of faults were identified. On all program pairs, more than one of these potential sources did in fact produce the reported fault. In practice, if the engineer debugging the applications does not monitor all sources, it is possible that a patch may not repair all defective sources. By using RCRF, developers can locate additional causes of faults.

Table 4.4: Additional Fault Types Found

<table>
<thead>
<tr>
<th>PuD1</th>
<th>PuD2</th>
<th>Additional Fault Types Found</th>
<th>Number of Faults Found</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCATE</td>
<td>UPDATEDB</td>
<td>UPDATEDB delete the database file while LOCATE is using that database.</td>
<td>120</td>
</tr>
<tr>
<td>UPDATEDB</td>
<td>UPDATEDB</td>
<td>Only one UPDATEDB can really update.</td>
<td>204</td>
</tr>
</tbody>
</table>

We also applied different oracles to assess whether other types of concurrency faults can occur on the same shared resource in an application. To do this, we chose oracles that operate on similar resources (e.g., files, sockets) and applied those oracles
to validate the event traces. Table 4.4 reports our findings. In addition to the reported faults, two pairs of programs also contained other types of faults involving the shared resources that we monitor. First, updatedb-updatedb can suffer from a second type of race that leads to a nullified update; we found 204 additional instances of this type of fault. Second, we also found 120 instances of premature file removal due to races in locate-updatedb. Validating traces using additional oracles incurred only modest overhead. For example, using one additional oracle to validate locate and updatedb adds only 18 more seconds per run.

It is worth noting that the quality of a bug report can affect the effectiveness of RCRF. If a report is incomplete, RCRF must conservatively identify more relevant events; this can result in increased debugging time. Vaguely described symptoms can also lead to imprecise fault localization by requiring RCRF to consider additional events and resources. Because RCRF is dynamic, the quality of test suites can also affect its ability to localize concurrency faults; inadequate test suites can lead to fewer events that can be identified as relevant.

4.5.2 RCRF versus RacePro

According to Laaden et al. [59], RACEPro is capable of detecting the types of races that occur in our object programs, as well as additional race types. To do this, first, RACEPro identifies all potential locations that may lead to a race that needs to be reproduced. Second, it uses replay to cause races to occur. Based on this information, we also attempted to compare RCRF to RACEPro. Our investigation focuses only on process-level races. This is because we applied RACEPro to detect the other two types of faults that we target and found that RACEPro cannot detect deadlocks or
excessive delays. Unfortunately, we were only able to run it on 13 of our 17 pairs of racy programs. For the four that failed, RACEPro encountered runtime errors.

One limitation of RACEPro involves its limited ability to perform resource accessing and anchoring of the `sleep` calls needed to exercise races. This ultimately results in many instances of replay divergence. We also discovered several faults in RACEPro that have not been addressed, perhaps because RACEPro has not been actively maintained since 2012. These faults prevented four pairs of programs from running [72, 73]. Even for pairs of programs that we were able to run, we needed to carefully rewrite them first. For example, we needed to change the programs so that they created processes in certain orders using specific syntaxes.

Table 4.5: Results from RacePro

<table>
<thead>
<tr>
<th>PuD1 (A)</th>
<th>PuD2 (B)</th>
<th>Number of Races Detected (C)</th>
<th>Number of Races Diverged (D)</th>
<th>Harmful (E)</th>
<th>Reported Faults Found (F)</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPDATEDB UPDADBD</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>LOCATE   LOCATE</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>BASH     BASH</td>
<td>25</td>
<td>15</td>
<td>10</td>
<td>0</td>
<td>Yes</td>
</tr>
<tr>
<td>MV       MV</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>RM(RTS)  RM</td>
<td>19</td>
<td>19</td>
<td>0</td>
<td>0</td>
<td>Yes</td>
</tr>
<tr>
<td>MKDIR    CLEANER</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>Yes</td>
</tr>
<tr>
<td>MKFILE   CLEANER</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>Yes</td>
</tr>
<tr>
<td>MKDIR2   CLEANER</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>Yes</td>
</tr>
<tr>
<td>MKFILE2  CLEANER</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>Yes</td>
</tr>
<tr>
<td>LN       CLEANER</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>Yes</td>
</tr>
<tr>
<td>FIND     CLEANER2</td>
<td>7</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>Yes</td>
</tr>
<tr>
<td>CSPLIT   MSIGSHOOTER</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>No</td>
</tr>
<tr>
<td>PXZ      STEALER</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>LOGROTATE STEALER2</td>
<td>15</td>
<td>0</td>
<td>15</td>
<td>0</td>
<td>No</td>
</tr>
<tr>
<td>PS       GREP_SCRIPT</td>
<td>8</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>Yes</td>
</tr>
<tr>
<td>MARS_HIGH MARS_LOW</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>MIXXX    MIXXX</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>MULTITHREAD MULTITHREAD</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 4.5 provides data on our attempt to run RACEPro. As the table shows, RACEPro was able to identify some locations that can potentially cause the known races; however, not all of these locations could be used to reproduce the race on the
bug reports, as shown in Column G. For csplt-msigshooter, RacePro tended to
detect races occurring on other shared resources and not the reported races. In this
case, RacePro detected races occurring on the file that both programs try to access.
This happened with logrotate-stealer2 as well.

We also tried to apply RacePro to detect deadlocks and QoE faults. Unfortunately,
RacePro has been designed to detect races but not the other types of faults we
are targeting and it was unable to reproduce any fault in the last three pairs of
applications.

4.5.3 Incidental Faults

In some programs, faults are particularly difficult to reproduce and in these cases,
amplification is necessary to exercise them. In other cases, faults can occur naturally
even without amplification, and such other faults, in addition to the ones we intend
to exercise, can also occur in a program execution. We report the results of our
investigation of incidental faults during our experiments in Table 4.6.

As the table shows, two of our pairs of programs exhibit faulty behaviors more
frequently than others. In these cases, the faults we observed included both targeted
and incidental faults and additional analysis of the data was required to distinguish
them. All of the cases show, however, that RCRF was effective for reproducing the
targeted faults.

4.5.4 Detecting Unreported Faults

We also discovered that in some cases, the repair specified in a bug report [37] does
not completely fix faults. As shown in Table 4.4, in our application pair (locate,
updatedb), the repair described in the report fixes the UPDATEDB script but races
Table 4.6: Numbers of Targeted Races and Incidental Races

<table>
<thead>
<tr>
<th>PuD1</th>
<th>PuD2</th>
<th>Type of Races</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPDATEDB</td>
<td>UPDATEDB</td>
<td>Targeted 85</td>
</tr>
<tr>
<td>LOCATE</td>
<td>UPDATEDB</td>
<td>Incidental 25</td>
</tr>
<tr>
<td>BASH</td>
<td>BASH</td>
<td>Targeted 562</td>
</tr>
<tr>
<td>MV</td>
<td>RM</td>
<td>Incidental 746</td>
</tr>
<tr>
<td>RM(FTS)</td>
<td>SYMLINK</td>
<td>Targeted 7</td>
</tr>
<tr>
<td>MKNOD</td>
<td>CLEANER</td>
<td>Incidental 0</td>
</tr>
<tr>
<td>MKFIFO</td>
<td>CLEANER</td>
<td>Targeted 10</td>
</tr>
<tr>
<td>MKDIR2</td>
<td>CLEANER</td>
<td>Incidental 2</td>
</tr>
<tr>
<td>MKNOD2</td>
<td>CLEANER</td>
<td>Targeted 10</td>
</tr>
<tr>
<td>MKFIFO2</td>
<td>CLEANER</td>
<td>Incidental 0</td>
</tr>
<tr>
<td>LN</td>
<td>CLEANER</td>
<td>Targeted 40</td>
</tr>
<tr>
<td>FIND</td>
<td>CLEANER2</td>
<td>Incidental 30</td>
</tr>
<tr>
<td>C_SPLIT</td>
<td>MSIGSHOOTER</td>
<td>Targeted 60</td>
</tr>
<tr>
<td>PXZ</td>
<td>STALER</td>
<td>Incidental 20</td>
</tr>
<tr>
<td>LOGROTATE</td>
<td>STEALER2</td>
<td>Targeted 270</td>
</tr>
<tr>
<td>PS</td>
<td>GREP_SCRIPT</td>
<td>Incidental 2</td>
</tr>
<tr>
<td>MARS_HIGH</td>
<td>MARS_LOW</td>
<td>Targeted 24</td>
</tr>
<tr>
<td>MIXXX</td>
<td>MIXXX</td>
<td>Incidental 54</td>
</tr>
<tr>
<td>MULTITHREAD</td>
<td>MULTITHREAD</td>
<td>Targeted 11</td>
</tr>
</tbody>
</table>

can still occur in LOCATE. Using RCRF we were still able to reproduce races after the suggested repair has been applied. We also found another fault in UPDATEDB-UPDATEDB where a race can cause one update to succeed and the other simultaneous update to fail.

We also tried to reproduce previously known faults in a subsequent version of the same library. A bug report [41] indicates that process-level races can occur in MKDIR, MKNOD, and MKFIFO commands from GNU Core Utils 5.2.1. However, a subsequent bug report [42], indicates only that MKDIR2 still suffers from process-level races. The report does not refer to MKNOD2 and MKFIFO2. Since they are all from the same version of GNU Core Utils 5.97, we decide to test all of them and we can still reproduce the same previously reported races in version 5.97.
4.5.5 Evaluating Repair Effectiveness

So far, we have applied RCRF to reproduce reported faults. However, RCRF can also be used by developers to observe the effectiveness of a bug fix based on applying delays to avoid concurrency faults. In ps-grep_script, races are inevitable due to the way the command-line interface sets up the communication pipe. A possible fix is to inject a delay prior to launching grep. While this delay injection is done manually by a developer, we can still apply RCRF to observe (via the use of our runtime monitoring tool and oracle) whether the solution is effective.

4.6 Work Related to RCRF

There has been a great deal of research on debugging concurrent programs at the thread level [74,75,76,77,78]. These techniques, however, focus on multi-threaded applications within single processes, and have rarely been adapted to deal with concurrency faults occurring across processes.

There are several dynamic techniques that replay or reproduce concurrency faults by permuting thread interleavings. We have already described active testing techniques including CALFuzzzer [64] and UPC-Thrille [65]. CHESS [32] captures information on nondeterminism during program execution and allows the scheduler to later reproduce a faulty execution by replaying nondeterministic choices. These techniques, however, focus on thread-level concurrency while ignoring concurrency faults at the process level.

There has been much research on fault localization for concurrent programs [27, 79,80,81,82]. For example, Park et al. [27] monitor memory-access patterns among threads and detect data-access patterns associated with a program’s pass/fail results. They then use statistical debugging to report data access patterns with suspiciousness
scores. Wang et al. [83] identify of shared memory access pairs that behave distinctively in failed and successful runs, and pinpoint root causes using different test procedures. Again, these techniques focus on thread-level concurrency faults. In addition, they are applied only during pre-deployment.

There have been a few approaches created for detecting races at the process level [3, 66, 84, 85]. We have already discussed RACEPro [3]. In addition, detecting TOCTOU races [85] has been a topic in the security community. TOCTOU races typically occur when the permission check and use of a file in one process is not atomic, allowing a malicious process to slip in and attack the file. Our prior work on SimRacer leverages active testing [17] to reduce false positives during dynamic race detection. However, these techniques all focus on testing and detection and do not consider the debugging of process-level concurrency faults.

There have been several approaches to detect synchronization contention [86, 87, 88]. There are also prior studies on contention reproduction and optimization [89, 90, 91, 92]. Work by Carril and Tichy [89] targets C Programs using the pthread library on LLVM [93] framework to reproduce race and deadlocks. Again, this work operates at the thread level.

Other work [90, 91] attempts to reproduce deadlocks and concurrency bugs in Java applications. Work by Yu and Pradel [92] attempts to detect, localize, and optimize synchronization bottlenecks by using PIN [67], a binary instrumentation framework. Our framework, on the other hand, focuses on concurrency fault reproduction at the process level. Thus, our approach needs to overcome issues such as language heterogeneity, dealing with system calls and signals, and dealing with different types of shared resources. Our approach is also simple and requires no complicated mechanisms such as recording and replaying [3]. Instead, we attack the error prone process of manually injecting delays and turn it into an automated process. As shown in this
work, the approach is also flexible and can handle various types of concurrency faults and support different kernels and tools to monitor system events.

