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

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

Summer 6-16-2014

SimExplorer: A Testing Framework to Detect Elusive Software Faults

Tingting Yu
University of Nebraska-Lincoln, tyu@cse.unl.edu

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

Part of the Software Engineering Commons

Yu, Tingting, "SimExplorer: A Testing Framework to Detect Elusive Software Faults" (2014). *Computer Science and Engineering: Theses, Dissertations, and Student Research*. 77.
http://digitalcommons.unl.edu/computerscidiss/77

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.

SIMEXPLORER:
A TESTING FRAMEWORK TO DETECT ELUSIVE SOFTWARE FAULTS

by

Tingting Yu

A DISSERTATION

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Doctor of Philosophy

Major: Computer Science

Under the Supervision of Professors Gregg Rothermel and Witawas Srisa-an

Lincoln, Nebraska

August, 2014

SIMEXPLORER:

A TESTING FRAMEWORK TO DETECT ELUSIVE SOFTWARE FAULTS

Tingting Yu, Ph. D.

University of Nebraska, 2014

Advisors: Gregg Rothermel and Witawas Srisa-an

Modern computer software systems are prone to various classes of runtime faults due to their reliance on features such as concurrency and peripheral devices such as sensors. Testing remains a common method for uncovering faults in these systems. However, commonly used testing techniques that execute the program with test inputs and inspect program outputs to detect failures are often ineffective on these systems. In addition, existing testing techniques focus primarily on single applications, neglecting elusive faults that occur at the whole-system level due to complex system event interactions.

This research provides a framework, SIMEXPLORER, that allows engineers to effectively test for subtle and intermittent faults in modern software systems by providing them with greater *observability* and *controllability*. The testing framework first employs dynamic analysis techniques to observe system execution, identify program locations of interest, and report faults related to oracles. It next employs virtualization to achieve fine-grained controllability that exercises event interleavings that are likely to expose faults. The framework also supports testing of evolving software. The ultimate benefit of SIMEXPLORER is its ability to test for broader classes of elusive faults that cannot be effectively detected by existing approaches. These faults include: (1) concurrency faults between applications and hardware interrupts; (2) concurrency faults caused by improper shared resource accesses among multiple processes, and software signals; (3) violations of expected worst-case interrupt latencies (WCILs); and (4) data races induced in evolving software.

DEDICATION

To my parents and my loving spouse.

ACKNOWLEDGMENTS

I would like to say thanks to those who accompanied me throughout the course of my PhD study. This dissertation would not have been possible without your guidance, support, and encouragement.

I would like to thank my advisors Dr. Gregg Rothermel and Dr. Witawas Srisa-an. Dr. Gregg Rothermel has been advising and mentoring me since the very beginning of my graduate study. I would like to thank him for everything he did for me that has lead to my current accomplishment. As a well-respected scholar, he taught me how to do rigorous research. As a good mentor, he provided me with valuable advice all the time. I also feel so grateful that he spent countless hours on improving my writing and presentations. He helped me grow from a zero-level student into an independent researcher.

Dr. Witawas Srisa-an has also had a great influence on me. He taught me many things in the system's area which has been of a great benefit for my research. I would like to thank him for his knowledge, insights, and constant encouragement. He always believes in me and helps me achieve my full potential. It was through his encouragement that I could realize dreams that I once thought to be impossible. All in all, I am extremely grateful and fortunate to have Gregg and Witty as advisors. I do not know how I can possibly thank them enough. I hope I can repay them by making them proud of me in the future.

My special thanks go to Dr. Matthew Dwyer and Dr. Yaoqing Yang, for offering their time to serve as my committee members, and for delivering valuable feedbacks and insights.

I would like to thank my collaborators, Dr. Myra Cohen, Dr. Mithun Acharya and Dr. Xiao Qu. Especially, I would like to thank Dr. Myra Cohen for her contributions in Chapter 5. It was her "Search-based Software Engineering" class that inspired me and made SIMLATTE possible.

I would like to thank Ahyoung Sung for training me and sharing her valuable experience with me through my junior years. I would like to thank Xueling Chen for helping me with the Simics virtual platform.

I would like to thank all my friends in the Esquared lab and UNL, Xiao, Zhihong, Ahyoung, Isis, Yurong, Sandeep, Supat, Shuai, etc. I cannot imagine how I would have survived through all these years without your friendship. Especially, I would like to thank our girl group, Zhihong, Ahyoung and Isis, for many happy moments. I believe the ISSTA trip at Chicago is one of the best memories in my life.

Lastly, I would like to thank my family, Mom, Dad, and my loving spouse - Ziyuan, for their continuous love and encouragement.

GRANT INFORMATION

This work is supported in part by the Air Force Office of Scientific Research through award FA9550-10-1-0406, the Army Research Office through award W911NF-13-1-0154, the Air Force Research Laboratory through award FA8750-14-2-0053, the National Science Foundation through awards CNS-0720757, CCF-1161767 and CNS-1205472. W911NF-13-1-0154.

Contents

Contents **vii**

List of Figures **xi**

List of Tables **xiii**

1 Introduction **1**

1.1 A Summary of the SimExplorer Framework 4

1.2 Contributions 5

2 Background and Related Work **8**

2.1 Background 8

2.1.1 Modern Software Systems 8

2.1.2 Test Oracles 9

2.1.3 Regression Testing 9

2.1.4 Interrupts 11

2.1.5 Elusive Faults in Modern Software Systems 13

2.1.6 Genetic Algorithms 14

2.1.7 Virtual Platforms 15

2.2 Related Work on Verifying Modern Software Systems 17

2.2.1 Testing-based Verification 17
2.2.2 Non-testing-based Verification 23

3 SimTester: Automatic Testing for Concurrency Faults in Embedded Software 26

3.1 Introducing SIMTESTER 29
3.2 Utilizing SIMTESTER 30
3.2.1 Data Race Detection 31
3.2.2 Deadlock Detection 40
3.3 Empirical Study 44
3.3.1 Results and Analysis 46
3.3.1.1 RQ1: Testing for Race Conditions 46
3.3.1.2 RQ2: Testing for Deadlock 48
3.3.1.3 Effectiveness of Techniques at Detecting Seeded Faults . . 49
3.4 Further Discussion 50
3.5 Conclusion 53

4 SimRacer: Automatic Testing for Process-Level Races 54

4.1 Motivation 57
4.2 Approach 60
4.2.1 Recording Events 62
4.2.2 Phase 1: Identify Potential Race Sources 63
4.2.3 Phase 2: Test for Real Races 67
4.2.4 Further Discussion 71
4.3 Empirical Study 73
4.3.1 Objects of Analysis 73
4.3.2 Variables and Measures 76
4.3.3 Study Operation 78

4.3.4 Threats to Validity . . . 79
4.4 Results and Analysis . . . 80
4.5 Discussion . . . 82
4.6 Conclusion . . . 86

5 SimLatte: Using Testing to Estimate Worst-Case Interrupt Latencies in Embedded Software 88

5.1 Background: Factors that Affect Latency . . . 89
5.2 SimLatte . . . 93
5.2.1 SimLatte's Architecture . . . 93
5.2.2 SimLatte's Genetic Algorithm . . . 94
5.3 Empirical Study . . . 98
5.3.1 Objects of Analysis . . . 98
5.3.2 Setting GA Parameters . . . 100
5.3.3 Variables and Measures . . . 101
5.3.4 Study Operation . . . 102
5.3.5 Threats to Validity . . . 103
5.3.6 Results and Analysis . . . 103
5.3.6.1 RQ1: Effectiveness of SIMLATTE vs. RANDOM . . . 104
5.3.6.2 RQ2: The Role of the GA and Opportunistic Interrupt Invocation in SIMLATTE . . . 105
5.3.6.3 RQ3: Dependability of Techniques . . . 106
5.4 Discussion . . . 108
5.5 Conclusion . . . 112

6 SimRT: Automatic Regression Testing for Data Races 114

6.1 Background . . . 115

6.1.1 Race Detection and Verification . . . 115
6.1.2 Testing for Races . . . 116
6.2 Approach . . . 117
6.2.1 Overview of SimRT . . . 119
6.2.2 Shared Variable Identification . . . 120
6.2.3 Shared Variable Impact Analysis . . . 121
6.2.4 Matching Shared Variables . . . 126
6.2.5 Regression Test Selection . . . 126
6.2.6 Test Case Prioritization . . . 129
6.3 Empirical Study . . . 130
6.3.1 Objects of Analysis . . . 131
6.3.2 Variables and Measures . . . 134
6.3.2.1 Independent Variable . . . 134
6.3.2.2 Dependent Variables . . . 135
6.3.3 Study Operation . . . 136
6.3.4 Threats to Validity . . . 137
6.3.5 Results and Analysis . . . 138
6.3.5.1 RQ1: Regression Test Selection . . . 138
6.3.5.2 RQ2 and RQ3: Test Case Prioritization . . . 140
6.4 Summary and Discussion . . . 142
6.5 Conclusion . . . 145

7 Conclusion and Future Work 147

Bibliography 150

List of Figures

1.1 SIMEXPLOXER Framework 4

2.1 Interrupt latency 12

2.2 HW/SW co-design when using a Virtual Platform. 17

3.1 Overview of the SimTester architecture. 29

3.2 Faulty code that can cause data races in the UART driver in Linux. 32

3.3 Sample trace information for race detection. 38

3.4 Algorithm to determine whether it is possible to issue an interrupt. 39

3.5 Faulty code that can cause deadlocks. 41

4.1 Two processes race on a file 57

4.2 Process-level race due to a software signal 57

4.3 A deadlock in the Apache Web-server 58

4.4 A process-level race in an Android application 58

4.5 A process-level race in a UART device driver 59

4.6 Overview of SimRacer 61

4.7 Happens-before example (solid arrow for happen-before, dashes for potential races) . . 66

4.8 Phase 2 algorithm to test for races between processes 68

4.9 Phase 2 algorithm to test for races between a process and a signal 69

5.1 SimLatte Architecture 93

5.2 WCILs calculated during testing of L_{UT0}, H_{INT1} and S_{ADC} 107

6.1 Original HM program P (left) and a modified version of the HM program P' (right) . . 118

6.2 Overview of SimRT 119

6.3 ImpAnalyzer algorithm 122

6.4 ImpAnalyzer algorithm (cont'd) 123

6.5 Regression test selection algorithm 127

6.6 Efficiency: testing times for smaller objects 138

6.7 Efficiency: testing times for larger objects 139

List of Tables

4.1 Object Program Characteristics 75
4.2 Object Program Statistics 77
4.3 Results 81

5.1 Interrupt Examples 90
5.2 Initial Test Case Population 96
5.3 Result of Test Case Selection 97
5.4 Result of a CrossOver Operation 97
5.5 Result of a Mutation Operation 97
5.6 Object Program Characteristics 99
5.7 Technique Effectiveness 104
5.8 Interrupt Density of Random Testing 110
5.9 WCILs via SIMLATTE and Static Analysis 111

6.1 Objects of Analysis and their Characteristics 133
6.2 Race Detection Effectiveness 140
6.3 APFD Values for Selected and All Test Cases 141

Chapter 1

Introduction

Modern computer software systems are highly concurrent, memory intensive, and sensor intensive. For example, most computer software systems currently employ multi-core processors, making concurrent programming a natural way for developers to achieve higher performance. Furthermore, today's embedded systems ranging from consumer electronics to safety-critical devices are equipped with various sensors and peripherals to enable advanced features. These characteristics make these systems very complex and can result in varieties of elusive faults [1] that can be difficult to identify, isolate, and correct. While verification techniques such as model checking have been effective for detecting such faults in certain contexts, it can be still challenging to use these techniques in practice. Therefore, testing is still commonly used to assess and find faults in these systems.

To efficiently and effectively test software, developers must be able to *observe* and *control* execution. Where *observability* is concerned, *test oracles* are needed to inspect system behavior for correctness. Unfortunately, testing for faults in modern software systems is difficult simply because the classes of faults (e.g., data races) that occur in these systems are often "intermittent", making the traditional testing approaches of using *output-based*

[1] Elusive faults are intricate enough that their activation conditions depend on complex combinations of internal states and external requests, that occur rarely and can be very difficult to reproduce [1].

oracles ineffective [2, 3]. Furthermore, the ability to observe system runtime information is required to guide controllability at certain program points.

Over the past decade, researchers have developed approaches for achieving observability through runtime monitors (e.g., [4]). These approaches tend to focus only on application execution and do not monitor events occurring in lower-level software components such as device drivers and OS modules. As such, they can be ineffective for revealing subtle faults that can appear in hardware, device drivers, and kernels. In addition, these approaches can obscure lower-level information because they rely on instrumentation that can perturb lower-level system states (e.g., cache, bus, and register usage).

Where *controllability* is concerned, testing techniques must be developed to increase the chance of exposing faults. This means that sources of unpredictability in program execution due to scheduling and interrupt events must be controllable during testing. As an example, to reveal an interrupt related fault, engineers should be able to force interrupts to occur at a particular location. Unfortunately, existing approaches for randomly forcing interrupts (e.g., [5]) are not powerful enough to support such a precise requirement. In the ideal case in which randomly invoking interrupts does expose faults, it may miss faults that can occur due to other interleavings.

Several existing approaches have tried to abstract away scheduling non-determinism in concurrent programs to achieve greater execution control (e.g., [6]). These approaches often control thread scheduling within a single application process. However, they have rarely been adapted to detect concurrency faults that occur due to shared hardware resources or across different applications in modern software systems.

In addition, embedded systems tend to be interrupt-driven, yet the presence of interrupts can affect system dependability because there can be delays in servicing interrupts. Such delays can occur when multiple interrupt service routines and interrupts of different priorities compete for resources on a given CPU. For this reason, researchers have sought

approaches by which to estimate worst-case interrupt latencies (WCILs) for systems. While many advances have been made in the area of real-time system verification, particularly in the area of static analysis (e.g., [7, 8, 9, 10, 11]), the correctness of interrupt-driven systems remains difficult to verify. Without running the system being assessed, it is difficult for static analysis to identify locations where interrupts can occur because interrupts are platform dependent, and can be disabled and enabled by other software components or underlying systems at arbitrary execution points. Further, with the complexity of real-time systems and the varieties of interrupt sources, estimating the number of nested interrupts that can occur within an execution path can be difficult. Conservative static analysis techniques can over-approximate WCILs, and this affects their precision and applicability.

While the challenges for testing modern software systems are extensive, additional challenges arise as software evolves. Regression testing is used to perform re-validation of evolving software. In practice, traditional regression testing can be expensive when extensive existing regression test suites must be executed following program changes. Furthermore, applying high overhead testing techniques following modifications (e.g., dynamic race detection) using large regression test suites can make regression testing processes become more expensive. There has been research on reducing the cost of regression testing including research on techniques for regression test selection and test case prioritization targeting particular fault classes related to internal oracles [12,13]. However, these techniques focus on sequential software and have not considered concurrent software systems.

Given the foregoing discussion, the overall goal of our research is to provide a testing framework for modern software systems, SIMEXPLORER, with the aid of *observability* and *controllability* support at lower system layers. We present a set of techniques applied to both single software versions and evolving software.

It is important to investigate the SIMEXPLORER framework on different types of programs since program characteristics may impact how well various techniques work. There-

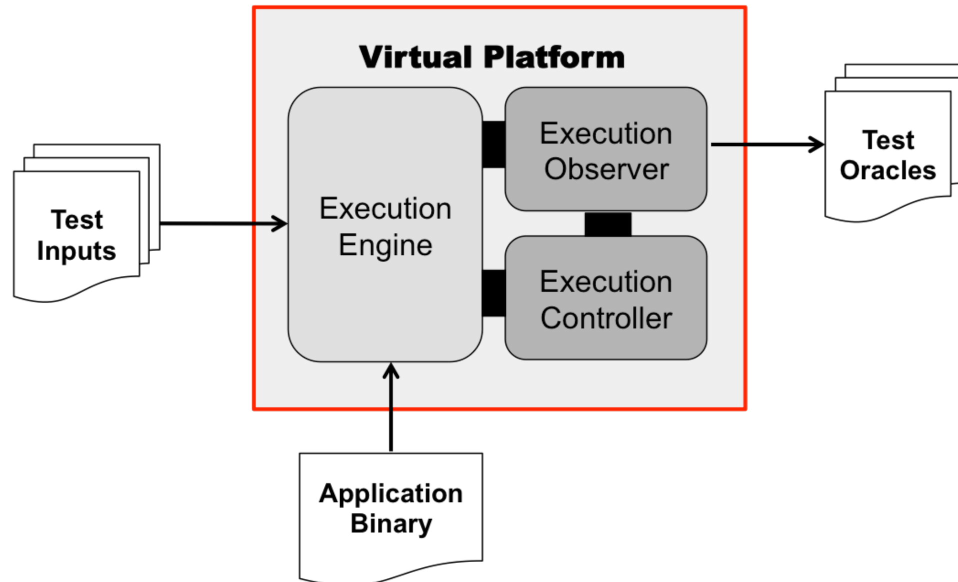

Figure 1.1: SIMEXPLOXER Framework

fore, a major element of our work involves empirical investigation of the SIMEXPLORER framework on real software systems. Our results offer useful suggestions for practical use.

1.1 A Summary of the SimExplorer Framework

SIMEXPLORER takes advantage of virtual machines (VMs) to achieve deep observability and fine-grained controllability. Figure 1.1 shows an overview of SIMEXPLORER framework. Given a set of test inputs, a target application binary is executed by the *execution engine*, which is an internal component in the VM. The *execution observer* and *execution controller* are two major components implemented as external modules in the VM. The *execution observer* observes system execution, including events occurring in both application and lower-level software components such as hardware, device drivers and OS modules. The *execution controller* controls occurrences of events across the entire system such as thread scheduling, process scheduling, signals, and hardware interrupts. The test oracle

specifies in which condition a fault occurs so that engineers can determine whether a test has elicited a fault.

Based on this generic SIMEXPLORER framework, we propose a set of techniques that instantiate SIMEXPLORER to effectively and efficiently test for classes of elusive faults in modern computer software systems. These faults include: (1) concurrency faults caused by interactions between applications and hardware interrupts; (2) concurrency faults caused by improper shared resource access among multiple processes; (3) concurrency faults caused by incorrect arrivals of software signals; and (4) violations of expected worst-case interrupt latencies. In addition, we propose an automated regression testing technique for use in detecting data races that are induced in concurrent programs by code modifications.

1.2 Contributions

This dissertation makes the following contributions:

1. We introduce an instantiation of SIMEXPLORER that allows test engineers to detect concurrency faults that occur due to interactions between software and hardware. We consider two types of concurrency faults: data races and deadlocks. Our technique is implemented on the Simics Virtualization Platform [14]. The technique achieves observability by utilizing the VM's abilities to monitor memory accesses and hardware states as a program executes and report faults related to oracles; the observed information is also used to guide controllability at certain program points. Controllability is achieved by utilizing the VM's abilities to interrupt execution without affecting the states of the virtualized system. As such, engineers can manipulate memory and buses directly to force interrupts so that they test only the locations that have potential races and deadlocks. We evaluate the effectiveness of our technique on a real

Linux device driver, and compare it to that of two existing baseline approaches. We present this work in Chapter 3.

2. We propose an instantiation of SIMEXPLORER that allows engineers to effectively test for races at the process level (running in both user and kernel modes) that involve files, shared memory, hardware components, signals, and processing cores. The technique is implemented on Simics and operates in two phases. The first phase employs observability to generate runtime information related to shared resource accesses (e.g., accessing system-level resources through system calls, accessing hardware registers) and process-level synchronization operations (e.g., `fork`, `wait`) that are used to analyze the system for potential sources of races. The runtime information is also used to direct the scheduling effort in the second phase. The second phase employs controllability to permute process interleavings by causing the kernel scheduler to force races to occur at potential racing points obtained in the first phase. The technique allows engineers to focus only on applications under test without worrying about other events or applications in the system. Side effects from execution of applications not under test is determined by querying current system states. By using the VM, our technique is transparent; that is, it does not require any instrumentation or kernel modification. In addition, our technique operates at the binary level so it is capable of detecting races between applications written in different languages. We evaluate the effectiveness of our technique on a set of real-world applications, and compare it to that of robust stress testing. This work is presented in Chapter 4.

3. In addition, the SIMEXPLORER framework is integrated with existing test case generation approaches to enhance testing effectiveness for timing related faults including those caused by excessive interrupt latency. We present a testing-based approach, SIMLATTE, another instantiation of the SIMEXPLORER framework for estimating

WCILs. First, we identify a set of factors that impact WCILs. Based on these factors, we next use a genetic algorithm (GA) to generate inputs and interrupt arrival points that favor paths with longer WCILs. The exploration guided by the GA is expected to cover a wide range of combinations of inputs and locations. The technique then employs an opportunistic interrupt invocation mechanism to invoke interrupts at feasible locations that are likely to cause longer WCILs. Our technique is implemented on AVRORA, a cycle-accurate simulator for Atmel microcontrollers [15] that achieves the observability and controllability needed to effectively estimate WCILs. We evaluate the effectiveness and efficiency of our technique on several non-trivial embedded systems, and compare them to those of random testing and static analysis. This work is presented in Chapter 5.

4. Finally, to cost-effectively test for data races that are induced in concurrent programs by code modifications, we present SIMRT, an automated regression testing technique for use in detecting races introduced by code modifications. First, SIMRT employs a regression test selection technique, focused on sets of program elements related to race detection, to reduce the number of test cases that must be run on a changed program to detect races that occur due to code modifications. Specifically, it employs escape analysis and code differencing to identify a set of impacted shared variable pairs as coverage targets, and then select test cases to cover these targets. Second, SIMRT employs a test case prioritization technique to improve the rate at which such races are detected. Specifically, it employs a greedy prioritization algorithm to schedule test cases in orders that detect races faster. We evaluate the effectiveness and efficiency of our technique on several real-world concurrent programs, and compare them to those of baseline regression test selection and test case prioritization techniques. We present this work in Chapter 6.

Chapter 2

Background and Related Work

In this chapter we first describe background related to testing modern software systems. We then discuss related work.

2.1 Background

2.1.1 Modern Software Systems

Modern computer systems ranging from personal computers to consumer electronic devices are becoming increasingly complex. These systems are utilizing high-performance multicore processors to ensure adequate responsiveness and performance. They also utilize a full array of peripheral devices and sensors to support required features. Competition for market share means that new features are frequently added to these systems, making their product life-cycles last only one to two years.

The classification of modern software systems is broad, and we summarize five characteristics of the software running on top of these systems: 1) it can frequently interact with hardware to control system behaviors and thus it is hardware dependent, 2) it can employ different concurrency mechanisms to coordinate threads and processes, 3) it can be pro-

grammed with interrupts to interact with the external environment, 4) it can have timing constraints, and 5) it can produce internal faults that cannot be detected using traditional output-based oracles.

Note that these characteristics may occur to different extents in different systems, and they can also be present in other classes of systems, however they are centrally important to the modern computer software systems we are considering in this work, and our solutions must accommodate them.

2.1.2 Test Oracles

In the testing literature, a *test oracle* is the device by which engineers determine whether a test has elicited a failure in a system.

The most typical approach to verifying test results involves using *output-based oracles*; that is, oracles that check system outputs. However, faults may escape detection if a test suite fails to propagate their effects to program outputs. In such cases, output-based test oracles are inadequate.

In contrast, *internal oracles* monitor and enforce constraints on internal program states seeking evidence that *infections* have occurred – that is, cases in which program states have been altered in violation of the constraints enforced by the given oracles. These oracles then signal the presence of those infections to alert testers to the possible presence of faults.

2.1.3 Regression Testing

Let P be a program, let P' be a modified version of P, and let T be a test suite for P. Regression testing is concerned with validating P'. To facilitate this, engineers often begin by reusing T, but reusing all of T (the *retest-all approach*) can be inordinately expensive. Thus, a wide variety of approaches have been developed for rendering reuse more cost-

effective via regression test selection and test case prioritization ([16] provides a recent survey).

Regression test selection (RTS) techniques attempt to select, from test suite T, a subset T' that contains test cases that are important to re-run, and omit test cases that are not as important. Rothermel et al. [17] presented the RTS technique DEJAVU that performs simultaneous depth-first traversals on control flow graphs (CFGs) for procedures in P and P' to find *dangerous edges* that lead to code that has changed. Execution traces of test cases (bit vectors indicating whether basic blocks were covered) on P are then used to select test cases that traversed dangerous edges in P. The authors have shown [18] that when certain conditions are met, DEJAVU is *safe*; i.e., it cannot omit test cases which, if executed on P', would reveal faults in P' due to code modifications.

Test case prioritization (TCP) techniques attempt to reorder the test cases in T such that testing objectives can be met more quickly, and one potential objective involves revealing faults. A wide range of TCP techniques have been proposed and studied, and one particular successful one is the "additional-block-coverage" technique [19]. Given the results of executing tests and gathering trace information, this technique prioritizes test cases in terms of the numbers of new (not-yet-covered) basic blocks they cover, by iteratively selecting the test case that covers the most not-yet-covered blocks until all blocks are covered, then repeating this process until all blocks have been covered.

Because TCP techniques do not themselves discard test cases, they can avoid the drawbacks that can occur when regression test selection cannot achieve safety. Alternatively, in cases where discarding test cases is acceptable, test case prioritization can be used in conjunction with regression test selection to prioritize the test cases in the selected test suite. Further, test case prioritization can increase the likelihood that, if regression testing activities are unexpectedly terminated, testing time will have been spent more beneficially than if test cases were not prioritized.

A key insight behind the use of regression testing techniques such as those that we have just described is that certain testing-related tasks such as that of gathering coverage information can be performed in the "preliminary period" of testing, before changes to a new version are complete. The information derived from these tasks can then be used during the "critical period" of testing after changes are complete and where time is more limited.

2.1.4 Interrupts

Interrupts are widely used in modern software systems such as microcontroller firmware, operating systems and device drivers. For highly resource constrained embedded software with a thin OS layer, such as software in flight control systems and sensor network nodes, it is common to implement interrupt handlers using applications that run on top of microcontrollers.

There are several varieties of interrupt handling mechanisms available, depending on the types of processors being used. The Atmel AVR processor, for example, is commonly used for interrupt-driven applications, supports a wide range of peripheral devices, is energy efficient, and offers good cross-platform support. Each device in this processor is associated with one unique interrupt, and each interrupt is associated with an *interrupt pending bit* that is set when data is available for the device. Applications written for this microcontroller can be programmed with *interrupt service routines* (ISRs) that handle data relevant to each particular interrupt. The pending bit for an interrupt remains set until the program jumps to the ISR for that interrupt. The microcontroller has a *global interrupt enable bit*, and each interrupt has an associated *local interrupt enable bit*. An interrupt can be invoked only when both its local interrupt enable bit and global interrupt enable

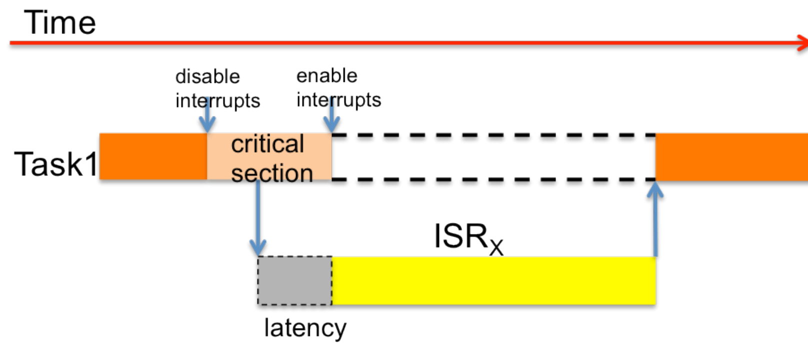

Figure 2.1: Interrupt latency

bit are set. When multiple interrupts are about to execute, the processor selects the lowest numbered interrupt and executes it.

Interrupt latency is the time that elapses from when an interrupt is generated (its interrupt pending bit is set) to when the source of the interrupt is serviced (the associated ISR starts executing). Figure 2.1 shows one scenario involving interrupt latency. In this case, an interrupt X occurs while task1 is executing its critical section; at this point the execution of ISR_X is postponed until the critical section is exited. A system's *worst-case interrupt latency* (WCIL) is the longest possible latency across all possible executions of that system.

Interrupt latency can be calculated theoretically by summing up expected hardware latency and software latency [20] times. Hardware latency time is related to the design of microprocessors such as cache and pipelines. Software latency is related primarily to the length of critical sections, the priority assignments made to different interrupts, and the implementation of ISRs.

There are several principles that interrupts must follow. First, unlike threads, interrupts cannot be preempted by normal program routines; instead, they can be preempted only by other interrupts. Second, *nested* interrupts occur when one interrupt handler preempts another, and this can occur only when the ISR is set to be preemptible. Third, *reentrant* interrupts are interrupts that can preempt themselves. Reentrant interrupts are used

only in special situations [5] such as when it is necessary need to reduce interrupt latency. Fourth, lower priority interrupts can never preempt higher priority interrupts. Fifth, there is typically a *minimum inter-arrival time*, or its inverse, a *maximum interrupt frequency* associated with interrupts. The inter-arrival time imposes a restriction on how frequently an interrupt can be invoked.

2.1.5 Elusive Faults in Modern Software Systems

We now discuss several classes of common faults in modern software systems with respect to their characteristics. These include various forms of concurrency faults that occur at the thread-level, interrupt-level and process-level, and faults that violate real-time constraints. Such faults can lead to well-known, but intermittent and hard-to-reproduce failures [21,22].

Thread-level concurrency faults. There are three general classes of concurrency faults: data races, atomicity violations and deadlocks. A data race occurs when two threads can simultaneously access a shared variable, with at least one access being a write [23]. Atomicity violations are introduced when programmers assume some code regions to be atomic, but fail to guarantee the atomicity in their implementation [24]. Deadlock occurs when threads circularly wait for each other to release acquired resources [25].

Interrupt-level concurrency faults. In modern systems, the frequent use of interrupts for timing, sensing, and I/O processing can cause concurrency faults to occur due to interactions between applications and hardware interrupt handlers. For example, a data race occurs when a hardware interrupt is triggered during the execution of an application, and the interrupt handler modifies a shared variable that is incorrectly read by the application later on. It is common for interrupt handlers to use non-preemptive mechanisms such as spin-locks instead of preemptive mechanisms such as semaphores to protect critical code regions. This is because typically, ISR code is not allowed to sleep. As such, a deadlock

occurs when the application is preempted by the interrupt handler while the application is holding a spin-lock, and the interrupt handler tries to acquire the same spin-lock and becomes stuck in the spin-lock loop.

Process-level concurrency faults. Numerous program analysis and testing techniques have been proposed to detect concurrency faults between threads. Little work, however, has addressed the problem of detecting and testing for concurrency faults, such as data races at the process-level. *Process-level races* occur when multiple processes access a system-wide shared resource (e.g., file, device, hardware register) without proper synchronization [26]. Process-level races often involve shared resources accessed through system calls. Thus, testing for process-level races requires the effects of read/write and synchronization operations involving system calls to be accounted for. However, system calls operating on shared resources are typically treated as black boxes by engineers who use them to develop applications. Furthermore, most race detection algorithms are designed to work only with memory read/write and well-defined synchronization operations at the thread level [26].

Interrupt latency faults. One factor affecting the occurrence of timing constraints violations is interrupts. Modern embedded systems tend to be interrupt-driven, and these systems can suffer from untimely interrupts due to poor system design or erroneous implementations. In this work, we consider faults related to timing constraint violations caused by interrupt latency. An interrupt latency fault is exposed when the actual interrupt latency exceeds the system's worst-case interrupt latency (WCIL).

2.1.6 Genetic Algorithms

A genetic algorithm (GA) is a programming technique that mimics some of the processes observed in biological evolution. GAs are a combination of search space and optimal solutions and are good for navigating very large search spaces. As such, GAs are often used

to search for optimal solutions that might not be found easily. Given the advantages of GAs, they have been widely applied in many fields in the engineering world. For example, in software testing, GAs have been successfully used to automate the process of test case generation [27, 28, 29]. GAs have also been used to test for timing constraint violations in real-time systems [30] and for faults unrelated to timing constraint violations (e.g., stack overflow) in interrupt-driven software [5]. A GA begins with an initial (often randomly generated) test data population and "evolves" the population toward goals (targets) such as worst-case execution time (WCET) and worst-case stack usage. To apply a GA to a system, that system's test inputs must be represented in the form of a "chromosome", and a "fitness function" must be provided that defines how well a chromosome satisfies the intended goal. The algorithm proceeds iteratively by evaluating all chromosomes in the population and then selecting a subset of the fittest to "mate". These are combined in a "crossover" stage in which information from one half of the chromosomes is exchanged with information from the other half to generate a new population. A small percentage of chromosomes in the new population are then "mutated" to add diversity back into the population. This concludes a single generation of the algorithm. The process is repeated until a stopping criterion such as a time limit has been met.

2.1.7 Virtual Platforms

Virtual platforms (VMs) have been used to support tracing, replay, and debugging [31, 32, 33, 34, 35]; however, they have rarely been used to support testing. One notable example is work by Goh et al. [36]. They introduce a VM-based online testing approach to supplement off-line testing. With off-line testing, all possible test inputs may not be known ahead of time since embedded software systems are often influenced by external environments. Their work, however, did not consider the use of observability and controllability

to enhance testing effectiveness. On the other hand, there have been some efforts to extend virtual platforms to provide greater observability in the security domain [37, 38, 39]. However, these efforts did not utilize the additional observational power for testing.

Virtual platforms provide an attractive platform on which to carry out testing for three reasons:

1. *Non-intrusive instrumentation of executables.* Instrumentation occurs at the binary level and without disturbing execution or affecting the virtualized state of a system. Therefore, full-system simulators can simulate and profile systems accurately in the presence of instrumentation. Furthermore, these simulators can collect exact profile data instead of relying on sampling or probability. As such, the profiled information is more complete. For the problem we are addressing, this is an important consideration.