4.7 Conclusion

Debugging process-level concurrency faults is challenging because the sources of these faults can occur in system-level events, and reproducing them requires favorable process scheduling. To empower engineers with the tools needed to effectively debug process-level races, we have introduced RCRF, a bug reproduction framework that provides the observability and controllability necessary to effectively reproduce reported faults. Our evaluation shows that RCRF is effective at reproducing challenging faults due to under- and over-constrained process interactions (e.g., races, deadlock, and contention delays). Given twenty previously reported, real-world faults, RCRF was able to reproduce all of them while stress testing was able to reproduce only six. RCRF also reveals other sources within a program that can produce reported faults, allowing engineers to formulate repairs that address all sources. The framework also allows engineers to explore whether other types of concurrency faults can occur on these sources at modest cost.
Chapter 5

RDSC: Applying Reaching Definitions to Create a Generic Race and Atomicity Violation Detector

5.1 Introduction

A prior study has shown that the most common cause of concurrency faults are atomicity violations [1]. As previously mentioned, an atomicity violation occurs when one process or thread can affect a shared resource’s state within a block of code that is being executed by another process or thread. This means that concurrent executions of these processes would generate different results than if these processes were executed serially. The most challenging aspect of detecting atomicity violations, as well as most types of concurrency faults, is that these faults can occur nondeterministically. As such, we need to be able to observe the concurrent execution events and control execution interleavings that give us the best chance to expose these faults.

In the context of our work, understanding how data is accessed and how it can flow
from one application to the next is critical for detecting and reproducing concurrency faults. More specifically, our work focuses on shared components that cannot be easily seen by simply analyzing the source code. It requires that we observe various operating system events to understand how two or more applications are related. As such, our approach relies on dynamic analysis to gather these necessary kernel events that can be used for processing.

As we have shown in Chapter 4, a mechanism that can effectively detect concurrency faults is internal oracles or detectors, which specify correct execution interleavings so that dynamic execution interleavings that do not match the correct ones are identified as faults. For RCRF, we have used Linear Temporal Logic to construct internal oracles to detect specific types of process-level races. We then used code injection to create yield points that can amplify the chance of reproducing the reported faults. However, to detect atomicity violations, instead of looking for conflicting pair of operations that can race, we look for two sequences of operations or blocks of code that should be atomically executed but can interleave with each other. To do so, we rely on a data-flow technique based on reaching definitions that developers can use to semantically and syntactically observe programs [94].

While it is possible to use mechanisms such as linear temporal logic or happen-before relationship to detect atomicity violations [1], such mechanisms often require that we reduce a sequence of instructions that must be executed atomically into an event. In doing so, any parallel execution of two related events would indicate an atomicity violations. While such approaches have been effective for detecting atomicity violations, they do not provide sufficient information to reveal exactly where in an atomic sequence a violation occurs and which operation causes the violation. Knowing this information is important for atomicity violation reproduction based on known or previously reported symptoms or behaviors. Our approach, based on reaching
definitions, can provide such information because our approach considers all operations and tracks the sources that cause a violation.

As an example, when an application deletes a file, there are multiple data structures in addition to the file itself that must be modified. A typical file access often involves accessing an i-node (data structure of filesystem objects of operating system). It may also require changing permission, removing links to the file, and then creating a new link that may be used to recover the file in the case of accidental delete or drive corruption. These operations need to be done atomically to ensure that another process that may need to access the same file does not incorrectly update the state of the file. However, if we collapse all these operations into a single event, we may not be able to identify the actual operation within the atomic block that causes the violation. Furthermore, the violation may appear in the data structures that maintain the file’s metadata or the actual file itself. Being able to observe every operation within an atomic block provides us with more precise information about the violation.

In the example just given, we need to understand all the necessary operations performed by the OS that are related to each file access. This is done by modeling an event (e.g., writing to a file) to include relevant operations. Note that this is a common procedure in prior techniques that deal with process-level races including RACEPRO [3], SimRacer [66], and our own RCRF. Such modeling also gives us insights into which sequences of system calls and operations must be done atomically. Therefore, through this modeling process, we are able to define atomic blocks that exist in these programs. We then apply reaching definitions analysis to help identify atomic sequences of events from each application that can interact with those in other applications through shared system resources, resulting in atomicity violations.

Because many types of races are also related to atomicity violations, we also hypothesize that by developing an oracle to detect atomicity violations, we may be
able to cover different types of races using just one oracle. In RCRF, we created 16 custom oracles based on fault descriptions reported in various repositories for the applications we used. Each of these oracles looks for a specific pair of conflicting operations that can be performed on a specific resource. This means that the window for a race to occur is bound to those two operations colliding. The definition of atomicity violation, on the other hand, is higher level and less restrictive. If a state within an atomic block executed by a process can be seen by another process, we have a violation. If each of those racy operations can be bound to an atomic block, we would be able to detect races by using an atomicity violation detector.

RDSC is a new oracle that extends the collection of custom oracles used by RCRF. It is based on an adaptive reaching definition algorithm for checking atomicity violations found in kernel event log files. This adaptive algorithm is more precise than a typical reaching definitions algorithm. By tracing the kernel event log file, the RDSC semantically selects one execution path from the code and traces the path backward to identify shared resources for potential atomicity violations. Because the traditional reaching definitions algorithm statically traces back for usages of all variables on all possible execution paths, this can lead to more generated information that must be processed. In addition, it can also lead to more false positives.

To highlight the differences between our adaptive and a typical reaching definitions algorithm, we compare the two algorithms in the next section. We then describe the RDSC framework and empirically evaluate it using a subset of the applications (16 pairs) described in Chapter 3. We then report the results and discuss them in greater detail. We then describe prior work related to RDSC.
5.2 Background

5.2.1 Reaching Definitions (RD)

Reaching definitions is a static forward may data-flow approach that propagates a value assigned to each variable from the beginning of a program to all reachable lines of code or blocks. Each line or block of code has its own \( Gen() \) and \( Kill() \) function to generate a new definition for new values stored in a variable and to remove prior definitions. The analysis propagates information forward using transferring functions. During the propagation along the line or the block of code, the definitions are modified by \( Entry() \) and \( Exit() \) functions.

\[
Gen_{RD} : \begin{cases}
\text{Gen}_{RD}(x=e^{line}) = \{(x, \text{line})\} \\
\text{Gen}_{RD}(\ldots^{line}) = \emptyset
\end{cases}
\]  

(5.1)

\[
Kill_{RD} : \begin{cases}
\text{Kill}_{RD}(x=e^{line}) = \forall (x, l) | l \in \cup \text{line} \\
\text{Kill}_{RD}(\ldots^{line}) = \emptyset
\end{cases}
\]  

(5.2)

\[Entry_{RD}(\text{line}) = \cup \{Exit_{RD}(\text{line}') | \text{line}' \in \text{pred(\text{line})}\}\]  

(5.3)

\[Exit_{RD}(\text{line}) = (Entry_{RD}(\text{line}) \setminus Kill_{RD}(\text{line})) \cup Gen_{RD}(\text{line})\]  

(5.4)

Each line or block of code has its own \( Gen_{RD}() \) and \( Kill_{RD}() \) functions and processes things differently depending on the operation. The \( Gen_{RD}() \) (Equation 5.1) returns variable \( x \) and line of code if it is an assignment statement \( (x = e) \) and the empty set if it is not. Also for the case of \( Kill_{RD}() \) (Equation 5.2), if the code is
assigning a value to the variable $x$, then $Kill_{RD}()$ returns a set of all existing definitions of variable $x$, and returns $\emptyset$ for everything else.

The analysis propagates definitions forward along the control flow graph of a basic block toward the next basic blocks. The goal of this process is to see if the definition reaches a point in a program. $Entry_{RD}()$ and $Exit_{RD}()$ work as the transfer functions to modify the definitions before reaching and after leaving the block or line of code, respectively. With respect to reaching definitions, $Entry_{RD}()$ (Equation 5.3) is a union of all $Exit_{RD}(line')$ values or the value of definitions from the previous steps or line, which can come from more than one direction based on the control flow graph. $Exit_{RD}()$ (Equation 5.4) is simpler, getting $Entry_{RD}()$, removing everything from $Kill_{RD}()$ and adding everything from $Gen_{RD}()$.

![Figure 5.1: The control flow graph of a function "Average"](image)

Tables 5.1 and 5.2 show examples of how reaching definitions works. From Table 5.1, Columns C and D show the $Gen()$ and $Kill()$ values for each line of code. From Table 5.2, Columns C and D show the $Entry()$ and $Exit()$ values along the way as the analysis goes on.
In Table 5.1, source code lines 1 - 3 are the initializations for new variables, so they have only Gen() values without Kill() values. Lines 4 and 8 are not assignment statements. In line 5 of the source code, the variable sum is added to the variable x, so Gen() returns a set with a pair of variable sum and line number 5, while Kill() kills all existing definitions of variable sum, which include the variable sum from lines 3 and 5. The code in line 6 with variable x is handled similarly; Gen() returns \{(x, 6)\} and Kill() returns all existing definitions of variable x: \{(x, 2)(x, 6)\}.

The propagation carries all definitions of all variables along the control flow graph, allowing us to better understand the behavior of the code. As an example, the reaching definitions analysis proceeds iteratively from line 1 of the source code down to line 9 along the data flow path shown in Figure 5.1. According to Table 5.2, the analysis starts at line 1 with Entry() returning the empty set and Exit() returning the first definition of variable i. The definition from Exit() is passed through its successor...
(line 2) as \textit{Exit()} returns another new definitions of \textit{x}, and these steps occur again in line 3.

The analysis begins to perform differently at line 4 of the source code. This line has two data flow directions from lines 3 and 6; the \textit{Entry()} unions the exit from lines 3 and 6 and the \textit{Exit()} results from \textit{Entry()} deleting the values from \textit{Kill()} and adding the values from \textit{Gen()}. The analysis works the same on lines 5 - 6.

The analysis focuses on the definition of one variable and determines if it can naturally reach or block of code. For example, if we want to determine if the value of variable \textit{x} in line 2 can reach line 8, we look for the definition on line 8. Definition \((x, 2)\) on the \textit{Exit()} of line 8 shows that the value from line 2 can reach the end.

\subsection*{5.2.2 Adaptive Reaching Definitions (ARD)}

There are two alternative ways we can control and observe the behavior of RD analysis. First, we can run the analysis through a sequential program and see how the analysis steps through the data flow from the first line of code to the last line (Figure 5.2). Second, since each definition of each variable is independent of the others, we can alter the algorithm to focus on one variable at a time. This simplifies the set of definitions that we need to track and reduces the amount of information being generated. Next we describe these two approaches in turn.

\subsubsection*{5.2.2.1 Using a sequential program}

We use a simple a sequential program to observe how RD analysis works and how the definitions from sequential program are modified. Note that there is only one entry into each point of code, which changes the reaching definitions analysis from a \textit{may} analysis to be a \textit{must} analysis. Also, there is only one definition for one variable as
01: double c = 0;
02: System.out.print("enter a: ");
03: int data = new Scanner(System.in).nextInt();
04: c = (data * data);
05: System.out.print("enter b: ");
06: data = new Scanner(System.in).nextInt();
07: c += (data * data);
08: System.out.println("c = "+Math.sqrt(c));

Figure 5.2: Sequential flow code to calculate the hypotenuse in the Pythagorean theorem

Table 5.3: Gen() and Kill() for Each Line in a Sequential Program

<table>
<thead>
<tr>
<th>Line (A)</th>
<th>Source code (B)</th>
<th>Gen() (C)</th>
<th>Kill() (D)</th>
</tr>
</thead>
<tbody>
<tr>
<td>01:</td>
<td>double c = 0;</td>
<td>{(c,1)}</td>
<td>∅</td>
</tr>
<tr>
<td>02:</td>
<td>System.out.print(&quot;enter a: &quot;);</td>
<td></td>
<td>∅</td>
</tr>
<tr>
<td>03:</td>
<td>int data = new Scanner(System.in).nextInt();</td>
<td>{(data,3)}</td>
<td>{(data,3)(data,6)}</td>
</tr>
<tr>
<td>04:</td>
<td>c = (data * data);</td>
<td>{(c,4)}</td>
<td>{(c,1)(c,4)(c,7)}</td>
</tr>
<tr>
<td>05:</td>
<td>System.out.print(&quot;enter b: &quot;);</td>
<td></td>
<td>∅</td>
</tr>
<tr>
<td>06:</td>
<td>data = new Scanner(System.in).nextInt();</td>
<td>{(data,6)}</td>
<td>{(data,3)(data,6)}</td>
</tr>
<tr>
<td>07:</td>
<td>c += (data * data);</td>
<td>{(c,7)}</td>
<td>{(c,1)(c,4)(c,7)}</td>
</tr>
<tr>
<td>08:</td>
<td>System.out.println(&quot;c = &quot;+Math.sqrt(c));</td>
<td></td>
<td>∅</td>
</tr>
</tbody>
</table>