2. *Support for more types of executables.* Full-system simulators support executables with or without operating systems. This is different than other approaches, which are operating system dependent (e.g., Pin works only on Linux or Mac OS X binaries). As such, our approach can work in diverse applications and systems ranging from executables running in stand-alone embedded devices with no operating systems, to executables running in large computing clusters.

3. *Popularity.* HW/SW co-design is a widely adopted method for creating modern computer systems. As such, full-system simulators such as Simics already play a prominent role in the development process. Developers who already use VMs for co-design can easily integrate the proposed extension into their testing and debugging toolkits. Figure 2.2 shows the co-design process a developer typically follows when working with a VM. The process starts with the design of both software and hardware based on their specifications. After that, both software and virtual hardware are created using the VM. In a co-simulation step, the software is simulated together with the virtual hardware.

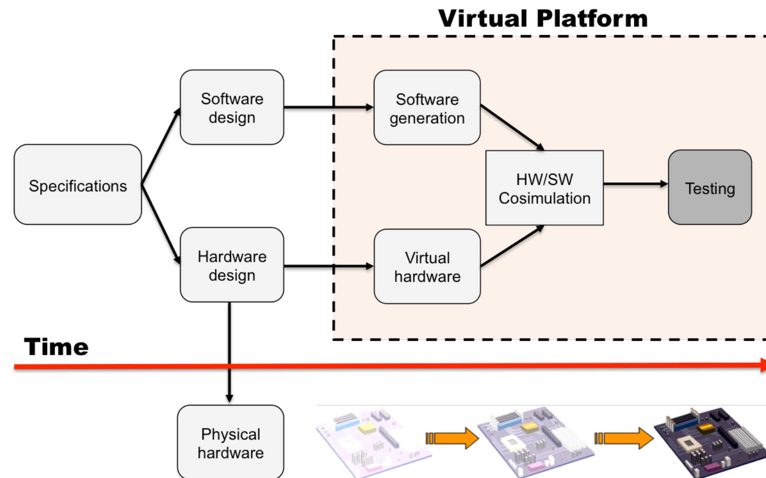

Figure 2.2: HW/SW co-design when using a Virtual Platform.

Finally, testing or other verification techniques are applied to verify the correctness of target software. In the traditional co-design process, software is integrated with hardware after hardware is fabricated. This may delay the co-design process and cause late discovery of software faults. By using VMs, software and hardware can be jointly simulated and tested earlier in the design cycle. This allows parallel development of physical hardware. As such, testing can be done while physical hardware is under development, and the costs of testing are amortized into the co-design process.

2.2 Related Work on Verifying Modern Software Systems

In this dissertation, we classify modern software system verification into testing and non-testing-based approaches. We discuss related work for both categories.

2.2.1 Testing-based Verification

Testing for thread-level concurrency faults. There has been a great deal of research on basic techniques for testing multi-threaded programs for data races [40, 41], atomic-

ity violations [24, 42], actual deadlock [43, 44], and potential deadlock [45, 46]. There are common algorithms for detecting data races involving lock-sets [23, 47] and vector-clocks [4, 40]. Lock-set based techniques are efficient but can report false warnings. Vector clock based techniques are precise but can be expensive. As such, hybrid approaches [41,48] such as ThreadSanitizer have been created to yield better precision with lower overhead. These techniques enhance observability by monitoring events such as memory access and synchronization operations. Some techniques have been applied to large systems (e.g., Apache, Mozilla) [42,49,50]. These techniques use algorithms to analyze synchronization operations (e.g., lock acquire/release) among multiple threads in the execution traces of successful program executions.

For *controllability*, there have been some testing techniques that permute thread interleavings to increase the possibility of exposing faults. For example, active testing techniques (e.g., [6]) first identify potential concurrency faults, and then control the underlying scheduler by inserting delays at context switch points. CHESS systematically explores program state spaces to detect potentially erroneous interleavings [51]. These techniques, however, focus on thread level concurrency while ignoring concurrency faults caused by system-level events such as interrupts, signals and different processes

Testing for interrupt-level concurrency faults. As noted above, existing techniques on testing for thread-level concurrency faults have rarely been adapted to work in scenarios in which concurrency faults occur due to asynchronous interrupts. It is unclear whether these approaches can work in such a scenario for two reasons. First, controlling interrupts requires *fine-grained* execution control; that is, it must be possible to control execution at the machine code level rather than the program statement level, which is the granularity at which many existing techniques operate [6, 52]. Second, occurrences of interrupts are highly dependent on hardware states; that is, interrupts can occur only when hardware

components are in certain states. Existing techniques are often not cognizant of hardware states [5,53].

There are several techniques for testing embedded systems with a particular focus on interrupt-level concurrency faults. Regehr et al. [5] use random testing to test Tiny OS applications. They propose a technique called restricted interrupt discipline (RID) to improve naive random testing (i.e., firing interrupts at random times) by eliminating aberrant interrupts. In our approach, as will be discussed in Chapter 3, however, interrupts are fired conditionally instead of randomly. Our evaluation shows that conditionally fired interrupts increase the chances of revealing faults while reducing cost. Lai et al. [54] present a notation for modeling interrupt-driven nesC applications for testing purposes. They formulate two test adequacy criteria based on the notation. Their approach does not provide observability or controllability. In their approach, test cases are randomly selected from a test pool until coverage has been achieved. We believe that this process can be made more effective by adding controllability. Higashi et al. [53] improve random testing via a mechanism that causes interrupts to occur at all instruction points to detect interrupt related data races. However, this approach can be both ineffective and inefficient as it cannot determine the location at which to issue an interrupt. More details will be presented in Chapter 3.

Testing for process-level concurrency faults. Unlike thread-level race detection, which often relies on binary/bytecode instrumentation to insert yield points at the application level (e.g., CHESS [55]), process-level race detection techniques must be able to control the kernel's scheduler. Furthermore, these techniques must have access to various system states to ensure process and event schedulability. Achieving even partial system control often requires custom kernel modifications, limiting the practicality of the techniques. For example, RACEPRO can be used only in a custom Linux kernel that has been modified to provide event recording and replay capabilities [26]. It also cannot precisely control process

scheduling, making it ineffective as a platform for testing for various types of process-level races.

There have been several approaches created for detecting races between processes [26, 56,57]. Detecting TOCTOU (time of check to time of use) races [57] has been a topic in the security community. TOCTOU races typically happens when the permission check and use of a file in one process is not atomic so that a malicious process can slip in and attacks the file. However, these techniques are non-testing based approaches; the techniques depend on the current execution at deployment time. In addition, TOCTOU races are just one specific type of process race while our technique deals with general process races.

RACEPRO [26] leverages the vector-clock algorithm that has been widely used in dynamic techniques for thread-level race detection [4,58]. As such, this technique may detect a race only if it happens during a test execution, and it cannot predict a potential race; this is one of the limitations of vector-clock algorithms. Our technique, as will be discussed in Chapter 4, on the other hand, considers all shared resource access pairs that are potentially racy; over-estimation can be overcome by the race verification phase. In addition, our technique can detect races that occur due to unsynchronized hardware accesses by monitoring hardware states, and races that occur specifically on multi-core systems while RACEPRO deals with neither of them.

To deal with false alarms generated by imprecise race detectors, replay techniques have been used to identify false positives [26, 48]. ThreadSanitizer [48], for example, inserts sleeps after racing instructions; a real thread-level race is reported if the two sleeps happen simultaneously. RACEPRO [26], on the other hand, replays the original execution and its variations (reordering system calls) to validate whether a detected race is harmful by observing the output. RACEPRO is scalable and effective for detecting faults in large software systems. A major drawback of this approach, however, is replay divergence failure (i.e., a

mismatch between replayed system calls and actual system calls). Such a failure can occur when the actual system states do not allow the system to replay or switch system calls.

There has been some work on testing for races caused by software signals [59,60]. For example, Tahara et al. [60] issue signals when an application process attempts to access a global variable, amplifying the chance of exposing faults. However, this technique does not attempt to reorder the global variable access and the signal handler. Thus, it is possible that the signal handler and the global variable will be synchronized, resulting in false positives. Again, these techniques can handle only signal races while our technique focuses on general process races.

Testing for timing constraint violations. There has been some work on dealing with interrupts using testing and dynamic analysis techniques [5, 53, 61, 62, 63]. None of the techniques, however, attempt to address interrupt-related timing constraint violations such as interrupt latencies.

There has been work on using black-box testing to test embedded systems for timing contraint using search-based or evolutionary algorithms [30, 64, 65, 66, 67, 68, 69]. For example, Iqbal et al. [30] model the environments of real-time embedded systems, and guide searches to generate test cases reaching a system's error states. Briand et al. [66] apply model-based stress testing to real-time systems using genetic algorithms. Their approach encodes chromosomes as task arrival times, and searches for schedules that can cause the system to miss deadlines. These methods are based on software modeling and do not consider a system's real runtime states. In addition, it is unclear how much manual effort is involved in creating system models. In contrast, as will be discussed in Chapter 5, our technique does not require source code annotation or modeling, and our inputs and interrupt schedules are generated under real system runtime environments.

Search-based algorithms have also been used in white-box testing to test for timing constraint violations [64, 65, 68]. Hartmut et al. [65] use an evolutionary algorithm to find

worst case execution times (WCET) and best case execution times (BCET) to test against timing constraint specifications. Bhattacharya et al. [69] use a genetic algorithm to maximize the possibility of exposing concurrency faults such as data races. Their technique generates locations of delays that can be injected in the execution of a thread. This technique does not apply in the context of interrupt-driven software.

Regression testing. As noted above, there has been a great deal of work on analyzing, detecting, and testing for data races [6, 23, 58, 70, 71]; however, existing techniques do not consider software evolution.

There has been some work on selecting and prioritizing schedules for testing multi-threaded programs such that faults can be exposed faster [51, 72]. For example, CHESS prioritizes the exploration of program state spaces to detect potentially erroneous interleavings. However, these techniques do not consider code changes.

There have been several techniques presented for systematically exploring schedules in multi-threaded programs across versions [73, 74]. Gligoric et al. [73] reuse results from exploration of one program version to speed up exploration of the next program version. Jagannath et al. [74] use information about program changes in software evolution to prioritize the exploration of schedules. These techniques, however, target exploration of schedules within individual test cases and do not address the challenges of regression testing involving large sets of test cases.

Recent work by Deng et. al [75] studies how existing concurrency fault detection tools work for a set of test inputs. They propose a technique that first measures coverage of a program, and then selects a subset of test inputs to test for data races and atomicity violations on that program. This technique focuses, however, on single version programs and does not consider code changes. In contrast, as will be discussed in Chapter 6, we reuse coverage information on the old programs and select test inputs to test the new programs.

There has been a great deal of research on improving regression testing through regression test selection and test case prioritization (e.g., [17, 19, 76, 77, 78, 79, 80, 81, 82, 83]) – Yoo and Harman [16] provide a recent survey. In this work , we restrict our attention to techniques that share similarities with ours.

Some RTS techniques target specific fault classes [12, 84, 85]. Our own previous work [12] selects test cases associated with system changes, but only those that are relevant to a set of internal oracles that are known to be important for the system under test. Staats et. al [13] propose a TCP technique that favors test orderings in which many variables impact the test oracle's result early in test execution. However, all the foregoing techniques focus on sequential programs, and do not address concurrency faults.

2.2.2 Non-testing-based Verification

Our focus in this dissertation is on testing-based approaches. For completeness, however, we note that there have been many non-testing-based verification approaches proposed for use on modern software systems, and we summarize these here.

Model checking for verifying system correctness. Model checking is a verification technique that can can be used to prove the absence of software faults. Given a model of a software system, a model checker exhaustively checks whether this model meets a given specification. The key strength of model checking is that it can detect faults that would be hard to detect using testing. This makes model checking particularly useful when verifying safety-critical systems. Furthermore, if a specification does not hold, a model checker can produce a counterexample, which can make fault localization become easier.

There has been research that uses model checking to verify concurrency properties of modern software systems. Chen and MacDonald [52] verify multithreaded Java programs by overcoming scheduling non-determinism through value schedules. One major

component of their approach involves identifying concurrent definition-use pairs, which then serve as targets for model checking. There are also approaches that overcome the difficulties of using model checking on multi-threaded programs with large search spaces [51, 86, 87, 88, 89]. For example, Musuvathi [51] bounds the number of context switches to alleviate the state explosion problem.

There has been a great deal of work on verifying real-time properties of modern software systems using model checking [7, 8]. For example, Hessel et al. [8] use UPPAAL, a model checker, to verify embedded systems that can be modeled as timed automata against their real time specifications. En-Nouaary et al. [90] focus on timed input/output signals for real-time embedded systems and present the timed Wp-method for generating timed test cases from a nondeterministic timed finite state machine. However, these techniques do not consider interrupts.

There has some research that uses model checking techniques for interrupt-driven software [91, 92]. For example, Schlich et al. [92] use model checking of assembly code software for microcontrollers. They propose a technique called *interrupt handler reduction* to reduce the number of program locations at which an interrupt needs to be considered. The goal of this technique is to reduce program states in model checking. This approach, however, does not check real-time properties of the software. Brylow et al. [93] apply a model checker to verify interrupt related properties such as interrupt latency, but their approach cannot handle loops waiting for hardware changes. In later work, the same authors [91] address this drawback by identifying the code segments that cannot be analyzed using static analysis, leaving them for testing. The approach still requires annotations of all operations on manipulated interrupt bits in assembly code, which is platform dependent and requires substantial manual effort. Moreover, their timing analysis is applicable only on certain code segments, not the entire program.

Static analysis for verifying system correctness. Another common approach used to verify modern software systems is to apply static analysis techniques to discover paths and regions in code that might be susceptible to certain faults. Techniques based on program state modeling and transitions (e.g., [94]) have been used to verify device drivers and kernels [70, 95]. There are also static analysis techniques used to verify embedded software with interrupts. Tan et al. [96] design a type of annotation that can be used to detect OS concurrency faults related to interrupts. Their approach is based on statically analyzing comments and code. Jonathan et al. [97] first statically translate interrupt-driven programs into sequential programs by bounding the number of interrupts, and then use testing to measure execution time.

Sehlberg et al. [11] use static analysis to estimate WCETs (worst-case execution times) of vehicle control systems. The steps for performing such an analysis involve creating processor timing models and manual code annotations which require substantial manual effort. In addition, the results of the work show that static analysis overapproximates actual WCETs in most tasks under consideration. To overcome the drawbacks of static analysis, Kirner et al. [10] present a hybrid WCET analysis framework using testing together with static program analysis. All of the foregoing techniques, however, focus on sequential software systems, and none of them provides support for interrupts.

Static analysis is powerful because it is fast and cheap while generally effective. It can detect faults in the software that testing may not easily expose. However, there are several drawbacks associated with static analysis. First, static analysis can report false positives due to imprecise local information and infeasible paths. Second, as embedded systems are highly dependent on hardware, it is difficult for static analysis to annotate all operations on manipulated hardware bits; moreover, hardware events such as interrupts usually rely on several operations among different hardware bits.

Chapter 3

SimTester: Testing for Concurrency Faults in Embedded Software [2]

In this chapter, we describe SIMTESTER, an instantiation of the SIMEXPLORER framework that provides *observability* and *fine-grained controllability* features sufficient to allow test engineers to detect concurrency faults in embedded software. In software for embedded systems, the frequent use of interrupts for timing, sensing, and I/O processing can cause concurrency faults to occur due to interactions between applications and interrupt handlers. As an example, occurrences of data races between interrupt handlers and applications have been reported in a previous release of uCLinux, a Linux OS designed for real-time embedded systems. In this particular case the serial communication line can be shared by an application through a device driver and an interrupt handler. In common instances, the execution of both the driver and the handler would be correct. However, in an exceptional operating scenario, the driver could execute a rarely executed path. If an interrupt occurs at that particular time, simultaneous transmission of data is possible.

[2] The contents of this chapter have appeared in [62].

SIMTESTER takes advantage of features readily available in many virtual platforms to tackle the challenges of testing for concurrency errors in embedded software. Particularly, SIMTESTER achieves the levels of observability and controllability needed to test such systems by utilizing the virtual platform's abilities to interrupt execution without affecting the states of the virtualized system, to monitor function calls, variable values and system states, and to manipulate memory and buses directly to force events such as interrupts and traps. As such, SIMTESTER is able to stop execution at a point of interest and force a traditionally non-deterministic event to occur. The system then monitors the effects of the event on the system and determines whether there are any anomalies.

Many existing approaches for detecting concurrency faults are not widely used because they require significant deployment effort. We designed SIMTESTER to overcome deployment obstacles by implementing it on a commercial virtual platform called Simics [14, 98, 99]. We chose Simics for several reasons. First, similar to other full-system simulators, Simics provides functional and behavioral characteristics similar to those of the target hardware system, enabling software components to be developed, verified, and tested as if they are executing on the actual systems. Second, through a rich set of Simics APIs, software engineers have the ability to non-intrusively observe and control various system behaviors without ever needing the source code. Third, due to its powerful device modeling infrastructure, Simics already plays a critical role in hardware/software (HW/SW) co-design; therefore, adding the proposed capabilities to it will enable adoption without requiring much effort [100]. Thus, we envision that SIMTESTER will allow several aspects of product integration testing to be moved up to the co-design phase of system development. Fourth, licensing of Simics is free for academic institutions, making it a good platform for research.

SIMTESTER is implemented for applications running on x86/Linux environments. There are four major components that interact with Simics:

- A **configuration repository** stores initialization scripts containing information that includes execution break-points and variable locations that must be observed.
- An **execution controller** is an external module that can be attached to Simics. It invokes callback functions when events of interest occur (e.g., interrupts, memory read/write operations).
- An **execution observer** is another external module that can be attached to Simics. It monitors information generated by the execution controller and then records it in a file.
- An **oracle repository** stores test oracle files in the form of property requirements. For example, the oracle can specify in which condition data races or deadlocks occur. Each log file is compared against an oracle file to detect a particular type of anomalous execution behavior.

By using SIMTESTER, engineers can directly observe races and causes of deadlocks as they occur. They can also precisely control the occurrences of interrupts so that they can test every variable that can be accessed by both the application and interrupt handler for vulnerabilities to concurrency faults. SIMTESTER yields precise detection of data races; that is, it produces no false positives. It is also effective; that is, if races or deadlocks are possible on a shared variable under test, they can be found more easily than with other testing approaches. To evaluate the potential usefulness of SIMTESTER we apply it to test for two classes of errors. These include data races (using an approach similar to that introduced by Higashi et al. [53] but with additional optimization) and deadlocks between device drivers and interrupt handlers. Our results show that SIMTESTER can be effective and efficient.

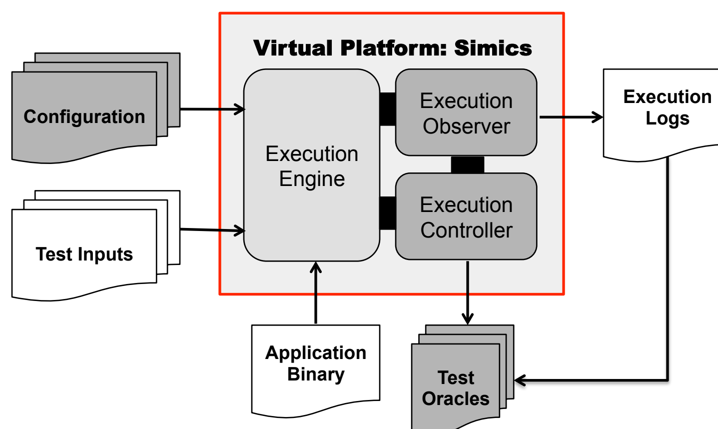

Figure 3.1: Overview of the SimTester architecture.

3.1 Introducing SIMTESTER

Figure 3.1 depicts the overall architecture of SIMTESTER. There are four major components in SIMTESTER in addition to Simics itself. As stated earlier, Simics provides APIs that can be accessed via Python scripts; thus, all components except the test oracles are Python scripts.

The first component is the *configuration* script, the content of which includes information such as locations at which to set execution breakpoints, addresses of variables that need to be monitored, and machine instructions that need to be monitored.

The second component is the *execution controller* program. This program specifies certain events to be invoked at particular points in executions. It can generate data that can cause the system to take different execution paths. As an example, SIMTESTER can artificially create I/O interrupts simply by writing data to the I/O bus or to the memory locations that have been mapped to hardware devices. It can also force the system to execute a particular exception handling routine by artificially creating that exception. Finally, it can force the system to execute a particular path by specifically setting a conditional value.

The third component is the *execution observer* program, which monitors and generates information that can either be recorded into a file for offline analyses (used to detect data races) or fed directly into test oracles for online analyses (used to detect deadlocks). Any anomalies are reported in the *result file*.

To use SIMTESTER, a test engineer first configures Simics to model the system to be tested. The engineer then writes a configuration script to set up breakpoints and a variable watch list, and specific instructions to be monitored (e.g., procedure calls and return instructions). In addition, the engineer also writes a script to specify actions to be taken when monitored events occur (Simics refers to these actions as *handlers*.) As an example, a handler for reaching an execution breakpoint could fire a timer interrupt. Another possible handler for writing to a monitored address could make the written data available for logging or on-line analysis. Any generated information is processed by the execution observer.

In addition to the components illustrated in Figure 3.1, a test driver is also needed to automate the testing process. Typically, engineers conduct testing by running test programs on a system. A test driver is a program that automates the process of running test programs in a suite and managing the generated log files.

After a test driver executes a test case an event log file is generated. Log files can then be analyzed to detect anomalies, or saved for further off-line analysis. In this chapter, we consider both output-based oracles and internal oracles.

3.2 Utilizing SIMTESTER

In this section, we describe and illustrate how SIMTESTER can be used to test for two significant classes of concurrency faults: data races and deadlocks. These two classes of faults have been identified as the most “nasty” faults to test for in embedded software [101]. When we conduct testing for data races, the components under test include the main ap-

plication, the UART device driver, and the ISRs that are associated with specified serial ports. The focus of our illustration is testing for races that occur when the application coupled with the device driver interact with an ISR.

Note that in the illustration we present, interrupts are not nested, but our algorithms do also support nested interrupts. Also note that our illustration considers only a single ISR but our algorithm can be generically applied to handle multiple ISRs; however, in that case it is more difficult to isolate events related to any one particular ISR. In addition, because SIMTESTER forces interrupts to occur, we would like to distinguish between interrupts that occur naturally as part of program execution and forced interrupts issued by the execution control module of SIMTESTER. We refer to the former type of interrupts as *self-generated*, and to the latter as *controlled*.

3.2.1 Data Race Detection

In early releases of the 2.6 Linux kernel, there is a particular data race that occurs between the routine `serial8250_start_tx` and the UART ISR `serial8250_interrupt` in UART driver program [102]. This fault remained in the source code for three years before being fixed. We provide code snippets that illustrate the fault in Figure 3.2.

Routine `serial8250_startup` is responsible for testing and initializing the UART port, and assigning the ISR. This routine is called before the UART port is ready to transmit or receive data. Routine `serial8250_start_tx` is used to initialize data transmission, and is called when data is ready to transmit via a UART port. Routine `serial8250_interrupt` is the actual ISR, and is called by satisfying two conditions in terms of data transmission: (1) the data is ready to transmit; and (2) the interrupt enable register (IER) is enabled by the `serial8250_start_tx` routine.

```
static int serial8250_startup(struct uart_port *port){
1.      up = (struct uart_8250_port *)port;
2.      ...
        /* Do a quick test to see if we receive an
        * interrupt when we enable the TX irq.
        */
3.      serial_outp(up, UART_IER, UART_IER_THRI);
4.      lsr = serial_in(up, UART_LSR);
5.      iir = serial_in(up, UART_IIR);
6.      serial_outp(up, UART_IER, 0);
7.      if (lsr & UART_LSR_TEMT && iir & UART_IIR_NO_INT) {
8.      if (!(up->bugs & UART_BUG_TXEN)) {
9.              up->bugs |= UART_BUG_TXEN;
10.             }
11.     }
12.     ...
        /* Finally, enable interrupts. */
13      up->ier = UART_IER_RLSI | UART_IER_RDI;
14.     serial_outp(up, UART_IER, up->ier);
15.     ...
16.}

static void serial8250_start_tx(...){
17.     serial_out(up, UART_IER, up->ier);
        ...
18.     if (up->bugs & UART_BUG_TXEN) {
19.             ...
20.             transmit_chars(up);
21.     }
22.}

static irqreturn_t serial8250_interrupt(...){
23.     ...
24.     transmit_chars(up);
25.     ...
26.}

static void transmit_chars(...){
27.     struct circ_buf *xmit = &up->port.info->xmit;
28.     ...
29.     xmit->tail = (xmit->tail + 1) & (UART_XMIT_SIZE - 1);
30.     ...
31.}
```

Figure 3.2: Faulty code that can cause data races in the UART driver in Linux.

Under normal operating conditions, the ISR is always responsible for transmitting data. To ensure that an ISR is assigned correctly, the driver issues an interrupt and monitors the response from the ISR (lines 3-6 in Figure 3.2). Several sources have shown that problems such as races with other processors on the system or intermittent port problems can cause the response from the ISR to get lost or cause a failure to correctly install the ISR, respectively. When that happens, the port is registered as buggy (line 9) and workaround code based on polling instead of interrupts is used (lines 18-21). Unfortunately, the enabled interrupts (lines 13-14) are not disabled in the workaround code region so by the time the workaround code is executed, it is possible for both the ISR and the workaround code to be transmitting or receiving data through the same serial port at the same time by calling the routine `transmit_chars`, and simultaneously read/write the shared variable `xmit→tail` (line 29). As such, a race in this illustration occurs when:

1. the device driver program is preempted by the ISR after a shared memory access before it can proceed to the next instruction;
2. the ISR manipulates the content of this shared memory.

Higashi et al. [53] introduce an approach to test for this fault by controlling invocations of interrupts. In that work, they used an ARM-based processor simulator and modified version of *uCLinux* with the same fault so that it could run on that simulator. Their modifications included porting the code over from PPC to ARM and removal of irrelevant code to reduce the simulation time. Their methodology was to invoke an interrupt at every memory read and write operation.

For this illustration, we replicated the fault in the example of Figure 3.2 in Fedora Core 2.6.15. We also recreated a similar testing system based on SIMTESTER with two additional optimization techniques. In the first optimization technique, we apply static program analysis to detect the resources that can be affected by the UART driver and the ISR. With

this optimization, we invoke interrupts only when these shared resources are accessed. Second, we also check system states to ensure that it is possible to invoke interrupts when those resources are accessed. These two optimizations should significantly reduce the time required to conduct testing. Next, we discuss the configuration of SimTester that allows engineers to test for this fault.

Configuration of the Test System

To test for data races we need to provide two components to the system under test: the test input and conditions governing when to invoke interrupts from within the system. In this case, a *test case* is used as the test input for the program under test (*PuT*) (which in this case includes the application and any device driver running under non-interrupt service routine context that is called by the application.) In the case we are considering, the *PuT* includes an application that interacts with a serial port and the UART driver. We refer to the interrupt service routine for the tested UART port as the ISR. Note that test cases for the *PuT* can be generated based on various criteria. We discuss the criteria we use in the next section.

Next, we need to describe each *interrupt condition* (*IC*). We express an IC as a tuple: $< loc, pin >$. The first element, loc, specifies a code location at which to invoke an interrupt. The second element, pin, specifies an *Interrupt Request* (*IRQ*) line number at which to invoke the interrupt. This is needed because typically, an interrupt service routine can be associated with multiple IRQ lines. ICs are used only when the controllability module is enabled.

Observer Module. We configure the observer module so that it records runtime information that can be used by test engineers to perform off-line analysis for races. In this example, the generated information includes:

- when functions of the *PuT* and ISR execute and when they return, and
- when shared variables are accessed by the *PuT* and written by the ISR.

As such, one of our configuration tasks is to set execution breakpoints in Simics to detect when functions in the *PuT* and ISR execute and when shared variables are accessed. The algorithms to accomplish this task are provided next.

---

procedure *BasicConfig*
1: **begin**
2: **for** each function f in the *PuT* and ISR
3: set execution `breakpoint` at entry point f_{entry} of f
4: **endfor**
5: set execution `breakpoint` on function return instruction ret
6: set execution `breakpoint` on interrupt return instruction $iretd$
7: **end**

procedure *RaceObserver*
require: procedure *BasicConfig*
8: **begin**
9: **for** each SV in list l_{SV}
10: set memory read/write `breakpoint` at SV
11: **endfor**
12: **switch** (`breakpoint`)
13: **case** f_{entry}:
14: f_{list}.`push`($\{f_{entry}$, *ebp*, *esp*$\}$)
15: **if** f_{entry} == ISR_{entry}
16: log "ISR entry"
17: $isISR$ = `true`
18: **endif**
19: **case** ret:
20: **if** *esp* == f_{list}.`top`(*esp*)
21: **if** f_{list}.`top`(*entry*) == ISR_{entry}
22: $isISR$ = `false`
23: log "ISR exit"
24: **endif**
25: f_{list}.`pop()`
26: **endif**
27: **case** $iretd$:
28: **if** *ebp* == ebp_{switch}

```
29:            log "IRETD"
31:            log program counter in next instruction PC_next
32:        endif
33:    case SV:
34:        if ebp == f_list.top(ebp) /*check if SV is accessed by the PuT*/
35:            if isISR == false
36:                log "PuT, SV,SV_access, PC"
37:                ebp_switch = ebp /*save Reg[ebp] content*/
38:            else /*interrupt handler context*/
39:                if SV_access == write
40:                    log "ISR, SV, SV_access"
41:                endif
42:            endif
43:        endif
44: end
```

Execution memory breakpoints are set in *BasicConfig*, including function entry addresses f_{entry} (line 3), function return instructions ret in the ***PuT*** and ISR (line 5), and interrupt return instructions $iretd$ (line 6). Races occur when both the ***PuT*** and ISR access the same memory location, so a memory breakpoint for each shared variable (SV) address is set in callback function *RaceObserver*, which is invoked whenever the execution reaches a breakpoint.

One challenge in setting up a breakpoint for each shared variable is that we need to be able to obtain the dynamic address of that shared variable. One option for doing this is to parse the symbol table. However, the symbol table provides only global variable addresses so our system may miss other shared variables such as local pointers. To obtain the address of each shared variable that has been identified by our static analysis, we first create an instrumented version of the ISR so that it dynamically prints each shared variable address. We then iteratively inject data into the device port or adjust the device's register so that we can obtain the addresses of all shared variables in the ISR stored in l_{SV}. Our function *RaceObserver* sets breakpoints at these shared variable addresses in line 3 of *RaceObserver*.

To isolate the ***PuT*** and ISR from other applications in the system or other ISR invocations, we statically identify all function names in the ***PuT*** and their entry addresses. We also identify the entry address to the ISR. These addresses can be obtained by parsing the symbol tables. Furthermore, we monitor the function return instruction (ret in X86) to determine whether a function or the ISR has returned, and the interrupt return instruction ($iretd$ in X86) to determine whether the ***PuT*** has been recovered from the interrupt context.

At runtime, we keep a *calling stack* named f_{list}. When a function or an ISR from ***PuT*** is invoked its <address, frame pointer, stack pointer> is added to *calling stack* (line 14). When a shared variable is accessed, we compare the current frame pointer ebp with the frame pointer on top of *calling stack* (line 34). This mechanism allows us to ignore those shared variables that might be accessed by a different ISR or a different program on the same ISR. If a ret instruction is encountered, by comparing the current stack pointer esp with the stack pointer on top of *calling stack* (line 20), we can determine whether the current function or the ISR has returned.

A function is popped from *calling stack* if its ret instruction is reached. Program counter PC is recorded twice to determine whether the ***PuT*** is actually preempted between a shared variable access and its following instruction. The first time is when a shared variable is accessed (line 36) under a non-interrupt service routine context, and the second time is after an interrupt returns (line 31). An interrupt return instruction $iretd$ is recorded to indicate termination of an interrupt context. Note that the mere presence of an $iretd$ does not imply that an interrupt will jump back to the ***PuT***, because more than one device can issue interrupts and call $iretd$ instructions. To overcome this problem, an $iretd$ is logged only when its frame pointer is equal to the frame pointer when a shared variable is accessed in the ***PuT*** (line 28).

In summary, events logged for testing race conditions include: (1) read/write accesses to shared variables (SV_{access} by the ***PuT***); (2) entry to the ISR; (3) a write to an SV by the

ISR; (4) return from the ISR; (5) context switches from the ISR to the *PuT*. Figure 3.3 illustrates a sample of trace information recording these events for this example.

```
...
PuT, $xmit->tail$, read, pc1
ISR entry
ISR, $xmit->tail$, write
ISR exit
IRETD
pc1+1
...
```

Figure 3.3: Sample trace information for race detection.

Note that there is a race in the trace given in Figure 3.3. By observing the program counter when SV is accessed by the PuT and the interrupt recovery point, we can determine that an interrupt occurs right after `xmit→tail` is read by the *PuT*.

Controller Module. When engineers enable the controller module *RaceController*, a *controlled* interrupt is invoked right after a shared variable access by the *PuT*. Simics allows us to issue an interrupt on a specific IRQ line from the simulator itself. The interrupt will happen before the subsequent instruction. As such, when our test system reaches a memory breakpoint, the observer module is called. If the controller module is enabled, the observer module tries to invoke an interrupt right after the access to shared variable.

It is not always realistic, however, to invoke an interrupt whenever we want. For example, the interrupt enable register and possibly other control registers have to be set to enable interrupts. In the example of Figure 3.2, before invoking an interrupt, the interrupt enable register IER of the UART must be set while the interrupt identification register IIR must be cleared. Even if interrupts are enabled, they can be temporarily disabled. Figure 3.4 is the routine in *RaceController* used to determine whether it is possible to issue an interrupt.

---

procedure *ISR_enabled(int p)*

/*p is the pin number for a certain interrupt*/

45: **begin**

46: **if** $eflags$[9] != 0 and `ioapic.redirection[p]` == 0 and `ioapic.pin_raised[p]` == LOW:

47: *return* `true`

48: **else**

49: *return* `false`

50: **endif**

51: **end**

Figure 3.4: Algorithm to determine whether it is possible to issue an interrupt.

There are two general steps that our system takes prior to invoking a *controlled interrupt*. First, the controller module checks the status of the local interrupt and global interrupt bits to see if interrupts are enabled. In an X86 architecture the global interrupt bit is the ninth bit of the eflags register (line 46 in Figure 3.4). When this bit is set to 1, the global interrupt is disabled, otherwise it is enabled. For local interrupts, Simics uses the Advanced Programmable Interrupt Controller (APIC) as its interrupt controller. As such, our system checks whether the bit controlling the UART device is masked or not. Our system also checks whether a *self-generated* interrupt is about to be issued by examining the current pin status. If this is true, the *controlled* interrupt will not be invoked.

Second, our system invokes only one *controlled* interrupt per test run. This is done to avoid *fault masking* effects, which may occur in cases where multiple interrupts fire and cause a failure that would be evident in the presence of a single interrupt to be “masked” by the presence of the second. Thus, our system needs to first check a flag to determine whether a controlled interrupt has already been invoked in the current run. If it has, the test system does not monitor any further events in this run. Once it has been determined that there has not been any invocation of a controlled interrupt in this run, the system then checks to see whether the last accessed shared variable has already been tested in prior

runs. If it has not, the system enables the control register for the UART and then invokes an interrupt.

Note that, given the foregoing approach, there can be multiple runs of each test case, and the number of runs depends on the number of shared variables that must be tested. With the controller module disabled, the ***PuT*** runs $|tc|$ times during a testing process, where $|tc|$ is the number of test cases. With the controller module enabled, the number of runs is $|tc| * (|int| + 1)$, where $|tc|$ is the number of test cases and $|int|$ is the number of controlled interrupts issued. We also need to run the ***PuT*** one additional time for each test case to determine whether all shared variables have been accessed.

3.2.2 Deadlock Detection

It is common for interrupt handlers to use non-preemptive mechanisms such as spin-locks instead of preemptive mechanisms such as semaphores to protect critical code regions. This is because typically, ISR code is not allowed to sleep. As such, interrupt code is short and deterministic so the amount of time that a task must wait to enter the critical region should be predictably short [103]. That said, there are many real world examples including the one used in this illustration (Figure 3.5) that show how incorrect usage of spinlocking mechanisms can cause priority inversions and ultimately deadlocks between the ***PuT*** and ISR. In the example of Figure 3.5, deadlocks occur because:

- the ***PuT*** is preempted by the ISR while the ***PuT*** is holding a spin lock,
- the ISR tries to acquire the same spinlock and becomes stuck in the spinlock loop.

Because the ISR has higher priority, it can become stuck in this loop. To test for such deadlocks, we configure our test system in a manner similar to that used to test for races. However, test cases for the system are generated to adequately cover `spin_lock`

```
void xhci_watchdog(...)
1.{
2.        xhci = ep->xhci;
3.        ...
4.        spin_lock(&xhci->lock);
5.        ...
6.        spin_unlock(&xhci->lock);
7.}

irqreturn_t interrupt(int irq, void *dev_id)
8.{
9.        xhci = get_dev(dev_id);
10.       ...
11.       spin_lock(&xhci->lock);
12.       ...
13.}
```

Figure 3.5: Faulty code that can cause deadlocks.

and `spin_unlock` pairs in the *PuT* instead of shared variables. In this illustration, we assume that `spin_lock` and `spin_unlock` are properly paired. Next, we describe the configurations of the execution observer and execution controller.

Configuration of the Test System

In a uniprocessor environment, an observer does not increase fault detection effectiveness because deadlocks can be easily observed (e.g., systems hang or show no response). However, SIMTESTER can still be useful in informing testers of the sources of deadlocks. Furthermore, it can support deadlock detection in multiprocessor systems where deadlock might cause one or more processors to stop making execution progress but other processors continue to do useful work.

Observer Module. We configure the observer module so that it records runtime information that can be used to both issue a runtime warning and log for offline analysis. In this deadlock example, the generated information includes:

- when functions of the *PuT* and ISR execute and return;

- when a spinlock is acquired by the *PuT* and by the ISR, and
- when a spinlock is released by the *PuT*.

As in the example of Figure 3.5, when the spinlock is acquired by the the *PuT*, the observer sets the lock condition to true. If the ISR tries to obtain the same lock, the observer first checks whether the lock condition is true, and then compares the requested lock with that held by the *PuT*. If they are the same, a deadlock has occurred. In this case, the observer issues a warning once the deadlock is detected. In both environments, it also records the event so that engineers can perform analysis offline. Function *DeadlockObserver* describes our algorithm to detect deadlocks.

---

procedure *DeadlockObserver*
require: procedure *BasicConfig*
52: **begin**
53: set execution `breakpoint` on entry point of $spin_lock$ and $spin_unlock$
54: **switch** (`breakpoint`)
55: **case** f_{entry}:
56: f_{list}.`push`($\{f_{entry}, ebp, esp, lock_{obj}\}$)
57: **if** f_{entry} == ISR_{entry}
58: $isISR$ = `true`
59: **endif**
60: **case** $spin_lock$:
61: **if** ebp == f_{list}.`top`(ebp)
62: **if** $isISR$ and $isLock$ and eax == $lock_{obj}$
63: print"deadlock occurs"
64: **else**
65: f_{list}.`push`($\{f_{entry}, ebp, esp, eax\}$)
66: **endif**
67: **endif**
68: **case** $spin_unlock$:
69: **if** ebp == f_{list}.`top`(ebp)
70: f_{list}.`push`($\{f_{entry}, ebp, esp, eax\}$)
71: **endif**
72: **case** *ret*:
73: **if** esp == f_{list}.`top`(esp)
74: **if** f_{list}.`top`($entry$) == ISR_{entry}

```
75:                 isISR = false
76:             endif
77:             if f_list.top(entry) == spin_lock /*spinlock returns*/
78:                 if !isISR
79:                     isLock = true
80:                     lock_obj = f_list.top(eax)
81:                 endif
82:             endif
83:             if f_list.top(entry) == spin_unlock
84:                 if !isISR
85:                     isLock = false
86:                     lock_obj = null
87:                 endif
88:             endif
89:             f_list.pop()
90:         endif
91: end
```

Functions `spin_lock` and `spin_unlock` are commonly used by various applications. As such, we need to isolate calls to these two functions that come from the *PuT* and ISR. Again, we use the *calling stack* to dynamically store information on called functions during virtualization (lines 56, 65, 70).

Initially, $lock_{obj}$ is set to `null`. The lock is acquired after `spin_lock` returns (line 79), and released after `spin_unlock` returns (line 85). Besides the frame pointer and stack pointer, each function in *calling stack* carries a lock object, indicating whether the current running function is holding a lock or which lock it is holding. As such, a locked object can always be obtained by examining the top of the stack (line 80).

A deadlock occurs under three conditions (line 62): (1) the ISR is executing; (2) a lock is held by the *PuT*; (3) the ISR is stuck in the same spinlock loop as the object of the held lock.

Deadlock detection is different from race detection in that a deadlock warning is issued instead of simply recording deadlock information. Once a deadlock is detected, the test

script terminates execution and reinitializes the test system for the next test run. Because a lock object is passed as a parameter to `spin_lock` or `spin_unlock`, the object can be obtained by reading the CPU eax register in the X86 architecture. Note that this is architecture and compiler dependent. However, the approach should be generalizable to other architectures and compilers as long as we know how the lock object is passed.

Controller Module. The controller module *DeadlockController* is implemented following the same steps as the controllability module for race conditions, except that an interrupt is issued after a spinlock is acquired by the *PuT* instead of invoking interrupts on each shared variable access.

3.3 Empirical Study

To evaluate SIMTESTER, we consider two research questions.

RQ1: How does the effectiveness of SIMTESTER at detecting data races compare to that of other techniques?

RQ2: How does the effectiveness of SIMTESTER at detecting deadlock compare to that of other techniques?

The two research questions let us investigate whether the use of controllability and observability can affect the effectiveness of SIMTESTER at detecting data races and deadlock, respectively.

We applied our approach to the UART device driver on a preemptive kernel version of Fedora Core 2.6.15. The driver includes two files, serial_core.c and 8250.c, containing 1896 and 1445 lines of non-comment code, respectively. The main application transmits character strings to and receives character strings from the console via the UART port. Note that in this work we apply our testing process only to the UART driver. However, the same process is also applicable to other types of device drivers.

Our approach requires the use of existing test cases, so we generated test cases for the system based on a code-coverage-based test adequacy criterion. To generate test cases relevant to race conditions, we first statically identified shared variables between the PuT and ISR. We use the precise shared variable detection algorithm proposed in [104], but we are interested only in shared variables that are read by the *PuT* and written by the ISR, or written by both the *PuT* and the ISR. We labeled each shared variable as a "definition" or "use" through our analysis. After shared variables were identified, we generated a set of test cases that cover the feasible shared variables (shared variables for which there exists a possible execution of the program which executes them) in the *PuT*. This process produced 12 test cases.

To generate test cases relevant to deadlocks, we sought to find test cases that adequately cover `spin_lock` and `spin_unlock` pairs in the PuT instead of shared variables. (In the case of our target program, we know that all occurrences of `spin_lock` and `spin_unlock` are properly paired; in practice a static analysis could initially determine this and flag unpaired occurrences for attention by the test engineer.) In the case of our target program, which contains only three spin-lock pairs, this process resulted in two test cases (one of which covered two of the pairs).

To better assess the cost and effectiveness of our approach, we considered both the approach and two alternative baseline approaches. In the discussion that follows, we refer to our approach as the *conditional controllability* approach, because it involves issuing controlled interrupts under certain conditions. The second approach that we considered, *no controllability*, involves testing the program without any controlled interrupts; this is the approach that test engineers would normally use. The third approach that we consider, *random controllability*, involves issuing controlled interrupts at random program locations after shared variable accesses and without checking interrupt conditions.

We measured execution times for the foregoing approaches by embedding a timer in the Simics module. As such, the reported times are the actual times spent by Simics to execute the program.

3.3.1 Results and Analysis

3.3.1.1 RQ1: Testing for Race Conditions

We begin by considering the target program as given, and evaluate the effectiveness and efficiency of our race condition testing approach on that program.

We first applied conditional controllability together with observability. Under this approach, across the 12 test cases utilized, 84 controlled interrupts were applied, and for each test case, one extra run was needed to determine whether all shared variables had been accessed. Thus, 96 test runs were required to finish testing the target program with an average execution time of 77.91s per test run. Including self-generated interrupts, the number of interrupts generated for the target program was 352. In the course of applying the approach, we detected a race in function `uart_write_room` of `serial_core.c`, which we later determined had been corrected in subsequent versions of the system. By running the system with observability turned off, we determined that this fault can be detected only with observability enabled; in other words, it is a fault that did not propagate to output on our particular test inputs.

We next tested our target program with no controllability. In this case, the only interrupts that occur are self-generated interrupts. Because runs of each test case can conceivably differ, we ran each test case on the program 500 times. The total number of interrupts observed was 16,500. Over the 6000 total test runs, average execution time was 74.08s per test, only 3.83 seconds less than with controllability added. None of these test runs de-

tected the race condition detected by our first approach, however, either with observability enabled or disabled.

Finally, we tested our target program using the random controllability approach. For each test case, we ran the target program three times more than the number of runs performed under the conditional controllability approach, on each run generating an interrupt at a randomly selected program location. The total number of test runs was 288 and the number of interrupts generated was 1044. In this case the average execution time per test case was 75.17s, only 2.74 seconds less than with controllability added. Again, the race was not detected, either with observability enabled or disabled.

One important characteristic of our technique is that checking is performed before issuing a controlled interrupt. When a shared variable is accessed in the main program, the controllability module first checks to see whether it is possible to issue an interrupt, and if not, it proceeds to the next possible location. This approach can save test runs, but at the cost of checking. To quantify the tradeoffs involved, we also applied our conditional controllability approach without the checking step enabled. Recall that with checking enabled, 96 test runs were needed to issue controlled interrupts, with an average execution time of 77.91s per test. With checking disabled, on the other hand, 1428 test runs were needed to issue controlled interrupts, with an average execution time of 75.66s per test run. Clearly, the checking approach saves time overall.

A second characteristic of our technique is that interrupts are issued only after shared memory accesses, and this can be much less expensive than issuing interrupts after each memory access, which is the approach used by Higashi et al. [53]. For our target program, there are 94,941 data accesses made in the course of running the 12 test cases. If an interrupt were issued after each data access, we would need 82.6 days to finish testing the target program.

3.3.1.2 RQ2: Testing for Deadlock

We next considered the target program as given, to evaluate the effectiveness and efficiency of our deadlock testing approach on that program. (In this case, because our experiment is running on a uniprocessor and observability does not increase fault detection power, we consider only the effects of controllability, not the effects of observability.)

We began by running our two deadlock test cases on the target program under conditional controllability. Under this approach, only two controlled interrupts were generated, and they detected one deadlock in function `serial8250_handle_port` of `8250.c`, which had been reported [105] and corrected in later versions. The average execution time was 69.88s per test run.

We next ran the target program with no controllability. For each test case, we ran the program 500 times; the 1000 test runs had an average execution time of 75.24s, and 4000 self-generated interrupts were observed. However, no deadlocks were detected.

Finally, we ran the program using the random controllability approach. Here we ran the program three times as many as the number of runs performed for conditional controllability, issuing an interrupt at a random program location. The total number of controlled interrupts was six and the total number of interrupts generated was 15. The average run time was 75.24s per test. Here also, no deadlocks were detected.

Notice that the average execution times of the latter two approaches are higher than that of the conditional controllability approach. This result, at first, appears counter-intuitive. However, in our experiment with conditional controllability, we terminated the execution once a deadlock has been detected. On the other hand, the latter two approaches did not detect deadlocks, so the program ran to completion.

3.3.1.3 Effectiveness of Techniques at Detecting Seeded Faults

While the results of the foregoing studies are encouraging, the numbers of naturally occurring faults found in the target program was low, rendering comparisons of the fault detection effectiveness of the approaches less meaningful. To further investigate fault detection effectiveness we followed a process often utilized in the software testing research community [106]; namely, the use of seeded faults.

In this case, we injected 12 potential race condition faults and 11 potential deadlock faults into 8250.c by making syntactic changes to the code. For race conditions, we removed statements corresponding to critical section protection (e.g., `spin_lock`, `spin_lock_irq`). For deadlock, we changed statements corresponding to interrupt disable and enable pairs (e.g., `spin_lock_irq` and `spin_unlock_irq`) into pure spin lock pairs (e.g., `spin_lock`, `spin_unlock`). Of the 23 potential faults thus created, further examination revealed that seven of the potential race condition faults, and seven of the potential deadlock faults, could not possibly be triggered on the system on its given hardware platform, so we removed those. This left us with five potentially revealable race condition faults and four potentially revealable deadlock faults.

Given the faults thus seeded, we ran our test cases on the faulty systems using conditional and random controllability, and in the case of race detection, with observability enabled and disabled. For the race condition detection approach, conditional controllability detected two of the five faults. One of these faults was detected both with and without observability. The same fault was also detected with random controllability, but only with observability enabled because in this case the fault does not propagate to output. This occurred because interrupts issued by conditional controllability visited more unprotected shared variables that can cause incorrect output, and these shared variables are not visited by random controllability.

The second fault revealed in our race detection trial was revealed not through observability, but rather, through output, for both conditional controllability and random controllability. The reason this occurred is because the fault was not actually caused by our defined race condition, but rather, by another type of atomicity violation. In particular, a read-write shared variable pair in the main program is supposed to be atomic, but the ISR read this shared variable before it was updated in the main program. This outcome shows that, while our approach does not specifically target other types of faults, it may catch them as by-products.

We also inspected the three potential race condition faults that were not detected by any techniques. We determined that the reason for their omission was that the interrupt handler in each of the versions does not share variables or read variables with the main program. This does not mean that the code regions involved do not need to be protected, because other ISRs may share memory locations, or programmers may intentionally cause the regions to execute without interruption.

Where deadlock faults were concerned, we discovered that conditional controllability detected all four, while random controllability detected two.

3.4 Further Discussion

Our observer module considers one type of definition of a race condition. In practice, testers can adopt different definitions because there is not a single general definition for the class of race conditions that occur between an ISR and a *PuT*. As noted above, for the four faulty versions on which the ISR and the *PuT* do not share read-write and write-write variables, we still found one fault with controllability enabled. This fault is related to an atomicity violation, because a code region in the main program is supposed to execute

atomically, e.g., before a shared variable is updated in the main program, and an interrupt occurs and the wrong data is read.

Our approach injects data into device ports and forces an interrupt handler to execute one path. The data we inject is the same as the test input given to the program. For example, if an application sends the string "hello" to the UART console passed by UART *transmitter buffer*, a controllability module would inject "world" into the UART *transmitter buffer* to force an interrupt to occur after a certain access. It is also possible to have multiple paths by which shared variables can exist in interrupt handlers. Testers can extend our method by forcing interrupt handlers to execute different paths, which may increase the probability of revealing faults.

However, no faults are left undetected due to missing shared variables or spin locks in the other paths of the ISR in our target program. It is also possible to force an interrupt handler to execute only the paths that have definition-use relationships with the main program. This may further reduce the number of controlled interrupts and test runs. To do this, the value schedule approach proposed by Chen et al. [52] could be adapted.

To force an interrupt to occur, our controllability module issues a new interrupt. However, races and deadlocks can occur relative to the interrupt generated by the target program itself. For example, suppose an interrupt is requested by device driver code, but is not immediately processed for some reason (e.g., device port delay). The interrupt handler associated with this interrupt may be executed later within a spin lock pair or after a shared variable access, and thus a race condition or a deadlock may occur. Our controllability module can be further extended to deal with such cases. For example, when an interrupt is triggered, the module can delay this interrupt by masking its interrupt enable register, and issue the interrupt after a certain event happens (e.g., shared variable access, spin lock is acquired). If there is no such event, the interrupt is issued on exiting the *PuT*.

In practice, when testing software components (e.g., device drivers and interrupt handlers), the first task that a test engineer must accomplish is to gain confidence that the software component is developed correctly. In our study, the analysis involves a test program, the interrupt handler that interacts with the device driver, and the device driver code. The key point here is that the tester focuses on a specific component and how it interacts with the rest of the components. If the focus changes to a different component, the same analysis can be applied to test the new component. As such, the proposed approach is not designed to test the entire system at once. Instead, it is more suitable for component testing.

In our current work, the test generation process was done manually (which is currently the norm in practice). Our study considers a test input to include input values and interrupt scheduling. However, there is no reason the approach could not also utilize input values created using existing test case generation approaches (such as dynamic symbolic execution [107, 108].) A problem with such approaches by themselves is that they generate large numbers of test cases with no methodology for judging system correctness beyond looking for crashes. Our approach provides more powerful, automated oracles, and thus should ultimately facilitate the use of larger numbers of automatically generated test cases.

As noted in Chapter 2.2, there are several techniques for testing embedded systems with a particular focus on interrupt-level concurrency faults. For example, Higashi et al. [53] detect race conditions caused by interrupt handlers via a mechanism that causes interrupts to occur at all possible times. Their method and ours artificially amplify the frequency of interrupts to evaluate whether these interrupts can cause faults. However, our work has several advantages. First, by adding observability, we increase the power of fault detection while Higashi's method focuses on controllability. Second, instead of firing interrupts at all memory access instructions of an application, we issue interrupts only at shared variable accesses, which significantly reduces testing cost. Third, we adopt coverage criteria to cover all feasible shared variables in the application instead of using arbitrary inputs;

this can help the program execute code regions that are more race-prone. Fourth, Higashi's method issues all interrupts during one program run. This may cause problems with fault masking and cascading errors. In contrast, we issue just one interrupt during a given program run. Fifth, Higashi's method does not consider situations in which interrupts cannot occur; however, our technique can determine whether it is possible to issue interrupts at runtime by observing hardware states, which further improves testing efficiency. Another implementation difference is that we built SIMTESTER on a virtual platform while their approach is built on an processor emulator.

3.5 Conclusion

The frequent use of interrupts for timing, sensing, and I/O processing in embedded software can cause concurrency faults to occur due to interactions between applications, device drivers, and interrupt handlers. This type of fault is considered by many practitioners to be among the most difficult to detect, isolate, and correct, in part because it can be sensitive to interleavings and often occurs without causing any observable incorrect output. In this chapter, we introduced SIMTESTER, an instantiation of the SIMEXPLORER framework, that provides test engineers with the ability to precisely *control* execution events and *observe* runtime context at critical code locations. SIMTESTER is built on a commercial virtual platform that is commonly used as part of the hardware/software co-design process.

Chapter 4

SimRacer: Automatic Testing for Process-Level Races [3]

In this chapter, we propose SIMRACER, another instantiation of the SIMEXPLORER framework, that allows engineers to effectively test for process-level races by providing deterministic execution in multiprogramming systems. Similar to SIMTESTER (Chapter 3), SIMRACER employs Simics to observe system execution, to deterministically control occurrences of events, and to allow testing to target specific types of races. SIMRACER allows engineers to focus only on applications under test without worrying about other events or applications in the system. Side effects from execution of applications not under test can be easily determined by querying current system states. By using a VM, SIMRACER also operates at the binary level so it is capable of detecting races between applications written in different languages. It is also generalized, as it can be used on any system that can run on Simics.

To achieve the foregoing benefits, SIMRACER works as follows:

[3] The contents of this chapter have appeared in [109].

1. *Employ dynamic analysis to identify potential races.* SIMRACER executes tests on the target processes and records synchronization and memory operations, hardware operations, and system calls that access shared resources. It derives happens-before relationships involving shared resource accesses between processes based on the recorded information, and computes sets of pairs of program instructions that could potentially race in a concurrent execution.

2. *Employ virtualization to achieve controllability.* Virtualization allows us to achieve controllability without modifying the kernel. This is because SIMRACER can stop execution at each specified program location and observe system information to determine whether events can occur. For example, it can query the status register to determine whether it is possible for signals to occur (e.g., check signal masking bits). If so, a signal can be raised to exercise different signal interleavings. Further, through a special device driver, SIMRACER can selectively execute processes that can participate in races. These capabilities provide the controllability needed to test for process-level races.

3. *Employ active testing to permute process interleavings.* SIMRACER executes the target processes following a random schedule. During execution, if the next instruction of a process is contained in a potential race pair computed in the recording phase, then the execution of this process is suspended until the other process executes an instruction in the race pair. Next, SIMRACER reorders the two racing events recorded in the original execution. If the current system states do not allow the reordering to occur, SIMRACER adjusts the executions of the processes until the reordering succeeds, which results in a real race.

4. *Detect damaging results.* SIMRACER allows engineers to observe test oracles for correctness. In the case of race detection, results that do not cause observable failures are

further examined. If there are no test inputs that can cause the race to expose an observable failure, the race is considered benign. Because we implemented SIMRACER on Simics, scripting mechanisms that can be used to validate the system outputs with provided oracles and generate error notifications are readily available.

To evaluate our approach we conducted an empirical study using 16 Unix programs that contain actual process-level races. Our goal was to answer two research questions: (1) how effective is SIMRACER at detecting process-level races, and (2) how effective is SIMRACER at detecting observable faults related to those races. The results of our study show that our approach is effective at revealing races, and far more effective than stress testing at revealing faults related to those races. Further, while our approach takes longer than stress testing to execute test runs due to virtualization overhead, it can reveal more races while using far fewer test runs.

We provide five real-world examples to illustrate how process-level races can occur. Figure 4.1 provides an instance of a process-level race modeled after a pattern of commonly occurring races in Linux systems [110, 111, 112]. In this example, a race occurs on a file, causing the recorded information to be incorrect. The parent process $P1$ creates a file "foo" and writes to it (lines 1-3). Next, $P1$ spawns a child process $P2$ using `fork`. While $P2$ is sleeping (line 6), $P1$ renames "foo" to "bar" and waits for $P2$ to exit (lines 16-17). After $P2$ wakes up, it writes a string into "bar" (lines 7-8). Next it checks for the existence of "foo" and writes a message into "bar" (lines 9-10). The expected output is a "bar" file containing the string "hello kitty" followed by the string "foo does not exist". However, races can occur if $P1$ is suspended while $P2$ is sleeping. In such a scenario, $P2$ creates a new file "bar", and writes "kitty" into it (lines 7-8). Next, $P1$ overwrites the contents of "bar" by renaming "foo" to "bar" (line 16). As a result, the output information in "bar" is just "hello".

```
1  fd_foo = fopen("foo", "w");
2  fwrite("hello ", 1, 256, fd_foo);
3  fclose(fd_foo);
4  childpid = fork();
5  if(childpid == 0) { /*child process*/
6          sleep(1);
7          fd_bar = fopen("bar", "a+");
8          fwrite("kitty\n", 1, 256, fd_bar);
9          if(stat("foo") != 0){
10             fwrite("foo does not exist\n",1,256,fd_bar);
11         }
12         fclose(fd_bar);
13         exit(retval);
14 }
15 else { /*parent process*/
16         rename("foo", "bar");
17         wait(&status);
18         exit(0);
19 }
```

Figure 4.1: Two processes race on a file

```
1  int files_created = 0; /*global variable*/
2  create_output_file() {
3       output_stream = fopen(output_filename, "w");
4       /*signaled*/
5       files_created++;
6  }
7  signal() {
8       for (i = 0; i < files_created; i++) {
9               name = make_filename(i);
10              if (unlink(name))
11                      error(0, errno, "%s", name);
12      }
13 }
```

Figure 4.2: Process-level race due to a software signal

4.1 Motivation

Figure 4.2 provides an example of a race between a regular process and a software signal handler. This race occurs in a program, CSPLIT, considered in our study. Under normal execution, the process calls `create_output_file` to create a set of files (line 3), and increments the counter (line 5). If a signal occurs, the process iterates over the files that

have been created (lines 8-9) and deletes them (line 10). However, if a signal occurs just before the counter is incremented (line 4), the files will not be deleted by the handler.

Figure 4.3 provides an example of a deadlock due to a data race in Apache 2.0.49 [113] that is detectable by our technique. It involves the CGI process and the HTTPD process that handles the CGI request. Apache redirects CGI's stderr and stdout outputs into two pipes. Under normal execution, the HTTPD process reads data after the CGI process writes stderr and stdout into the two pipes. However, a deadlock occurs when line 2 happens before line 8. The HTTPD process is blocked, waiting for the CGI process to write data to pipe2, and the CGI process is blocked, waiting for the HTTPD process to read data from pipe1.

```
HTTPD{                              CGI{
1. ...                              6. write(pipe1, stderr);
2. read(pipe2, stdout);             7. ...
3. ...                              8. write(pipe2, stdout);
4. read(pipe1, stderr);             9. ...
5. }                                10. }
```

Figure 4.3: A deadlock in the Apache Web-server

Figure 4.4 provides an example of a data race in an Android application [114]. The `init` process watches for a new device (line 1) and sets the permission on it (line 3). At the same time, if the server process tries to open this device, this results in an error because the server does not yet have access permission for this device.

```
init{                               server{
1. open(dev);                       5. ...
2. ...                              6. open(dev);
3. set_permission(dev);             7. ...
4. }                                8. }
```

Figure 4.4: A process-level race in an Android application

```
backup_timer{                          start_tx{
1. write_reg(IER);                     8. ...
2. ...                                 9. write_reg(IER);
3. if(isEnabled(IER)==TRUE);           10. /*transmit by ISR*/
4. /*transmit by ISR*/                 11. ...
5. else                                12. }
6. /*transmit by polling*/
7. }
```

Figure 4.5: A process-level race in a UART device driver

Figure 4.5 provides an example of a process-level race in a UART device driver that runs in kernel mode [115]. This race occurs on multi-core systems and was detectable by our approach. When the UART port is buggy (the interrupt does not work initially), the driver calls a backup timer periodically. The backup timer first disables interrupts by setting the interrupt enable register (IER) (line 1) and later checks the IER to see if it has been enabled. If it has, the timer uses an interrupt handler to transmit data (line 4); otherwise, it uses polling to slowly transmit the data (line 6). If two CPU cores try to transmit data simultaneously (i.e., calling $start_tx$), races can occur on the IER (line 1 races with line 9), causing both the backup timer and transmission routines to simultaneously transmit data through the same serial port.

Generalizing from these examples, *a process-level race 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, which is based only on synchronization operations [4, 6]. We also consider order violations; that is, cases in which the desired order of shared resource accesses is reversed and the accesses may or may not be protected by a common lock [22, 116]. An order violation is a necessary but not sufficient condition for a typical race.

4.2 Approach

SIMRACER, which is implemented on Simics, is designed to support testing for races between processes (running in both user and kernel modes) and between processes and signal handlers. We consider three classes of processes which include: (1) regular user processes (created through `fork` and `execv`), (2) software signals and handlers (software interrupts used for event notifications), and (3) kernel threads that run in the kernel mode so they can access kernel functions and data structures. The scheduling of processes is controlled by the kernel process scheduler. When a software signal is sent to a process, a signal handler may be entered. The servicing of the signal may or may not be immediate depending on the system state and the current disposition of the signal.

SIMRACER operates in two phases. The first phase analyzes the system under test for resources that can potentially race under normal execution. The result of this phase is an event trace that is used in the second phase to test for races. The second phase (the primary contribution of this work) permutes process interleavings to force races to occur at potential racing points obtained in the first phase. SIMRACER outputs both races and observable results.

SIMRACER has two major components: *Execution Observer* and *Execution Controller* (Figure 4.6). The Observer monitors the system under test in the first phase, and generates runtime information that is used to analyze the system for potential sources of races. The Observer also monitors runtime events to direct the scheduling effort in the second phase. We configured the Observer so that it sets memory breakpoints on shared resource operations enacted by each *process under test* (*PUT*). To distinguish events invoked by a *PUT* from other events, we keep a call stack for each *PUT* at runtime. When a function in a PUT is invoked, its frame pointer is added to the call stack. When a shared resource is accessed, we compare the current frame pointer with the frame pointer on top of the call stack. This

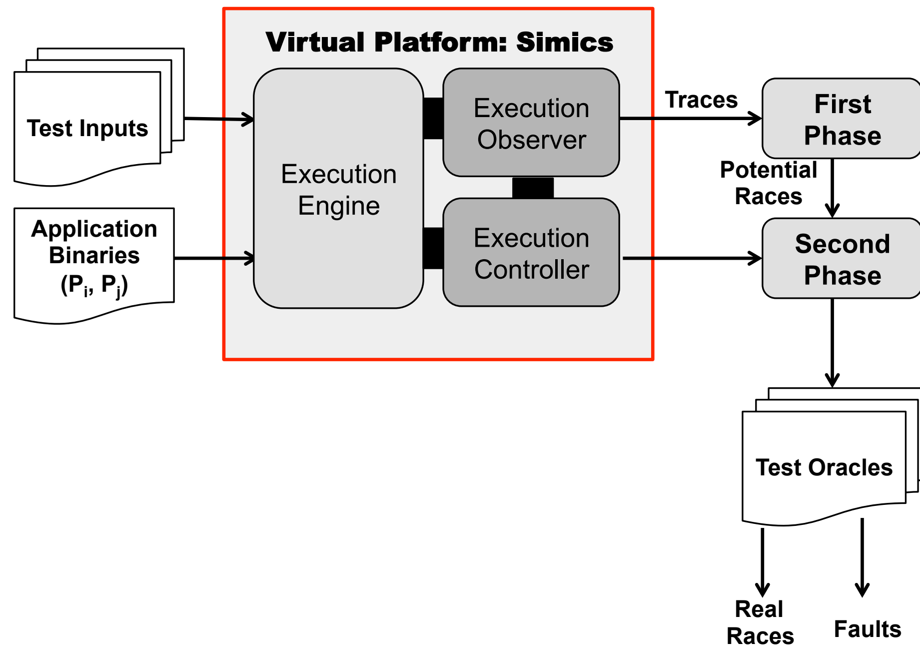

Figure 4.6: Overview of SimRacer

allows us to ignore shared resources that might be accessed by kernel threads or user-level processes that are not under test.

The Controller specifies events (e.g., process scheduling, signal handler invocations) to be invoked at particular points in execution. This is critical to supporting testing because it allows non-deterministic events to occur predictably. We implemented the Controller by installing a dummy device and its driver program. When an event such as a shared resource access occurs, the Observer notifies the Controller, which writes to the programmable interrupt controller to raise an interrupt for the dummy device. Next, the interrupt handler in the dummy device driver executes and checks the state of the target process using the $task_struct$ data structure to determine whether to suspend or resume a process or send out a software signal. Our mechanism can precisely force events to occur at particular binary instructions because hardware interrupts always have the highest priority of execution. In addition, our approach is applicable to any operating system that can run on Simics.

Because modern systems are multicore, we designed SIMRACER to be able to detect races that occur due to concurrent executions on different cores. For example, a device driver that is invoked simultaneously on two different cores can race if it manipulates the same hardware resource. To detect such races, we use a dummy device for each core. During testing, we use the Linux `sched_setaffinity` command to assign kernel threads to two cores. Instead of suspending the kernel thread when necessary, SIMRACER makes the thread spin on its core; this is necessary because the interrupt handler cannot call `sleep` functions.

4.2.1 Recording Events

The Observer component of SIMRACER records operations that access the following shared resources (SRs): shared memory, file descriptors (file handles of opened files), inodes (metadata about files, such as time and access permission), file contents, process data (process file table, data in $/proc$, e.g., PID), and hardware components (e.g., registers). Many of these SRs are also used by Laadan et al. [26] to perform process-level race detection; however, we also consider hardware components that are accessible by applications and device drivers. This lets our approach detect more types of races. Note that SRs need not be limited to the aforementioned items. In this work, however, we consider only those items that are likely to produce harmful results. For example, we do not track accesses to standard I/O, because races related to these are likely to be benign (when running multiple processes, outputs are expected to be non-deterministic). In addition, we consider only SRs that are directly accessible by application developers. For example, we do not track kernel process data. However, SIMRACER provides a user interface that lets developers specify other SRs.