Table 5.4: Entry() and Exit() for Each Line in a Sequential Program

<table>
<thead>
<tr>
<th>Line (A)</th>
<th>Source code (B)</th>
<th>Entry() (C)</th>
<th>Exit() (D)</th>
</tr>
</thead>
<tbody>
<tr>
<td>01:</td>
<td>double c = 0;</td>
<td>∅</td>
<td>{(c,1)}</td>
</tr>
<tr>
<td>02:</td>
<td>System.out.print(&quot;enter a: &quot;);</td>
<td>{(c,1)}</td>
<td>{(c,1)}</td>
</tr>
<tr>
<td>03:</td>
<td>int data = new Scanner(System.in).nextInt();</td>
<td>{(c,1)}</td>
<td>{(c,1)}</td>
</tr>
<tr>
<td>04:</td>
<td>c = (data * data);</td>
<td>{(c,1)(data,3)}</td>
<td>{(c,1)(data,3)}</td>
</tr>
<tr>
<td>05:</td>
<td>System.out.print(&quot;enter b: &quot;);</td>
<td>{(c,4)(data,3)}</td>
<td>{(c,4)(data,3)}</td>
</tr>
<tr>
<td>06:</td>
<td>data = new Scanner(System.in).nextInt();</td>
<td>{(c,4)(data,3)}</td>
<td>{(c,4)(data,3)}</td>
</tr>
<tr>
<td>07:</td>
<td>c += (data * data);</td>
<td>{(c,4)(data,6)}</td>
<td>{(c,4)(data,6)}</td>
</tr>
<tr>
<td>08:</td>
<td>System.out.println(&quot;c = &quot;+Math.sqrt(c));</td>
<td>{(c,7)(data,6)}</td>
<td>{(c,7)(data,6)}</td>
</tr>
</tbody>
</table>

shown in Table 5.4. Kill() kills all prior definitions whenever the variable has been newly reassigned and Gen() creates new definitions (Table 5.3).
5.2.2.2 Focusing on one resource at a time

RD does not focus on the reading of variables; instead, it focuses on assignments to variables. Thus, we can treat each variable as independent in our analysis. To do so, we can modify the algorithm to concentrate on one variable at a time. This is beneficial for eliminating long sequences of code that rarely touch the variable. The algorithm can also eliminate several irrelevant lines of code before running the analysis as shown in Tables 5.5 and 5.6, which renders the analysis faster and more efficient.

5.3 Introducing RDSC

In this section, we introduce RDSC, an oracle based on an adaptive reaching definitions algorithm, that analyzes kernel event logs for atomicity violations. We focus on atomicity violations that can occur in each shared resource due to accesses by concurrent threads or processes. Since these threads or processes may use many
shared resources, we want to efficiently optimize the algorithm, by processing one
shared resource at a time, and eliminating kernel events that have nothing to do with
the targeted shared resources. As such, the process iteratively identifies all possible
shared resources among concurrent threads or processes and eliminates kernel events
not related to such resources. Afterward, it applies the proposed atomicity violation
detection approach based on reaching definitions as the oracle.

5.3.1 RDSC Workflow

Figure 5.3: Using RDSC to detect atomicity violations in system call logs

Figure 5.3 shows how to iterate over one shared resource at a time from the
resource pool (A) and filter out irrelevant lines of kernel events (Box 1) from the log
file. The new pruned log is sent to the atomicity violation detector (Box 2) and a final
atomicity violation report is produced. If there are more shared resources in the pool,
the process repeats.

Algorithm 2 details the workflow. Line 2 filters out unrelated kernel event lines
from the log, while the functions in lines 3 and 4 segregate the new list (OSh) into
two groups according to the program that generates the events (e.g., PN1 and PN2).
All the information is then provided to the atomicity violation detector and used to
produce the violation report (Rs).
Algorithm 2 The framework

Require: a set of shared resource \((Sh)\),
- a list of system calls from parallel processing of processes \((LSC)\),
- an optimized list of system calls accessing the shared resource \(s_i\) \((OSh)\)

1. for all \((s_i) \in Sh\) do
2. \(OSh \leftarrow \text{filteringOut\_Irrelevant\_SystemCalls}(s_i, LSC)\)
3. \(PN_1 \leftarrow \text{collectingAllSubProcess}(OSh, 1)\)
4. \(PN_2 \leftarrow \text{collectingAllSubProcess}(OSh, 2)\)
5. \(Rs[s_i] \leftarrow isRDSC(s_i, OSh, PN_1, PN_2)\)
6. end for
7. return a set of result for each shared resource \((Rs)\)

5.3.2 Applying ARD on System Call Logs (RDSC)

Algorithm 3 presents the process used to check for atomicity violations inside a targeted list of system calls by applying the ARD algorithm. Given a list of system calls, we capture the dynamic kernel event log that lists the kernel’s operations, including accessing the specific shared resources and possibly containing atomicity violations \((OSh)\). The proposed RDSC look in the log for atomicity violations caused by processes.

Inside the log, there are main processes and sub processes. The \(RDSC\) creates two sets of events \((PN_1, PN_2)\), each belonging to a process. For example, a log may include operations by the main process and operations by its child processes.

Algorithm 3 focuses on atomicity violations that occur when multiple processes race \((Osh)\) for a particular shared resource \((s)\). In this algorithm, we use three variables to trace the list of system calls \((OSh)\). First, \(LatestWritting\) (initialized in line 1) is a collection of the latest "writing" system call (into the shared resource), so we can look back for it. The second variable, \(Counter\) (initialized in line 2) is for counting the number of context switch. The atomic block is not completed until the second context switch happens. The last one is \(prev\) (initialized in line 3). This is a collection of the previous system calls to use for comparison.
Algorithm 3 isRDSC (Reaching Definition Algorithm on System Call Log)

Require:  the shared resource \((s)\),
          a list of suspected SysCalls \((OSh)\),
          a set of process number on Group one \((PN_1)\),
          a set of process number on Group two \((PN_2)\)

1: LatestWriting \(\leftarrow \{sc_0\}\)
2: Counter = 0
3: Prev \(\leftarrow \{sc_0\}\)
4: for all \((sc_i \in OSh)\) do
5:   if (isSwitched(Prev, \{sc_i\})) then
6:      Counter ++
7:   if (Counter > 1) then
8:      if (not fromTheSameSide(LatestWriting, \{sc_i\})) then
9:         return true
10:     end if
11:   end if
12: end if
13: if (isWriting(sc_i)) then
14:   LatestWriting \(\leftarrow \{sc_i\}\)
15: end if
16: Prev = \{sc_i\}
17: end for
18: return false

With respect to data flow analysis, \(Gen()\) and \(Kill()\) are the sets for generating the transfer function of each block or statement in the source code. The transfer function contains all variables that have been modified inside that block or statement and the definitions depend on the algorithm or analysis. \(RDSC\), however, analyzes one shared resource at a time as shown from lines 4 to 19. Variable \(LatestWriting\) represents the definition of that particular shared resource. \(Gen()\) and \(Kill()\) are processed in lines 14 to 16. \(Gen()\) generates \(\{sc_i\}\) and \(Kill()\) works by simply replacing the value into the variable \(LatestWriting\) in line 15.

Typically, the classic reaching definitions algorithm starts from the beginning of the program and propagates information through the end. However, \(RDSC\) is designed
to detect atomicity violations. As such, we terminate propagations of definitions as soon as RDSC detects an atomicity violation. Inside the algorithm, the code in lines 5 to 12 and line 18 works as an atomicity violation checking mechanism. The detection process works in concert with RCRF, which helps amplify the chance of reproducing atomicity violations by inserting yield points to suspend each process.

To illustrate this algorithm, assume that a process $P_0$ is suspended due to quantum expiration. At this point, the system observes the state of a faulty shared resource. This is done by simply identifying the last write operation, by reading the value stored in $LatestWriting$. Assume that while $P_0$ is blocked, $P_1$ is scheduled and writes to the same shared resource. This would cause an update in $LatestWriting$. When $P_0$ returns, it checks $LatestWriting$ and finds that the state of the shared resource has been changed by $P_1$.

5.4 Empirical Study

The adaptive reaching definitions algorithm can help us dynamically detect atomicity violations. With the reproduction capability of RCRF, we can exercise a large range of process interleavings. When we study RCRF, we create an oracle to be applicable to a type of reported race. Typically, such an oracle cannot be generically applied to other types of races as they may use different resources and the system calls used to access those resources may also be different. The proposed algorithm based on ARD produces a more generic oracle so it can be applicable across diverse types of races.

We conducted an empirical study to answer the following two research questions.

**RQ1:** By including a RDSC oracle in RCRF, how effective is RDSC on detecting known races during reproduction?
RQ2: By including a RDSC oracle in RCRF, can we reduce the number of custom oracles previously used to evaluate RCRF?

5.4.1 Object of Analysis

We use the first 16 pairs of applications described in Chapter 3. For the time being, RDSC is implemented to detect atomicity violations on files. The other four application pairs not included in this study use shared resources other than files so our oracle would not be able to detect violations. However, we can extend RDSC to cover atomicity violations on these and other types of shared resources.

5.4.2 Variables and Measures

Independent Variable. We wish to measure the effectiveness of our new algorithm, RDSC, in generically detecting atomicity violations and subsequently races that were previously reproduced by RCRF. We also wish to measure the savings in terms of the number of custom oracles that must be developed to effectively reproduce races. As such, our baseline system is RCRF.

Dependent Variable. The effectiveness of RDSC (RQ1) is determined by the numbers of races in each application that the custom oracle and RDSC can detect during the reproduction attempts. We again define two types of races that can be detected, dynamic and static. When an execution detects one or more races, we count these as detected races. However, we exercise each interleaving 100 times, so it is possible that the same race may be detected repeatedly across these 100 runs. As such, the usefulness of dynamic races is to show the effectiveness of our technique to reproduce races. Static races, on the other hand, report the unique sources of races that have been detected across the 100 runs.
With respect to RQ2, we compare the number of oracles that we need to reproduce races in the 16 pairs of applications. For RCRF, we use custom oracles. We then use RDSC alone to try to detect these races. For the cases in which we could not detect all races that have been previously detected by the custom oracles, we simply add the custom oracle for that particular type of race to improve detection effectiveness. At the end of the evaluation process, we record the number of oracles (i.e., RDSC + custom oracles) that we need to detect the same races detected by using custom oracles alone.

5.4.3 Study Methodology

The methodology we used to evaluate RDSC is similar to that used to evaluate RCRF. Again, we executed each pair multiple times to help neutralize the effect of non-determinism. In addition, as we only tried to use a generic oracle (RDSC) instead of multiple customized oracles to detect multiple types of races, we want to ensure we use enough runs to detect as many types of races that can occur within each pair of programs. As such, we run each pairs of applications 100 times instead of 10.

5.4.4 Threats to Validity

The primary threat to external validity in this study involves the object programs utilized. We have studied just 16 pairs of faulty programs. However, the programs are real and the faults are real and non-trivial to reproduce. A second threat involves the representativeness of our oracles. These threats can be addressed only through further studies.

The primary threat to internal validity involves potential errors in the implemen-
The primary threat to construct validity relates to the fact that we study effectiveness and savings relative to applications of RDSC, but do not yet assess whether the approach helps engineers localize and correct targeted faults more quickly than current approaches.

5.4.5 Results

Tables 5.7 and 5.8 provide data for both research questions. Columns A and B are the program pair (PuD1 and PuD2). These program pairs are the subset of objects of analysis used to evaluate RCRF. Column C in Table 5.7 is the total number of runs, using 100 runs per interleaving. For example, in the case of UPDATEDB – UPDATEDB, we have 36 potential interleavings that involve shared resources. We exercise each interleaving 100 times and therefore, the number of total runs is 3,600. Column C in Table 5.8 is the total number of potential sources of races based on a particular resource.
Table 5.8: Comparing Unique Faults Between RCRF and RDSC

<table>
<thead>
<tr>
<th>PuD1 (A)</th>
<th>PuD2 (B)</th>
<th>Potential Source of Race (C)</th>
<th>RCRF (D)</th>
<th>RDSC (E)</th>
<th>RCRF ∩ RDSC (F)</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPDATEDB</td>
<td>UPDATEDB</td>
<td>9</td>
<td>5</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>LOCATE</td>
<td>UPDATEDB</td>
<td>9</td>
<td>2</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>BASH</td>
<td>BASH</td>
<td>196</td>
<td>70</td>
<td>84</td>
<td>70</td>
</tr>
<tr>
<td>MV</td>
<td>RM</td>
<td>16</td>
<td>5</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>RM(FST)</td>
<td>SYMLINK</td>
<td>6</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MKDIR</td>
<td>CLEANER</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MKNOD</td>
<td>CLEANER</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MKFIPO</td>
<td>CLEANER</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MKDIR2</td>
<td>CLEANER</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MKNOD2</td>
<td>CLEANER</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MKFIPO2</td>
<td>CLEANER</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>LN</td>
<td>CLEANER</td>
<td>6</td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>FIND</td>
<td>CLEANER2</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CSPLIT</td>
<td>MSGSHOOTER</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>PXZ</td>
<td>STEALER</td>
<td>8</td>
<td>7</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>LOGROTATE</td>
<td>STEALER2</td>
<td>77</td>
<td>16</td>
<td>16</td>
<td>15</td>
</tr>
</tbody>
</table>

between the two programs. For example, in the case of UPDATEDB – UPDATEDB, we have nine potential sources of races that involve a shared resource.