After identifying SRs, the Observer tracks operations that access them in the system under test. We denote these operations as `OP`. The system stores recorded information as a log file for future use. When engineers wish to identify potential sources of races involving a pair of processes, they can provide the traces of the processes as inputs to our Phase 1 shared resource analyzer.

We categorize `OP`s as *reads* or *writes*. For `OP`s related to memory and hardware accesses, SIMRACER automatically identifies *reads* and *writes* using features provided by Simics. For `OP`s related to system calls on shared objects, SIMRACER models their effects on *reads* and *writes*. SIMRACER tracks 23 common system calls (e.g., `open`, `read`, `write`, `lstat`, `clone`, `wait`, `execve`, and `exit`); these involve *reads* and *writes* on the defined 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.

The `clone` system call creates a new process inode under the $/proc$ directory (write). The `wait` system call changes the state of the process id of its child process and removes the inode of its child process inode under the $/proc$ directory (write). The `exit` call changes the state of the pid of the current process. A system call passed with different options may have different effects on an object. For example, the `open` system call on file f reads the data for f with the O_RDONLY option, writes to the inode of f's parent directory with O_CREATE, and writes to the data for f with the O_TRUNC option. SIMRACER provides a user interface that lets developers define other system calls of interest.

4.2.2 Phase 1: Identify Potential Race Sources

The Phase 1 algorithm computes a set of potential sources of races, $PRaceSet$, between processes P_i and P_j or between a process and signal handler P_i and S_j. Two events have the *potential to race* if P_i and P_j (S_j) access the same SR, at least one of the accesses is

a write, and one access can *potentially* happen both before and after the other. We next introduce notation used to describe our algorithms.

1. The log files of P_i and P_j (S_j) obtained from a normal execution are denoted by L_i and L_j, respectively.
2. `OP(`SR_n`)` denotes an SR access by P_n, and includes both the entry and exit of an SR access operation.
3. σ = (`OP(`SR_i`)`, `OP(`SR_j`)`) denotes a pair of access operations involving the same SR in P_i and P_j (S_j).
4. `ENT(OP(`SR_n`))` denotes the entry of an SR access in process P_n. If `OP(`SR_n`)` is a system call, `ENT(OP(`SR_n`))` is the instruction that invokes that system call. If `OP(`SR_n`)` is a memory or hardware access, `ENT(OP(`SR_n`))` is the instruction that executes the load/store instruction.
5. `EXT(OP(`SR_n`))` denotes the exit of an SR access in process P_n. If `OP(`SR_n`)` is a system call, `EXT(OP(`SR_n`))` is the return instruction of the system call. If `OP(`SR_n`)` is a memory or hardware access, `EXT(OP(`SR_n`))` is the instruction immediately after `ENT(OP(`SR_n`))`.
6. `State(`P_n`)` returns the current state of process P_n.
7. `Execute(`P_n`)` returns the state after executing the current instruction of process P_n.
8. `Output`(P_i, P_j) returns the outputs of P_i and P_j.

Given an event e_i and an event e_j, we use $\prec$ to denote the *happens-before* relationship for e_i and e_j; this is a relationship that can be satisfied by the following rules:

1. if e_i and e_j are from the same process or the same signal handler, and e_i happens before e_j, then $e_i \prec e_j$

2. if e_i and e_j are from different processes or signal handlers, and there is a dependency such that e_j would not happen unless e_i happens, then $e_i \prec e_j$
3. if e_i is from a process and e_j is from a signal handler S, and $e_i \prec e_j$, then $e_i \prec S_{all-the-events}$
4. if e_i is from a process and e_j is from a signal handler S, and $e_j \prec e_i$, then $S_{all-the-events} \prec e_i$
5. $\prec$ is transitively closed.

On the second of the foregoing rules, we infer dependencies from the semantics of the system calls. For example, if Process P_i `forks` a child process P_j, all events in P_i prior to calling `fork` happen before all events in P_j. The following two semantics are also identified as *happens-before* system calls: 1) a `wait` occurs in the parent process and an `exit` occurs in the child process; 2) a `pipe` in one process blocks the data in another process until the data is available. For the third and fourth of the foregoing rules, dependencies are inferred from the properties of signal handlers. Because a signal handler can never be preempted by a regular process, all events are synchronized in the signal handler.

The Phase 1 algorithm takes traces L_i and L_j as inputs and outputs a set of potential races $PRaceSet$. For each shared resource SR_i accessed in L_i, the algorithm iterates over L_j to see if the same shared resource is accessed. If two events do not have a *happens-before* relationship between them and at least one access is a write, the pair of events is added to $PRaceSet$.

To simplify the Phase 2 algorithm, event pairs are recorded based on execution order. For example, an event pair recorded as (σ_1, σ_2) indicates that σ_1 occurs before σ_2 in the original execution trace, and the Phase 2 algorithm will need to reverse the two events to make σ_2 happen before σ_1. Note that only distinct event pairs are identified in terms of

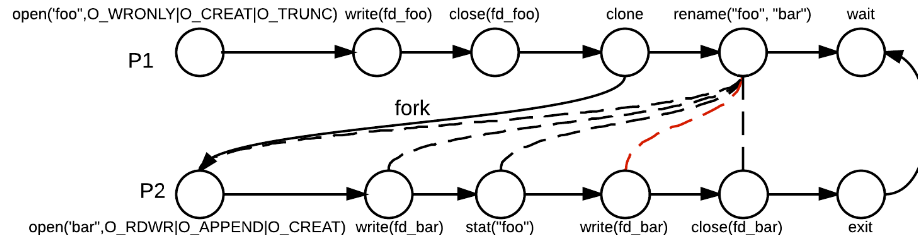

Figure 4.7: Happens-before example (solid arrow for happen-before, dashes for potential races)

instruction locations; we ignore redundant pairs that occur due to recursive calls or loops. As an example, we refer to the program in Figure 4.1, which has the following execution trace:

$P1$:open(“foo”) $\rightarrow$ $P1$:write(fd_foo) $\rightarrow$ $P1$:close(fd_foo) $\rightarrow$ $P1$:clone() $\rightarrow$ $P1$:rename(“foo”,“bar”) $\rightarrow$ $P1$:wait() $\rightarrow$ $P2$:open(“bar”) $\rightarrow$ $P2$:fstat(fd_bar) $\rightarrow$ $P2$:write(fd_bar) $\rightarrow$ $P2$:stat(“foo”) $\rightarrow$ $P2$:write(fd_bar) $\rightarrow$ $P2$:close(fd_bar) $\rightarrow$ $P2$:exit()

Figure 4.7 shows the *happens-before* relationships and the potential-race relationships between operations in this trace. Solid arrows indicate *happens-before* relationships between pairs of events. For example, all events in $P2$ happen after events in $P1$ prior to the fork call. The return from wait in $P1$ happens after the exit of $P2$ because $P1$ has to wait for $P2$ to finish. The system call rename in $P1$ modifies the inodes of “foo” and “bar”, thus it races with the system calls accessing the inodes in $P2$. Events connected by dashes are potential race pairs.

This Phase 1 algorithm may generate false positives because it does not identify all synchronization operations. In Figure 4.1, the statement in line 10 cannot be executed unless the statement in line 16 is executed. Thus, the potential race pairs ($P1$:rename (“foo”, “bar”), $P2$:write(fd_bar)) (the second write(fd_bar) in $P2$) and ($P1$:rename (“foo”, “bar”), $P2$:close(fd_bar)) in Figure 4.1 can never result in real races. We describe how our proposed system deals with false positives as part of Phase 2.

4.2.3 Phase 2: Test for Real Races

The main objective of the Phase 2 algorithm is to automatically control process scheduling based on each pair in $PRaceSet$. Since signal handlers use different synchronization mechanisms than regular processes, we also develop an algorithm to automatically control signal events. We present these two algorithms as $Phase2$-$RegPro$ (Figure 4.8) and $Phase2$-$SoftSig$ (Figure 6.5).

We first describe the $Phase2$-$RegPro$ algorithm. Let σ = (σ_i, σ_j) be a pair of instructions that have been identified as potential race sources. In this case, σ_i happens before σ_j when P_i and P_j are originally executed. The algorithm takes σ, P_i and P_j, and test oracle O as inputs; it outputs real races and observable faults. The goal of the algorithm is to switch the order of σ_i and σ_j to let σ_i happen after σ_j. Specifically, whenever a process is about to execute an instruction in σ, it is suspended (lines 7-9). After both processes have been suspended, P_j is allowed to resume (lines 10-12) until it reaches the exit of its SR access (lines 13-14).

Next, P_i is allowed to resume (line 15) until it reaches the exit instruction of its SR access (line 17). At this point, we have switched the execution order of σ_i and σ_j. A real race is added into $RaceSet$ and the processes continue executing normally (lines 18-19). If one process terminates, the algorithm checks the state of the other, and allows it to resume if it is still suspended (lines 21-23); in this test execution the potential race is not a real race because the terminated process does not reach the SR after the other process reaches the SR and being suspended. When both processes terminate, the algorithm checks their outputs against a test oracle to determine whether a fault has been identified (lines 24-26).

To see how this algorithm detects process-level races, consider the example shown in Figure 4.1. Suppose we select the potential race pair σ = ($P1$:`rename`("foo", "bar"), $P2$:`write`(fd_bar)) (the first `write`(fd_bar) in $P2$) identified in the first phase. The

Algorithm SimRacer: Phase2-RegPro

```
1: Inputs: PRaceSet, P_i, P_j, O
2: Outputs: RaceSet, Faults
3: begin
4: RaceSet = φ
5: for each σ = (σ_i, σ_j) ∈ PRaceSet
6:     execute P_i and P_j
7:     if Execute(P_n) ∈ σ and n ∈ (i, j)
8:         suspend P_n
9:     endif
10:    if State(P_i) is suspended and State(P_j) is suspended
11:        continue P_j
12:    endif
13:    if Execute(P_j) == EXT(σ_j)
14:        suspend P_j
15:        continue P_i
16:    endif
17:    if Execute(P_i) == EXT(σ_i)
18:        RaceSet = RaceSet ∪ σ /*race occurs*/
19:        continue P_j
20:    endif
21:    if P_m terminates ∧ State(P_n) is suspended ∧ m ∈ (i, j) ∧ n ∈ (i, j) ∧ m ≠ n
22:        continue P_n
23:    endif
24:    if Output(P_i, P_j) ≠ O
25:        print "Error: fault found"
26:    endif
27: endfor
28: end
```

Figure 4.8: Phase 2 algorithm to test for races between processes

system call `rename` comes before the `write` system call in the original execution trace; thus the goal of the second phase is to let the `rename` happen after the `write`. First, the algorithm suspends the execution of $P1$ and $P2$ when they are about to execute the instructions in the race pair. Next, $P2$ is allowed to resume until the `write` returns. $P2$ is then suspended again, and $P1$ is allowed to resume until it returns from `rename`. At this point, we conclude that σ is a real race. In fact, if we let $P1$ and $P2$ continue, we will discover that this race is harmful.

By default, SIMRACER switches the order of operations to allow one to happen immediately after the other. However, in a pair (σ_i, σ_j), it may not always be possible to

Algorithm SimRacer: Phase2-SoftSig

```
1: Inputs: PRaceSet, P, S
2: Outputs: RaceSet, Faults
3: begin
4: RaceSet = φ
5: for each σ = (σ_i, σ_j) in PRaceSet
6:     execute P
7:     if (Execute(P) == EXT(σ_j) ∧ σ_i ∈ S) ∨ (Execute(P) == ENT(σ_i) ∧ σ_j ∈ S)
8:         if State(S) is not masked
9:             raise signal S
10:        else
11:            find another possible location
12:        endif
13:        if S accesses SR
14:            RaceSet = RaceSet ∪ σ /*race occurs*/
15:        endif
16:    endif
17:    if Output(P, S) ≠ O
18:        print "Error: fault found"
19:    endif
20: endfor
21: end
```

Figure 4.9: Phase 2 algorithm to test for races between a process and a signal

let σ_i happen immediately after σ_j due to changes in system states (e.g., disabled process scheduling, page fault). In such cases, SIMRACER continues the execution of P_j (line 14) until it is possible to schedule P_i and execute σ_i (line 17). If the entry instruction of the operation in another potential race pair is reached and it is still not possible to execute P_i, then (σ_i, σ_j) is not considered to be a race.

We now describe Algorithm $Phase2$-$SoftSig$. The goal of this algorithm is to manipulate the order of potentially racing accesses to SR between a user process P and the signal handler S. According to the third and fourth rules described in the *happens-before* relationship, the goal of this algorithm can be restated as that of swapping the order of an SR access in P and the arrival of S. The algorithm first executes process P (line 6). If the signal occurs before SR is accessed by P in the original execution trace, the algorithm forces the signal to occur immediately after SR is accessed by P (first condition in line 7).

If the signal occurs after SR is accessed by P in the original execution trace, the algorithm forces the signal to occur just before SR is accessed by P (second condition in line 7). If a race occurs, it is added to $RaceSet$ (line 14).

Because it may not be possible to raise a signal immediately (e.g., the signal is currently masked), the algorithm checks the current state of P (line 8) before raising a signal. Similar to $Phase2$-$RegPro$, it also checks outputs on termination of the processes (lines 17-19) to determine whether a fault has been identified.

If the signal handler (S) cannot be raised *immediately after* the SR access in P (lines 10-12), the algorithm postpones S until it can feasibly be raised, or until the entry instruction of the operation in another potential race pair is reached. In the case in which the signal handler cannot be raised *immediately before* the SR access in P, the algorithm finds another possible location in P at which to raise S, before the SR access and after the entry instruction of the operation for another potential race pair.

To illustrate the algorithm's operation, using Figure 4.2 as an example, the variable $files_created$ is a global variable and its memory read and write accesses are denoted by `load` and `store`. The execution trace is as follows:

P:`open`($file1$) $\rightarrow$ P:`load`($files_created$) $\rightarrow$ P:`store` ($files_created$) $\rightarrow$ P:`open`($file2$) $\rightarrow$ P:`load`($files_created$) $\rightarrow$ P:`store`($files_created$) $\rightarrow$ S:`load`($files_created$) $\rightarrow$ S:`unlink`($file1$) $\rightarrow$ S:`load`($files_created$) $\rightarrow$ S:`unlink`($file2$)

There are two potential race pairs:

σ_1=(P:`open`($file1$), S:`unlink`($file1$)) and

σ_2=(P:`store`($files_created$), S:`load`($files_created$)).

Although (P:`open`($file2$), S:`unlink`($file2$)) can potentially race, the elements of the pair are redundant because they are part of loops. Considering σ_2, S happens after P:`store`($files_created$) occurs in the original execution; thus, the algorithm forces S to

be raised before P:`store`$(files_created)$ in the second phase. In this scenario, $files_created$ is incorrectly read by S. As such, the race is real and harmful.

4.2.4 Further Discussion

Several additional aspects of our approach and the foregoing algorithms bear further discussion. First, SIMRACER tests only one potential race pair in each test run. This is done to avoid *fault masking* effects, which may occur in cases where the presence of a single conflicting pair is "masked" by the presence of the second. Thus, in a given run, our system needs to first determine whether a pair has already been tested; if it has, the system does not monitor events related to that pair in that run.

Second, in Phase 2, SIMRACER may encounter a potential race pair more than once during a test run (e.g., due to a loop). Once a pair is identified, there is no need to keep recording it after each encounter; thus, our algorithm records only the first instance. This reduces the amount of redundant information in the race report.

Third, SIMRACER uses different testing approaches in its two phases. In the first phase it runs all tests, and in the second phase it runs only tests that detected potential races in the first phase. If a potential race pair is exercised by multiple tests in the first phase, SIMRACER selects only one test in the second phase; if this test does not find the fault, SIMRACER selects the next test, and so forth. This process may result in more test runs than the original number of tests. In fact, the actual number of test runs is:

$$TR = |TC| + \sum_{i=1}^{|PRaceSet|} |tc_i|$$

Here, $|TC|$ is the total number of original tests, $|PRaceSet|$ is the number of potential races, and $|tc_i|$ is the number of tests required to test for the ith pair in $PRaceSet$.

Fourth, SIMRACER reports both harmful and benign races. There are two steps that can be followed to distinguish these. First, races that cause failures that are observable

through test oracles are clearly harmful. Second, although SIMRACER is fully automated, additional manual effort can be used to determine whether the program may fail under other test inputs or interleavings. We will study the identification of benign races in future work by leveraging a recently proposed symbolic execution technique [117].

Fifth, SIMRACER can also be used to *replay* concurrent executions of multiple threads or processes. This is achieved by using the Controller to stop the execution of threads or processes at the same execution points as in prior runs. Note that in Linux, every thread is a lightweight process with its own PID, so SIMRACER can track it just like a process.

Sixth, SIMRACER can cause processes to deadlock in ways in which they should not. Deadlock occurs when one process is waiting for an event in another process that has been suspended by SIMRACER. We observed such a deadlock in four of the object programs we considered in our empirical study. To address this problem, SIMRACER periodically checks the states of the processes, and allows the process that is being waited for to resume.

Seventh, SIMRACER decomposes a complex system into a collection of controllable processes. This allows us to focus only on the two processes under test. Processes other than these run as normal. SIMRACER monitors and controls the two processes without interfering with others to guarantee that the system can execute.

Eighth, we have used SIMRACER only to test for races between a process and a signal handler that interrupts the process, because a signal handler cannot run in parallel with the process that it interrupts. However, it is possible for a signal handler to race with another process running concurrently with the handler. In this scenario, we can apply the $Phase2$-$RegPro$ algorithm to detect races between the handler and the concurrently running process.

Ninth, by viewing every shared resource as a potential race source, SIMRACER can produce false positives in Phase 1. We use the happens-before relationship to reduce these false positives. In some cases, false positives can be reduced in Phase 1 by strictly mod-

eling communication primitives as happens-before relationships. However, when synchronization occurs at the process-level there can be ambiguities at the implementation level because the implementations may be unknown (many system calls are blackbox), and their synchronization effects cannot easily be modeled. Instead, SimRacer takes an approach that does not involve preconceptions about implementation details.

Finally, SIMRACER employs random schedules [6] in Phase 2. As such, it is precise and cost effective, but can be incomplete. On the other hand, complete replay techniques such as PENELOPE [118] may incur much higher overhead due to context switches. In our future work, we will further evaluate cost-effectiveness by adopting complete replay techniques.

4.3 Empirical Study

To evaluate SIMRACER we consider two research questions:

RQ1: How effective is SIMRACER at detecting process-level races across the phases of its operation?

RQ2: How effective is SIMRACER at detecting observable faults related to those races?

RQ1 allows us to evaluate the effectiveness of our Phase 2 algorithm at reducing false positives while RQ2 lets us consider the ultimate effectiveness of our approach.

4.3.1 Objects of Analysis

To obtain objects of analysis, we searched several bug repositories (e.g., GNU, bugzilla, debian) using the key words “concurrency”, “race”, “deadlock”, “atomicity”, and “signal”. By examining bug descriptions (not code), we identified applications written in C or C++ for which existing descriptions contained reports of concurrency faults related to re-

source dependencies among multiple processes and between processes and software signal handlers. We randomly selected 30 of these applications. We next eliminated from consideration those applications that could not compile in our environment; this process left us with 16 applications.

The selected programs are listed in Column 1 of Table 4.1 (some are listed multiple times, for reasons explained below). BASH and TCSH are command interpreters that execute commands read from standard input or from a file. We use BASH versions 3.0, 3.2, and 4.2 and TCSH version 6.17. STRACE is a debugging tool that intercepts and records the system calls made by a process and the signals received by a process; we use version 4.5.18. UPDATEDB, LOCATE, and FIND are Linux utilities: UPDATEDB creates or updates a database used by LOCATE to find files, FIND is used to locate files in the file system; we use versions 4.1.2 of each of these. UART is a UART device driver from the Linux kernel; we use version 2.6.31. All other programs are Linux core utilities, including MV (version 6.1.9), CSPLIT (version 5.2), LN (version 5.97), MKDIR (version 5.97), MKDIR (version 5.2), MKNOD (version 5.2) and MKFIFO (version 5.2). Column 3 of Table 4.1 lists the numbers of lines of non-comment code in the applications.

To address our research questions we also needed to identify specific faults in the object programs that could result in process-level races. By examining descriptions in the bug repositories, for each object program, we located faults related to the two race types. Column 5 of Table 4.1 lists the numbers of faults found.

To cause process-level races to occur, the basic programs shown in Column 1 of Table 4.1 must each be paired with at least one other process or signal handler. To select second programs to pair with the basic programs, we again consulted the bug repositories for the basic programs, and where possible, selected as paired programs those programs indicated as having led to the faults previously identified when run concurrently with the basic programs.

Table 4.1: Object Program Characteristics

Program		NLOC		Faults	Tests
Basic	Paired	Basic	Paired		
BASH1	BASH1	39102	39102	1	30
BASH2	BASH2	40944	40944	1	30
BASH2	SIG	40944	493	1	30
BASH3	SIG	48263	553	2	30
TCSH	TCSH	47167	47167	1	5
STRACE	MYPROG	25192	491	1	49*
STRACE	SIG	25192	1814	1	49
UPDATEDB	UPDATEDB	1924	1924	1	11*
UPDATEDB	LOCATE	1924	669	1	34*
FIND	MYPROG	4004	491	1	25*
UART	UART	1944	1944	1	16
MV	MYPROG	346	491	1	45
CSPLIT	SIG	1032	73	3	30*
PS	GREP	4695	4695	1	57
LN 5.94	MYPROG	410	491	1	28
MKDIR1	MYPROG	143	491	1	23
MKDIR2	MYPROG	146	491	1	23
MKNOD	MYPROG	184	491	1	16*
MKFIFO	MYPROG	112	491	1	17*

In several cases, bug reports did not specify problematic paired programs. To handle these cases, we asked a person who was not familiar with our testing approach and had no access to the source code to create a program, MYPROG, that can be paired with multiple basic programs. This program accepts shared resources as inputs, and uses various operations to manipulate the shared resources accessed by the basic programs. For example, on one test for MV, "`mv file1 file2`", that operates on two shared resources `file1` and `file2`, MYPROG uses system calls (e.g., `open, stat, access`) to access `file1` or `file2` or both.

The paired programs and signal handlers we used are listed in Column 2 of Table 4.1, and the numbers of lines of code contained in these programs are reported in Column 4. In three cases (on programs BASH2, STRACE and UPDATEDB) two different paired programs or signal handlers involved in process-level races were identified, yielding two pairs of programs that could be studied – this accounts for the repetitions in Column 1.

To address our research questions we also required tests. For basic programs that had been released with tests, we selected, from those, tests that could potentially exercise shared resources (e.g., tests of operations on BASH history files). Seven of our object programs, however, were not equipped with tests. For these programs, we created black-box test suites. Engineers often use such test suites designed based on system parameters and knowledge of functionality [119]. We followed this approach, using the category-partition method [120], which employs a Test Specification Language (TSL) to encode choices of parameters and environmental conditions that affect system operations and combine them into test inputs. We asked a person who was not familiar with our testing approach and had no access to the source code to employ this approach. Column 6 of Table 4.1 lists the numbers of tests ultimately utilized for each object program pair; numbers marked with a "⋆" indicate instances in which tests were generated using TSL.

Testing also requires test oracles. For programs released with existing test suites and with built-in oracles provided, we used those. Otherwise we checked program outputs, including messages printed on the console and files generated and written by the programs. The oracles were obtained by running the two processes under test sequentially, which did not involve any interleavings.

4.3.2 Variables and Measures

Independent variable. Our independent variable is the testing technique used. We use two randomized stress testing techniques as the baseline for comparison with our approach.

Although other baseline techniques could be considered (e.g., random scheduler fuzzing technique) we chose stress testing for several reasons. First, stress testing is a practical approach that has been widely used in industrial settings [121, 122, 123]. Second the bug reports for some of our object programs described stress testing as the approach used to

Table 4.2: Object Program Statistics

Program		TR-SR	TT-SR	S1	S2	S3
Basic	Paired					
BASH1	BASH1	42	87.3	42	170	170000
BASH2	BASH2	46	103.6	46	186	186000
BASH2	SIG	38	97.3	38	184	184000
BASH3	SIG	36	77.6	36	127	127000
TCSH	TCSH	37	28.7	9	37	37000
STRACE	MYPROG	59	88.7	59	253	253000
STRACE	SIG	57	65.6	57	234	234000
UPDATEDB	UPDATEDB	35	570.4	35	144	144000
UPDATEDB	LOCATE	40	669.7	40	167	167000
FIND	MYPROG	41	9.0	41	180	180000
UART	UART	18	16.7	18	75	75000
MV	MYPROG	72	157.4	72	290	290000
CSPLIT	SIG	36	7.4	36	185	185000
PS	GREP	64	58.4	64	365	365000
LN 5.94	MYPROG	49	9.6	49	320	320000
MKDIR1	MYPROG	45	7.8	45	195	195000
MKDIR2	MYPROG	45	7.8	45	197	197000
MKNOD	MYPROG	30	5.4	30	136	136000
MKFIFO	MYPROG	22	4.1	22	24	24000

reproduce those bugs. Third, stress testing has been well studied as a baseline approach for detecting thread-level concurrency faults [55, 124].

In the case of program pairs that do not involve signal handlers, the stress testing technique that we used runs a target program pair multiple times and invokes 20 processes (the largest number obtained from bug repositories) for each of the basic and paired programs. In the case of program pairs that do involve signal handlers, the stress testing technique that we used runs the basic program multiple times and randomly raises a software signal at an arbitrary location in the basic program during each test run. The type of signal we chose was SIGINT. As in SIMRACER, this signal was raised only when it was determined that it was possible to do so by examining system states; otherwise, the signal was postponed to another arbitrary location in the program.

For each stress testing technique we used three different numbers of runs (levels). At the first level ($S1$) we used the same number of test runs as the number of test runs required

by SIMRACER; this lets us examine the relative effectiveness of the two approaches on equivalent numbers of test runs. At the second level ($S2$) we used the same amount of testing time required by SIMRACER; this lets us examine the relative effectiveness of the approaches when each is given the same amount of time. At the third level ($S3$) we used the number of tests run at the second level multiplied by 100; this lets us examine how the effectiveness of SIMRACER compares to that of a more robust stress testing process. In all cases, tests are randomly selected from the sets of tests utilized by SIMRACER.

Dependent variables. We consider two dependent variables. Our first dependent variable measures technique *effectiveness* in terms of the *number of races detected*. This variable applies only to SIMRACER, not the stress testing approach, but it lets us evaluate the effectiveness of our Phase 2 algorithm at reducing false alarms.

Our second dependent variable focuses on technique *effectiveness* measured in terms of the *ability of techniques to detect faults*. For both SIMRACER and the stress testing techniques that we consider, we measure the number of observable faults reported by test oracles. These can include both the known faults discovered through inspection of bug repositories, and newly discovered faults.

4.3.3 Study Operation

We conducted our study by configuring Simics to virtualize a single core X86 system running a Linux 2.6.15 kernel. As noted in Section 4.2, when concurrency faults occur on single core systems they can also occur in multi-core systems, so results obtained on a single core system would be obtainable on a multi-core system. To confirm that SIMRACER can function in multi-core environments, we did gather data on the UART object on such a system.

To implement the testing process of SIMRACER, as noted above, we ran both basic and paired programs simultaneously with each test case, and for pairs involving signal handlers raised signals at arbitrary locations in the basic program. We collected the runtime trace for each test case and applied the Phase 1 algorithm. We next applied the Phase 2 algorithm to check for real races based on the potential races identified in the first phase. Finally, we inspected all of the reported real races that did not result in detectable failures to determine whether they were harmful or benign.

Column 3 of Table 4.2 lists the number of tests (TR-SR) run by SIMRACER. (The approach by which the number of test runs is determined is discussed in Section 4.2.4.) Column 4 reports the time required for these test runs (TT-SR) in seconds. To implement the stress testing techniques, we wrote a script that performed the techniques at the three levels of effort described in Section 4.3.2. Columns 5, 6 and 7 list the number of test runs at the first, second, and third levels of stress testing, respectively.

4.3.4 Threats to Validity

The primary threat to external validity for this study involves the representativeness of our programs and test suites. Other programs may exhibit different behaviors and cost-benefit tradeoffs, as may other forms of test suites. However, the programs we investigate are popular and the faults we consider are real. Furthermore, the test cases are either those provided with the programs, or are created using a commonly used process (TSL in this case) so they are representative of test cases that could be used in practice by engineers to test these programs.

The primary threat to internal validity for this study is possible faults in the implementation of our approach and in the tools that we use to perform evaluation. We controlled for this threat by extensively testing our tools and verifying their results against a smaller

program for which we can manually determine the correct results. We also chose to use popular and established tools (e.g., Simics) for implementing the various modules in our approach. A second source of potential threats involves instrumentation used to detect failures. To test for races between a process and a signal handler, more than one signal handler can be applied. For our study, we chose *SIGINT* because it is commonly used (Ctrl-C). Applying other handlers could increase fault detection effectiveness but with additional testing cost. Fault detection effectiveness can also depend on the test oracles used. We report failures using output-based oracles. Using internal oracles may increase fault detection effectiveness [2, 125], but at the cost of manual effort, testing time, and false positives. Finally, it is possible that faults revealed by our test oracles are due to effects other than the propagation of races. We controlled for this threat by comparing error results to those described in the bug reports; we report only faults related to races.

Where construct validity is concerned, numbers of faults and races detected are just two variables of interest. Other metrics such as the cost of trace analysis could be valuable.

4.4 Results and Analysis

RQ1: Race Detection Effectiveness of SIMRACER. Columns 3-5 of Table 4.3 report the number of potential process-level races detected by the Phase 1 algorithm, the number of these identified as real races by the Phase 2 algorithm, and the number of harmful races identified by further examination, respectively. The Phase 2 algorithm reduced the number of false negatives contained in the sets of potential races by 53.2% overall, with reductions ranging from 20% to 100% across the 19 program pairs. The overall percentage of harmful races relative to all real races detected in the second phase was 43.7%.

RQ2: Fault Detection Effectiveness of SIMRACER. Column 6 of Table 4.3 reports the number of faults detected by SIMRACER. (Note that the mapping between harmful races

Table 4.3: Results

Program		SIMRACER			Faults Detected			
Basic	Paired	Ph.1	Ph.2	Hm.	SR	S1	S2	S3
BASH1	BASH1	13	8	2	1	1	1	1
BASH2	BASH2	9	4	0	0⋆	0	0	0
BASH2	SIG	6	3	2	**0**⋆+	0	0	0
BASH3	SIG	5	4	3	1⋆	0	1	1
TCSH	TCSH	7	4	2	1	0	1	1
STRACE	MYPROG	3	0	0	0⋆	0	0	0
STRACE	SIG	5	1	1	1	0	0	0
UPDATEDB	UPDATEDB	17	3	3	1	0	0	1
UPDATEDB	LOCATE	12	4	1	1	0	0	0
FIND	MYPROG	4	2	2	1	0	0	0
UART	UART	11	7	2	1	0	0	0
MV	MYPROG	16	6	3	1	0	0	0
CSPLIT	SIG	9	3	3	2⋆	1	1	1
PS	GREP	19	13	3	1	0	0	0
LN 5.94	MYPROG	14	4	2	1	0	0	0
MKDIR1	MYPROG	10	5	2	1	0	0	0
MKDIR2	MYPROG	10	5	2	1	0	0	0
MKNOD	MYPROG	8	6	3	1	0	0	0
MKFIFO	MYPROG	8	5	2	1	0	0	0

and faults can be many-to-one or one-to-one; this is because a single fault such as a missing lock pair may result in multiple races.) For all program pairs, SIMRACER detected 17 of the 22 previously known faults; a fault detection rate of 77.3%. The five cases in which known faults were not detected are indicated by a "⋆". SIMRACER also detected one fault that had not been reported; this fault was related to a signal handler in BASH2 and is indicated by a "+".

There is one special case worth noting. One of the known faults missed by SIMRACER, on BASH2 paired with a signal handler (indicated by bold font), failed to appear in the results reported by output-based test oracles. SIMRACER did identify an actual race in this case, but the effects of the race did not propagate to output.

Columns 7-9 in Table 4.3 report the numbers of faults detected by the three levels of stress testing techniques. As the data shows, in total, stress testing detected only four of the 22 known faults; a fault detection rate of 18.2%. For program pairs (BASH1, BASH1) and

(CSPLIT,SIG), all three levels detected the same known faults. For program pairs (BASH3, SIG) and (TCSH,TCSH), levels $S2$ and $S3$ detected the same known fault while level $S1$ detected none. For program pairs (UPDATEDB, UPDATEDB), only $S3$ detected the known fault. The data shows that given longer running times, stress testing may demonstrate more fault detection effectiveness than with shorter times; however, at all three levels, SIMRACER performed as well as or better than stress testing.

4.5 Discussion

Faults detected. We first describe the detected known faults. Program pairs (BASH1, BASH1), (TCSH, TCSH), and (UPDATEDB, UPDATEDB) [110, 111, 112] suffer from classic races in which two processes access the same resource at the same time. The races result in corrupted history files for BASH1 and TCSH, and a corrupted database file for UPDATEDB.

For program pairs (STRACE, SIG) and (CSPLIT, SIG) [126, 127], races occurs due to untimely signals. In the case of (CSPLIT, SIG), a signal arrives before the statement `files_created` in the file deletion routine [127] is executed, causing the signal handler to print an error message. In the case of (STRACE, SIG), SIGINT arrives to kill a process before an STRACE system call can complete a write to the $/proc$ directory. As a result, the process is not cleanly detached (i.e., it does not print correct messages).

Program pairs (UPDATEDB, LOCATE), (FIND, MYPROG), (MV, MYPROG), and (LN, MYPROG) [112, 128, 129, 130] suffer from races that result in premature freeing of resources. For example, for (MV, MYPROG), the MV process calls `unlink` before `rename` and they are not atomic [129]. The MYPROG process attempts to open the file in between the operations, causing the `open` to fail.

On program pair (PS, GREP), when the command "`ps aux | grep string`" is run, the command may (incorrectly) print the command "`grep string`" itself [131].