To answer RQ1, we compare the results listed in Columns D and E in both tables and report in the intersection in Column F. For Table 5.7, Column D reports the number of total faults dynamically detected by the customized oracles based on Linear Temporal Logic (LTL) used to evaluate RCRF, and Column E is the number of faults dynamically detected by RDSC. We identify the same faults that have been detected by both RCRF (Column D) and RDSC (Column E) for each pair of PuDs, and report that result in Column F.

An atomicity violation may result in a type of race that our customized oracle is not designed to detect (e.g., a customized oracle is designed to detect file corruptions but not i-node corruptions), so we turn our attention to the atomicity violations that are also detected as races by our customized oracles. As shown in Column F of Table 5.7, RDSC can detect over 99% of the faults that were detected by our customized oracles in nine pairs of apps. These are BASH-BASH, MKDIR-CLEANER, MKNOD-CLEANER, MKFIPO-CLEANER, MKDIR2-CLEANER, MKNOD2-CLEANER, MKFIPO2-CLEANER, FIND-.
CLEANER2, and CSPLIT-MSIGSHOOTER. It can partially detect similar faults in five of the remaining seven pairs of applications (81% or less). It cannot detect any similar faults in the remaining two pairs of applications.

In terms of atomicity violation effectiveness, UPDATEDB-UPDATEDB has the greatest number of runs with detectable atomicity violations at 70%. LOCATE-UPDATEDB and LN-CLEANER have 66% and 50% of the total runs that have detectable atomicity violations. RDSC detects atomicity violations in fewer than 40% of the total runs for the remaining pairs of applications. Note that RDSC can detect atomicity violations in 14 out of 16 pairs of apps.

In terms of fault detection effectiveness, Column D in Table 5.8 reports the number of unique faults that have been reproduced by RCRF during the 100 runs. Again, a unique fault is referred to as at least one instance of a possible fault that has been exercised. A potential fault may have been exercised many times in a sequence of repetitive runs. As long as at least one run exercises that fault, we count it as a unique fault. Column E is the number of unique faults that have been reproduced by RDSC during the 100 runs. We identify the same faults that have been detected by both RCRF (Column D) and RDSC (Column E) for each pair of PuDs, and report that result in Column F.

As shown in Column F in Table 5.8, RDSC can detect the same faults detected by RCRF in 13 out of 16 pairs of applications. The only exceptions are the last three pairs of applications, wherein the majority of races are not due to atomicity violations. This suggests that RDSC can be used as a general race detector for process-level races.

Turning to RQ2, we first list the names of all oracles that are available in Table 5.9. Next, we compare the numbers of oracle that RCRF and RDSC use by listing them in Table 5.10. Column (A) shows the framework, Columns (B) and (C) are the pairs of PuDs, and Columns (D) to (V) are oracle numbers. In Table 5.10, we list the
Table 5.9: List of Oracle Numbers and Names Used in RCRF and RDSC

<table>
<thead>
<tr>
<th>Number</th>
<th>Name of Oracle</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ARD</td>
</tr>
<tr>
<td>2</td>
<td>File Existing</td>
</tr>
<tr>
<td>3</td>
<td>One Rename</td>
</tr>
<tr>
<td>4</td>
<td>Two Rename</td>
</tr>
<tr>
<td>5</td>
<td>Modified While Using</td>
</tr>
<tr>
<td>6</td>
<td>Read Once</td>
</tr>
<tr>
<td>7</td>
<td>Double Open</td>
</tr>
<tr>
<td>8</td>
<td>Unlink Non-Existing</td>
</tr>
<tr>
<td>9</td>
<td>Unlink Once</td>
</tr>
<tr>
<td>10</td>
<td>Unlink Twice</td>
</tr>
<tr>
<td>11</td>
<td>Chmod Once</td>
</tr>
<tr>
<td>12</td>
<td>Rmdir Once</td>
</tr>
<tr>
<td>13</td>
<td>Cannot Remove</td>
</tr>
<tr>
<td>14</td>
<td>Error between stat + unlink</td>
</tr>
<tr>
<td>15</td>
<td>Error between unlink + symlink</td>
</tr>
<tr>
<td>16</td>
<td>Cannot remove folder</td>
</tr>
<tr>
<td>17</td>
<td>Clear all files</td>
</tr>
<tr>
<td>18</td>
<td>Unzip the Log file</td>
</tr>
<tr>
<td>19</td>
<td>Unzip the PXZ file</td>
</tr>
</tbody>
</table>

oracle that each pair of PuDs use and mark them as "1". In the last row of each framework we mark the oracle they use with "Y", before counting the number of "Ys" and showing it in Column (C) on the same row.

According to the results, we are able to reduce the number of custom oracles based on LTLs from 18 to 4. This is because RDSC alone can replace 14 LTL oracles. This results in a 77.78% reduction in the number of oracles. We also see three cases where RDSC is not effective in detecting those types of races and therefore, custom oracles designed to detect such races would still be needed.

5.5 Discussion

5.5.1 Effectiveness

The results in Columns D and E of Table 5.7 collaborate that RDSC is more effective than LTLs. RDSC can detect more faults than LTLs. However, we also have three cases wherein RDSC is not effective. The reason that RDSC cannot detect one
Table 5.10: Results Comparing Oracles Used Between RCRF and RDSC

<table>
<thead>
<tr>
<th>Oracle Number</th>
<th>Framework</th>
<th>RCRF</th>
<th>RDSC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oracle</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(A)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(B)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(C)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(D)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(E)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(F)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(G)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(H)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(I)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(J)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(K)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(L)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(M)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(N)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(O)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(P)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Q)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(R)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(S)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(T)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(U)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(V)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

particular race in CSPLIT-MSIGSHOOTER is because the race is a low-level data race. The specific LTL to detect this race has been designed to monitor the results from
manipulating the interleaving by RCRF. However, the RDSC can detect a different fault in csplit that is not detected by RCRF. This particular race is due to the signal handler trying to remove a file. This is an example of a difference in capability of RCRF and RDSC.

For the other two pairs of applications, pxz-stealer and logrotate-stealer2, RDSC cannot detect all the races detected by RCRF. Similar to the case above, the LTLs used by RCRF have been created to detect specific faults based on the reports, while RDSC is more about code analysis via data-flow. However, RDSC is able to detect one atomicity violation in pxz and logrotate that is not detectable by any customized oracle.

5.6 Work Related to RDSC

Atomicity violation at the thread level has been well studied by several prior research efforts [16, 28, 33, 74, 75, 77]. While many approaches such as Eraser [28], CHESS [74], and DEJAVU [16] find the concurrency bug in general, tools like Atomizer [77] and CTrigger [75] focus on the atomicity violation or both atomicity violation and order violations [33].

Eraser [28] uses the lock-set approach to find the data races in a multithread program. However CHESS expands on this to cover more on memory corruption, deadlock, and livelock [74]. In addition, Atomizer [33] and Ai [77] use the lockset approach to capture atomicity violation dynamically.

DEJAVU [16] uses a yield point technique to enforce interleavings and also uses the record/replay mechanism to observe non-deterministic execution later. In addition to DEJAVU, CTrigger [75] also uses an approach based on record/replay to capture atomicity violations. The aforementioned approaches have been designed to target
races, order violations, and atomicity violations in multithreaded programs. Therefore, their main focus is on detecting and reproducing shared memory access violations. Our approach, on the other hand, has been designed specifically to address the need of reproducing process-level races and atomicity violations. Our work requires the observation a much larger class of shared resources including those that can be accessed by system calls and signals. Our work also supports fault detection when programs are written in different languages.

With respect to reachability testing, there have been many approaches used to dynamically test concurrent programs [95,96,97]. To test for synchronization faults, a reachability based approach exhaustively exercises the programs without previously constructing any static model [95]. The approach combines nondeterministic and deterministic testing [96] to create sequences called SYN-sequences representing the interleavings between processes and threads. Monitoring tools such as Butterfly Analysis [98] and ParaLog [99] apply reaching definitions to monitor running applications for bugs and security attacks. They have been shown to be effective. However, a major shortcoming of this approach is that it can incur high runtime overhead as the number of resources that must be monitored can be quite large.

Our approach exploits the dynamic nature of this approach to reproduce races. However, as a race reproduction framework, our approach does not need to monitor all resources that can possibly encounter faults. Instead, we use bug reports to determine which resources must be monitored and then apply reaching definitions to observe accesses to these resources as part of fault reproduction. This allows our approach to reduce the overhead of reaching definitions.
5.7 Conclusion

In this chapter, we have shown that one way to reduce the effort required to reproduce a specific concurrency fault is to adopt a reaching definitions based oracle that can detect a large class of races. By using the proposed RDSC, we are able to replace 15 customized oracles with a single oracle without suffering from degradation in effectiveness. As our empirical results indicate, RDSC can detect the same races as those detected by customized oracles in 13 out of 16 pairs of applications. At the same time, the study also reveals limitations of RDSC that can be addressed as part of future work.
Chapter 6

CFI: Concurrency Faults Inspector

6.1 Introduction

As previously mentioned, the most prolific type of real-world concurrency fault is the data race [1]. This particular type of fault occurs when multiple processes, signals, and interrupts improperly access shared resources. Due to non-determinism, even races that have been reported by users in bug repositories are still quite difficult to reproduce [3,66,100]. Typically, there are three critical elements needed for a good race report. They are (i) the faulty application, (ii) its symptoms, and (iii) the inducing application(s). However, race reports do not always contain all three elements. For example, in one particular report [101], users did not disclose the inducing application. Reports with such missing information make reproducing races even more challenging.

To ease the process of reproducing races from incomplete bug reports, we need a framework that is able to identify possible applications from within a given set (or a universe) of applications that can interact with the reported application to reproduce the reported race. One possible approach is to exhaustively run every application within the universe alongside the reported application using testing techniques such
as RCRF to try to reproduce the reported race. However, this process can be highly inefficient especially if the number of applications in a universe is large.

An alternative approach, introduced in this work, is to project possible interleavings and then compute whether those projected interleavings can cause the reported race. In this approach, we do not need to run these applications alongside the reported application. Instead, we create a list of Virtual Interleavings (VIs) between the vulnerable application and a universe of possibly inducing applications. The selection of applications in the universe can be based on correlating known shared resources. Deriving known shared resources requires us to run each app once through our observation tools (see Chapter 4 for the tool we used and the list of other possible tools) without forcing specific interleavings. We can then pass these VIs through oracles to predict whether an application can race with the reported application. With this approach, we hypothesize that it is possible to develop an efficient and effective testing framework to reproduce races from incomplete bug reports.

In this chapter, we propose *Concurrency Fault Inspector* (CFI), a framework that can find a set of applications sharing a similar resource usage pattern as that of the reported application so that they can possibly induce the same race as indicated in an incomplete bug report. This framework explores a universe of applications that can run on the same system as that of the reported application to find one or many race-inducing applications. The framework first executes each application using its test suite to identify shared resources and sequences of system calls and signals that it uses. Next, it generates a list of unique and possible VIs before predicting the result using a set of classifiers; each is designed to detect a specific class of concurrency faults. Finally, it employs RCRF to confirm the presence of races by trying to reproduce them between the reported application and the predicted applications.

We evaluate the performance of CFI using 117 applications from four universes
of applications with known faults. Our results indicate that CFI is effective (100% accuracy in our case study) in predicting applications that can induce races in the reported apps. In doing so, it can reduce the testing time by 35%.

In summary, the contributions of this chapter are as follows.

1. We propose Concurrency Fault Inspector (CFI) to identify race-inducing applications in a universe of applications.

2. We design a feature model, generate a training dataset, and experiment with multiple classification algorithms to determine the most accurate algorithm for the given problem. We then perform training and testing to construct a model to classify whether generated VIs can cause a reported race.

3. We perform a case study to evaluate the effectiveness and efficiency of the CFI Framework against that of RCRF to perform an exhaustive evaluation of all applications in a universe. Our results indicate that CFI is more efficient than an approach that performs exhaustive testing using RCRF. It can also accurately predict applications that can race with a given application.

The rest of this chapter is organized as follows. Section 6.2 provides an overview of background information related to this work. Section 6.3 describes the architecture of the CFI framework. Section 6.4 reports our experimental setup to conduct our case study using an incomplete race report and a universe of 117 applications, it also reports the results of our investigation. Section 6.5 discusses the implications of our results and the usefulness of CFI to potentially isolate and debug race conditions. Section 6.6 highlights prior work related to CFI. The last section concludes this chapter.
6.2 Background

This section discusses background information related to incomplete bug reports and how we can extract information that can be used to help CFI identify race-inducing applications.