This race occurs between the system call `execve` of GREP and the system call `read` of PS; they both access the inode of `/proc/pid/cmd_line`, where `pid` corresponds to the GREP process. If the `read` happens after the `execve`, the GREP command itself will be printed.

For program pairs (MKDIR1, MYPROG), (MKDIR2, MYPROG), (MKNOD, MYPROG), and (MKFIFO, MYPROG), faults are all caused by TOCTTOU (time of check to time of use) races; that is, a change is made by one process between the time at which the condition is checked and the time at which the results of that check are used by another process [132, 133]. For each of the four programs, `myprog` slips in and changes the file permission, in between the file permission check and the use of the file in the basic program.

The previously unknown fault was detected when BASH2 was paired with its signal handler. In this case, SIGINT occurs just before executing `history_lines_this_session++`. The signal handler reads the incorrect value of `history_lines_this_session`. As a result, the history file is not saved when BASH terminates. Section 4.1 described the fault in UART.

Faults not detected. We also examined the undetected faults. On the (BASH2, BASH2) program pair, the race is exposed when the two BASH processes run under different sessions (a parent BASH and a child BASH) [134]. Initially, we did not consider this scenario in our study. However, once we created such a scenario in order to reproduce the fault, SIMRACER *did detect this race*. The lesson to be learned here is that engineers may do well to realistically explore potential execution scenarios (e.g., considered sessions, simultaneous logins) because doing so may uncover additional faults.

On the (BASH2, SIG) program pair, as noted in Section 4.4, SIMRACER detected a race that did not propagate to output. This occurred because BASH and the signal handler both wrote the same value to a shared memory location [135]. A different test may cause the signal handler to write a different value, potentially causing the program to fail. The lesson

to be learned here is that our approach should consider redundant writes as a criterion to eliminate tests as part of the selection process in Phase 2 (see Section 4.2.4).

On the (BASH3, SIG) program pair, the fault related to one race [136] was not detected because our tests did not exercise the shared variable that can cause the race to occur. Subsequently, we manually created a test to exercise this shared variable and SIMRACER was then able to detect this race. The lesson to be learned here is that the effectiveness of our approach can depend on the initial test suite and the oracles used. Selecting a proper adequacy criteria (e.g., data-flow) could increase the effectiveness of our approach.

On the (STRACE, MYPROG) program pair, the race occurs only on an IA64 platform whereas our experiments were conducted on an X86 platform [126]. This confirms an argument made by Laadan et al. [26] that process level races are sensitive to system configurations. The result also suggests that engineers may wish to test under different system configurations.

On the (CSPLIT, SIG) program pair two signals are required to trigger the race while SIMRACER issued only one signal each time. We configured SIMRACER to handle two signals and it successfully detected this fault [127].

Benign races. When a race occurs but cannot cause any observable fault with any inputs, it is benign [137]. In this study we found several benign races. For example, when two BASH processes issued an `open(``dev/null'', O_TRUNC)` system call with the `O_TRUNC` option set (file length is truncated to 0), the two system calls race because they both write to the entire data range of the inode of `dev/null` without proper synchronization. However, they are each immediately followed by a `close` system call. As such, `dev/null` is not disturbed and faults will never occur.

In the case of BASH3, a race is reported because the signal handler modifies a global variable of BASH. However, the old value of this variable is restored before it is read again by the handler. As such, a fault will not occur. In the case of UART, one kernel thread

always writes the same value to the IIR register. Even though races occur, the read value is always the same.

Runtime cost. One shortcoming of using virtual platforms such as Simics is that virtualization time can be much longer than execution time on a real system. Columns 3, 4 and 8 of Table 4.1 indicate that the average runtime per test of SIMRACER is significantly greater than the runtime per test for the stress testing technique. In other words, given the same testing time, stress testing invokes more test runs than SIMRACER. However, we have also seen that additional runs through stress testing do not render testing more effective in most cases. As such, when we consider both fault detection rate and testing time, SIMRACER is more cost-effective. In addition, as noted in Chapter 1, the costs of testing can be amortized into the development workflow when using VMs.

System-specific properties. Our experiment with SIMRACER was conducted on a system running on Linux. As such, runtime properties including shared resources, system calls, and schedulers that we considered are Linux specific. Detecting different types of elusive faults, however, does require engineers to look deeper to examine system-specific properties. In practice, arguably, companies focus on specific systems, so they could implement our approach on those systems. Thus, the general idea of our approach would translate to other systems.

Testing efficiency. Modern software systems evolve (e.g., through software updates, availability of new configurations, and creation of new applications). SIMRACER is designed with evolution in mind. When a new application is installed in a system, it is necessary to test the interactions between that application and other applications. One approach is to test the entire system again [121]. With SIMRACER, developers can analyze the resource usage of this new application with previously tested applications. Only applications that share resources with this new application need to be tested again.

Environment dependency. Given the same test input, it is possible that a potential race pair computed in Phase 1 is not reachable in Phase 2. Such a scenario can occur if the system environment or the status of a device changes (e.g., in the case of a buggy device port), causing a program to take a path that does not access the targeted shared resource. We did not encounter such a case in our study. This is because by using a virtual platform, we can maintain environment consistency across different runs.

In contrast, SIMRACER is more "active" in that it executes processes with a random schedule and considers system states whenever it tries to reorder two potential racing events. As described in Section 4.2, SIMRACER adjusts process scheduling to adaptively reorder the two events. Unlike RACEPRO, which erroneously classifies a race as benign if there are no detectable failures at the output, SIMRACER does not do this. As shown in our study, some races that are classified as benign may not always be benign. Different inputs may show that such races are harmful.

As noted in Section 2.1.1, existing thread-level race detectors do not directly apply at the process-level. Nonetheless, the ideas behind existing thread-level approaches can be useful in our context if system call effects are modeled as read/write operations. For instance, we may leverage the approach used in CHESS to ensure exhaustive interleavings of processes.

4.6 Conclusion

In this chapter, we presented another instantiation of our SIMEXPLORER framework, SIMRACER, for effectively testing for process-level races. We have conducted an empirical study applying SIMRACER to fifteen user-level programs and one device driver program. We have empirically compared SIMRACER to traditional stress testing techniques, and our

results suggest that SIMRACER is more effective than these techniques at detecting faults that are caused by the process-level races.

Chapter 5

SimLatte: Using Testing to Estimate Worst-Case Interrupt Latencies in Embedded Software [4]

We have created two instantiations of the SIMEXPLORER framework that can detect concurrency faults. In this chapter, we introduce SIMLATTE, another instantiation of the SIMEXPLORER framework meant to uncover WCILs, that is informed by a set of factors that cause WCILs to occur. SIMLATTE utilizes a genetic algorithm (GA) for test case generation to find inputs and interrupt arrival points. We use a GA because research has shown that GAs are effective for automatic test case generation [139]. For interrupt-driven applications, GAs are a particularly appropriate choice, because such applications can have large combinations of inputs and many interrupt locations that can affect latencies. The guided exploration used by evolutionary algorithms allows a GA to cover a wide range of combinations of inputs and locations. SIMLATTE then employs an opportunistic inter-

[4] The contents of this chapter have appeared in [138].

rupt invocation approach to trigger interrupts, at feasible locations, that are likely to cause significant interrupt latencies.

To evaluate SIMLATTE, we compare its results to those achieved with random test case generation and without opportunistic interrupt invocation. We also compare it to a static analysis approach. Our results show that SIMLATTE is effective for calculating WCILs and can handle a larger number of nested interrupts than the random approach. SIMLATTE is also efficient, identifying WCILs in significantly less testing time. By further investigating the effectiveness and efficiency of the two components of SIMLATTE, we show that applying either of the two components yields better results than random testing, but employing both components yields the best results. Finally, we observe that as the number of interrupts in a system increases, SIMLATTE produces more precise WCILs than the static analysis approach that we consider.

The contributions of this work are:

1. the first testing based approach for finding WCILs in interrupt driven software that is informed by the factors that impact interrupt latency;
2. an instantiation of the SIMEXPLORER framework, SIMLATTE, that uses a GA along with opportunistic interrupt invocation to find WCILs;
3. an empirical study demonstrating the effectiveness of our approach.

5.1 Background: Factors that Affect Latency

In this section we provide background on, and discuss three factors that can affect, interrupt latencies (**F1-F3**). We then show how SIMLATTE leverages these factors to determine WCILs. We first explain the notation used in this work.

We denote an interrupt-driven program by $P = \texttt{Main} \,\|ISR_1\|ISR_2\| \ldots \|ISR_N$, where `Main` is the main program and ISR_1, ISR_2, ..., ISR_N are interrupt service routines, and

Table 5.1: Interrupt Examples

1	main{	1	ISR2{
2	...	2	...
3	irq_disable();	3	isr_disable();
4	**if**(input > 10){	4	**if**(push_button){
5	*/*execute 100 cycles*/*	5	*/*execute 290 cycles*/*
6	}	6	}
7	**else**{	7	**else**{
8	*/*execute 500 cycles*/*	8	*/*execute 60 cycles*/*
9	}	9	}
10	irq_enable();	10	isr_enable();
11	...	11	...
12	}	12	}
1	ISR1{	1	ISR3{
2	**if**(data < 128){	2	**if**(signal){
3	*/*execute 50 cycles*/*	3	*/*execute 300 cycles*/*
4	}	4	}
5	**else**{	5	**else**{
6	*/*execute 180 cycles*/*	6	*/*execute 960 cycles*/*
7	}	7	}
8	}	8	}

where subscripts indicate interrupt numbers, with larger numbers denoting lower priorities. Typically, P receives two types of *incoming data*: command inputs as entered by users and sensor inputs such as data received through specific devices (e.g., a UART port). An *interrupt schedule* specifies a sequence of interrupts occurring at specified program locations. The input data and an interrupt schedule for P together form a *test case* for P. In this work, we do not consider reentrant interrupts (interrupts that can preempt themselves); these are uncommon and used only in special situations [5]. Next, we discuss three factors that can affect interrupt latencies.

F1: Execution time of critical sections. A major contributor to increased interrupt latency is the number and length of critical sections in which interrupts are disabled [140]. Consider two types of critical sections. The first type is programmed in the main program `Main`. By disabling interrupts in `Main`, the processor delays the handling of interrupts until the critical section exits. The top left quadrant of Table 5.1 illustrates this. In `Main`, there are two paths in the critical section, and these have different execution cycles (100 and 500 cycles, respectively). Interrupts that occur within the `else` branch can have longer

latencies because this branch requires more time to execute. As a good programming practice, developers are encouraged to keep critical sections short.

The second type of critical section exists in ISRs. If nested interrupts are not allowed in a microcontroller, the processor disables global interrupts before transferring control to an ISR and re-enables them when control returns to the main program. In this case the entire ISR is considered to be a critical section. If nested interrupts are allowed, the ISR is specified as interruptible by setting specific hardware bits. In this scenario, a critical section is defined as a non-interruptible code region in the ISR. Long execution of critical sections in ISRs can block the invocation of higher priority interrupts, thereby increasing their interrupt latencies.

Table 5.1 illustrates a system with three interrupt service routines, ISR_1, ISR_2 and ISR_3, with priority levels ranging from high to low, respectively, and with ISR_3 being nonpreemptable. If ISR_1 is invoked within the critical section of ISR_2, it will be delayed because interrupts are disabled. This delay may be longer if ISR_2 takes the `if` branch and ISR_1 takes the `else` branch because these paths involve more cycles. Similarly, ISR_1 will be blocked if it is invoked while ISR_3 is executing because ISR_3 is nonpreemptable, and the latency of ISR_1 may be longer if the `else` branch is taken in ISR_3. It is thus reasonable to suggest that critical sections involving ISRs be kept short and simple to avoid leaving higher priority interrupts disabled for too long.

F2: Execution time of ISRs of higher priority interrupts. When multiple interrupts occur, another cause of increased interrupt latency is the length of ISRs with higher priority. Suppose a low priority interrupt is issued during the execution of an ISR with higher priority. The longer the execution time of the high priority ISR, the longer the latency of the low priority interrupt. This occurs because, typically, a low priority interrupt cannot preempt a higher priority interrupt. For example, in Table 5.1, ISR_1 is preemptable and does not contain critical sections. However, if ISR_2 is invoked while ISR_1 is executing

it will be delayed, because ISR_1 cannot be preempted by a lower priority interrupt. If the `else` branch of ISR_1 is taken, the delay for ISR_2 may be even longer.

Changes in hardware states including preemption delays due to cache can also affect the execution time of interrupts. In this work, we do not consider these factors because they can be implementation dependent. For example, the Atmega processor that we model does not use cache memory. However, to apply SIMLATTE to test for WCILs in systems that utilize cache memory, existing techniques such as *Evicting Cache Block-Union* [141] can be integrated with SIMLATTE to estimate delays due to cache preemption.

F3: Locations where interrupts occur. The occurrence of interrupts at particular program execution points can also impact interrupt latency. We summarize three cases. First, for a given non-interruptible path p in the main program or an ISR, interrupts that occur at an early execution point in p will be delayed longer. In Table 5.1, suppose the `else` branch is taken in the main function. An interrupt has a longer latency if it is issued at the starting point of the critical section (line 3) than if it is issued at later points.

Second, for a given path p in an ISR with high priority, if a low priority interrupt is issued at an early execution point in p, it will be delayed longer. In the example of Table 5.1, suppose the `else` branch is taken while executing ISR_1. In this case, ISR_2 can have a longer latency if it is issued at the entry of ISR_1 than when issued at later execution points.

Third, nested high priority interrupts can affect the latency of lower priority interrupts. In the example of Table 5.1, suppose that all three interrupts are issued at the critical section entry of the main function (line 3). ISR_1 is serviced first, then ISR_2 is serviced. Suppose that as ISR_2 executes, ISR_1 again preempts ISR_2, causing ISR_3 to wait longer. The worst case occurs when ISR_2 is frequently preempted by ISR_1 such that ISR_3 is never serviced. High interrupt frequency is a major cause of nested interrupts; thus, it is important to employ a system design involving minimum interrupt inter-arrival times, or the inverse, maximum interrupt frequency.

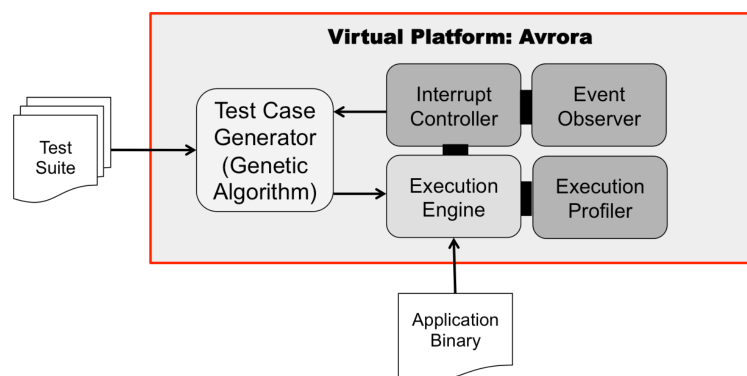

Figure 5.1: SimLatte Architecture

5.2 SimLatte

We now present SIMLATTE, an automated approach for finding WCILs, that leverages the three factors (F1, F2, and F3) just described, that is, the test cases generated by SIMLATTE manipulate these three factors to potentially produce longer interrupt latencies.

5.2.1 SimLatte's Architecture

SIMLATTE contains two main components: an *execution controller* and a *test case generator* – see Figure 6.2. SIMLATTE is implemented on AVRORA, a simulator platform that supports programmable event monitoring and application profiling [15]. We use the programming interfaces AVRORA provides to create two additional modules to support testing: an *interrupt controller* module and an *event monitoring* module. The test case generator and interrupt controller modules are configurable so that we can, for example, disable the GA portion of the test case generator and use random test generation instead. We can also disable opportunistic interrupts.

The input to SIMLATTE is an initial (possibly empty) test suite. The GA uses this as a starting point. As test cases are generated, they are executed to determine fitness (described later in this section). To do this the GA interacts with the execution controller. For

example, when a critical section is detected by the event monitoring module, the execution engine pauses the current program execution and requests the interrupt controller to invoke a specific number of interrupts. Upon completing a test execution, the test case generator is updated with new test cases. The profiler is then used to measure performance and latencies based on execution clock cycles.

5.2.2 SimLatte's Genetic Algorithm

SimLatte's genetic algorithm accepts three parameters: a time limit t_{iter}, a maximum interrupt latency l_{max} (optional), and constraints, C, that need to be enforced. The algorithm returns a set of new test cases NTC.

Chromosome. In SIMLATTE, the chromosome is a combination of input data (main program inputs and sensor device inputs) and an interrupt schedule (instruction locations) that consists of all interrupts in the program. A gene in the chromosome is denoted as a triple (`num`, `instr[cycle]`, `data`), where `num` is the interrupt number, `instr` is the instruction location where interrupt `num` is issued, and `data` is the sensor input data associated with interrupt `num`. The array following `instr` indicates that interrupt `num` is issued again after `cycle` cycles based on the previous invocation during a test execution.

Fitness Function. Our optimization problem is defined as *finding test cases to exhibit the worst case interrupt latency*. Thus, the goal of our fitness function is to maximize the interrupt latency of each *IUT* in P. The fitness function, f, is defined as $f = T_{isr} - T_{invok}$, where T_{isr} is the time at which the *IUT* is issued, and T_{invok} is the time at which the corresponding ISR for the *IUT* is invoked. Since the fitness of a test case depends on the interrupt latency of the *IUT*, this is calculated during the execution of test cases. Interrupts are dynamically issued for three conditions. First, when an interrupt location `instr` defined in the chromosome is reached, its associated interrupt is issued. Second, when a critical

section entry is encountered, the *IUT* and each interrupt with higher priority are issued. Third, when an *IUT* is pending, the algorithm checks whether a higher priority interrupt is available and if so, issues that interrupt. Note that the same interrupt can be issued multiple times per test run. SIMLATTE selects only the longest latency per test case. The instruction locations `instr` in the chromosome for interrupts issued at specific points are updated in terms of the longest latency.

Constraints. There are two constraints that must be adhered to in order for the test cases generated by SIMLATTE to be valid. First, the instruction location `instr` must be generated within the instruction range of the program text section. Second, the sensor data accepted by a device must conform to the range defined in the device specifications. For example, the sensor data accepted by the UART port must exist in the ASCII table. We enforce these constraints in the initial randomly generated population and subsequent generations that require new genes.

Some constraints cannot be determined statically; SIMLATTE checks these dynamically. First, SIMLATTE checks hardware states (e.g., interrupt bits) and issues only interrupts that are allowable by the hardware. Second, SIMLATTE avoids invoking reentrant ISRs by checking the current stack and suppressing an interrupt if its ISR has been invoked but has not returned. Third, whenever an interrupt is about to be issued, SIMLATTE checks its minimum inter-arrival time and determines whether it is allowable to issue this interrupt.

During the testing process all issued interrupts are serviced by the processor. As for the encoding of chromosomes, there are two cases considered in our approach. First, the interrupt does not require input data (e.g., periodic timer). In this case, both `instr` and `data` are set to `null` during the entire evolution process. Second, the interrupt requires input data (e.g., data arrives through UART). In this case, `instr` is set to null, but the input data is passed into the device when an interrupt is invoked by the processor.

Table 5.2: Initial Test Case Population

C.S.	Main	ISR_1	ISR_2	ISR_3	ISR_4	Fit.
tc1	128	(1,0x980c,'a')	(2,0x29bd,'d')	(3,0x792b,'m')	(4,0x8441,'b')	1597
tc2	...	...	...	...	...	1428
tc3	64	(1,0x75fa,'z')	(2,0x8412,'a')	(3,0x410c,'k')	(4,0x69bc,'m')	1433
tc4	...	...	...	...	...	1509
tc5	56	(1,0x22ad,'c')	(2,0x4890,'a')	(3,0x592d,'l')	(4,0x81da,'o')	1288
tc6	...	...	...	...	...	1410
tc7	225	(1,0x81bc,'f')	(2,0x7988,'n')	(3,0x8012,'x')	(4,0x8441,'p')	1402
tc8	...	...	...	...	...	1382

SimLatte GA Example. To illustrate SIMLATTE's GA we present an example. Let P = `Main` $\|ISR_1\|ISR_2\|ISR_3\|ISR_4$. The *IUT* is the interrupt associated with ISR_3. Table 5.2 shows the initial population consisting of eight randomly generated test cases. Fitness values are shown in the column labeled *Fit*.

After running these test cases, SIMLATTE selects the four best test cases in terms of fitness values; in the example, this includes the first four (shown in Table 5.3). At this point, the instruction locations of ISR_1, ISR_2 and ISR_3 have been updated to the entry instructions of the critical sections that cause the longest latency of interrupt 3 (shown in bold font).

Taking $tc1$ as an example, ISR_1 is invoked again after 1088 cycles of the execution of the critical section entry point. The reason for this is as follows. After ISR_1 returns, ISR_2 is invoked and returns. Since ISR_3 is still pending, SIMLATTE invokes ISR_1 again to further delay ISR_3. In this case, the inter-arrival time of ISR_1 is specified to be greater than or equal to 1088 cycles, otherwise it violates constraints.

Next, the algorithm performs a crossover between two pairs of chromosomes. In our example, pairing is done by matching evens and odds (i.e., 1 with 3, 2 with 4, etc.), so we cross $tc1$ and $tc3$. We use a 1-point crossover with a randomly selected division point in this example (crossover choice is flexible).

Table 5.4 shows the results of the crossover operation on two parent chromosomes and the generated offspring. The double line between the third and fourth columns indicates

Table 5.3: Result of Test Case Selection

C.S.	Main	ISR_1	ISR_2	ISR_3	ISR_4	Fit.
tc1	128	(1,**0x783c[1088]**,‘a’)	(2,**0x783c**,‘d’)	(3,**0x783c**,‘m’)	(4,0x8441,‘b’)	1597
tc4	...	...	...	...	...	1509
tc3	64	(1,**0x8010[3200]**,‘z’)	(2,**0x8010**,‘a’)	(3,**0x8010**,‘k’)	(4,0x69bc,‘m’)	1433
tc2	...	...	...	...	...	1428

Table 5.4: Result of a CrossOver Operation

C.S.	Main	ISR_1	ISR_2	ISR_3	ISR_4	Fit.
tc1	128	(1,0x783c[1088],‘a’)	(2,0x783c,‘d’)	(3,0x783c,‘m’)	(4,0x8441,‘b’)	1597
tc3	64	(1,0x8010[3200],‘z’)	(2,0x8010,‘a’)	(3,0x8010,‘k’)	(4,0x69bc,‘m’)	1433
ch1	128	(1,0x783c[1088],‘a’)	(2,0x8010,‘a’)	(3,0x8010,‘k’)	(4,0x69bc,‘m’)	1682
ch2	64	(1,0x8010[3200],‘z’)	(2, 0x783c,‘d’)	(3,0x783c,‘m’)	(4,0x8441,‘b’)	1569

Table 5.5: Result of a Mutation Operation

C.S.	Main	ISR_1	ISR_2	ISR_3	ISR_4	Fit.
tc1	128	(1,0x783c[1088],‘a’)	(2,0x783c,‘d’)	(3,0x783c,‘m’)	(4,0x8441,‘b’)	1597
tc3	64	(1,0x8010[3200],**‘k’**)	(2,0x8010,‘a’)	(3,0x8010,‘k’)	(4,0x69bc,‘m’)	1532
ch1	128	(1,0x783c[1088],‘a’)	(2,0x8010,‘a’)	(3,0x8010,‘k’)	(4,**0x604d**,‘m’)	1608
ch2	64	(1,0x8010[3200],‘z’)	(2,0x783c,‘d’)	(3,0x783c,‘m’)	(4,0x8441,‘b’)	1569

the crossover point. The non-shaded areas represent genes in the first parent ($tc1$), and the shaded areas represent genes in the second parent ($tc3$). The rightmost column shows the fitness values of the chromosomes.

Next, mutation is performed by altering either $data$ or $instr$ in a gene. Table 5.5 shows the results for four chromosomes after the mutation operator has been applied (mutated elements in bold). Following mutation, the four test cases are executed. The rightmost column shows the resulting fitness values.

5.3 Empirical Study

To assess SIMLATTE we explore three research questions.

RQ1: How does the effectiveness of SIMLATTE compare to that of a random testing technique?

RQ2: To what extent do the choices of using a GA to generate inputs and employing opportunistic interrupt invocation in SIMLATTE affect its effectiveness?

RQ3: How does the dependability of SIMLATTE in calculating WCILs compare to that of the other techniques considered?

The first research question evaluates SIMLATTE by comparing it to a state-of-the-art interrupt testing technique based on random inputs [5]. The second research question lets us further investigate whether the use of the GA and of opportunistic interrupt invocation can affect SIMLATTE's effectiveness. The third research question explores the extent to which we can depend on particular techniques to converge on or otherwise successfully calculate an appropriate WCIL.

5.3.1 Objects of Analysis

As objects of analysis we chose three embedded system applications. These include LARGE-DEMO, an open source program downloaded from GNU Savannah [142], HAND-MOTION CHESS, a student project developed in a microcontroller class at Cornell University [143], and SNUGGLZ, a student project in a graduate-level embedded systems course at the University of Nebraska - Lincoln. Table 5.6 lists these programs, the numbers of lines of non-comment code they contain, and the interrupt sources utilized by the programs with priorities ranging from *highest to lowest*. The *interrupt complexity* (the number of interrupts, critical sections, and branches within critical sections) of these programs ranges from lowest to highest across the three programs, respectively. The numbers in the columns cor-

Table 5.6: Object Program Characteristics

App.	*NLoC*	*Interrupt Source (frequency KHz)* In Decreasing Order of Priority INT1	INT2	M1	TM3	TM0	UT0	UT1	ADC	I2C
LARGEDEMO	341	-	-	-	-	0.1	19.2 (p)		125	-
CHESS	1550	800	800	1	0.05	-	-	-	-	-
SNUGGLZ	2373	-	-	-	-	0.06	19.2 (p)	19.2 (p)	125	400

responding to interrupts denote interrupt frequencies (the inverse of interrupt inter-arrival time). The notation "-" indicates that an interrupt is not utilized by the corresponding program. The notation "(p)" indicates that the interrupt is preemptable for the corresponding program; interrupts not so marked are not preemptable.

Our objects of analysis utilize various interrupt sources. `INT1` and `INT2` are external interrupt interfaces that can be used to attach non-built-in devices. `TM1` and `TM3` are "compare timers" that trigger two interrupts when each of the timers reaches two different compare values. `TM0` is an "overflow timer" that triggers an interrupt when it reaches its top value. `UT0` and `UT1` are UART devices used to receive and send ASCII data through UART ports; interrupts are triggered when data is available. `ADC` is an analog-to-digital converter, and triggers an interrupt when a new value is converted. `I2C` is a two-wire interface that sets communications between two devices; interrupts are generated based on their events. The timer interrupts (`TM1`, `TM3` and `TM0`) are *periodic interrupts* that are issued at a periodic interval. The other interrupts are *non-periodic interrupts* that can occur any time after being issued by their associated devices.

LARGEDEMO controls the brightness of an LED with a PWM (Pulse Width Modulation) output. Its `main` function accepts an operation mode (e.g., ADC, button, serial) and PWM values. HAND-MOTION CHESS simulates the physical (hand) motions involved in playing chess without the need for a physical chess set. Its `main` function performs actions based on the motion values sensed by two contact sensors that are used to sense the motion of

players (e.g., to determine the position of hand clicks on the chess board). INT0 and INT1 generate interrupts for the contact sensors. TM1 ensures a constant sampling interval of ADC conversion, and TM3 ensures that the cursor's position on the chess board is updated. SNUGGLZ implements code for motion detection by a hovercraft, and runs on microcontrollers with voltages 3.3V and 5V. Its main function (on the 5V processor) accepts three inputs as commands to control directions, and sends them to the 3.3V processor through the I2C bus for processing. UT0 and UT1 are a pair of ZigBee radios used to receive and send data. I2C sends commands from the 5V processor to the 3V processor. TM0 controls periodic task scheduling.

Our study focuses on interrupts under test (*IUTs*). We selected two interrupt sources for each of the three object programs. These include UT0 and ADC in LARGEDEMO, INT1 and TM1 in HAND-MOTION CHESS, and ADC and I2C in SNUGGLZ, and we denote them by L_{UT0}, L_{ADC}, H_{INT1}, H_{TM1}, S_{ADC} and S_{I2C}, respectively. We chose these interrupt sources because they encompass a wide range of *interrupt complexities*, including the number of higher priority interrupts, the number of lower priority interrupts with respect to an *IUT*, and the categories of interrupts (e.g., periodic or non-periodic) associated with an *IUT*. For example, L_{UT0} involves one higher priority and one lower priority interrupt with respect to UT0, H_{TM1} involves two higher priority interrupts and one lower priority interrupt with respect to TM1, and S_{ADC} involves four higher priority interrupts with respect to I2C. H_{TM1} also allows us to study periodic interrupts.

5.3.2 Setting GA Parameters

To investigate our research questions we required an implementation of SIMLATTE appropriate for our object programs. As GA parameters, we chose 32 as an initial population size, and 16 as the population size. The number of new test cases that results is 16. For se-

lection, we configured the algorithm to select the best half of the population from which to generate the next generation; the selected chromosomes are retained in the new generation. The chromosomes are ranked, and evens and odds are paired, to generate offspring. We configured the algorithm to perform a one-point crossover by randomly selecting a division point in the chromosome. Smith et al. [144] conclude that mutation rates considering both the length of chromosomes and population size perform significantly better than those that do not. Thus, we utilize a mutation rate of $\frac{1.75}{\lambda\sqrt{l}}$ as suggested by Haupt et al. [145], where λ is the population size and l is the the length of chromosome.

Our object programs do not specify fittest values (i.e., maximum WCILs) for each device. Thus, iteration limits govern the stopping points for SIMLATTE. In this study, we used time limits to control iteration limits. To compare the effectiveness of SIMLATTE to other techniques, we set a maximum time limit of 24 hours for all techniques including SIMLATTE; this lets us examine the relative effectiveness of the techniques when each is given the same amount of time. Because iteration limits can affect both the effectiveness and the cost of genetic algorithms, we further investigate the effects of iteration limits on SIMLATTE in Section 6.4.

5.3.3 Variables and Measures

Independent Variable. Our independent variable involves the techniques used to calculate WCILs. In addition to SIMLATTE, to address our first research question we consider a random testing approach, RANDOM, based on the approach presented in [5]. RANDOM neither applies a GA nor invokes opportunistic interrupts at the locations of interest; instead, it randomly generates test cases following static constraints, and randomly issues interrupts with the same interrupt density achieved by SIMLATTE. For example, if the average interrupt

density for device number 5 when using SIMLATTE is 8%, RANDOM randomly selects interrupt locations for device number 5 with interrupt density 8% during testing.

To address our second research question we consider two techniques. The first technique, SIMLATTE_{GA}, is used to evaluate the effects of applying a GA in SIMLATTE. SIMLATTE_{GA} is implemented based on SIMLATTE (Figure 6.2), except that it does *not* invoke opportunistic interrupts at locations of interest using the interrupt controller. Instead, it randomly invokes interrupts at the same density in which interrupts are issued by SIMLATTE. The second technique that we consider is SIMLATTE_{LOC}, and is used to evaluate the effects of employing opportunistic interrupt invocation in SIMLATTE. SIMLATTE_{LOC} utilizes SIMLATTE in that it issues interrupts at the locations of interest, but it does *not* invoke the GA test case generator to evolve test cases or to generate an initial population. Instead, it randomly generates test cases using static constraints.

To address our third research question we utilize all four of the foregoing techniques.

Dependent Variables. As dependent variables, we measure the *effectiveness* and *dependability* of the foregoing techniques. To measure effectiveness, we report the WCILs calculated by the techniques for a given amount of testing time. To measure dependability, we plot trend lines for each technique to show the WCILs estimated as testing time increases.

5.3.4 Study Operation

We implemented SIMLATTE and the other three techniques on AVRORA, by tailoring the algorithm described in Section 5.3.3. For each test case, we pass the sensor inputs to the devices. This is easy to control in the simulator because devices are all implemented as software. By utilizing the monitor features, we can monitor each instruction access such that whenever an interrupt location of interest is reached, this interrupt is issued. AVRORA provides an API `forceInterrupt`(*num*) that lets users force a specific interrupt to

occur. For constraint checking, AVRORA allows us to monitor hardware states (determine whether it is possible to issue an interrupt) and the return of an ISR (this prevents re-entrant interrupts). Calculating fitness values (interrupt latencies) for the GA can also be done using monitoring features.

All four techniques we study involve randomization. To control for variance due to randomization we ran each of the techniques five times. We did this separately on each of the six *IUTs* under consideration in the object programs (Table 5.7). In total, then, we conducted 120 runs (four techniques * six *IUTs* * five runs). We used a Linux cluster to perform the executions, distributing each job on a distinct node.

5.3.5 Threats to Validity

The primary threat to external validity for this study involves the representativeness of our object programs. Additional studies with other programs are needed.

The primary threat to internal validity for this study involves possible faults in the implementation of the algorithms and tools we used to perform the evaluation. We controlled for this threat by extensively testing our tools and verifying their results against a smaller program for which we can manually determine the correct results. We also chose to use the well-established Avrora simulator to implement our algorithm.

Where construct validity is concerned, there are other metrics that could be considered. Given tight implementations and environment controls, time could be measured. Costs of engineer time could also be considered.