6.2.1 Analyzing Incomplete Bug Reports

Bug repositories provide a way for users to report faulty behaviors of applications that occur in the field back to the developers. In many systems, bug tracking applications are embedded into systems (e.g., Apple software products or Microsoft operating systems) to automatically formulate reports and request user’s permissions to send the report back to developers when the systems encounter problems. However, there are also many applications that require users to manually report faulty symptoms. For example, BugZilla [102] is the web-based bug-tracker for Mozilla projects. Launchpad [103] is the website to help developers maintaining GNU software and a part of this site also includes a large collection of bug reports from users.

Information that can help developers debug process-level races includes the reported application (i.e., the application that exhibits symptom), the symptoms, and the race-inducing application(s). The report should also include system configurations or other specific information such as the system specification (e.g., software version, OS version). It should also include any generated error messages or log files. Often, each of the symptoms or faulty behaviors requires different evidence or information to help developers debug the applications. Thus, it is common for developers to ask users to provide as much information as possible when reporting faults.

Bug analysis, however, is not an easy task. Most users try to perform a preliminary diagnosis before reporting them. However, the complexity of process-level races can
cause users to mis-diagnose the problems. We have seen bug reports that do not contain sufficient information to diagnose the faults [104]. There are also bug reports that contain incorrect information about the real causes of races [105, 106]. For example, in report [36], the error appears to originate from calling the `find` command. As such, the problem was initially sent to be fixed by the configuration team. A year later, this problem was rerouted to the team that maintains lower-level software that resides in the GNU FINDUTILS.

We have also seen many examples of incomplete reports related to concurrency faults. There are race reports that provide the wrong diagnosis of the sources of the problem [36, 105, 106] or omit important information [104]. Some race reports disclose the vulnerable applications but do not identify the inducing applications [101, 107]. For example, reports may show a symptom of simply displaying an error message, where that particular message alone does not identify the inducing application [101, 107]. While typical users can often identify the presence of data races in an application, knowing which other application(s) can induce that particular race may not be trivial. The goal of CFI is to help engineers identify race-inducing applications when a race report does not disclose such information.

Figure 6.1 [101] is an example of an incomplete bug report. A user reports an error in MySQL from the signal handler `epoll`. The report includes the symptom (i.e., a bad file descriptor error message), an explanation of how the race occurs, and a repair that has been applied. However, the proposed repair does not fix this fault; it simply changes some configurations and makes a simple repair to an API. This repair, in effect, masks the fault. It does not address the sources that corrupt the file descriptor. Further diagnostics reveal that a data race is the source. However, the engineer still does not know which application races with it.

Figure 6.2 [107] is another race report that does not disclose the inducing application.
Figure 6.1: Bug report from MySQL number 36537

[6 May 2008 14:58] Cyril SCETBON

Description:

Nodes are crashing when inserting data.

Here is the error encountered:

Failed to add fd to epoll-set...giving up! Bad file descriptor
2008-05-06 16:37:06 [ndbd] INFO -- Signal 6 received; Aborted

We are on a X86_64

How to repeat:

restart node.
**Figure 6.2: Bug report from Slidematch number 641521**

**Intermittent paused song restart**

Bug #641521 reported by Eric Parker on 2010-09-17

This bug affects 1 person

<table>
<thead>
<tr>
<th>Affects</th>
<th>Status</th>
<th>Importance</th>
<th>Assigned to</th>
<th>Milestone</th>
</tr>
</thead>
<tbody>
<tr>
<td>slidematch</td>
<td>Incomplete</td>
<td>Undecided</td>
<td>Unassigned</td>
<td></td>
</tr>
</tbody>
</table>

Also affects project, Also affects distribution/package, Nominate for series

**Bug Description**

Very occasionally a paused song appears to have restarted without any user action. In one case this led to the currently playing song being removed from on deck, in another case it led to the app playing things after pausing and leaving for lunch. Unable to reproduce. Best guess at this point is we are looking for an orphan NextItemTimer, probably as a result of some kind of race condition. Generally this all works fine, steps to reproduce would be most welcome.

Add tags

<table>
<thead>
<tr>
<th>Eric Parker (theriex) on 2010-10-06</th>
</tr>
</thead>
<tbody>
<tr>
<td>Changed in slidematch:</td>
</tr>
<tr>
<td>status: New $\rightarrow$ Incomplete</td>
</tr>
</tbody>
</table>
The report contains the symptom and the incorrect output from the application. However, there is no diagnostic information. As such, this fault could not be reproduced and therefore, has not been repaired. The report now has the status of *incomplete*.

Part of analyzing a race report is to determine for certain whether there is indeed a data race. If there is, then we want to know what type of race it is (e.g., a race on file-related resources or a race on shared memory). The critical information that must be extracted is where the engineer can observe the race if she is able to reproduce it. Typically, shared resources are files, file structures (iNodes), memory, pipes, or devices like network ports or channels. However, not all races show clear symptoms [37,45]. Sometimes, races can cause silent errors that may not be noticeable [38]. As such, the ability to extract information from incomplete bug reports may be based on the prior experience of developers or engineers. However, in Chapter 4 we have described tools that can help identify resource usage within an application.

### 6.2.2 Converting A Happen-Before Graph to Feature Matrix

In Chapter 4, we used Happen-Before graphs [70] to represent sequences of kernel events. The concept of Happen-before has also been used in prior work to represent interleavings between processes [59,66]. For CFI, we also use Happen-before graphs in the form of a feature matrix to train and test our classifiers.

Creating a feature matrix to represent a Happen-before graph requires more than just the edge before and after our target node. We also need to encode the relationships between our target node and the rest of the nodes inside the graph. For example, Figure 6.3 shows the relationship among the nodes inside a Happen-before graph. To represent the graph as a matrix, we need to include all nodes and their relationships.

To generate a feature matrix for every single interleaving, we use the data from
the kernel event log. After the kernel event log is processed to select only the relevant events, we set a mark on each event to indicate the caller of the event. This is done by setting a cell in the matrix to ‘1’. An illustration of our feature matrix is shown in Figure 6.4.

We use the relevant kernel events to represent the column and the row of each feature matrix and initialize every cell inside the matrix to 0. On every graph, there are edges to represent the relationship between the nodes. We define every edge on the graph by marking it as 1 on the feature matrix. For example, Figure 6.5 shows an edge between the system call 1stat64 and 1open. As such, we set the cell that is defined by 1stat64 (row 1) and 1open (column 3) to 1. As a reminder, the number before each event (e.g., 1stat64) indicates the process that calls that event. Figure 6.6 illustrates a complete feature matrix.

Next, we introduce the proposed Concurrency Fault Inspector (CFI) framework.
### 6.3 Introducing Concurrency Fault Inspector (CFI)

Concurrency Fault Inspector or CFI is a framework to help developers identify inducing applications that cause concurrency faults in incomplete race reports. To identify inducing applications for a particular fault, the framework needs three types of inputs that can be extracted from a bug report. The first input is the vulnerable application and its test suite that can be used to exercise the application. This is necessary as CFI operates based on dynamic information. The second input is the symptom. As previously shown, analyzing a reported symptom can often reveal the location where the fault occurs. Knowing the location also leads us to the third input, which is the previously designed and implemented oracle that can be used to detect occurrences of that particular fault. In this work, we use the same oracles as those used in RCRF.

Figure 6.7 illustrates the overview framework of the CFI. As shown, there are four major components: Collector, Synthesizer, Trainer, and Verifier. Each component processes information and then passes the results to the subsequent

![Feature Matrix Diagram](image)

**Figure 6.5: Initializing the feature matrix**

<table>
<thead>
<tr>
<th></th>
<th>1stat64</th>
<th>1open</th>
<th>1read</th>
<th>2stat64</th>
<th>2unlink</th>
<th>1close</th>
</tr>
</thead>
<tbody>
<tr>
<td>1stat64</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1open</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1read</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2stat64</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2unlink</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1close</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
component. As such, the workflow is sequential. The workflow starts with COLLECTOR (component A). The inputs to this component are the reported application and its test suite. In addition, it also takes the entire universe of applications and the corresponding test suites of all applications in the universe.

Next, COLLECTOR runs each application using its test suite on a system that employs tools that can observe and record kernel events (such tools include Scribe [59] and Strace [108]). At the end of this process, COLLECTOR produces kernel event log files of the reported application and all applications in the provided universe. It then processes these log files to include only events that are potentially relevant to the reported fault. This is done by previously analyzing the reported symptom to identify possible resources that can suffer from the reported concurrency fault. In this step, COLLECTOR also removes duplicated events prior to passing these lists of kernel events to SYNTHESIZER, component B in Figure 6.7.
Synthesizer consists of two additional sub-components: Virtual interleavings Generator (VIG) and Dataset Synthesizer (DS). These two components work independently, but they share the same system call event logs provided by Collector.

To identify a fault-inducing application, it is necessary to consider every possible interleaving between the vulnerable application and the application under inspection. However, running all possible interleavings can be an expensive process. Instead, we developed a Virtual interleaving Generator to reason about each possible interleaving and determine whether it can cause the reported fault. Our VIG can generate Virtual interleavings (VIs) by manipulating kernel events invoked by the two applications to create a list of possible interleavings. Each list is then evaluated by a corresponding Oracle to determine whether it can possibly lead to
the reported concurrency fault. Once race-leading lists of interleavings have been identified, they will be verified by our last component, Verifier (component D).

In addition to generating VIs, Synthesizer also produces datasets based on the system calls and signals that have been used in every pair of applications. This is done for the purpose of training our machine learning-based classifier that can be used to predict applications in a universe that can induce the reported fault. Because the amount of information generated by virtual interleavings can be large, it is cost effective to use the information to train a classifier that can be used in the future. Our Dataset Synthesizer (DS) generates synthetic data based on VIs and actual interleavings. These datasets are then used by our next component, Trainer (component C) to train our proposed classifier.

Trainer (component C) takes three inputs. The first two inputs are generated by Synthesizer; they are the lists of VIs and synthesized datasets. The third input is the dataset consisting of interleavings from actual executions. Trainer uses both types of datasets for both training and testing. We apply different machine learning algorithms to build our classifier. Once Trainer is done building our classifier, we use it to predict the result of each list of VIs. This allows our approach to identify a subset of applications in a universe that can possibly induce the reported race.

Although the prediction results come from the well-trained classifier, they still need to be verified to ensure that we have indeed identified fault-inducing applications. The proposed Verifier (shown in Figure 6.7 as component D) performs this important task. It takes two inputs: the lists of VIs from Synthesizer and the prediction results from Trainer. The verification process leverages our prior work, RCRF. However, other concurrency fault reproduction approaches can also be used. Once the identified fault-inducing application has been verified, developers can use the actual execution interleavings that produce faults as additional training data.
It is worth noting that we make one assumption in this work. We assume that for every reported application, the engineer working on replicating the reported faults can access the path, branch, or def-use adequate test suite for that application. This is reasonable as the application is being maintained or repaired by the same people who developed it. In addition, because our work focuses on applications written for Linux, the applications in the universe that we are using should be open-source so accessing their test suites is also possible. Next, we provide a detailed description of each CFI’s component.

6.3.1 COLLECTOR: Identify Race-Inducing Applications and Relevant System Events

As the first step, an engineer analyzes an incomplete race report. At a minimum, the engineer needs to be able to diagnose the reported faulty behaviors and identify possible resources for which occurrences of that fault can be observed (e.g., files, shared memory, system calls). Based on the identified resources, the engineer subsequently identifies existing oracles that can detect the presence of such faults. Next, the engineer executes the reported application using the existing test suite to generate a list of kernel events. The list is analyzed along with lists from other applications in the universe. If these lists do not yet exist, these applications will need to be executed with their corresponding test suites to produce these lists.

Algorithm 4 shows how the proposed COLLECTOR gathers all the system kernel event logs. CFI considers all feasible execution paths inside that application (Line 1) by running the test suite (Line 2). It also considers all system kernel events (Line 3) before filtering out duplicated and those irrelevant events (Line 4).
Algorithm 4 The Collector and its filtering

Require: a list of application name $f_i (F)$,
a list of test case $t_j (TC)$,
shared resource name $(Sh)$,
a list of original kernel event log file and the line number of code ($OSCLog$),
a list of filtered kernel event log file and the line number of code ($FSCLog$),
a list of unique kernel event log file and the line number of code ($USCLog$)

1: for all $f_i \in F$ do
2:   for all $t_j \in TC$ do
3:      $OSCLog(i,j) \leftarrow$ KernelEventLogTool($f_i, t_i$)
4:      $FSCLog(i,j) \leftarrow$ FilterForReleventEventOnly($OSCLog(i,j)$, $Sh$)
5:   end for
6:   $USCLog_i \leftarrow$ Unique($FSCLog(i,j)$)
7: end for
8: return $USCLog$

6.3.2 SYNTHESIZER: Generating the VIs and Synthesizing Datasets for Classifier

As previously mentioned, this second feature has two sub-components, the VIRTUAL interleavings Generator (VIG) and the DATASET SYNTHESIZER (DS). Both components use information from the COLLECTOR to generate VIs and synthesize datasets.
6.3.2.1 Virtual interleavings Generator (VIG)

The Virtual interleaving Generator or VIG is a tool to generate Virtual interleavings (VIs) that can be analyzed for possible faults without executing the applications. In a nutshell, CFI can use these VIs to predict the outcome and verify the result without performing exhaustive verification. VIG generates every possible interleaving pattern as briefly illustrated in Figure 6.10.