5.3.6 Results and Analysis

Table 5.7 reports the effectiveness results observed (the WCILs estimated) in our study; we use this table to address our first two research questions. Results are shown per *IUT*

Table 5.7: Technique Effectiveness

Config.	*Techn.*	*WCIL (cycles)*				
		R1	R2	R3	R4	R5
L_{UT0}	SL	**15414**	**15414**	**15414**	**15414**	**15414**
	RAND	14984	14967	14192	14800	14360
	SL_{GA}	15324	15194	15223	15353	15106
	SL_{LOC}	**15414**	**15414**	**15414**	**15414**	**15414**
L_{ADC}	SL	**18925**	**18925**	18433	18433	18433
	RAND	15012	15048	15070	15012	15012
	SL_{GA}	15892	15730	15828	15904	15794
	SL_{LOC}	16782	16782	16782	16782	16782
H_{INT1}	SL	**20223**	20014	**20223**	**20223**	19938
	RAND	15235	15256	15279	15235	15235
	SL_{GA}	15312	15375	15328	15235	15235
	SL_{LOC}	20204	20204	20166	20184	20146
H_{TM1}	SL	5271	**5742**	5271	**5742**	**5742**
	RAND	2472	2468	2467	2472	2467
	SL_{GA}	4796	4647	4922	4994	4758
	SL_{LOC}	4708	4767	4841	4894	4967
S_{ADC}	SL	**113446**	104541	107044	109706	104257
	RAND	41038	14161	19690	44147	15353
	SL_{GA}	50978	61038	59536	61537	60998
	SL_{LOC}	91011	91011	91011	88304	91011
S_{I2C}	SL	149820	141929	143697	**149988**	140323
	RAND	35842	34161	28737	29752	25353
	SL_{GA}	84023	75313	86004	78144	78422
	SL_{LOC}	104602	108838	103800	104602	108838

(Column 1) and technique (Column 2). Columns 3-7 show the WCILs estimated by the four techniques on each of the five sets of runs, for each *IUT* and technique. The numbers rendered in bold font indicate the largest WCILs for each *IUT*, among all five sets runs, for all techniques.

5.3.6.1 RQ1: Effectiveness of SIMLATTE vs. RANDOM

As Table 5.7 shows, for each *IUT* and set of runs, SIMLATTE (SL) is more effective than RANDOM (RAND). On all six *IUT*s, when averaging the five sets of runs, SIMLATTE improved WCILs over the best runs of RANDOM by amounts ranging from 2.9% to 305.0%. The lowest level of improvement (2.9%) occurred on L_{UT0}, while the highest levels of (144.2% and 305.0%) occurred on S_{ADC} and S_{I2C}, respectively. These results further imply

that as the level of interrupt complexity increased, SIMLATTE was even more effective than RANDOM. There are two reasons for this. First, higher complexity is accompanied by a larger search space, which can amplify the relative effectiveness of the GA. Second, higher complexity indicates the presence of more potential critical sections and nested interrupts, which can amplify the effectiveness of using opportunistic interrupts.

5.3.6.2 RQ2: The Role of the GA and Opportunistic Interrupt Invocation in SIMLATTE

To address RQ2, we first compare the effectiveness of SIMLATTE$_{GA}$ (SL$_{GA}$) to that of SIMLATTE and RANDOM. As shown in Table 5.7, for each *IUT*, for each set of runs, SIMLATTE was more effective than SIMLATTE$_{GA}$. The improvement for individual *IUT*s with averaged sets ranged from 1.1% to 83.3%. The highest level of improvement occurred on S$_{ADC}$ and the lowest occurred on L$_{UT0}$. These results show that the use of opportunistic interrupts did impact the effectiveness of SIMLATTE in estimating WCILs. This impact was more obvious as the complexity level increased.

Comparing SIMLATTE$_{GA}$ and RANDOM, SIMLATTE$_{GA}$ was more effective in most of the sets of runs across all six *IUT*s. Only on sets of runs $R4$ and $R5$ on H$_{INT1}$ did RANDOM calculate the same WCILs as SIMLATTE$_{GA}$. Using average WCILs, across the five sets of individual *IUT*s, the effectiveness improvement achieved by SIMLATTE$_{GA}$ with respect to RANDOM ranged from 0.3% to 226.5%. Again, S$_{ADC}$ and S$_{I2C}$ exhibited the most substantial improvements, of 141.8% and 226.5%, respectively. These results indicate that the use of the GA amplifies the effectiveness of the WCIL calculation process when larger search spaces are present.

We next compare the effectiveness of SIMLATTE$_{LOC}$ (SL$_{LOC}$) to that of SIMLATTE and RANDOM. As Table 5.7 shows, on L$_{UT0}$, SIMLATTE$_{LOC}$ and SIMLATTE are equally effective for each set of runs. On H$_{INT1}$, on sets of runs $R2$ and $R5$, SIMLATTE$_{LOC}$ is

more effective than SIMLATTE. This suggests that SIMLATTE$_{LOC}$ can be as effective as SIMLATTE when smaller search spaces exist. On the other four *IUT*s, on each set of runs, SIMLATTE$_{LOC}$ is less effective than SIMLATTE. In fact, for the four *IUT*s, averaging the five sets, SIMLATTE improves WCIL estimation over SIMLATTE$_{LOC}$ by amounts ranging from 11.0% to 35.5%.

Comparing SIMLATTE$_{LOC}$ and RANDOM, for each set of runs and *IUT*, SIMLATTE$_{LOC}$ was more effective than RANDOM. The improvement for individual *IUT*s with averaged sets ranged from 5.1% to 335.2%. Again, S_{ADC} and S_{I2C} exhibited the greatest improvements (271.9% and 335.2%, respectively) while L_{UT0} exhibited the least.

Overall, these results indicate that both the use of the GA, and the use of opportunistic interrupt invocation, contributed to enhancing the effectiveness of techniques for calculating WCILs. For opportunistic interrupt invocation, however, this improvement was evident only in cases where programs had higher complexity.

5.3.6.3 RQ3: Dependability of Techniques

To address our third research question we rely on Figure 5.2, which displays trend lines observed for the four techniques during testing. We show results only for L_{UT0}, H_{INT1}, and S_{ADC}, which we consider to be representative regarding levels of complexity. The x-axes indicate testing time measured in minutes, and the y-axes indicate the WCILs calculated in the testing process in terms of cycles. For SIMLATTE, WCILs are plotted by averaging the five sets of runs. For the other three techniques, the data points used represent only a subset of the total data points; this is because SIMLATTE takes longer to execute each run so it generates fewer results within a given time. For these three techniques, we select the largest WCIL present among the five sets; thus, the graphs present the most favorable possible view of their operation. At the same time, using only a subset of data removes

clutter and renders a visualization that still illustrates the fluctuations among data points throughout the testing period.

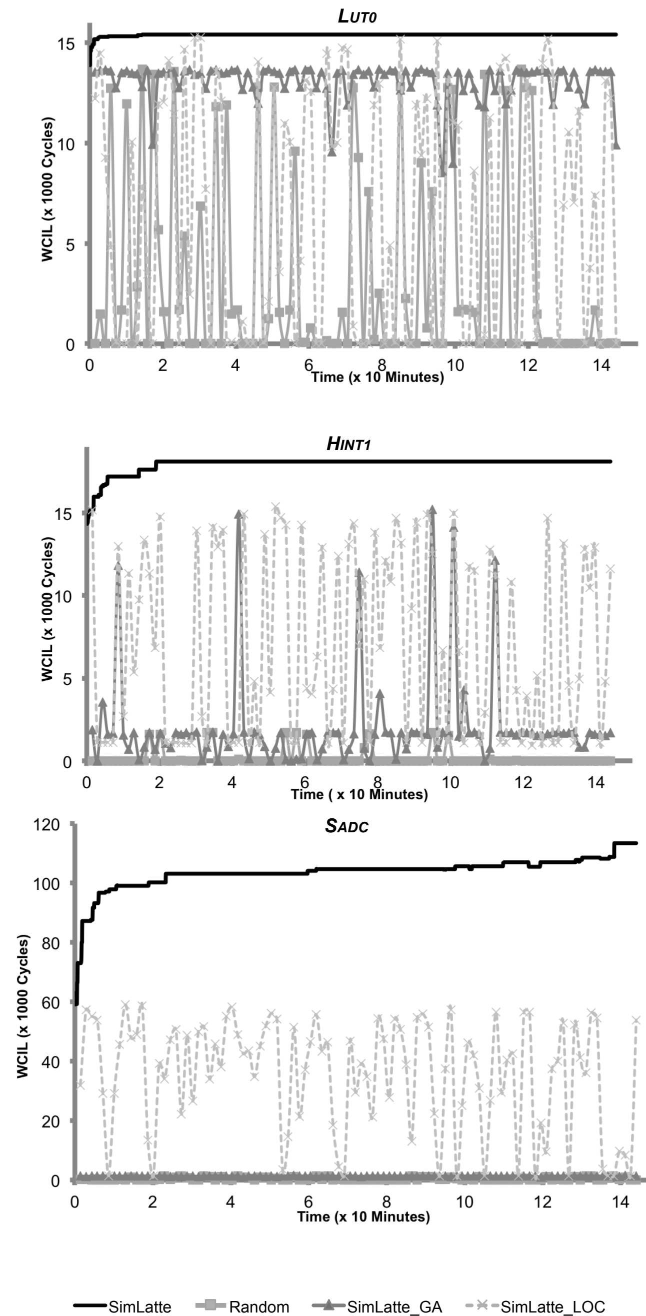

Figure 5.2: WCILs calculated during testing of L_{UT0}, H_{INT1} and S_{ADC}

The top graph in Figure 5.2 shows an example in which both SIMLATTE$_{LOC}$ and SIMLATTE achieved the same effectiveness (for L_{UT0}). Note, however, that SIMLATTE began to converge on an answer in this case after just 14 minutes. WCILs estimated by SIMLATTE$_{LOC}$, on the other hand, dramatically fluctuated across the testing time with the first maximum appearing at around 33 minutes. More generally, for each *IUT*, SIMLATTE increased its estimated WCILs until it reached a plateau. This indicates that by using SIMLATTE, we can potentially decide to stop testing early as results converge, making the testing process more efficient. The other three techniques, on the other hand, did not converge in any obvious manner, so identifying when to terminate the testing process when using them could be difficult.

Figure 5.2 also indicates that the complexity of the program under test can affect the time required for SIMLATTE to converge. For example, the convergence times for SIMLATTE on L_{UT0}, H_{INT1} and S_{ADC} are 14 minutes, 19 minutes, and 138 minutes, respectively. This implies that, as interrupt complexity increases, SIMLATTE takes longer to converge. However, it is worth noting that for S_{ADC}, SIMLATTE reached 88% of the estimated maximum WCIL (one million cycles) within the first 13% of its execution (18 minutes), whereas other approaches did not estimate WCILs of more than 60,000 cycles. This indicates that SIMLATTE yielded much smaller estimation errors in complex systems.

5.4 Discussion

We next examine the influence of several tunable parameters on the effectiveness of SIMLATTE. We then compare SIMLATTE to an approach for estimating WCILs based on static analysis.

Time Limits. We further examined our data to assess the effects of time limits on SIMLATTE. For each *IUT*, for each set of runs, we increased the time limit imposed on SIM-

LATTE by 10 hours. In the case of S_{ADC} and S_{I2C}, WCILs do appear to increase by 0.5% and 0.4% cycles as the time limit increases. All other cases fail to exhibit increasing trends; this indicates that our algorithm is converging in these cases. An implication of this discovery is that time limit matters more to programs with higher complexity; thus, longer testing times should be allocated to such programs.

Mutation Rate. To investigate whether the convergence of our GA was due to local optimums, we automatically tuned the mutation rate when the WCIL appeared to freeze for an hour. We did see effects from increasing the mutation rate.

For example, in the case of H_{INT1}, two out of five runs did not reach the best WCIL estimate (i.e., 20,223) without tuning the mutation rate; this occurred even when the time limit was increased. By increasing the mutation rate, however, we enabled all five runs to reach the best WCIL estimate. This case also occurred with L_{ADC} and H_{TM1}. In the case of S_{ADC}, increasing only the mutation rate did not have any effect. We conjecture that the algorithm requires more time to converge in cases where program complexity is higher. Therefore, we conducted additional experiments with 10 hours of additional testing time. In this case, we achieved new higher WCILs than without any change. When we increased both mutation rate and testing time, we achieved the highest WCIL. A similar case also occurred with S_{I2C}.

Interrupt Density. Interrupt density is the frequency at which interrupts occur in a test run. In our study, we use the same interrupt density for RANDOM and SIMLATTE. When using RANDOM, however, increasing interrupt density may enhance WCIL estimation effectiveness. This is especially true for systems employing one or more interrupts with higher priorities than an *IUT* because a high level of interrupt density may increase the likelihood that nested interrupts occur.

Table 5.8: Interrupt Density of Random Testing

Density	L_{UT0}	L_{ADC}	H_{INT1}	H_{TM1}	S_{ADC}	S_{I2C}
8%	14984	15070	15279	2472	44147	35842
30%	15012	19900	19224	2686	49554	37932
50%	14597	18494	19224	2762	48946	26847
70%	8332	9651	14383	1763	25633	12882

To further investigate the effectiveness of SIMLATTE, we compare its result to those of RANDOM at densities of 30%, 50% and 70%. Table 5.8 shows the results of RANDOM under four levels of interrupt density. The first level in each *IUT* is the value used in our study to answer the three RQs in Section 5.3.6. The numbers indicate the largest WCILs for each *IUT*, among all five sets of all levels of interrupt density. The effectiveness of RANDOM does increase as the interrupt density increases to a certain level, and then it begins to decline. This occurs because multiple pending interrupts under a heavy interrupt load can cause the CPU to select only the highest priority interrupt, which may cause lower priority interrupts (including *IUT*) to starve and lose data. Even at peak effectiveness, the best WCIL estimates obtained by RANDOM are far worse than those obtained by SIMLATTE.

These results further suggest that appropriately adjusting interrupt density may enhance the effectiveness of random techniques for estimating WCILs. However, blindly increasing interrupt density may also decrease technique effectiveness. Such an observation also confirms a point made by Regehr et al. [5]; namely, that in random testing, interrupts should be neither too sparse nor too dense.

Static Analysis. To measure the accuracy of the WCILs estimated by SIMLATTE, we compared its results to those estimated by BOUND-T, a commercial static analysis tool that analyzes Atmel AVR machine code to compute worst case execution times [146]. We tailored the tool to estimate WCILs for the five *IUT*s of our object program. We did this by using BOUND-T to compute the WCETs of all ISRs and the WCETs of the critical sections in both `Main` and the ISRs. Next, we computed the longest delay based on all

Table 5.9: WCILs via SIMLATTE and Static Analysis

Techniques	L_{UT0}	L_{ADC}	H_{INT1}	H_{TM1}	S_{ADC}	S_{I2C}
SIMLATTE	15414	18925	20223	5742	113446	149988
STATIC ANAL.	15414	18925	20223	15414	147548	173954

possible combinations of the critical sections, interrupt sequences, and minimum interrupt inter-arrival times. For example, $\text{WCIL}_{H_{INT1}}$ is the largest WCET among all WCETs in the critical sections in `Main` and $\text{WCET}_{H_{INT0}}$. For $\text{WCIL}_{S_{ADC}}$ and $\text{WCIL}_{S_{I2C}}$, the minimum interrupt inter-arrival time for `UART0` is considered because `UART0` can be nested in `UART1`.

Table 5.9 shows the WCILs calculated by both SIMLATTE and the static analysis approach. In general, code-level static analysis is conservative, and therefore can yield inaccurate WCIL estimates. For L_{UT0}, L_{ADC} and H_{INT1}, however, SIMLATTE estimated the same results as those computed by the static analysis tool. These results indicate that SIMLATTE can achieve the same effectiveness as static analysis for programs of lesser complexity. This is because interrupt invocation patterns in such systems are straightforward and can be accurately computed by the static analysis tool.

We measured the amount of time required to estimate WCILs on our object programs using static analysis; in no case did the analysis require more than one minute, and thus, the analysis was more efficient than the analysis conducted by SIMLATTE. On the three object programs just discussed, static analysis is thus more cost-effective. However, on the other three object programs, static analysis, while efficient, is much less effective than SIMLATTE.

For H_{TM1}, the result of static analysis is much higher than the estimates produced by SIMLATTE. The static analysis estimate tool assumes that `TIMER1` is invoked at the beginning of the critical section. During testing, however, this particularly rare scenario never occurs. When we shorten the period of `TIMER1`, however, the WCIL estimated by

SIMLATTE becomes longer. This occurs because the interrupt density for TIMER1 is now higher.

For S_{ADC} and S_{I2C}, static analysis estimated a higher WCIL than that estimated by SIMLATTE. On further examination of the program, we ascertained that the static analysis approach overestimated WCIL for two reasons. First, when I2C is enabled, both Main and UART0 have data dependencies with I2C. For example, when the WCET path is selected in MAIN with a shared variable defined in I2C, the WCET path in UART0 cannot be executed because it also has a dependency with this shared variable. Second, the static analysis approach does not consider data dependencies between ISRs in UART0 and UART1. In this case, the interrupt invocation pattern is quite complex; thus, the global interrupt bit is frequently disabled and enabled to keep the entire system working properly. We believe that these events can be observed only at runtime.

This result leads to two possible conclusions. First, a WCIL computed by static analysis may be based on a path that is infeasible in a system *IUT*. As such, the result could be an overapproximation. Second, test inputs are not sufficiently adequate to exercise the WCIL computed by static analysis – a main limitation of testing. As such, the result could be an underapproximation. As we lower the interrupt period, we can force the WCIL to go up. However, the value we test may not be applicable to the actual system (e.g., the period is too short). We leave further investigation as future work.

5.5 Conclusion

In this chapter, we have presented SIMLATTE, another instantiation of the SIMEXPLORER framework, that helps detect interrupt latency faults by determining worst-case interrupt latencies. Our empirical study shows that SIMLATTE can be more effective and dependable for calculating WCILs than state-of-the-art random testing. Further, by investigating

the two components of SIMLATTE (opportunistic interrupt invocation and the genetic algorithm), our results show that each component contributes to the effectiveness and dependability of the approach. Finally, we show that SIMLATTE can produce more precise WCILs than static analysis.

Chapter 6

SimRT: Automatic Regression Testing for Data Races [5]

As mentioned in Chapter 1, traditional regression testing techniques for dynamic race detection may not be cost-effective. To address this problem, in this chapter, we propose SIMRT, an automated regression testing technique for use in detecting races that are induced in concurrent programs by code modifications. SIMRT identifies variables that can be accessed by multiple threads in a modified program, and that are impacted by modifications. SIMRT then employs a regression test selection technique to select the test cases from the program's regression test suite that exercise these shared variables in a manner that involves more than one thread. It is these test cases that are specifically relevant to detecting races.

While regression test selection targeting shared variables helps SIMRT reduce the cost of regression testing, the testing process is still unnecessarily burdened by the cost of dynamic race detection approaches, because different inputs tend to repeatedly execute the same memory locations and interleavings. Deng et al. [75] observe that up to 88% of test

[5] The contents of this chapter have appeared in [147].

inputs always execute the same shared memory locations and interleavings and thus do not increase race detection rate as test cases execute. Thus, SIMRT next applies a greedy test case prioritization algorithm to schedule the test cases selected in its prior phase in an order that detects races faster.

To assess SIMRT, we conducted an empirical study in which we applied the approach to modified versions of nine concurrent source code objects, all of which contain data races that are the result of code modifications. We assessed both the regression test selection and test case prioritization components of our approach by comparing SIMRT to traditional and baseline regression test selection and test case prioritization techniques. Our results show that SIMRT is more efficient than a common baseline regression test selection technique, and substantially more efficient than the default technique of executing all test cases, while retaining the effectiveness of those techniques. Our results also show that the prioritization of test cases by SIMRT substantially increases the rate at which races are detected with respect to a common baseline prioritization technique, and a random ordering of test cases.

6.1 Background

6.1.1 Race Detection and Verification

As noted in Chapter 2.2, many static and dynamic analysis techniques have been developed to detect data races. In this work, we consider dynamic race detection techniques. Dynamic race detection techniques based on lock-set and vector-clock algorithms can report false positives, because they cannot guarantee that a race can really occur under specific thread interleavings. We refer to the races reported by these algorithms as *potential races*. Techniques have also been proposed to determine whether potential races identified by a detector can actually occur. We refer to races that have been verified in this way as *real*

races. However, even race verifiers cannot distinguish between harmful races and benign races; they report any pair of unsynchronized accesses (at least one of which is a write) as a race.

RACEFUZZER [6] is a race detection tool that combines dynamic race detection and race verification. RACEFUZZER computes pairs of program instructions that could *potentially* race during concurrent execution. It then *randomly schedules* the program under test based on the computed instruction pairs to allow *real* racing events to be placed temporally next to each other.

Employing random schedules for race verification is precise and-cost effective, but can be incomplete [6, 109]. Random scheduling does not guarantee that verification can distinguish all real races from potential races; rather, it guarantees that if it can cause a race to occur, this potential race will become a real race. On the other hand, complete replay techniques such as PENELOPE [118] may incur much higher overhead due to context switches.

6.1.2 Testing for Races

Software testing requires test inputs. Given a set of test inputs, the race detection process for a multi-threaded program involves two steps. First, for each test input, the program is executed by a dynamic race detection tool that identifies potential races (race detection). Second, for any input t that produces a set of potential races $PRaceSet$ in the first step, and for each $pr \in PRaceSet$, the program is executed n ($1 \leq n \leq N$) times to verify pr under t, where N is defined by users in terms of a testing budget (race verification). We define $N = 1$ for SIMRT to simulate a resource-constrained testing environment.

Our SIMRT approach uses RACEFUZZER to test for races by following the foregoing two steps. However, in the race verification phase, RACEFUZZER verifies every potential

race reported in the race detection phase for the given input regardless of whether this race has already been verified by previous inputs. This can be expensive when given large test suites, particularly when each test case reports redundant potential races in the race detection phase. To alleviate this problem, we adjusted RACEFUZZER so that, once a potential racing pair is confirmed to be a real race, it is not verified again under another input. Specifically, for each test input, SIMRT first invokes RACEFUZZER and reports potential races. If there exist any potential races that have not been confirmed as real races, SIMRT invokes RACEFUZZER to verify each of them, under the same test input, before proceeding to the next test input. As such, the actual number of test runs for an entire test suite T is:

$$TR = |T| + \sum_{i=1}^{|PRaceSet|} NT_i$$

Here, $|T|$ is the total number of original tests, $|PRaceSet|$ is the number of potential races, and NT_i is the number of tests required to verify the ith pair in $PRaceSet$.

6.2 Approach

Figure 6.1 contains an example that we use to illustrate our approach. The figure provides code snippets from two versions of the JDK's `HashMap` utility (slightly modified, and referred to here as `HM`). The variables `table`, `tab[i]`, `next` and `modCount` are identified as shared variables by thread escape analysis (see Section 6.2.2). A test driver for this code instantiates an `hm` object from class `HM`. The two parameters in the constructor of `HM` specify the initial capacity and load factor. Each of the test cases for the code involves two components, each executing in its own thread. We define four test case components:

$case_0$: hm.clear()

$case_1$: hm.containsValue(new HM(100, 10.0f));

```
1 public boolean containsValue(Object value)
      {
2        Entry tab[] = table; //block ID: 1
3        if(value == null){
4           for (int i=tab.length; i-- >
               0;) //block ID: 2
5                 ...
6        }
7        else {
8                 ...
9                 return true; //block ID
                    : 4
10       }
11       return false;
12 }
13
14
15
16
17
18
19
20
21 public void clear() {
22       ...
23       synchronized(this){
24       Entry tab[] = table; //block ID: 3
25       for (int i=0; i<tab.length; i++)
26          tab[i] = null; //block ID: 4
27       }
28       //block ID: 5
29       ...
30 }
```

```
1 public boolean containsValue(Object value)
      {
2        Entry tab[] = table;
3        if(value == null){
4           for (int i=0; i<tab.length; i
               ++) //change
5                 ...
6        }
7        else {
8           ...
9           return containsNull(); //change
10       }
11       return false;
12 }
13
14 private boolean containsNull() {
15       Entry tab[] = table;
16       for (int i=0; i<tab.length ; i++)
17          for (Entry e=tab[i]; e!=null; e
               =e.next)
18          ...
19    }
20
21 public void clear() {
22       ...
23       synchronized(this){
24       Entry tab[] = table;
25       for (int i=0; i<tab.length; i++)
26             tab[i] = null;
27       }
28       modCount++; //change
29       ...
30 }
```

Figure 6.1: Original HM program P (left) and a modified version of the HM program P' (right)

case$_2$: hm.containsValue(new HM(0, 100.0f));

case$_3$: hm.containsValue(null)

A test case tc_i is denoted by (m, n) ($0 \leq m, n \leq 3$), where m and n denote two of the foregoing cases (i.e., $case_m$ and $case_n$).

Suppose there are six test cases generated for version P of `HM`: $tc_1 = (0, 1)$, $tc_2 = (0, 2)$, $tc_3 = (0, 0)$, $tc_4 = (0, 3)$, $tc_5 = (1, 2)$, and $tc_6 = (3, 3)$. In version P' of `HM`, there are two changes that cause three races to occur. The first change involves addition of a new method `containsNull()` and a call to it from line 9. This change causes a read-write

race involving `tab[i]` at line 17 and `tab[i]` at line 26 when executing tc_1 or tc_2 (i.e., concurrently executing `containsNull()` and `clear()`). The second change involves addition of a new line of code (line 28) that reads and writes a class variable `modCount`. Two data races occur in this case, involving `modCount` when tc_3 is executed; one is a write-write race, and the other is a read-write race.

6.2.1 Overview of SimRT

Figure 6.2 provides an overview of SIMRT. SIMRT contains five components: *SVLocator*, *ImpAnalyzer*, *Matcher*, *Selector*, *Ranker*. Let P be a program, let P' be a modified version of P, let C' be the set of changes made to P to produce P' (a set of program locations in P'), let B'_{SV} denote a block in a method in P' that contains a shared variable SV, and let B_{SV} be a block in a method in P that corresponds to B'_{SV}. SIMRT first computes a list of shared variables L'_{SV} in P' using *SVLocator*. Next, *ImpAnalyzer* updates L'_{SV} by mapping B'_{SV} to B_{SV}, and identifying the shared variables in L'_{SV} that are potentially impacted by one of the changes C' in P'. Next, *Matcher* iterates over each SV in L'_{SV}, and selects the SVs that can be matched into pairs. The output of *Matcher* is a list of impacted shared

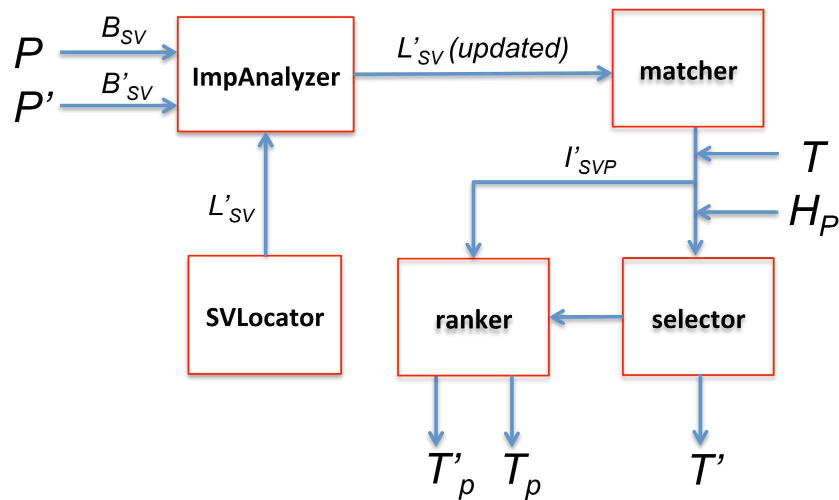

Figure 6.2: Overview of SimRT

variable pairs (I'_{SVP}) that are coverage targets for RTS and TCP techniques. Next, *Selector* selects $T' \in T$ for use in regression testing P'. Finally, *Ranker* prioritizes the test cases in T' (or T), producing T'_P (or T_P). Both *Selector* and *Ranker* utilize coverage history information provided as H_P. We describe each of these components in the sections that follow.

6.2.2 Shared Variable Identification

SVLocator produces a list of variables L'_{SV} that can be accessed by multiple threads in P'. In Java, shared variables can be identified by using thread escape analysis [104, 148]; we adopted the conservative shared variable detection algorithm proposed in [149], that uses the `ThreadLocalObjectAnalysis` API [150] provided by SOOT [151] to compute variables that can potentially be read from and written to by multiple threads simultaneously. We also used the alias analysis [152] provided by SOOT to identify a set of escaping variables that can potentially access the same location.

Each shared variable (SV) is defined as a 7-tuple $< C.M, N, D, L, A, B, I >$ where C is a class name, M is a method signature, N is the name of the SV, D is a memory access identifier (SVs that potentially access the same memory location share the same identifier), L is a line number at which the SV occurs, A denotes the access operation (read or write) performed on the SV, B is the block ID (a unique number in $C.M$) in P where SV is mapped to, and I is a boolean value indicating whether SV is impacted. Note that $C.M$ must exist in both P and P'. If a method M' in P' in which a SV occurs does not exist in P, we locate, in the call graphs for P', the method that most directly calls M' that also exists in P, and, if no such method is found, $C.M$ is defined as $\perp$.

In the HM program, shared variables in P' are displayed as the following initialized tuples:

SV_1:<HM.containsValue(Object), table, 1, 2, read, ⊥, ⊥>

SV_2:<HM.containsValue(Object), table, 1, 15, read, ⊥, ⊥>

SV_3:<HM.containsValue(Object), tab[i], 2, 17, read, ⊥, ⊥>

SV_4:<HM.containsValue(Object), next, 3, 17, read, ⊥, ⊥>

SV_5:<HM.clear(), table, 1, 24, read, ⊥, ⊥>

SV_6:<HM.clear(), tab[i], 2, 26, write, ⊥, ⊥>

SV_7:<HM.clear(), modCount, 4, 28, read, ⊥, ⊥>

SV_8:<HM.clear(), modCount, 4, 28, write, ⊥, ⊥>

Here, <HM.containsValue(Object), table, 1, 15, read, ⊥, ⊥> indicates that in line 15 in P', variable table is read, with access identifier 1. The method HM.containsNull() in which table occurs does not exist in P; thus, HM.containsValue(Object) is filled into $C.M$. Elements B and I are not specified until the comparison algorithm (Section 6.2.3) is invoked.

6.2.3 Shared Variable Impact Analysis

Figure 6.3 and Figure 6.4 displays the algorithm used by IMPANALYZER. The algorithm takes program P, modified version P', and a list of shared variables L'_{SV} with initialized tuples as inputs, and returns the updated L'_{SV}. First, the algorithm constructs control flow graphs (CFGs) G and G' for all methods in P and P' (line 4). Each node in a CFG is represented using a unique block ID. Next, the algorithm compares each CFG G in P to the corresponding G' in P' by calling *Compare* (line 6). The comparison begins with entry nodes E and E'. Given two CFG nodes N and N', *Compare* determines whether N and

N' have successors (S and S') whose code differs along pairs of identically labeled edges (lines 14-20, 23).

Up to this point, *ImpAnalyzer* behaves identically to the original DEJAVU algorithm [17]. However, *ImpAnalyzer* uses different mechanisms to handle node comparisons. *ImpAnalyzer* obtains a set of line numbers in block node S' (line 22). If the code associated with S and S' is the same (line 23), the algorithm iterates over L'_{SV}, and picks the SVs that are contained in S' (line 24-26). For each of these SVs, tuple element B is set to the block ID of S (line 27), and tuple element I is set to $false$ (line 28), indicating that this shared variable is not impacted.

---

procedure *UpdateSVInfo*

1: **Inputs:** P, P', L'_{SV}

2: **Outputs:** L'_{SV} for P' /*updated*/

3: **begin**

4: construct CFGs for P and P'

5: **for** each $G \in CFG$ and $G' \in CFG'$

6: `Compare(`E, E'`)` /*start from entry nodes*/

7: **endfor**

8: return L'_{SV}

9: **end**

Figure 6.3: ImpAnalyzer algorithm

```
procedure Compare
10: Inputs: N, N' /*nodes in G and G'*/
11: Outputs: L'_SV for G' /*updated*/
12: begin
13: isVisted(N) = true /*whether N is visited*/
14: for each successor S of N ∈ G
15:     if (N, S) is labeled
16:         L = the label on edge (N, S)
17:     else
18:         L = ε
19:     endif
20:     S' = the node in G' such that (N', S') has label L
21:     if isVisited(S) is false
22:         LNS_S' = getBlockLineNumbers(S')
23:         if LEquivalent(S, S') /*if code is equivalent*/
24:             for each SV ∈ L'_SV
25:                 ln_SV = getSVLineNumber(SV)
26:                 if ln_SV ∈ LNS_S'
27:                     setOldBlockID(SV, S)
28:                     setIsImpacted(SV, false)
29:                     break
30:                 endif
31:             endfor
32:             Compare (S, S')
33:         else
34:             for each SV ∈ L'_SV
35:                 if isImpactedBy(SV,S')
36:                     setOldBlockID(SV, S)
37:                     setIsImpacted(SV, true)
38:                     break
39:                 endif
40:             endfor
41:         endif
42:     endif
43: endfor
44: end
```

Figure 6.4: ImpAnalyzer algorithm (cont'd)

If the code associated with S and S' differs along some pair of identically labeled edges, *ImpAnalyzer* iterates over L'_{SV} and determines the impacted shared variables (line 34-35). If a shared variable SV is impacted by the change, tuple element B is set to old block ID S (line 36), and tuple element I is set to $true$ (line 37), indicating the existence of impact. A shared variable is considered impacted (line 35) in the following cases:

1. If a change to P' involves an SV in P' and this SV is either added to or modified in P', an SV is considered impacted.
2. If a change to P' involves adding or modifying a method, all SVs in this method are considered impacted.
3. If a change to P' involves a synchronization statement, all the SVs in the entire region of the synchronized block are considered impacted. For example, if line 23 in Program P is omitted, all shared variables inside the synchronization block (SV_5 and SV_6) are considered impacted. In addition to considering blocks using the synchronized keyword, we also consider blocks using explicit locks, including `lock()`, `unlock()`, and `trylock()`, and locks implementing a ReadWriteLock interface (`readLock()`, `writeLock()`). We do not consider changes involving `wait()`, `notify()`, or `notifyAll()`; these must be called inside a synchronized block [153], and changing them does not affect SVs in synchronized blocks.
4. If a change to P' involves removing a volatile keyword in the declaration of an SV, all subsequent uses of this SV are considered impacted.
5. If a change to P' does not involve an SV but its full impact set in P' (obtained by traditional impact analysis such as static forward slicing) includes one or more SVs, these SVs are considered impacted.