A Virtual interleaving (VI) is a representation of a possible interleaving can occur when two applications run simultaneously and share the same resource and/or time. The VIG simply runs all applications using test suites to produce kernel event logs.
that contain only relevant events and no duplicates. It then pairs up these events
one by one to create the list of VIs using the \texttt{recursiveMergeList()} as shown in the
Algorithm 5.

\textbf{Algorithm 5} \texttt{recursiveMergeList} for \textsc{Virtual interleafing Generator (VIG)}

\textbf{Require:} The first list of system calls ($L_1$),
The second list of system calls ($L_2$),
The list of System calls for output ($OL$)
1: if (sizeof($L_1$) = 0 && sizeof($L_2$) = 0) then
2: addOutputAndClearList($O$)
3: else
4: if (sizeof($L_1$) > 0) then
5: appendOneSC($OL$, FirstToken($L_1$))
6: RecursiveMergeList(RemoveFirstToken($L_1$), $L_2$, $OL$)
7: end if
8: if (sizeof($L_2$) > 0) then
9: appendOneSC($OL$, FirstToken($L_2$))
10: RecursiveMergeList($L_1$, RemoveFirstToken($L_2$), $OL$)
11: end if
12: end if

\textbf{6.3.2.2 Dataset Synthesizer (DS)}

From our previous study of RCRF (see Chapter 4), two applications are concurrently
executed to generate actual interleavings. RCRF manipulates these actual and unique
interleavings to reproduce races. However, the number of unique interleavings tends
to be small (Table 4.3 reports only 308 unique interleavings). On the other hand,
CFI does not execute the applications. Instead, it analyzes all possible VIs to identify
possible faults. Therefore, our approach synthesizes data.

The \textbf{Dataset Synthesizer} or DS is a tool to collect working system calls
from both actual execution and VIs. It then synthesizes a new dataset for training
and testing our classifier. DS is a subprocess inside the \textsc{Synthesizer} as shown in
Figure 6.9. Each data point in the dataset comes from a Happen-before graph of a
list of system calls. The length of the sequence depends on the minimum requirement of each Oracle \((\text{MinLengthRequiredByOracle})\). For example, the oracle in the case of \texttt{LOCATE} and \texttt{UPDATEDB} requires at least 3 system calls to see if a shared file is deleted before \texttt{LOCATE} is done using it.

Each dataset that will be used by our classifier (discussed next) consists of one feature matrix (discussed in Section 6.2) and one label as shown in Figure 6.11. It represents one iteration of a parallel run of the vulnerable and inducing applications. The interleavings between these two processes can create a faulty behavior, which the oracle should be able to detect. The labels are numbers to indicate the type of faults. In our study, we only use Boolean labels, true or false, to identify whether the fault we are seeking is present.

To create a dataset, we first convert the feature matrix to a line of data as shown in Figure 6.12. We merge the data line by line into a long row. We then append the result from the oracle at the end of the row. This represents one line in the dataset.
We repeat the process on every feature matrix until we have the entire dataset as shown in the Figure 6.11.

![Figure 6.12: Transforming the feature matrix and evaluating the output from the oracle into a line of dataset](image)

Algorithmically, DS uses a recursive algorithm to generate the dataset as shown in Algorithm 6. There are three inputs for this function. The first input is the list of system calls or $SC$, the second is the output string or $OutputString$, and the last is the counter or $Loop$. The main function of this DS calls this recursion function to generate the dataset $DataSynthesis(SC,””,MinLengthRequiredByOracle)$.

In Algorithm 6, the termination condition is when the loop counter is less than or equal to zero (Line 1). If the recursion has not terminated yet, it uses all of the system calls (Line 2) to add into a new output string or $newOutput$ (Line 3). This $newOutput$ is sent to the output if the $Loop$ counter is down to 1 (Line 4). Then the whole recursion moves forward (Line 7), decreasing the counter by 1.

The generated dataset is used by the next component for training and testing our classifier.
Algorithm 6 Data Synthesis (DS)

Require: The list of system calls ($SC$),
          An output string ($OutputString$),
          Loop counter ($Loop$)

1: if ($Loop > 0$) then
2:     for all ($sc_i$) $\in SC$ do
3:         newOutput = $OutputString + sc_i$
4:     if ($Loop = 1$) then
5:         PrintOutput($newOutput$)
6:     end if
7:     DataSynthesis($SC, newOutput, Loop - 1$)
8: end for
9: end if

6.3.3 TRAINER: Training and Testing the Classifier, then Predicting the Results of VIs

The third main component of CFI is the TRAINER that performs the process of training and testing our classifier, which is subsequently used to predict whether VIs can induce races. As shown in Figure 6.13, CFI receives the datasets from DS in SYNTHESIZER. Our classifier then performs the prediction on the VIs and reports the prediction results for each of the VIs as the output of this component.

![Figure 6.13: Workflow in TRAINER](image)

To determine what machine learning approaches to use, we experimented with five
machine learning algorithms: Naïve Bayes, Bayes Networks, ADTree, Random Forrest, and SMO to train and compare the results. We used the split percentage approach in WEKA [109] to perform training and testing. In this approach, 90% of the data is used for training and 10% of the data is used for testing. The dataset includes the data from running LOCATE against five versions of the BACKUP application. Our goal is to identify an algorithm that can predict accurately and can report the largest number of true positives. We report our observations in Table 6.1. Column A lists the algorithms used. Column B reports the time used by each algorithm to create a classifier. Column C reports the number of instances that each algorithm used for testing, which is 10% of the total number of instances. Column D reports the average F-Measure score [110]. Column E to H report the classification results as True Negatives (TN), False Negatives (FN), False Positives (FP), and True Positives (TP).

Table 6.1: Comparing the Results of Classification Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Building Time (second)</th>
<th>Testing Instances</th>
<th>Average F-Measure</th>
<th>True Negative</th>
<th>False Negative</th>
<th>False Positive</th>
<th>True Positive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naïve Bayes</td>
<td>8.52</td>
<td>3559</td>
<td>0.914</td>
<td>2094</td>
<td>267</td>
<td>46</td>
<td>1152</td>
</tr>
<tr>
<td>ADTree</td>
<td>1093.36</td>
<td>3559</td>
<td>0.845</td>
<td>2295</td>
<td>66</td>
<td>456</td>
<td>742</td>
</tr>
<tr>
<td>Random Forests</td>
<td>453.52</td>
<td>3559</td>
<td>0.991</td>
<td>2338</td>
<td>23</td>
<td>10</td>
<td>1188</td>
</tr>
<tr>
<td>SMO</td>
<td>2526.99</td>
<td>3559</td>
<td>0.976</td>
<td>2316</td>
<td>45</td>
<td>41</td>
<td>1157</td>
</tr>
<tr>
<td>Bayes Networks</td>
<td>37.78</td>
<td>3559</td>
<td>0.948</td>
<td>2183</td>
<td>178</td>
<td>11</td>
<td>1187</td>
</tr>
</tbody>
</table>

As shown in the table, the Random Forests algorithm performed best in our investigation. It reported the highest F-Measure score while also achieving the highest number of true positives. In addition, Random Forests is known to avoid overfitting [111]. On the other hand, ADTree and SMO are not efficient. Naïve Bayes and Bayes Networks have a high number of errors on the prediction for the False Negatives (Column F) and False Positives (Column G).

Next, we used our classifiers to perform predictions. We report the results using
Table 6.2: Predicting the VIs of LOCATE-BACKUP using classifiers from different algorithms

<table>
<thead>
<tr>
<th>Algorithm (A)</th>
<th>Virtual interleavings (B)</th>
<th>Predicted Instances (C)</th>
<th>True Negative (D)</th>
<th>False Negative (E)</th>
<th>False Positive (F)</th>
<th>True Positive (G)</th>
<th>Average F-Measure (H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naïve Bayes</td>
<td>LOCATE-BACKUP1</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>0.709</td>
</tr>
<tr>
<td></td>
<td>LOCATE-BACKUP2</td>
<td>15</td>
<td>5</td>
<td>0</td>
<td>5</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LOCATE-BACKUP3</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LOCATE-BACKUP4</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LOCATE-BACKUP5</td>
<td>15</td>
<td>15</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>ADTree</td>
<td>LOCATE-BACKUP1</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0.661</td>
</tr>
<tr>
<td></td>
<td>LOCATE-BACKUP2</td>
<td>15</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LOCATE-BACKUP3</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LOCATE-BACKUP4</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LOCATE-BACKUP5</td>
<td>15</td>
<td>15</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Random Forests</td>
<td>LOCATE-BACKUP1</td>
<td>5</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0.915</td>
</tr>
<tr>
<td></td>
<td>LOCATE-BACKUP2</td>
<td>15</td>
<td>7</td>
<td>0</td>
<td>3</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LOCATE-BACKUP3</td>
<td>5</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LOCATE-BACKUP4</td>
<td>5</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LOCATE-BACKUP5</td>
<td>15</td>
<td>15</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>SMO</td>
<td>LOCATE-BACKUP1</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>0.594</td>
</tr>
<tr>
<td></td>
<td>LOCATE-BACKUP2</td>
<td>15</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LOCATE-BACKUP3</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LOCATE-BACKUP4</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LOCATE-BACKUP5</td>
<td>15</td>
<td>15</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Bayes Networks</td>
<td>LOCATE-BACKUP1</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0.65</td>
</tr>
<tr>
<td></td>
<td>LOCATE-BACKUP2</td>
<td>15</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LOCATE-BACKUP3</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LOCATE-BACKUP4</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LOCATE-BACKUP5</td>
<td>15</td>
<td>15</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

all five classifiers in Table 6.2. Column A lists the algorithms. Column B reports the applications we pair with LOCATE to generate VIs. Columns C to G are the number of prediction instances as compared to the ground truth, which is the actual interleavings that race. Column H is the average F-measure. Again, we observe that the Random Forests algorithm still performs the best. It has a high number of True Negatives and True Positives and low number of False Negatives and False Positives. The average F-measure score is also the highest at 0.915.

After training and validating the classifier, we provide the VIs from the SYNTHESIZER to the classifier to get the prediction results and save the results into a file before passing them to the VERIFIER to reproduce the race in the last step.
6.3.4 VERIFIER: Confirming the Presence of Faults

By using VIs and our classifier, we can quickly identify applications in a universe that can possibly race with the reported application. To ensure that the identified application can induce concurrency faults with the reported application, we need to verify the reproducibility of the reported fault. We implement VERIFIER by using our proposed RCRF to execute these applications by focusing on the locations of the system calls or signals that are part of the VIs identified as faulty.

VERIFIER needs the correct oracle for the targeted fault, the VIs from SYNTHESIZER, and the prediction results from TRAINER as inputs and produces a final report as output as shown in Figure 6.14.

![Workflow in The verifier](image)

**Figure 6.14: Workflow in The verifier**

VERIFIER executes the interleavings repeatedly for a number of times (10 times in our case) to reduce non-determinism. If the faulty behaviors can be detected, VERIFIER then flags that interleaving as faulty in the final report. With the help of Oracle, the real faults found in this process can be confirmed.
6.4 A Case Study

Our case study focuses on how well CFI can help developers debug concurrency faults when only incomplete bug reports that do not contain information about the inducing applications are available. To find an inducing application that can cause a reported fault, CFI needs to efficiently analyze all applications in a universe along with the reported application. To make the process efficient, CFI uses a classifier to predict inducing applications to achieve better performance than an ordinary exhaustive approach that needs to run every application and exercise every interleaving. Clearly, the accuracy and efficiency of predictions are the two main performance metrics. As such, our evaluation aims answer the following two research questions.

**RQ1:** How effective is CFI compared to the exhaustive approach for finding the inducing applications?

**RQ2:** How efficient is CFI compared to the exhaustive approach for finding the inducing applications?

6.4.1 Objects of Analysis

We consider 117 open-source applications running on Linux 2.6.35+, including the applications described in Chapter 2 and Chapter 4. Because CFI analyzes system call usage, we further cluster the universe of applications into four groups using a simple k-Mean clustering algorithm based on their system call usages. We then identify which cluster to use based on the similarity between the system call usage of the reported application and the applications in a cluster. Table 6.3 reports the result after clustering.