In the example shown in Figure 6.1, IMPANALYZER compares the method `containsValue(Object)` in P and P'. The algorithm begins by visiting the node with block ID

1 in P (lines 2-3) and compares it to its corresponding node in P'. The two blocks are equal, so the algorithm updates tuple elements B and I for all SVs in lines 2-3 of P'. Thus, <HM.containsValue(Object), table, 1, 2, read, $\perp$, $\perp$> is set to <HM.containsValue(Object), table, 1, 2, read, 1, false>. Here, $B = 1$ is the block ID in P that table is mapped to, and $I = false$ indicates that table is not impacted.

Next, IMPANALYZER compares the nodes at line 4 in both P and P'. Although line 4 is changed in P', the change does not impact any shared variables. Thus, the comparison proceeds to the next node in P. When IMPANALYZER visits the node corresponding to block 4 in HM.containsValue(Object) in P, it discovers that line 9 in P' is changed by addition of the call to method containsNull(). Thus, all SVs in this method are considered impacted. For example, <HM.containsValue(Object), next, 3, 17, read, $\perp$, $\perp$> is set to <HM.containsValue(Object), next, 3, 17, read, 4, true>.

After the algorithm returns, the two undefined elements in each SV are noted. In the example shown in Figure 6.1, the list of shared variables is updated as follows:

SV_1: <HM.containsValue(Object), table, 1, 2, read, 1, false>,

SV_2: <HM.containsValue(Object), table, 1, 15, read, 4, true>,

SV_3: <HM.containsValue(Object), tab[i], 2, 17, read, 4, true>,

SV_4: <HM.containsValue(Object), next, 3, 17, read, 4, true>,

SV_5: <HM.clear(), table, 1, 24, read, 3, false>,

SV_6: <HM.clear(), tab[i], 2, 26, write, 4, false>,

SV_7: <HM.clear(), modCount, 4, 28, read, 5, true>,

SV_8: <HM.clear(), modCount, 4, 28, write, 5, true>.

6.2.4 Matching Shared Variables

Because race detection involves pairs of shared variable accesses, the next step is to use *Matcher* to identify a set of impacted shared variable pairs I'_{SVP} serving as coverage targets for the *Selector* and *Ranker* modules. We define an impacted shared variable pair $ISVP$ as a pair of shared variables SV_i and SV_j, such that at least one of them is impacted, and at least one of them is a write access.

To illustrate, *Matcher* takes the updated shared variable list L'_{SV} computed in Figure 6.4 as input, and outputs the potential impacted shared variable pairs I'_{SVP}. For each shared variable SV_i in L'_{SV}, if SV_i is impacted (determined by querying element I in the SV_i tuple), the algorithm iterates over L'_{SV} to identify each SV_j in L'_{SV} that is shared with SV_i, regardless of whether SV_j is impacted. As a result, each qualified $ISVP$ = (SV_i, SV_j) is added to I'_{SVP}. In our example, five variables are impacted (SV_2, SV_3, SV_4, SV_7, and SV_8), but only SV_3, SV_7, and SV_8 qualify for pairing, so I'_{SVP} = $<$(SV_3, SV_6), (SV_7, SV_8), (SV_8, SV_8)$>$.

6.2.5 Regression Test Selection

The next step is to use *Selector* to select test cases that are relevant to race detection. A naive approach is to select every test case that traverses any $(SV_i, SV_j) \in I'_{SVP}$. However, this may include test cases that execute shared variables but involve only one thread. Instead, SIMRT selects test cases by traversing the tuple elements B of SV_i and SV_j involved in different threads. To facilitate this, when collecting coverage information for each block in the original program P, which is done during the preliminary phase of testing, we also record the thread IDs that cover each block. As such, the constructed test history indicates (1) whether a block is covered, and (2) the IDs of the threads that cover this block.

```
procedure TestSelection
45: Inputs: T, H_P, I'_SVP
46: Outputs: T'
47: begin
48: T' = φ
49: for each (SV_i, SV_j) ∈ I'_SVP
50:       B1 = getOldBlockID(SV_i)
51:       B2 = getOldBlockID(SV_j)
52:       for each tc ∈ T
53:            if H_P(tc, B1) = true and H_P(tc, B2) = true
—:                        and H_P(tc, Tid_B1, Tid_B2) = true
54:                 T' = T' ∪ tc
55:            endif
56:       endfor
57: endfor
58: end
```

Figure 6.5: Regression test selection algorithm

Figure 6.5 shows the *Selector* algorithm. The algorithm takes three inputs: test suite T, the impacted shared variable pairs I'_{SVP}, and the test coverage history that indicates which test cases in T covered which statements in P with which thread. H_P is denoted by tc_i = $<(C.M,\ B,\ TID)>$, where tc_i is the test case with number i, $C.M$ is the class name combined with the method signature, B is the block ID that tc_i covers, and TID is the ID of the thread that covers B. For each shared variable pair (SV_i, SV_j) in I'_{SVP}, the algorithm obtains the matching block IDs $B1$ and $B2$ for SV_i and SV_j (line 50-51) as coverage targets. Based on the coverage history H_P, the algorithm selects all test cases from T that traversed $B1$ and $B2$ (the first two conditions in line 53) with different threads (the third condition in line 53), and adds them to T' (line 54).

In our example, suppose the coverage history H_P holds the following coverage information for each of the six test cases:

tc_1 = <(HM.clear(), 3, 1), (HM.clear(), 4, 1), (HM.clear(), 5, 1), (HM.containsValue(Object), 1, 2), (HM.containsValue(Object), 4, 2)>,

tc_2 = <(HM.clear(), 3, 1), (HM.clear(), 4, 1), (HM.clear(), 5, 1), (HM.containsValue(Object), 1, 2), (HM.containsValue(Object), 4, 2)>,

tc_3 = <(HM.clear(), 3, 1), (HM.clear(), 4, 1), (HM.clear(), 5, 1), (HM.clear(), 3, 2), (HM.clear(), 4, 2), (HM.clear(), 5, 2)>,

tc_4 = <(HM.clear(), 3, 1), (HM.clear(), 5, 1), (HM.containsValue(Object), 1, 2), (HM.containsValue(Object), 2, 2)>,

tc_5 = <(HM.containsValue(Object), 1, 1), (HM.containsValue(Object), 4, 1), (HM.containsValue(Object), 1, 2), (HM.containsValue(Object), 4, 2)>,

tc_6 = <(HM.containsValue(Object), 1, 1), (HM.containsValue(Object), 2, 1), (HM.containsValue(Object), 1, 2), (HM.containsValue(Object), 2, 2)>.

In this case, both tc_1 and tc_2 cover one target, (SV_3, SV_6), with different threads, and tc_3 covers two targets, (SV_7, SV_8) and (SV_8, SV_8), with different threads. Test cases tc_5, and tc_6 do not cover any targets. Test case tc_4 covers targets (SV_7, SV_8) and (SV_8, SV_8) but with only one thread. Therefore, $T' = < tc_1, tc_2, tc_3 >$. In contrast, most traditional RTS techniques would select all six test cases for T' because they all cover changed blocks. By running tests tc_1, tc_2 and tc_3 we can detect all three races, while tc_4, tc_5 and tc_6 do not help expose races.

Note that we select both tc_1 and tc_2 even though they cover the same target. The reason for this is because covering (executing) the two shared variables in the target pair under one test input is not necessarily sufficient to determine whether they access the same memory addresses [3]. Different inputs could cause different states to exist in the same code region,

causing two instructions in the target pair to access different memory locations under one input and the same memory location under a different input.

6.2.6 Test Case Prioritization

Ranker prioritizes test cases, beginning with those identified by *Selector*. *Ranker* iterates over T', selects the test case tc_i that covers the most impacted shared variable pairs ($ISVP$s) in I'_{SVP}, and places tc_i in T'_P. Next, *Ranker* iteratively selects the test case tc_i that covers the most $ISVP$s and appends tc_i to T'_P. When multiple test cases cover the same number of $ISVP$s, *Ranker* chooses one randomly. If each $ISVP$ has been covered by at least one test case, and the remaining unprioritized test cases cannot add additional coverage, *Ranker* resets the coverage vectors for all unprioritized test cases to their initial values, and reapplies the prioritization approach, ignoring previously prioritized test cases.

In our example program, tc_1 and tc_2 cover (SV_3, SV_6) with different threads, and tc_3 covers (SV_7, SV_8) and (SV_8, SV_8) with different threads. *Ranker* first places tc_3 in T_P. Because tc_1 and tc_2 achieve the same additional coverage, the algorithm randomly selects one of these (say, tc_1) and appends it to T'_P. Next, the algorithm resets the coverage data. As such, tc_2 becomes the test case that covers the most $ISVP$s and is appended to T'_P. Therefore, the output of *Ranker* is $T'_P = < tc_3, tc_1, tc_2 >$. When running T'_P, all three races can be exposed by the first two test cases.

As noted in Section 2.1.3, the safety of RTS techniques depends on certain conditions, and these include the condition that the program under test be executed deterministically. This condition cannot generally be met for the class of programs that we consider, and thus, SIMRT may omit test cases from T' that could reveal races in P'. In this context, it is important to note that even a retest-all approach can "omit" race-revealing test cases, because a given test case may or may not expose a fault in a given run when non-determinism

affects thread behavior. The motivation for selecting a subset of test cases, however, is to select test cases that are more likely to reveal races than others, allowing testers to focus on more worthwhile pursuits. Nevertheless, if test engineers wish, they can use *Ranker* to further prioritize the remaining test cases ($T - T'$), and execute those test cases as well. This process is performed using the approach just described, applied to $T - T'$, in two steps, where step 1 focuses on test cases that cover the most impacted shared variables, and step 2 focuses on test cases that cover only non-impacted shared variables.

In our example, when prioritizing the remaining test cases (having already set T_P to $< tc_3, tc_1, tc_2 >$), *Ranker* notes that (step 1) tc_4 covers two impacted shared variables, SV_7 and SV_8, and tc_5 covers three impacted shared variables, SV_2, SV_3, and SV_4, and appends these to T_P, obtaining $< tc_3$, tc_1, tc_2, tc_5, $tc_4 >$. In step 2, since tc_6 covers SV_1, a shared variable that is not impacted, *Ranker* appends that to T_P. Ultimately, T_P = $< tc_3$, tc_1, tc_2, tc_5, tc_4, $tc_6 >$.

6.3 Empirical Study

We wish to determine whether SIMRT is cost-effective, and ideally such an assessment would involve comparisons with existing state-of-the-art approaches for detecting races in modified software. There are, however, no existing approaches that have this specific goal. In the absence of such approaches, we can instead compare SIMRT to processes that are currently used in general regression testing applications, that might likewise be employed in our setting.

A second important issue regarding SIMRT involves whether either or both of its component techniques, selection and prioritization, play a role in its cost-effectiveness (or lack thereof), and if so, to what extent. Understanding this issue is important to any efforts to extend the approach further.

We thus designed a study focusing on three research questions:

RQ1: How do the *efficiency* and *effectiveness* of SIMRT, considering only its regression test selection component, compare to those of the retest-all technique and a state-of-the-art RTS technique.

RQ2: How does the *effectiveness* of the TCP technique employed by SIMRT compare to that of random test case orders when just the prioritized selected test cases are considered?

RQ3: How does the *effectiveness* of the TCP technique employed by SIMRT compare to that of traditional additional-block-coverage prioritization and random test case orders when the entire prioritized test suite is considered?

RQ1 lets us consider the overall efficiency and effectiveness of SIMRT focusing on its regression test selection component, compared to the baseline approach in which no selection is performed and to a state-of-the-art RTS technique. RQ2 lets us consider whether prioritization of just those test cases selected by SIMRT helps detect races more quickly than leaving them unprioritized. RQ3 lets us consider the overall effectiveness of prioritization if complete test suites are used.

6.3.1 Objects of Analysis

We chose nine open source Java objects, which are representative of real-world code and have been widely used in academic research. These included one object downloaded from the Software-artifact Infrastructure Repository (SIR) [154], five objects from JDK, and three larger objects [155, 156, 157]. The objects include both closed code units (code units equipped with test drivers) and open code units (libraries that require test drivers to close them).

Among the closed objects, WEBLECH is a multi-threaded web site download and mirror tool; it accepts both command line options and a configuration file that specifies settings

such as URL, search options (e.g., BFS, DFS), and number of threads. JIGSAW is a leading-edge Web server platform; it accepts configuration files on both client and server sites, and multiple main entry points with command line options.

Among the open objects, LANG is part of the Apache Common Lang project. LOG4J is a Java logging application. HASHMAP, TREEMAP, ARRAYLIST, HASHTABLE, and BITSET are synchronized collection classes provided by Sun Microsystems' JDK. To close these objects, we wrote test drivers that utilize unit test cases generated by RANDOOP (described later). Because RANDOOP does not support multi-threaded programs, the test drivers also create sets of threads that concurrently execute the methods.

We utilized two versions of each of the nine objects. Table 6.1 lists our object versions along with some of their characteristics, including the number of lines of non-comment code (NLOC, column 2) and the number of shared variables identified in the modified versions (SVs, column 6) using the technique described in Section 6.2 (the numbers in parentheses indicate the number of impacted shared variables). Other columns are described later.

To address our questions, we also required multi-threaded test cases. Our objects, however, were either not equipped with such test cases, or the test suites supplied with them were too small (e.g., fewer than 50 test cases) to allow SIMRT to operate as intended. To simulate a resource-constrained testing environment in which it makes sense to utilize approaches such as SIMRT, we chose to create test cases using a testing time budget, that limits the maximum number of test cases that can be executed for a given object. The testing time budgets we chose were selected to be relevant to the size and complexity of the objects. For the six smaller and less complex objects, we chose a testing budget of 12 hours (i.e., testing that can be performed overnight). For the three larger and more complex objects, we chose a testing budget of 60 hours (i.e., testing that can be completed over a weekend.) Both of these are practically realistic choices that engineers could make given

Table 6.1: Objects of Analysis and their Characteristics

Program	NLOC	|T|	PRs	RRs	SVs (ISVs)	ISV_{cov}s
LANG (V)	993	29104	-	-	-	-
LANG (V')	990		8	8	19 (6)	6
HASHMAP (V1.2)	852	26405	-	-	-	-
HASHMAP (V1.59)	1,014	-	59	10	230 (61)	51
TREEMAP (V1.2)	1,321	28203	-	-	-	-
TREEMAP (V1.56)	1,384	-	16	0	458 (17)	12
ARRAYLIST(V1.2)	727	34202	-	-	-	-
ARRAYLIST (V1.43)	747	-	22	3	352 (33)	24
HASHTABLE (V1.2)	3,041	28757	-	-	-	-
HASHTABLE (V1.55)	4,111	-	3	2	412 (36)	28
BITSET (V1.2)	194	30500	-	-	-	-
BITSET (V1.55)	486	-	4	2	240 (13)	10
LOG4J(V1.2.13)	15,331	32065	-	-	-	-
LOG4J (V1.2.8)	15,366	-	6	2	171 (63)	44
WEBLECH(V0.02)	16,393	9601	-	-	-	-
WEBLECH (V0.03)	16,633	-	29	1	23 (14)	12
JIGSAW(V2.2.0)	90,331	18805	-	-	-	-
JIGSAW (V2.2.6)	101,207	-	69	3	3452 (488)	209

that the testing required by our approach can be conducted automatically without the need for human intervention, yet must (like any validation effort) be conducted within some fixed time period. Note that the testing time we measured is based on the time required to run the instrumented original object with race detectors in place; thus, the numbers of generated test cases depend on the instrumentation overhead for different object.

The test cases we used were created using test case generation techniques relevant to the objects. Because our goal is to detect races, a test case must include two components: test input data and specified thread interleavings [6]. For objects WEBLECH and JIGSAW, which accept configuration files, we used incremental covering arrays [158] to generate test inputs. They also accept inputs via command line options, and such inputs were randomly generated. For the closed objects, which were not equipped with test drivers, we first used RANDOOP [159] to generate unit test cases for each method, and then we created multiple threads to concurrently execute these methods. Note that the number of threads is another input argument. For all object, the number of threads (if mutable) associated with each test

input was randomly generated in a range from 2 to 100. Column 3 of Table 6.1 (|T|) lists the numbers of test cases ultimately utilized for each object.

To address our research questions we also needed to know what races (if any) existed in our object programs. To determine this, we ran all test cases on the original version of each object using the modified version of RACEFUZZER (described in Section 6.1). We discovered that the original object versions contained many potential and real races. In this work we are interested only in regression races (i.e., races introduced by code modifications); not residual races (races that persist across versions of the modified objects). Hence, we located the causes of such races in the original object versions, and corrected the code in both versions to ensure that they would not occur in either. Next, we executed all of the test cases associated with each original object on its modified version with the race detector and verifier enabled. This yielded a set of regression races that had been introduced by the code changes made to the modified versions.

Because non-determinism may cause race detectors to report different results on different object executions, we repeated the foregoing process ten times for each object pair, and accumulated any newly detected races. (The fluctuation in numbers of races reported on different runs was actually quite small, and is discussed further in Section 6.4.) Columns 4 and 5 of Table 6.1 list the numbers of potential and real regression races (PRs and RRs) discovered in the foregoing process. Column 7 lists the number of impacted variables (ISV_{cov}s) that are covered in the modified objects.

6.3.2 Variables and Measures

6.3.2.1 Independent Variable

Our independent variable involves the techniques used in our study. As noted earlier, we consider SIMRT, the traditional retest-all technique (RTA), and the traditional DEJAVU

RTS technique (hereafter referred to as RTSC). With all three of these, we enable the race detector and verifier. We use SIMRT$_S$ to denote SIMRT with just the RTS technique enabled.

We refer to SIMRT when it also employs a TCP technique as SIMRT$_{SP}$. When considering RQ2 and just selected test cases, we compare SIMRT$_{SP}$ to a random test case ordering applied to just the selected test cases (denoted here by RANDOM$_{SP}$).

When considering RQ3, in which all test cases are prioritized, we compare SIMRT (denoting this version of the technique by SIMRT$_{AP}$) to two approaches: a random test case ordering applied to all test cases (denoted here by RANDOM$_{AP}$) and an application of the *additional block coverage* prioritization technique (denoted here by ABC$_{AP}$) described in Section 6.1.

6.3.2.2 Dependent Variables

As dependent variables, we chose metrics allowing us to answer each of our three research questions.

Regression test selection. To measure the *efficiency* of regression test selection we measured *testing time* by adding relevant measures, including the time required for escape and impact analyses (if any), the time required for running regression test selection algorithms (if any) and the time required to execute test cases.[6] To obtain these times, we applied RTSC and SIMRT$_S$ and measured relevant analysis and test selection times. Then, to account for possible differences in testing times per execution of sets of test cases, we executed each object ten times on the various sets of selected test cases (or in the case of RTA, on all test cases), and calculated the average time required to execute the test cases.

[6]When measuring testing time, we do not include time spent on activities performed in the preliminary phase of regression testing (prior to the time at which the modified object is available for testing, and when testing time becomes a critical issue); these include collection of test history information and construction of control flow graphs for the original object.

To measure race detection *effectiveness*, we compare the numbers of potential and real races detected by test cases selected by RTA, RTSC, and SIMRT$_S$.

Test case prioritization. To measure the *effectiveness* of test case prioritization we considered the rate at which test cases detect both potential and real races. To measure such rates, we use the well-known metric $APFD$ (Average Percentage Fault Detected) [19]. Let T be a test suite containing n test cases, let T_p be a prioritized version of T, and let R be a set of m races revealed by T. Let TR_i be the first test case in ordering T_p of T that reveals race i. The $APFD$ for test suite T_p is given by the equation:

$$APFD = \frac{\sum_{i=1}^{|m-1|} |TR_i|}{n * m} + \frac{1}{2n}$$

$APFD$ ranges from 0 to 100, with higher values indicating faster rates of race detection.

The $APFD$ metric can be applied to orderings of the entire test suite T, or to orderings of just the set of selected test cases T'. In the former case, n = $|T|$, and in the latter case, n = $|T'|$.

6.3.3 Study Operation

We conducted our experiment runs on a Linux cluster with 1440 AMD cores housed in 42 nodes with 128GB per node. We used a modified version of RACEFUZZER for race detection and verification (see Section 6.1). Among our objects, the instrumentation overhead incurred by our RACEFUZZER ranged from 1.5X to 20X. The remaining components in SIMRT (i.e., *ImpAnalyzer*, *Matcher*, *Selector* and *Ranker*) were implemented using Java by following the algorithms described in Section 6.2.

As noted in reference [75], executing a program once per input is sufficient to detect most, if not all, of the concurrent faults detectable under that input, because progams tend to follow the same interleaving patterns during different executions. As we discuss in Section

6.4, executing a program on the same input additional times may occasionally uncover additional races, but the benefit of doing this may be outweighed by the increased cost of testing. Therefore, the data reported in this study for each technique was obtained by executing each object once. However, we discuss the impact of race detection effectiveness given multiple test runs in Section 6.4.

6.3.4 Threats to Validity

The primary threat to external validity for this study involves the representativeness of our objects and test cases. Other objects and test cases may exhibit different behaviors and cost-benefit tradeoffs. However, we do reduce this threat to some extent by using several varieties of well studied open source code objects for our study, and test suites generated by practical approaches.

The primary threat to internal validity for this study is possible faults in the implementation of our approach and in the tools that we use to perform evaluation. We controlled for this threat by extensively testing our tools and verifying their results against a smaller program for which we can manually determine the correct results.

Where construct validity is concerned, our measurements of efficiency of regression test selection focus on the time required for analysis and test execution. However, other costs such as test setup and maintenance costs can play a role in technique efficiency. Our measurement of effectiveness for test case prioritization is $APFD$. Although $APFD$ does have certain limitations [76], it does provide a simple, intuitive measurement for rapid race detection.

6.3.5 Results and Analysis

6.3.5.1 RQ1: Regression Test Selection

Efficiency. Figures 6.6 and 6.7 show technique execution times in minutes for each of the three techniques and nine objects. For the six small objects using overnight testing, RTSC required less time than RTA, with savings ranging from 30.8% to 96.9%, and an average savings of 32.3% across all six objects. SIMRT achieved even greater savings than RTA on the smaller objects, with an average savings of 89.9%, and savings on individual objects ranging from 46.2% to 99.9%. For the three objects using over-the-weekend testing, RTSC required less time than RTA on only one object (LOG4J) with a savings of 44.8%. SIMRT achieved greater savings than RTA on all three objects, with savings of 96.5% on LOG4J, 47.9% on WEBLECH, and 52.9% on JIGSAW.

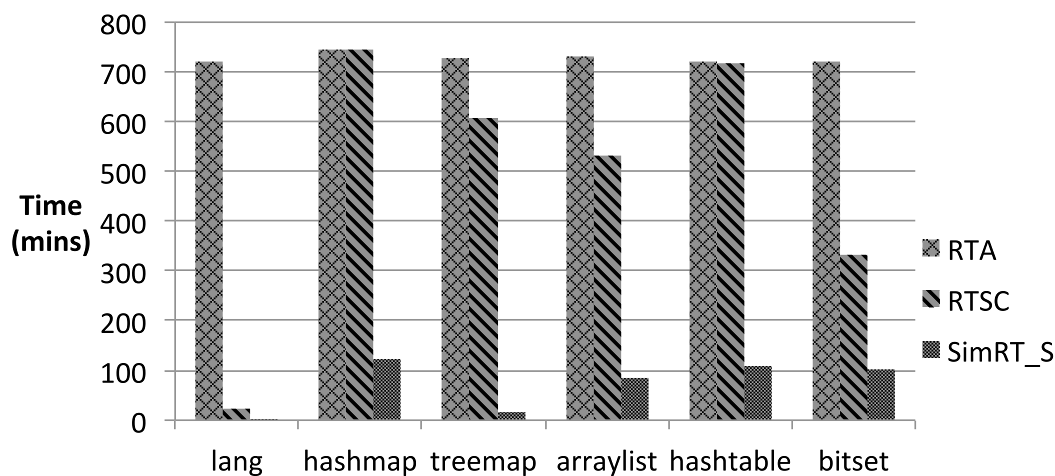

Figure 6.6: Efficiency: testing times for smaller objects

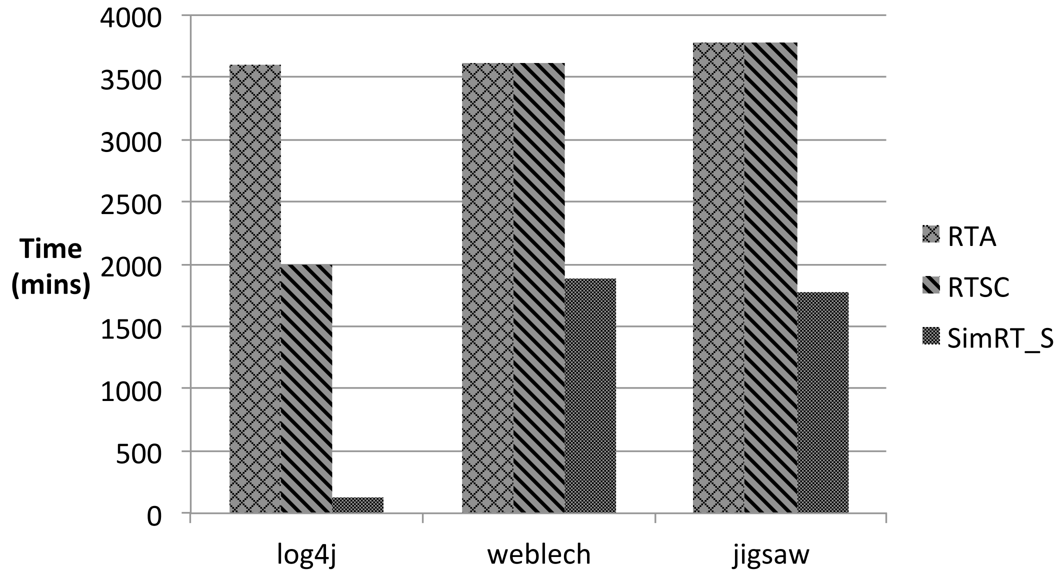

Figure 6.7: Efficiency: testing times for larger objects

We also measured the time required by RTSC and SIMRT to perform analysis tasks. Overall, while SIMRT required greater analysis times than RTSC, its analysis times never exceeded 45 seconds, which accounted for less than 0.5% of technique runtime overall. The time spent on escape analysis and impact analysis never exceeded 308 seconds. We also measured the time spent by SIMRT on race verification, and the results ranged from 7% to 11.9% of total technique runtime. The remaining time was spent on analysis tasks and race detection. Verification time varied with the number of detected potential races and the number of test runs needed to verify each potential race.

Testing times did vary across objects. The number of test cases selected by techniques depends on the program locations in which changes occur or in which shared variables are impacted, and this, in turn, affects testing time. For example, in WEBLECH, many changes occur inside the `main` function, causing all test cases to be selected by RTSC; thus, in this case, no time is saved.

Table 6.2: Race Detection Effectiveness

Prog.	*Potential races*			*Real races*		
	RTA	RTSC	SimRT	RTA	RTSC	SimRT
LANG	8	8	8	8	8	8
HASHMAP	59	59	58 (-1)	10	10	10
TREEMAP	16	16	16	0	0	0
ARRAYLIST	22	22	22	3	3	3
HASHTABLE	3	3	3	2	2	2
BITSET	4	4	4	2	2	2
LOG4J	6	6	6	2	2	2
WEBLECH	29	29	29	1	1	1
JIGSAW	68(-1)	68(-1)	66(-3)	3	3	3

Effectiveness. We consider whether races (both potential and real) detected by RTA and RTSC can also be detected by SIMRT. Table 6.2 shows the numbers of potential and real races detected for each of the nine objects by RTA, RTSC and SIMRT. Because the data reported for each technique is based on a single run, the numbers of races detected by RTA do not necessarily equal the numbers in Table 6.1; negative numbers in parentheses indicate the numbers of races missed compared to those known to be detectable.

For all nine objects, RTSC detected 100% of the potential and real races detected by RTA. SIMRT, on the other hand, detected 100% of the potential races on seven of nine objects and 100% of the real races on all nine. On LOG4J and JIGSAW SIMRT missed one and two potential races, respectively. This sacrifice in effectiveness is relatively small, with only 1.3% of the potential races detected by RTA and RTSC missed across the nine objects. Meanwhile, the associated savings in terms of testing time was large.

6.3.5.2 RQ2 and RQ3: Test Case Prioritization

Table 6.3 summarizes the APFD values computed for the test suites and test case orders utilized in our study across all nine objects, for both potential and real races, with $SimRT_{SP}$ and $Random_{SP}$ applied to selected test cases, and with $SimRT_{AP}$, ABC_{AP} and $Random_{AP}$ applied to the entire test suites.

Table 6.3: APFD Values for Selected and All Test Cases

Prog.	*Potential Races/Real Races*				
	SimRT_{SP}	Random_{SP}	SimRT_{AP}	ABC_{AP}	Random_{AP}
LANG	96.9/96.9	50.3/50.3	100.0/100.0	71.5/71.5	52.2/ 52.2
HASHMAP	94.2/96.8	58.9/66.2	95.7/97.2	57.4/58.8	59.1/60.0
TREEMAP	98.1/-	39.8/-	100.0/-	48.5/-	40.8/-
ARRAYLIST	89.1/92.3	54.8/64.5	91.4/93.8	70.6/72.9	48.4/51.4
HASHTABLE	92.2/92.2	42.4/42.4	94.2/94.3	58.9/58.9	54.2/54.8
BITSET	97.5/97.8	64.6/70.4	96.8/96.8	88.4/88.6	69.2/71.5
LOG4J	85.7/89.2	47.5/56.5	90.0/90.6	47.8/48.4	48.5/48.8
WEBLECH	100.0/100.0	99.8/99.8	100.0/100.0	83.2/83.4	64.4/64.5
JIGSAW	88.4/93.6	52.8/58.6	91.2/94.8	72.9/78.0	56.6/61.4

To compare the effectiveness of different techniques, we display results in terms of *improvement in race detection rate*, calculated as: $\frac{APFD_{T1}-APFD_{T2}}{APFD_{T2}}*100\%$. This indicates the percentage improvement in APFD achieved by technique $T1$ over technique $T2$. For example, for potential race detection, on object LANG, the improvement in APFD achieved by $SimRT_{SP}$ over $Random_{SP}$ was $\frac{96.9-50.3}{50.3}*100\%$ = 92.6%.

RQ2: Prioritization Results on Selected Test Cases.

On the six small objects, SIMRT_{SP} outperformed RANDOM_{SP} in terms of APFD. The improvement ranged from 50.9% to 146.5% for potential races, and from 38.9% to 117.5% for real races, and the average improvement across the five objects was 88.3% for potential races and 67.7% for real races. On the three large objects, SIMRT_{SP} outperformed RANDOM_{SP} in terms of rate of race detection, with improvements of 80.4%, 0.2% and 67.4% for potential races, and 57.9%, 0.2% and 59.7% for real races.

On WEBLECH, RANDOM_{SP} performed almost as effectively as SIMRT_{SP} because in that case, $ISVP$s were concentrated in several methods, making it easier to achieve high coverage per selected test case, and difficult to achieve additional coverage.

RQ3: Prioritization Results on Entire Test Suites.

On the six small objects, when compared to ABC_{AP}, SIMRT_{AP}, improved the potential race detection rate by 9.5% to 106.2%, with an average improvement of 52.0%. When compared to RANDOM_{AP}, SIMRT_{AP} improved the potential race detection rate by

between 39.9% and 145.1%, with an average of 83.5%. Where detection of actual races is concerned, when compared to ABC_{AP}, SIMRT_{AP} improved the detection rate by 9.3% to 65.3%, with an average of 40.7%. When compared to RANDOM_{AP}, SIMRT_{AP} improved the detection rate by 35.4% to 91.6%, with an average of 68.7%.

On the three large objects, when compared to ABC_{AP}, SIMRT_{AP} improved the potential race detection rate by 88.3%, 20.2% and 25.1%, respectively. When compared to RANDOM_{AP}, SIMRT_{AP} improved the potential race detection rate by 85.6%, 55.2% and 61.1%, respectively. Where detection of real races is concerned, when compared to ABC_{AP}, SIMRT_{AP} improved the race detection rate by 87.2%, 19.9% and 21.5%, respectively. When compared to RANDOM_{AP}, SIMRT_{AP} improved the race detection rate by 85.7%, 55.0% and 54.4%, respectively.

6.4 Summary and Discussion

Summary of Results.

SIMRT's RTS component was substantially more efficient in terms of testing time than the retest-all and traditional RTS techniques. Meanwhile, SIMRT detected the same sets of real races as the baseline techniques. Although a few potential races were missed, the number was small.

SIMRT's TCP component, employed on selected test cases, was substantially more effective in terms of $APFD$ than the random test case ordering. When employed on entire test suites, the TCP component was substantially more effective in terms of $APFD$ than both additional-block-coverage and random techniques. Meanwhile, in this case, SIMRT was able to detect potential races that were not detectable when operating on a subset of the test cases.

If these results generalize to other real objects and RTS and TCP techniques, then *if engineers wish to target race detection,* SIMRT *is the best technique to utilize.*

We now explore additional observations relevant to our study.

Multiple Runs. In our study, we executed each object once under the selected test cases for each technique to assess race detection effectiveness; a practical approach in resource-limited testing environments. Although empirical studies have shown that the fluctuation in race detection among multiple runs does not exceed 0.7% [75], we wished to determine whether multiple test runs could impact the effectiveness of the techniques studied.