Next, we identify applications in each cluster that access the same shared resources as the reported application. Our investigation in this study is based on the LOCATE-
Table 6.3: Universe of Applications

<table>
<thead>
<tr>
<th>Cluster No.</th>
<th>List of Applications</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>backup1, backup2, backup3, backup4, backup5, bash, migrate1, migrate2, migrate3, migrate4, migrate5, stealer, restore1, restore2, restore3, updatedb, zipper</td>
<td>14.53%</td>
</tr>
<tr>
<td>2</td>
<td>cp, date, dd, deadlock, dir, find, grep, ls, nice, nohup, pr, pxz, shred, shuf, sort, touch, vlc</td>
<td>14.53%</td>
</tr>
<tr>
<td>3</td>
<td>audacious, banshee, chromium, dia, inkscape, mixxx, opera</td>
<td>5.98%</td>
</tr>
<tr>
<td>4</td>
<td>base64, basename, cat, chgrp, chmod, chown, cksum, comm, csplit, cut, df, dircolors, dirname, du, echo, env, expand, expr, factor, false, fmt, fold, head, hostid, hostname, id, join, kill, link, ln, locate, logname, logroate, mSigShooter, md5sum, mkdir, mkfifo, mknod, mv, nl, od, paste, pathchk, pinky, printf, printenv, ps, ptx, pwd, readlink, rm, rmdir, seq, sleep, split, stat, stty, su, sum, sync, tac, tail, tee, test, true, tsort, tty, uname, unexpand, uniq, unlink, uptime, users, wc, whoami, yes</td>
<td>64.96%</td>
</tr>
</tbody>
</table>

Table 6.4 shows the list of applications grouped by clusters (the second column). We also list applications that access similar shared resources (the last column). The total number of applications that we need to consider is reduced to 42 applications as shown in the last column.

Table 6.4: The applications after refinement

<table>
<thead>
<tr>
<th>Cluster No.</th>
<th>List of Applications</th>
<th>List of Applications that touch the shared resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>backup1, backup2, backup3, backup4, backup5, bash, migrate1, migrate2, migrate3, migrate4, migrate5, stealer, restore1, restore2, restore3, updatedb, zipper</td>
<td>backup1, backup2, backup3, backup4, backup5, migrate1, migrate2, migrate3, migrate4, migrate5, restore1, restore2, restore3, updatedb, zipper</td>
</tr>
<tr>
<td>2</td>
<td>cp, date, dd, deadlock, dir, find, grep, ls, nice, nohup, pr, pxz, shred, shuf, sort, touch, vlc</td>
<td>cp, grep, ls, pr, pxz, shuf, sort, touch</td>
</tr>
<tr>
<td>3</td>
<td>audacious, banshee, chromium, dia, inkscape, mixxx, opera</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>base64, basename, cat, chgrp, chmod, chown, cksum, comm, csplit, cut, df, dircolors, dirname, du, echo, env, expand, expr, factor, false, fmt, fold, head, hostid, hostname, id, join, kill, link, ln, locate, logname, logroate, mSigShooter, md5sum, mkdir, mkfifo, mknod, mv, nl, od, paste, pathchk, printenv, pinky, printf, ps, ptx, pwd, readlink, rm, rmdir, seq, sleep, split, stat, stty, su, sum, sync, tac, tail, tee, test, true, tsort, tty, uname, unexpand, uniq, unlink, uptime, users, wc, whoami, yes</td>
<td>cat, csplit, cut, fold, head, join, ln, locate, mv, od, paste, ptx, rm, split, sum, tac, tail, tee, unlink, wc</td>
</tr>
</tbody>
</table>
6.4.2 Variables and Measures

**Independent Variable.** To assess effectiveness (RQ1) we wish to compare the effectiveness of CFI with that of the baseline approach from Chapter 4. To do so, we count the number of the applications in which we can detect faults. We consider all of the potential interleavings between the vulnerable application and the list of inducing applications on the same input and execution environment.

For the baseline technique, we use RCRF, which is an exhaustive approach to run every possible pair of applications to reproduce races. On the other hand, CFI uses a predictive approach.

**Dependent Variable.** To assess the cost of CFI, we measure the wall clock times needed to completely execute the baseline approach and CFI. We also count the number of correctly predicted race-inducing applications.

6.4.3 Study Methodology

We conduct our study using a report for LOCATE-UPDATEDB [37]. Based on the bug report, the vulnerable application is LOCATE and the fault can be observed in a shared file locatedb. We then try to identify applications in our universe that also try to access the same shared file. These applications can possibly induce the reported race. Next, we identify the oracles that can possibly detect occurrences of the reported fault. Last, we apply both approaches to identify applications that can induce the reported fault. We report the iteration that reproduces the reported fault and the time taken from the start of the process to reproduce the fault. Both approaches terminate as soon as a fault is found (referred to as early termination).
6.4.4 Threats to Validity

The primary threat to external validity in this study involves the object programs utilized. We have included only 117 open-source programs. Moreover, these applications may not be representative of the actual applications running on the user’s system when a reported fault occurred. We apply our system only to one case but since the set of data used by CFI is similar to that used by RCRF, we anticipate that we can apply CFI to other types of concurrency faults. However, we still need to conduct further studies to verify the results when CFI is applied to other fault types.

The primary threat to internal validity includes potential errors in the implementation of CFI and the infrastructure used to run both CFI and the exhaustive approach. To limit these errors we extensively validated all of the components and scripts.

The primary threat to construct validity relates to the fact that we study effectiveness and savings relative to applications of CFI, but do not yet assess whether the approach helps engineers localize and correct targeted faults more quickly than current approaches.

6.4.5 Results

Table 6.5 reports data showing the effectiveness of CFI. Column A lists the vulnerable program, which is LOCATE in this study. Column B lists all the applications in the universe that utilize similar system calls as those in the vulnerable program. Clustering based on system call usage results in 42 applications that we need to investigate. Column C is the total number of interleavings in each pair of applications that can possibly race. Columns D and E report the results of employing the exhaustive approach that uses RCRF to run every pair of applications. Column D reports the
Table 6.5: Effectiveness of CFI

<table>
<thead>
<tr>
<th>App1</th>
<th>App2</th>
<th>Total Interleavings</th>
<th>Exhaustive Approach</th>
<th>CFI</th>
<th>Race Found</th>
<th>(Early Termination)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A)</td>
<td>(B)</td>
<td>(C)</td>
<td>(D)</td>
<td>(E)</td>
<td>(F)</td>
<td>(G)</td>
</tr>
<tr>
<td>LOCATE</td>
<td>BACKUP1</td>
<td>5</td>
<td>2</td>
<td>Yes</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>LOCATE</td>
<td>BACKUP2</td>
<td>15</td>
<td>2</td>
<td>Yes</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>LOCATE</td>
<td>BACKUP3</td>
<td>5</td>
<td>2</td>
<td>Yes</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>LOCATE</td>
<td>BACKUP4</td>
<td>5</td>
<td>2</td>
<td>Yes</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>LOCATE</td>
<td>BACKUP5</td>
<td>15</td>
<td>15</td>
<td>No</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>CAT</td>
<td>35</td>
<td>35</td>
<td>No</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>Cp</td>
<td>336</td>
<td>336</td>
<td>No</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>Csplit</td>
<td>5</td>
<td>5</td>
<td>No</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>cut</td>
<td>15</td>
<td>15</td>
<td>No</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>fold</td>
<td>15</td>
<td>15</td>
<td>No</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>grep</td>
<td>15</td>
<td>15</td>
<td>No</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>head</td>
<td>15</td>
<td>15</td>
<td>No</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>join</td>
<td>70</td>
<td>70</td>
<td>No</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>ln</td>
<td>10</td>
<td>10</td>
<td>No</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>locate</td>
<td>70</td>
<td>70</td>
<td>No</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>ls</td>
<td>15</td>
<td>15</td>
<td>No</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>migrate1</td>
<td>5</td>
<td>2</td>
<td>Yes</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>LOCATE</td>
<td>migrate2</td>
<td>5</td>
<td>2</td>
<td>Yes</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>LOCATE</td>
<td>migrate3</td>
<td>5</td>
<td>2</td>
<td>Yes</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>LOCATE</td>
<td>migrate4</td>
<td>5</td>
<td>5</td>
<td>No</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>migrate5</td>
<td>15</td>
<td>2</td>
<td>Yes</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>LOCATE</td>
<td>mv</td>
<td>35</td>
<td>35</td>
<td>No</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>od</td>
<td>70</td>
<td>70</td>
<td>No</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>paste</td>
<td>15</td>
<td>15</td>
<td>No</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>pr</td>
<td>70</td>
<td>70</td>
<td>No</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>ptx</td>
<td>70</td>
<td>70</td>
<td>No</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>pxz</td>
<td>126</td>
<td>2</td>
<td>Yes</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>LOCATE</td>
<td>restore1</td>
<td>5</td>
<td>5</td>
<td>No</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>restore2</td>
<td>5</td>
<td>5</td>
<td>No</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>restore3</td>
<td>5</td>
<td>2</td>
<td>Yes</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>LOCATE</td>
<td>rm</td>
<td>5</td>
<td>2</td>
<td>Yes</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>LOCATE</td>
<td>shuf</td>
<td>700</td>
<td>700</td>
<td>No</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>sort</td>
<td>1260</td>
<td>1260</td>
<td>No</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>split</td>
<td>15</td>
<td>15</td>
<td>No</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>sum</td>
<td>15</td>
<td>15</td>
<td>No</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>tac</td>
<td>15</td>
<td>15</td>
<td>No</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>tail</td>
<td>35</td>
<td>35</td>
<td>No</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>tee</td>
<td>35</td>
<td>35</td>
<td>No</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>touch</td>
<td>5</td>
<td>5</td>
<td>No</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>unlink</td>
<td>5</td>
<td>2</td>
<td>Yes</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>LOCATE</td>
<td>updatedb</td>
<td>35</td>
<td>3</td>
<td>Yes</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>LOCATE</td>
<td>wc</td>
<td>35</td>
<td>35</td>
<td>No</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

The number of exercised interleavings for each pair of applications, and Column E reports the number of uncovered races.
Columns F and G report the results of employing CFI to predict and then RCRF to verify the prediction results. Note that we run only the pairs of applications that have been predicted by CFI to possibly race. For the pairs of applications that CFI does not predict to have races, we simply enter “-” in the corresponding cells. We apply early termination when possible (i.e., stop testing with RCRF as soon as a race is uncovered). The effect of early termination is that the number of exercised interleavings is fewer than the number of all interleavings that should be explored. For example, there are 15 possible interleavings that can cause backup2 to race with LOCATE. However, with early termination, the exhaustive approach can uncover the reported race after exercising only two interleavings. Once the race is uncovered, the execution of RCRF terminates.

**Table 6.6: Prediction Accuracy of CFI**

<table>
<thead>
<tr>
<th>Actual Result</th>
<th>No Race</th>
<th>Race</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Race</td>
<td>29</td>
<td>0</td>
</tr>
<tr>
<td>Race</td>
<td>0</td>
<td>13</td>
</tr>
</tbody>
</table>

To answer the first research question, Table 6.5 clearly shows that CFI can effectively predict applications that can race with LOCATE. In fact, Table 6.6 shows that its prediction is 100% correct; i.e., every predicted pair of applications does contain races. CFI also does not miss any pair of applications that contains races. Furthermore, by being able to investigate only the interleavings that are likely to race, we can take advantage of the early termination policy in all but one pair of applications.