To do this, for each RTS technique, we ran each modified object with the potential race detector and verifier ten times. The results indicated no differences in real race detection across the ten runs, and differences in potential race detection were revealed in only two cases. Specifically, on HASHMAP, SIMRT detected one extra race in two of the ten test suite runs, and missed one race in one test suite run. RTA and RTSC did not display any differences. On JIGSAW, RTA detected one extra race in two of the ten runs and missed one race in one run. RTSC detected one extra race in three of the ten runs, and SIMRT detected two extra races in three of the ten runs, missing one race in one run. These results imply that conducting one run for each test case is likely to be a reasonable level of effort in resource-limited testing environments.

Effects of Recording Thread Coverage. SIMRT selects test cases that potentially cover impacted shared variable pairs in the modified object with different threads. As such, SIMRT records thread IDs for test history construction. To determine whether this is useful, we further investigated whether savings could be attained by considering thread coverage. We considered percentages of test cases selected without using thread coverage, labeling this approach SIMRT_N. SIMRT_N selects test cases that potentially cover impacted shared

variable pairs in the modified object, regardless of the number of threads involved in covering each pair.

For all nine objects, SIMRT yielded savings over SIMRT$_N$; these ranged from 16.5% to 83.7% with an average of 77.4% for the six smaller objects and from 34.4% to 56.6% with an average of 44.9%, for the three larger objects. Therefore, the approach substantially improved the efficiency of regression test selection.

Numbers of Selected Test Cases. The numbers of test cases selected by various techniques is another metric for measuring the *efficiency* of regression test selection, providing an implementation-independent view of efficiency. For the nine objects, RTSC selected smaller test suites than RTA in many cases, with overall selection percentages ranging from 45.8% to 100%. On four of the nine objects, however, RTSC selected more than 90% of the test cases, and in two cases it selected 100% of the test cases. SIMRT, on the other hand, pruned away larger portions of the test suites, with selection percentages ranging from 0.1% to 51.8%. In fact, for seven objects, SIMRT selected fewer than 15% of the test cases.

Exposing Other Faults. SIMRT specifically targets races, and therefore, may not be effective at detecting other classes of faults that RTA or RTSC can reveal. As such, it is worth investigating the performance of RTSC when combined with SIMRT. We thus compared the testing time required by RTSC (with instrumentation for race detection) to that required by the use of both SIMRT and RTSC without instrumentation (denoted by RTSC$_N$). Both RTSC and RTSC$_N$ detect races and other output-based regression faults. This comparison showed that the savings of RTSC$_N$ over RTSC ranged from 43.5% and 67.6% (average 57.36%) across the six smaller objects. On the larger objects, the savings were 68.5%, 22.9% and 31.2%, respectively. This suggests that race detection and traditional fault detection should be considered separately.

Results such as these should be qualified. If SIMRT does not substantially reduce the number of selected test cases over RTSC, $RTSC_N$ may not achieve savings. Further, if engineers wish to detect both data races and other regression faults, and SIMRT substantially reduces the number of test cases selected over traditional RTS techniques, the best approach to use is: 1) run test cases selected by traditional RTS techniques on uninstrumented objects and 2) run test cases selected by SIMRT on instrumented objects. Race detection overhead varies across different types of applications. For example, I/O-intensive applications tend to have small race detection overhead, and hence would benefit less from SIMRT.

Further Discussion. We have used SIMRT to target data races, but it can be adapted to detect other types of concurrency faults such as atomicity violations [124, 160] and deadlocks [43,44] by adjusting the contents of coverage targets. For example, to detect atomicity violations, instead of identifying potential impacted shared variable pairs as coverage targets, SIMRT can identify potential *unserializable interleavings* [124] as coverage targets.

As Table 6.1 shows, for some objects, there were a few uncovered impacted shared variables remaining in the modified versions after running the existing test cases. These variables may also cause data races. Covering the impacted shared variables pairs associated with these variables requires additional test cases or possibly thread interleavings. To address this, we intend to further explore extending SIMRT using test suite augmentation techniques [161].

6.5 Conclusion

In this chapter, we have presented an automated regression testing approach, SIMRT, for use in detecting races that are induced due to code changes. SIMRT employs both RTS and TCP techniques. We have conducted an empirical study applying SIMRT to nine concur-

rent code objects. We have empirically compared SIMRT to traditional baseline techniques, and our results suggest that SIMRT's test selection component is more effective and efficient than other approaches in terms of race detection. Our results also show that the TCP technique employed by SIMRT is more effective than a traditional TCP technique and a random ordering in terms of rate of race detection.

Chapter 7

Conclusion and Future Work

In this dissertation, we have presented a testing-based verification framework, SIMEXPLORER, for modern software systems that allows engineers to effectively detect elusive faults that occur at the whole-system level. We have introduced SIMTESTER, an instantiation of the SIMEXPLORER framework that utilizes VMs to tackle the challenges of testing for concurrency errors involving interactions between software and hardware. We have presented SIMRACER, a second instantiation of the SIMEXPLORER framework that utilizes the VM to test for races involving multiple processes and signals. We have developed SIMLATTE, another instantiation of the SIMEXPLORER framework that uses a genetic algorithm for test case generation that converges on a set of inputs and interrupt arrival points that are likely to expose WCILs. Finally, we have presented SIMRT, a regression testing technique for use in detecting races introduced by code modifications.

Our results suggest that the four techniques are more effective than their various baseline techniques. First, we have evaluated SIMTESTER on a previous release of the Linux kernel. Our results clearly show that SIMTESTER is effective in detecting both real faults and seeded faults related to data races and deadlocks. These faults were not effectively revealed by baseline techniques, that lack either observability or controllability. Second,

we have conducted an empirical study applying SIMRACER to 16 Unix programs containing actual process-level races. Our results show that SIMRACER is effective at detecting process-level races. By comparing SIMRACER to traditional stress testing techniques, we also show that SIMRACER is more effective than these techniques at detecting faults that are caused by the process-level races. Third, we have evaluated SIMLATTE on three real-world embedded system applications. Our results show that SIMLATTE can calculate WCILs more precisely and efficiently than random approach. The results also suggest that applying either of the two components (genetic algorithm and opportunistic interrupt invocation) of SIMLATTE yields better results than random testing, but employing both components yields the best results. Finally, we have conducted an empirical study applying SIMRT to nine concurrent Java programs that contain naturally-occurring races caused by code modifications. Our results show that SIMRT is more efficient than other regression test selection techniques in terms of race detection. Our results also show that SIMRT's test prioritization component is substantially more effective than a traditional prioritization technique and a random ordering technique in terms of rate of race detection.

This research has made the following contributions, with impacts on both researchers and practitioners:

1. Developed techniques to test for concurrency faults that occur due to interactions between applications and interrupt handlers.

2. Developed techniques to test for races between processes and between processes and signal handlers.

3. Developed a testing-based approach to estimate worst-case interrupt latencies (WCIL).

4. Developed a regression testing technique to test for data races that are induced by software evolution.

For future work, we intend to improve our existing instantiations and create new instantiations of the SIMEXPLORER framework.

For SIMTESTER, we intend to study more device drivers that utilize a variety of interrupt sources. We also intend to consider nested interrupts to detect concurrency faults that occur among interrupt handlers. Finally, we intend to apply SIMTESTER on other architectures such as Atmel microcontrollers.

For SIMRACER, some faults were not detected due to characteristics of the test cases and system environment used. We intend to investigate the possibility of automatically creating test inputs and setting up system environments that allow us to effectively expose process-level data races.

For SIMLATTE, we intend to improve our fitness functions to further enhance its effectiveness. For example, we wish to consider the number of critical sections in the fitness function to guide SIMLATTE in exploring more critical sections. We also intend to extend SIMLATTE to combine both static and testing approaches to achieve higher effectiveness and accuracy.

On regression testing, future work includes extending SIMRT to detect other types of concurrency faults such as atomicity violations and deadlock and investigating the use of other regression testing techniques such as test suite augmentation in conjunction with SIMRT.

Finally, we plan to create new instantiations of SIMEXPLORER framework, that can address other types of elusive faults including worst-case memory usage and priority inversion.

Bibliography

[1] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, “Basic concepts and taxonomy of dependable and secure computing,” *IEEE Transactions on Dependable and Secure Computing*, vol. 1, pp. 11–33, Jan. 2004.

[2] M. Staats, M. W. Whalen, and M. P. Heimdahl, “Programs, tests, and oracles: The foundations of testing revisited,” in *Proceedings of the International Conference on Software Engineering*, pp. 391–400, 2011.

[3] T. Yu, W. Srisa-an, and G. Rothermel, “An empirical comparison of the fault-detection capabilities of internal oracles,” in *Proceedings of the International Symposium on Software Reliability Engineering*, pp. 11–20, 2013.

[4] M. D. Bond, K. E. Coons, and K. S. McKinley, “PACER: Proportional detection of data races,” in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 255–268, 2010.

[5] J. Regehr, “Random testing of interrupt-driven software,” in *Proceedings of the ACM International Conference on Embedded Software*, pp. 290–298, 2005.

[6] K. Sen, “Race directed random testing of concurrent programs,” in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 11–21, 2008.

[7] A. David, K. G. Larsen, A. Legay, U. Nyman, and A. Wasowski, “Timed I/O automata: A complete specification theory for real-time systems,” in *Proceedings of the ACM International Conference on Hybrid Systems: Computation and Control*, pp. 91–100, 2010.

[8] A. Hessel, K. G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou, “Testing real-time systems using UPPAAL,” in *Formal Methods in Testing*, pp. 77–117, Springer, 2008.

[9] J. Kim, H. Oh, H. Ha, S.-H. Kang, J. Choi, and S. Ha, “An ILP-based worst-case performance analysis technique for distributed real-time embedded systems,” in *Proceedings of the International Real-Time Systems Symposium*, pp. 363–372, 2012.

[10] L. Kong and J. Jiang, “A safe measurement-based worst-case execution time estimation using automatic test-data generation,” in *Proceedings of the IEEE Pacific Rim International Symposium on Dependable Computing*, pp. 245–246, 2010.

[11] D. Sehlberg, A. Ermedahl, J. Gustafsson, B. Lisper, and S. Wiegratz, “Static WCET analysis of real-time task-oriented code in vehicle control systems,” in *Proceedings of the International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, pp. 212–129, 2006.

[12] T. Yu, X. Qu, M. Acharya, and G. Rothermel, “Oracle-based regression test selection,” in *Proceedings of the International Conference on Software Testing, Verification and Validation*, pp. 292–301, 2013.

[13] M. Staats, P. Loyola, and G. Rothermel, “Oracle-centric test case prioritization,” in *Proceedings of the International Symposium on Software Reliability Engineering*, pp. 311–320, 2012.

[14] J. Engblom, D. Aarno, and B. Werner, *Full-system simulation from embedded to high-performance systems*, pp. 25–45. 2010.

[15] B. L. Titzer, D. K. Lee, and J. Palsberg, "Avrora: Scalable sensor network simulation with precise timing," in *Proceedings of the International Symposium on Information Processing in Sensor Networks*, pp. 477–482, 2005.

[16] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Journal of Software Testing, Verification, and Reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[17] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 2, pp. 173–210, 1997.

[18] G. Rothermel and M. J. Harrold, "Analyzing regression test selection techniques," *IEEE Transactions on Software Engineering*, vol. 22, no. 8, pp. 529–551, 1996.

[19] G. Rothermel, R. J. Untch, and C. Chu, "Prioritizing test cases for regression testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 10, pp. 102–112, 2001.

[20] interrupt latency. http://www.smxrtos.com/articles/lsr_art/lsr_art.htm, 2005.

[21] E. Farchi, Y. Nir, and S. Ur, "Concurrent bug patterns and how to test them," in *Proceedings of the International Symposium on Parallel and Distributed Processing*, pp. 286.2–286.2, 2003.

[22] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: A comprehensive study on real world concurrency bug characteristics," in *Proceedings of the Interna-*

tional Conference on Architectural Support for Programming Languages and Operating Systems, pp. 329–339, 2008.

[23] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," *ACM Transactions on Computer Systems*, vol. 15, no. 4, pp. 391–411, 1997.

[24] S. Lu, J. Tucek, F. Qin, and Y. Zhou, "Avio: Detecting atomicity violations via access interleaving invariants," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 37–48, 2006.

[25] A. Silberschatz, P. Galvin, and G. Gagne, *Applied Operating System Concepts*. New York, NY, USA: John Wiley & Sons, Inc., 1st ed., 2001.

[26] O. Laadan, N. Viennot, C.-C. Tsai, C. Blinn, J. Yang, and J. Nieh, "Pervasive detection of process races in deployed systems," in *Proceedings of the ACM Symposium on Operating Systems Principles*, pp. 353–367, 2011.

[27] P. McMinn, "Search-based software test data generation: A survey: Research articles," *Journal of Software Testing, Verification, and Reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[28] S. Wappler and F. Lammermann, "Using evolutionary algorithms for the unit testing of object-oriented software," in *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 1053–1060, 2005.

[29] P. Tonella, "Evolutionary testing of classes," in *Proceedings of the International Symposium on Software Testing and Analysis*, pp. 119–128, 2004.

[30] M. Z. Iqbal, A. Arcuri, and L. Briand, “Empirical investigation of search algorithms for environment model-based testing of real-time embedded software,” in *Proceedings of the International Symposium on Software Testing and Analysis*, pp. 199–209, 2012.

[31] L. Albertsson, “Simulation-based debugging of soft real-time applications,” in *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, pp. 107–108, 2001.

[32] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau, “Framework for instruction-level tracing and analysis of program executions,” in *Proceedings of the International Conference on Virtual Execution Environments*, pp. 154–163, 2006.

[33] M. Chang, E. Smith, R. Reitmaier, M. Bebenita, A. Gal, C. Wimmer, B. Eich, and M. Franz, “Tracing for web 3.0: Trace compilation for the next generation web applications,” in *Proceedings of the International Conference on Virtual Execution Environments*, pp. 71–80, 2009.

[34] T. Koju, S. Takada, and N. Doi, “An efficient and generic reversible debugger using the virtual machine based approach,” in *Proceedings of the International Conference on Virtual Execution Environments*, pp. 79–88, 2005.

[35] A. Pohle, B. Döbel, M. Roitzsch, and H. Härtig, “Capability wrangling made easy: Debugging on a microkernel with Valgrind,” in *Proceedings of the International Conference on Virtual Execution Environments*, pp. 3–12, 2010.

[36] O. Goh and Y.-H. Lee, “Schedulable online testing framework for real-time embedded applications in VM,” in *Proceedings of the International Conference on Embedded and Ubiquitous Computing*, pp. 730–741, 2007.

[37] K. Asrigo, L. Litty, and D. Lie, "Using VMM-based sensors to monitor honeypots," in *Proceedings of the International Conference on Virtual Execution Environments*, pp. 13–23, 2006.

[38] K. Kourai and S. Chiba, "HyperSpector: Virtual distributed monitoring environments for secure intrusion detection," in *Proceedings of the International Conference on Virtual Execution Environments*, pp. 197–207, 2005.

[39] J. Chow, T. Garfinkel, and P. M. Chen, "Decoupling dynamic program analysis from execution in virtual environments," in *USENIX Annual Technical Conference*, pp. 1–14, 2008.

[40] C. Flanagan and S. N. Freund, "FastTrack: Efficient and precise dynamic race detection," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 121–133, 2009.

[41] R. O'Callahan and J.-D. Choi, "Hybrid dynamic data race detection," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 167–178, 2003.

[42] W. Zhang, C. Sun, and S. Lu, "Conmem: Detecting severe concurrency bugs through an effect-oriented approach," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 179–192, 2010.

[43] R. Agarwal and S. D. Stoller, "Run-time detection of potential deadlocks for programs with locks, semaphores, and condition variables," in *Proceedings of the Workshop on Parallel and Distributed Systems: Testing and Debugging*, pp. 51–60, 2006.

[44] P. Joshi, C.-S. Park, K. Sen, and M. Naik, "A randomized dynamic program analysis technique for detecting real deadlocks," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 110–120, 2009.

[45] K. Havelund, "Using runtime analysis to guide model checking of Java programs," in *Proceedings of the International SPIN Workshop on SPIN Model Checking and Software Verification*, pp. 245–264, 2000.

[46] B.-C. Kim, S.-W. Jun, D. J. Hwang, and Y.-K. Jun, "Visualizing potential deadlocks in multithreaded programs," in *Proceedings of the International Conference on Parallel Computing Technologies*, pp. 321–330, 2009.

[47] P. Zhou, R. Teodorescu, and Y. Zhou, "Hard: Hardware-assisted lockset-based race detection," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, pp. 121–132, 2007.

[48] K. Serebryany and T. Iskhodzhanov, "ThreadSanitizer: Data race detection in practice," in *Proceedings of the Workshop on Binary Instrumentation and Applications*, pp. 62–71, 2009.

[49] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou, "Muvi: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs," in *Proceedings of the ACM Symposium on Operating Systems Principles*, pp. 103–116, 2007.

[50] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea, "Deadlock immunity: Enabling systems to defend against deadlocks," in *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, pp. 295–308, 2008.

[51] M. Musuvathi and S. Qadeer, "Iterative context bounding for systematic testing of multithreaded programs," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 446–455, 2007.

[52] J. Chen and S. MacDonald, "Testing concurrent programs using value schedules," in *Proceedings of International Conference on Automated Software Engineering*, pp. 313–322, 2007.

[53] M. Higashi, T. Yamamoto, Y. Hayase, T. Ishio, and K. Inoue, "An effective method to control interrupt handler for data race detection," in *Proceedings of the Workshop on Automation of Software Test*, pp. 79–86, 2010.

[54] Z. Lai, S. C. Cheung, and W. K. Chan, "Inter-context control-flow and data-flow test adequacy criteria for nesc applications," in *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, pp. 94–104, 2008.

[55] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and reproducing heisenbugs in concurrent programs," in *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, pp. 267–280, 2008.

[56] M. Payer and T. R. Gross, "Protecting applications against tocttou races by user-space caching of file metadata," in *Proceedings of the International Conference on Virtual Execution Environments*, pp. 215–226, 2012.

[57] D. Tsafrir, T. Hertz, D. Wagner, and D. Da Silva, "Portably solving file TOCTTOU races with hardness amplification," in *Proceedings of the USENIX Conference on File and Storage Technologies*, pp. 13:1–13:18, 2008.

[58] T. Sheng, N. Vachharajani, S. Eranian, R. Hundt, W. Chen, and W. Zheng, "RACEZ: A lightweight and non-invasive race detection tool for production applications," in

Proceedings of the International Conference on Software Engineering, pp. 401–410, 2011.

[59] K.-H. K. Guy Martin Tchamgoue, Ok-Kyoon Ha and Y.-K. Jun, "Lightweight labeling scheme for on-the-fly race detection of signal handlers," *Ubiquitous Computing and Multimedia Applications*, pp. 201–208, 2011.

[60] T. Tahara, K. Gondow, and S. Ohsuga, "DRACULA: Detector of data races in signals handlers," in *Proceedings of the Asia-Pacific Software Engineering Conference*, pp. 17–24, 2008.

[61] G. M. Tchamgoue, K. H. Kim, and Y.-K. Jun, "Dynamic race detection techniques for interrupt-driven programs," in *Proceedings of the International Conference on Future Generation Information Technology*, pp. 148–153, 2012.

[62] T. Yu, W. Srisa-an, and G. Rothermel, "SimTester: A controllable and observable testing framework for embedded systems," in *Proceedings of the International Conference on Virtual Execution Environments*, pp. 51–62, 2012.

[63] G. Gracioli and S. Fischmeister, "Tracing interrupts in embedded software," in *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pp. 137–146, 2009.

[64] M. Tlili, S. Wappler, and H. Sthamer, "Improving evolutionary real-time testing," in *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 885–891, 2006.

[65] H. Pohlheim and J. Wegener, "Testing the temporal behavior of real-time software modules using extended evolutionary algorithms," in *Proceedings of the Genetic and Evolutionary Computation Conference*, p. 1795, 1999.

[66] L. C. Briand, Y. Labiche, and M. Shousha, "Stress testing real-time systems with genetic algorithms," in *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 1021–1028, 2005.

[67] A. Arcuri, M. Z. Iqbal, and L. Briand, "Black-box system testing of real-time embedded systems using random and search-based testing," in *Proceedings of the IFIP WG 6.1 International Conference on Testing Software and Systems*, pp. 95–110, 2010.

[68] N. Al Moubayed and A. Windisch, "Temporal white-box testing using evolutionary algorithms," in *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops*, pp. 150–151, 2009.

[69] N. Bhattacharya, O. El-Mahi, E. Duclos, G. Beltrame, G. Antoniol, S. Le Digabel, and Y.-G. Guéhéneuc, "Optimizing thread schedule alignments to expose the interference bug pattern," in *Proceedings of the International Conference on Search Based Software Engineering*, pp. 90–104, 2012.

[70] D. Engler, B. Chelf, A. Chou, and S. Hallem, "Checking system rules using system-specific, programmer-written compiler extensions," in *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, pp. 1–1, 2000.

[71] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav, "Scalable and precise dynamic datarace detection for structured parallelism," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 531–542, 2012.

[72] K. E. Coons, S. Burckhardt, and M. Musuvathi, "GAMBIT: Effective unit testing for concurrency libraries," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 15–24, 2010.

[73] M. Gligoric, V. Jagannath, and D. Marinov, "Mutmut: Efficient exploration for mutation testing of multithreaded code," in *Proceedings of the International Conference on Software Testing, Verification and Validation*, pp. 55–64, 2010.

[74] V. Jagannath, Q. Luo, and D. Marinov, "Change-aware preemption prioritization," in *Proceedings of the International Symposium on Software Testing and Analysis*, pp. 133–143, 2011.

[75] D. Deng, W. Zhang, and S. Lu, "Efficient concurrency-bug detection across inputs," in *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, pp. 785–802, 2013.

[76] H. Do and G. Rothermel, "On the use of mutation faults in empirical assessments of test case prioritization techniques," *IEEE Transactions on Software Engineering*, vol. 32, no. 9, pp. 733–752, 2006.

[77] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 159–182, 2002.

[78] B. Korel, G. Koutsogiannakis, and L. Tahat, "Application of system models in regression test suite prioritization," in *Proceedings of the International Conference on Software Maintenance*, pp. 247–256, 2008.

[79] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *IEEE Transactions on Software Engineering*, vol. 33, no. 4, pp. 225–237, 2007.

[80] A. Orso, N. Shi, and M. J. Harrold, "Scaling regression testing to large software systems," in *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, pp. 241–251, 2004.

[81] A. Srivastava and J. Thiagarajan, "Effectively prioritizing tests in development environment," in *Proceedings of the International Symposium on Software Testing and Analysis*, pp. 97–106, 2002.

[82] W. E. Wong, J. R. Horgan, S. London, and H. A. Bellcore, "A study of effective regression testing in practice," in *Proceedings of the International Symposium on Software Reliability Engineering*, pp. 264–274, 1997.

[83] S. Yoo, M. Harman, P. Tonella, and A. Susi, "Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge," in *Proceedings of the International Symposium on Software Testing and Analysis*, pp. 201–212, 2009.

[84] H. K. N. Leung and L. White, "Insights into testing and regression testing global variables," *Journal of Software Maintenance*, vol. 2, no. 4, pp. 209–222, 1990.

[85] L. White and B. Robinson, "Industrial real-time regression testing and analysis using firewalls," in *Proceedings of the International Conference on Software Maintenance*, pp. 18–27, 2004.

[86] S. Bindal, S. Bansal, and A. Lal, "Variable and thread bounding for systematic testing of multithreaded programs," in *Proceedings of the International Symposium on Software Testing and Analysis*, pp. 145–155, 2013.

[87] L. Cordeiro and B. Fischer, "Verifying multi-threaded software using smt-based context-bounded model checking," in *Proceedings of the International Conference on Software Engineering*, pp. 331–349, 2011.

[88] H. Guo, M. Wu, L. Zhou, G. Hu, J. Yang, and L. Zhang, "Practical software model checking via dynamic interface reduction," in *Proceedings of the ACM Symposium on Operating Systems Principles*, pp. 265–278, 2011.

[89] B. Fischer, O. Inverso, and G. Parlato, "Cseq: A concurrency pre-processor for sequential c verification tools," in *Proceedings of International Conference on Automated Software Engineering*, pp. 710–713, 2013.

[90] A. En-Nouaary, R. Dssouli, and F. Khendek, "Timed Wp-method: Testing real-time systems," *IEEE Transactions on Software Engineering*, vol. 28, no. 11, pp. 1203–1238, 2002.

[91] D. Brylow and J. Palsberg, "Deadline analysis of interrupt-driven software," in *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, pp. 198–207, 2011.

[92] B. Schlich, "Model checking of software for microcontrollers," *ACM Transactions on Embedded Computing Systems*, pp. 36:1–36:27, 2010.

[93] D. Brylow, N. Damgaard, and J. Palsberg, "Static checking of interrupt-driven software," in *Proceedings of the International Conference on Software Engineering*, pp. 47–56, 2001.

[94] M. B. Dwyer and L. A. Clarke, "Data flow analysis for verifying properties of concurrent programs," in *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, pp. 62–75, 1994.

[95] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner, "Thorough static analysis of device drivers,"

in *Proceedings of the SIGOPS/EuroSys European Conference on Computer Systems*, pp. 73–85, 2006.

[96] L. Tan, Y. Zhou, and Y. Padioleau, “aComment: Mining annotations from comments and code to detect interrupt related concurrency bugs,” in *Proceedings of the International Conference on Software Engineering*, pp. 11–20, 2011.

[97] J. Kotker, D. Sadigh, and S. A. Seshia, “Timing analysis of interrupt-driven programs under context bounds,” in *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, pp. 81–90, 2011.

[98] J. Engblom, “Virtutech device modeling language.” White-Paper, 2009. http://www.virtutech.com/files/whitepapers/wp-dml-2009-03-30-letter.pdf.

[99] C. Haggstrom, *Detection of memory allocation bugs on system level*. PhD thesis, 2005. Adviser-Mikael Rannar.

[100] J. Takalo, J. Kaariainen, P. Parviainen, and T. Ihme, “Challenges of software-hardware codesign.” White Paper, 2007. http://www.vtt.fi/inf/pdf/workingpapers/2008/W91.pdf.

[101] EETIMES, “Five top causes of nasty bugs.” Web page. http://www.eetimes.com/design/embedded/4008917/Five-top-causes-of-nasty-embedded-software-bugs.

[102] I. Jackson, “IRQ handling race and spurious IIR read in 8250.c.” Web page. https://lkml.org/lkml/2009/3/12/379.

[103] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers, 3rd Edition*. O’Reilly and Associates, 2005.

[104] V. Kahlon, Y. Yang, S. Sankaranarayanan, and A. Gupta, "Fast and accurate static data-race detection for concurrent programs," in *Proceedings of the International Conference on Computer Aided Verification*, pp. 226–239, 2007.

[105] K. Trap, "deadlock in 3c59x driver." Web page. http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commit;h=4bf3631cdb012591667ab927fcd7719d92837833.

[106] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?," in *Proceedings of the International Conference on Software Engineering*, pp. 402–411, 2005.

[107] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," in *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, pp. 263–272, 2005.

[108] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, pp. 209–224, 2008.

[109] T. Yu, W. Srisa-an, and G. Rothermel, "SimRacer: An automated framework to support testing for process-level races," in *Proceedings of the International Symposium on Software Testing and Analysis*, pp. 167–177, 2013.

[110] 2004. https://bugs.launchpad.net/debian/+source/bash /+bug/10809/comments/0.

[111] 2011. http://www.mail-archive.com/debian-bugs-dist@lists.debian.org/msg918886.html.

[112] 2000. http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=67782.

[113] "Apache Deadlock," 2003. http://marc.info/?l=apache-httpd-bugs&m=105967988713871.

[114] "Android Race," 2010. https://android.googlesource.com/platform/frameworks-/base/+/e6b1bbd8acca3f6e174c24cf4eb23a66db2d08a2.

[115] "UART Race," 2011. http://lkml.org/lkml/2011/7/25/369.

[116] S. Park, R. W. Vuduc, and M. J. Harrold, "Falcon: Fault localization in concurrent programs," in *Proceedings of the International Conference on Software Engineering*, pp. 245–254, 2010.

[117] B. Kasikci, C. Zamfir, and G. Candea, "Data races vs. data race bugs: Telling the difference with portend," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 185–198, 2012.

[118] F. Sorrentino, A. Farzan, and P. Madhusudan, "PENELOPE: Weaving threads to expose atomicity violations," in *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, pp. 37–46, 2010.

[119] A. Causevic, D. Sundmark, and S. Punnekkat, "An industrial survey on contemporary aspects of software testing," in *Proceedings of the International Conference on Software Testing, Verification and Validation*, pp. 393–401, 2010.

[120] T. J. Ostrand and M. J. Balcer, "The category-partition method for specifying and generating functional tests," *Communications of the ACM*, pp. 676–686, 1988.

[121] "Test Strategies for Smartphones and Mobile Devices," 2010. http://www.macadamian.com/images/uploads/whitepapers-/MobileTestStrategies_Aug2010.pdf.

[122] "Stress Testing," 2006. http://msdn.microsoft.com/en-us /magazine/cc163613.aspx.

[123] "Web Site Test Tools and Site Management Tools," 2012. http://www.softwareqatest.com/qatweb1.html.

[124] S. Park, S. Lu, and Y. Zhou, "CTrigger: Exposing atomicity violation bugs from their hiding places," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 25–36, 2009.

[125] T. Yu, A. Sung, W. Srisa-an, and G. Rothermel, "Using property-based oracles when testing embedded system applications," in *Proceedings of the International Conference on Software Testing, Verification and Validation*, pp. 100–109, 2011.

[126] 2010. https://rhn.redhat.com/errata/RHBA-2010-0174.html.

[127] 2004. http://lists.gnu.org/archive/html/bug-coreutils/2004-04/msg00126.html.

[128] 2011. http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=461585.

[129] 2008. https://bugzilla.redhat.com/show_bug.cgi?id=438076.

[130] 2006. http://bugs.debian.org/cgi-bin/bugreport.cgi ?bug=357140.

[131] 2001. https://bugzilla.redhat.com/show_bug.cgi?id=54127.

[132] 2005. http://www.securityfocus.com/archive/1/395489.

[133] 2007. https://bugs.launchpad.net/ubuntu/+source /coreutils/5.97-5.6ubuntu1.

[134] 2009. https://bugs.gentoo.org/show_bug.cgi?id=231775.

[135] 2008. http://ftp.gnu.org/gnu/bash/bash-3.2-patches/bash32-051.

[136] 2011. http://ftp.gnu.org/gnu/bash/bash-4.2-patches/bash42-023.

[137] S. Narayanasamy, Z. Wang, W. Tigani, A. Edwards, and B. Calder, "Automatically classifying benign and harmful data races using replay analysis," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 22–31, 2007.

[138] T. Yu, W. Srisa-an, and G. Rothermel, "Simlatte: A framework to support testing for worst-case interrupt latencies in embedded software," in *Proceedings of the International Conference on Software Testing, Verification and Validation*, 2014.

[139] G. Fraser and A. Arcuri, "EvoSuite: Automatic test suite generation for object-oriented software," in *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, pp. 416–419, 2011.

[140] "Minimizing interrupt response time." http://www.ghs.com/download/articles/GHS_Minimixe_Interrupt_031405.pdf, 2005.

[141] S. Altmeyer, R. I. Davis, and C. Maiza, "Cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems," in *Proceedings of the International Real-Time Systems Symposium*, pp. 261–271, 2011.

[142] http://www.nongnu.org/avr-libc/user-manual/group__largedemo.html#largedemo_src, 2007.

[143] O. Quddus, R. Krieger, and C. Class, "Hand-motion chess." http://people.ece.cornell.edu/land/courses/ece4760/FinalProjects/f2012/oaq3_cig23_rk447/oaq3_cig23_rk447/index.html, 2012.

[144] J. Smith and T. C. Fogarty, "Adaptively parameterised evolutionary systems: Self-adaptive recombination and mutation in a genetic algorithm," in *Proceedings of the*

International Conference on Parallel Problem Solving from Nature, pp. 441–450, 1996.

[145] R. L. Haupt and S. E. Haupt, *Practical Genetic Algorithms*. John Wiley, 1998.

[146] N. Holsti, T. Lngbacka, and S. Saarinen, "Using a worst-case execution time tool for real-time verification of the Debie software," in *Proceedings [of] DASIA 2000, DAta Systems In Aerospace*, 2000.

[147] T. Yu, W. Srisa-an, and G. Rothermel, "SimRT: An Automated Framework to Support Regression Testing for Data Races," in *Proceedings of the International Conference on Software Engineering*, 2014.

[148] A. Salcianu and M. Rinard, "Pointer and escape analysis for multithreaded programs," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 12–23, 2001.

[149] J. Huang, P. Liu, and C. Zhang, "LEAP: Lightweight deterministic multi-processor replay of concurrent Java programs," in *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, pp. 207–216, 2010.

[150] R. L. Halpert, C. J. F. Pickett, and C. Verbrugge, "Component-based lock allocation," in *Proceedings of the International Conference on Parallel Computing Technologies*, pp. 353–364, 2007.

[151] Soot: A Java optimization framework. http://www.sable.mcgill.ca/soot/.

[152] M. Naik, A. Aiken, and J. Whaley, "Effective static race detection for Java," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 308–319, 2006.

[153] Guarded Blocks. http://docs.oracle.com/javase/tutorial /essential/concurrency/guardmeth.html.

[154] H. Do, S. G. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact.," *Empirical Software Engineering: An International Journal*, vol. 10, no. 4, pp. 405–435, 2005.

[155] http://jigsaw.w3.org.

[156] http://logging.apache.org/log4.

[157] http://weblech.sourceforge.net.

[158] S. Fouché, M. B. Cohen, and A. Porter, "Towards incremental adaptive covering arrays," in *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, pp. 557–560, 2007.

[159] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed random testing for Java," in *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, pp. 815–816, 2007.

[160] C.-S. Park and K. Sen, "Randomized active atomicity violation detection in concurrent programs," in *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, pp. 135–145, 2008.

[161] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold, "Test-suite augmentation for evolving software," in *Proceedings of International Conference on Automated Software Engineering*, pp. 218–227, 2008.