With respect to the second research question, we report the experimental results in Table 6.7. Again, Columns A and B are the pairs of the applications that we explore. Column C reports the execution time (wall-clock time) needed by RCRF to exercise
Table 6.7: Efficiency of CFI

<table>
<thead>
<tr>
<th>App1</th>
<th>App2</th>
<th>Exhaustive Approach</th>
<th>CFI</th>
<th>Time Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Processing Time</td>
<td>No of tested Interleavings</td>
<td>Processing Time</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(A)</td>
<td>(B)</td>
<td>(C)</td>
</tr>
<tr>
<td>LOCATE</td>
<td>BACKUP1</td>
<td>00:07:44</td>
<td>2</td>
<td>00:05:36</td>
</tr>
<tr>
<td>LOCATE</td>
<td>BACKUP2</td>
<td>00:11:27</td>
<td>2</td>
<td>00:08:28</td>
</tr>
<tr>
<td>LOCATE</td>
<td>BACKUP3</td>
<td>00:07:22</td>
<td>2</td>
<td>00:06:21</td>
</tr>
<tr>
<td>LOCATE</td>
<td>BACKUP4</td>
<td>00:07:19</td>
<td>2</td>
<td>00:05:15</td>
</tr>
<tr>
<td>LOCATE</td>
<td>BACKUP5</td>
<td>07:09:21</td>
<td>15</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>CAT</td>
<td>01:26:30</td>
<td>35</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>CP</td>
<td>14:30:40</td>
<td>336</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>CSPLIT</td>
<td>00:10:40</td>
<td>5</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>CUT</td>
<td>00:31:40</td>
<td>15</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>FOLD</td>
<td>00:35:17</td>
<td>15</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>GREP</td>
<td>00:43:01</td>
<td>15</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>HEAD</td>
<td>00:34:53</td>
<td>15</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>JOIN</td>
<td>03:00:02</td>
<td>70</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>LN</td>
<td>00:21:54</td>
<td>10</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>LOCATE</td>
<td>05:42:36</td>
<td>70</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>LS</td>
<td>00:33:49</td>
<td>15</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>MIGRATE1</td>
<td>00:09:33</td>
<td>2</td>
<td>00:06:45</td>
</tr>
<tr>
<td>LOCATE</td>
<td>MIGRATE2</td>
<td>00:09:59</td>
<td>2</td>
<td>00:07:04</td>
</tr>
<tr>
<td>LOCATE</td>
<td>MIGRATE3</td>
<td>00:32:56</td>
<td>2</td>
<td>00:06:00</td>
</tr>
<tr>
<td>LOCATE</td>
<td>MIGRATE4</td>
<td>00:27:15</td>
<td>5</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>MIGRATE5</td>
<td>00:09:09</td>
<td>2</td>
<td>00:07:08</td>
</tr>
<tr>
<td>LOCATE</td>
<td>MV</td>
<td>01:17:25</td>
<td>35</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>OD</td>
<td>07:05:10</td>
<td>70</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>PASTE</td>
<td>00:32:11</td>
<td>15</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>PR</td>
<td>02:54:14</td>
<td>70</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>PTX</td>
<td>03:00:58</td>
<td>70</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>PXZ</td>
<td>00:26:57</td>
<td>2</td>
<td>00:25:23</td>
</tr>
<tr>
<td>LOCATE</td>
<td>RESTORE1</td>
<td>00:27:02</td>
<td>5</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>RESTORE2</td>
<td>00:27:22</td>
<td>5</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>RESTORE3</td>
<td>00:07:05</td>
<td>2</td>
<td>00:05:05</td>
</tr>
<tr>
<td>LOCATE</td>
<td>RM</td>
<td>00:04:44</td>
<td>2</td>
<td>00:02:51</td>
</tr>
<tr>
<td>LOCATE</td>
<td>SHUF</td>
<td>02:55:28</td>
<td>700</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>SORT</td>
<td>05:17:28</td>
<td>1260</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>SPLIT</td>
<td>00:35:26</td>
<td>15</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>SUM</td>
<td>00:34:46</td>
<td>15</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>TAC</td>
<td>00:34:52</td>
<td>15</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>TAIL</td>
<td>01:29:59</td>
<td>35</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>TER</td>
<td>01:19:31</td>
<td>35</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>TOUCH</td>
<td>00:10:55</td>
<td>5</td>
<td>-</td>
</tr>
<tr>
<td>LOCATE</td>
<td>UNLINK</td>
<td>00:11:02</td>
<td>2</td>
<td>00:02:50</td>
</tr>
<tr>
<td>LOCATE</td>
<td>UPDATEDB</td>
<td>00:36:09</td>
<td>3</td>
<td>00:28:38</td>
</tr>
<tr>
<td>LOCATE</td>
<td>WC</td>
<td>01:29:13</td>
<td>35</td>
<td>-</td>
</tr>
</tbody>
</table>

**Summary for CFI**

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Each pair of applications. Column D reports the number of interleavings that RCRF exercises. Column E reports the execution time needed by RCRF to uncover one race.
in a predicted pair of applications. Column G reports the time difference between the
time in Column C and time in the Column E.

Table 6.8: Efficiency of CFI

<table>
<thead>
<tr>
<th>App1</th>
<th>App2</th>
<th>Exhaustive Approach</th>
<th>CFI</th>
<th>Time Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Processing Time</td>
<td>No of tested Interleaves</td>
<td>Processing Time</td>
</tr>
<tr>
<td>LOCATE</td>
<td>BACKUP1</td>
<td>00:07:44</td>
<td>2</td>
<td>00:05:36</td>
</tr>
<tr>
<td>LOCATE</td>
<td>BACKUP2</td>
<td>00:11:27</td>
<td>2</td>
<td>00:08:28</td>
</tr>
<tr>
<td>LOCATE</td>
<td>BACKUP3</td>
<td>00:07:22</td>
<td>2</td>
<td>00:05:21</td>
</tr>
<tr>
<td>LOCATE</td>
<td>BACKUP4</td>
<td>00:07:19</td>
<td>2</td>
<td>00:05:15</td>
</tr>
<tr>
<td>LOCATE</td>
<td>MIGRATE1</td>
<td>00:06:33</td>
<td>2</td>
<td>00:06:45</td>
</tr>
<tr>
<td>LOCATE</td>
<td>MIGRATE2</td>
<td>00:09:59</td>
<td>2</td>
<td>00:07:04</td>
</tr>
<tr>
<td>LOCATE</td>
<td>MIGRATE3</td>
<td>00:32:56</td>
<td>2</td>
<td>00:06:00</td>
</tr>
<tr>
<td>LOCATE</td>
<td>MIGRATE5</td>
<td>00:09:09</td>
<td>2</td>
<td>00:07:08</td>
</tr>
<tr>
<td>LOCATE</td>
<td>PXZ</td>
<td>00:26:57</td>
<td>2</td>
<td>00:25:23</td>
</tr>
<tr>
<td>LOCATE</td>
<td>RESTORE3</td>
<td>00:07:05</td>
<td>2</td>
<td>00:05:05</td>
</tr>
<tr>
<td>LOCATE</td>
<td>RM</td>
<td>00:04:44</td>
<td>2</td>
<td>00:02:51</td>
</tr>
<tr>
<td>LOCATE</td>
<td>UNLINK</td>
<td>00:11:02</td>
<td>2</td>
<td>00:02:50</td>
</tr>
<tr>
<td>LOCATE</td>
<td>UPDATEDB</td>
<td>00:36:09</td>
<td>3</td>
<td>00:28:38</td>
</tr>
<tr>
<td><strong>Summary for CFI</strong></td>
<td></td>
<td>03:01:26</td>
<td></td>
<td>01:56:24</td>
</tr>
</tbody>
</table>

There are three points to take away from the reported data.

1. Employing CFI has the potential to allow engineers to spend less time to test these applications. As our result collaborates, the reduction in the testing time is nearly 67 hours. This translate to 35 times speed-up of the testing time of CFI over that of the exhaustive approach.

2. For each pair of applications, CFI provides precise information about interleaveings that can cause races in each pair of applications so that the exploration time is also significantly shorter. Table 6.8 clearly collaborates this point. The table shows the 13 pairs of apps that can race. Because CFI prioritizes executions of interleaveings that can cause races, it would be able to terminate each verification run faster than the exhaustive approach that does not prioritize which interleaveing to execute first. We see a reduction of 1 hour and 5 minutes or 36% in execution time when CFI is used to uncover the first race in a pair
of applications instead of when the exhaustive approach is used to uncover the first race in the same pair of applications.

3. The use of CFI can potentially allow an engineer to quickly identify one application that can induce races with the reported application. In our case study, executing any of the applications predicted by CFI to contain the reported fault would immediately uncover the reported fault. On the other hand, the chance that the first application executed by the exhaustive approach would uncover the reported fault is only \( \frac{13}{42} \) or 31%.

6.5 Discussion

Column F in Table 6.5 shows that 12 out of 13 pairs of applications need to exercise only one interleaving to reproduce the reported races. The exception is LOCATE and UPDATEDB, which needs to exercise two interleavings before uncovering a race. This shows the power of CFI to pinpoint the interleavings that can cause races. Also, notice that there are many applications with one or more variations that cause races and others that do not. This shows that CFI is sensitive to some subtle changes.

To further investigate the pair of two interleavings that needs to be exercise, we illustrate a subset of VIs generated by CFI for that pair (LOCATE-UPDATEDB) in Figure 6.15. Figure 6.16 illustrates the results of exercising the interleavings by RCRF.

```
0:0:?:1Rstat64:1Wopen:1Wread:1Wclose:2Rfstatat64:2Wunlinkat:2Rstat64:2Wrename:2Rstat64:2Wfchmodat
1:0:?:2Rstat64:2Wopen:1Wread:2Rfstatat64:2Wunlinkat:2Rstat64:2Wrename:2Rstat64:2Wfchmodat
2:0:?:2Rstat64:2Wopen:1Wread:2Rfstatat64:2Wunlinkat:2Rstat64:2Wrename:2Rstat64:2Wfchmodat
3:0:?:1Rstat64:1Wopen:1Wread:2Rfstatat64:2Wunlinkat:2Rstat64:2Rlstat64:2Wrename:2Rstat64:2Wfchmodat
```

Figure 6.15: The example of virtual interleavings generated from LOCATE-UPDATEDB

Our analysis of the runtime trace indicates that the sleep injection technique used
by RCRF cannot inject a sleep statement between system calls `fstatat64` and `unlinkat`. As such, we cannot enforce the first predicted interleaving (first line in Figure 6.15). However, in the second line, the predicted interleaving has `Wclose` in between `fstatat64` and `unlinkat`. Our system can enforce this particular interleaving and uncovers a race (as shown in Figure 6.16).

Figure 6.16: The log file from executing the interleavings from LOCATE-UPDATEDB

6.6 Related Work

Previous work on race reproduction including RCRF, RACEPRO [3], and SIMRACER [66] requires that engineers specify pairs of applications that race (we discussed these approaches in Chapter 4.6). As such, they do need complete bug reports that disclose such information. Our work supplements these approaches by extending their abilities to tackle race reproduction even when bug reports are incomplete. (Specifically, when bug reports do not contain any information about race-inducing applications.)

A large class of concurrency fault detectors focus their efforts on applying dynamic analysis on actual interleavings [16,28,74,75,77,112]. For examples, DORA [113], an adaptive version of RACEPRO [3], performs concurrency bug prevention by mutating...
interleavings and employing record-and-replay of interleavings to avoid concurrency faults. iDNA-RECORDER [114] and iDNA-REPLAYER [115] perform record and replay at the binary execution level with data race analysis of Happens-Before relationships to prevent the fault. Dynamically detecting faults makes sense because, after all, concurrency faults occur dynamically. However, the main short-coming of such approaches is inefficiency. Exercising interleavings by precisely controlling their execution orders can result in significant runtime overhead and probe effects. CFI takes a different approach by using classifier-based prediction to identify pairs of applications and interleavings within each pair of applications that can race. Our case study shows that the proposed approach can be effective and efficient.

There are also prior studies that apply machine learning to detect and prevent concurrency faults. ATOM-AID [116] is a hardware-supported system to detect atomicity violations when concurrent programs try to access memory. The authors propose to use machine learning to achieve their goal. The work has not been released at the moment. NNPIN [117] extends the PIN tool [67] to build neural networks in hardware to detect atomicity violations in file accesses. The initial results show promise but more detailed investigation is still needed.

6.7 Conclusion

In this chapter, we propose a new approach to empower engineers to be more effective at reproducing and debugging data races. Because races can be difficult to diagnose by users during deployment, they can submit bug reports that are not complete. As such, these reports may not be useful in helping engineers to debug the reported faults.

We introduce CFI, a framework that can help engineers fill in missing information in incomplete bug reports. Specifically, CFI focuses on filling in information in bug
reports that do not disclose race-inducing applications. CFI aims to help an engineer identify applications that are accessible to her and that can possibly race with the reported application. Identifying applications can help the engineer reproduce the reported race.

The greatest benefit of CFI is its predictive power. By using prediction, it can significantly reduce the number of applications that must be tested for races. Our case study shows that CFI is accurate and can save a significant amount of testing time.
Chapter 7

Conclusions and Future Work

In this work, we have designed and implemented three complementary frameworks that can help developers reproduce process-level concurrency faults. The three proposed frameworks target three critical problems in reproducing concurrency faults which are:

1. dealing with executing non-determinism,
2. assisting developers in constructing oracles to detect various forms of concurrency faults, and
3. filling in the missing information in incomplete race reports submitted by users.

The three proposed frameworks establish a good starting point to address the complex issue of reproducing and debugging concurrency faults. In this chapter, we propose the next steps that should be taken to further advance state-of-the-art in this important research area.

With respect to the RCRF, we should explore additional types of concurrency faults to see if our framework can still replicate them. This will allow RCRF to be more general and can have much broader capability to tackle various types of concurrency faults. In addition, we should develop better user interfaces for RCRF.
Lastly, we should investigate the effectiveness of this tool in real-world testing and debugging environments.

With respect to RDSC, we would like to be able to create more generic oracles that can cover a broader range of concurrency faults including order violation, lock contention, and priority inversion.

With respect of CFI, we need to conduct more studies to evaluate the generality of this framework. If we can maintain the same level of accuracy, we may be able to use prediction alone to detect race. It can also be applied in scenarios where organizations want to ensure that newly installed applications would not negatively interact with existing applications that can lead to concurrency faults.
Bibliography


