Improving Software Reliability for Event-Driven Mobile Systems

by

Chun-Hung Hsiao

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2017

Doctoral Committee:

Associate Professor Satish Narayanasamy, Chair
Associate Professor Michael J. Cafarella
Professor Peter M. Chen
Associate Professor Robert Dick
Dr. Cristiano L. Pereira, Intel

Chun-Hung Hsiao
chhsiao@umich.edu
ORCID iD: 0000-0002-3507-9690

© Chun-Hung Hsiao 2017

ACKNOWLEDGEMENTS

First of all, I would like to express my gratitude to my advisors, Professor Satish Narayanasamy, and Professor Michael Cafarella. Both are brilliant and energetic researchers, and kind advisors. With their guidance, I have learned much about doing research. They granted me great freedom to explore interesting research problems, and guided me along my Ph.D. journey. They have been very kind and patient to me, are always willing to understand my thoughts when I have difficulty to articulate them clearly, recognize the values of my ideas, and help me to refine these ideas. They also taught me how to think about problems, how to address challenges, and how to present ideas. My Ph.D. study would not be successful without them.

I would like to thank my family for their endless support, especially my father, Sheng-Wu Hsiao. He provided me a great educational environment, and encourage me to pursue my Ph.D. study. Without his support, my life would be entirely different, and I would not have got any of my academic achievements.

I appreciate the opportunities to collaborate with Professer Peter Chen, Professor Jason Flinn, Dr. Cristiano Pereira, and Dr. Gilles Pokam. These people are brilliant researchers and gave me many advisers during our collaboration, and I enjoyed work with them. In addition, I would like to thank my committee members for their valuable feedback on my dissertation.

I would like to give a special thanks to Dr. Jie Yu. He was a senior member of our lab, and we collaborated for the first piece of research work presented in this dissertation. He helped me a lot during the collaboration, and without him, my thesis would have been totally different, and I would have missed a lot of interesting research problems in this area.

I cherish all friends I have met here during my Ph.D. journey. Thank you for always think of me and let me share my good and bad times with you. You all enrich my life in Ann Arbor.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS i

LIST OF TABLES vi

LIST OF FIGURES vii

ABSTRACT xii

CHAPTERS

1 Introduction 1
- 1.1 Software Reliability in Event-Driven Programs 2
- 1.2 Challenges in Detecting Concurrency Bugs for Event-Driven Programs 4
- 1.3 Thesis Statement 7
- 1.4 Contributions of and Organization 8

2 Background and Related Work 10
- 2.1 Event-Driven Programming Models 10
  - 2.1.1 Generic Model for Mobile Platforms 10
  - 2.1.2 Case Study: Android's Event-Driven Framework . . . 12
  - 2.1.3 Other Programming Models 14
- 2.2 Data Race Detection for Thread-Based Programs 14
  - 2.2.1 Addressing Benign Races 18
- 2.3 Bug Detection for Event-Driven Programs 19
  - 2.3.1 Data Race Detection 19
  - 2.3.2 Bug Detection for Mobile Applications 20

3 Race Detection for Event-Driven Mobile Applications 22
- 3.1 Modeling Causality 23
  - 3.1.1 Overview 23
  - 3.1.2 Event-driven Program Trace 24
  - 3.1.3 Generic Causality Model 25
  - 3.1.4 Case Study: Android's Causality Model 28
  - 3.1.5 Generalized Asynchronous Causality Rule 31

3.2 Race Detection 31
3.2.1 Use-Free Races 32
3.2.2 Finding Use-Free Races 33
3.2.3 Pruning False Positives 34
3.3 Implementation 36
3.3.1 Overview 37
3.3.2 Instrumentation for Capturing Causalities 38
3.3.3 Logging Potentially Racy Operations 40
3.4 Evaluation 41
3.4.1 Experimental Setup 41
3.4.2 A Survey of Use-After-Free Violations 42
3.4.3 Accuracy and Performance 43
3.5 Summary 46

4 AsyncClock: Scalable Inference of Asynchronous Event Causality 47
4.1 AsyncClock Design 48
4.1.1 Overview 48
4.1.2 AsyncClock Primitive 49
4.1.3 Maintaining AsyncClocks 50
4.1.4 Race Detection 51
4.2 Improving Scalability 51
4.2.1 Reclaiming Heirless Events 52
4.2.2 Reducing AsyncClock Size 55
4.3 Generalizing AsyncClock 57
4.3.1 Generalized AsyncClock 57
4.3.2 Case Study: Realizing AsyncClock for Android . . 58
4.4 Race Detection 60
4.5 Evaluation 61
4.5.1 Experimental Setup 62
4.5.2 Overall Performance 62
4.5.3 Comparison with EventRacer 62
4.5.4 Scalability Improvements 66
4.5.5 Reported Races 67
4.6 Summary 69

5 Statistical Commutativity Analysis for Asynchronous Events 70
5.1 Running Examples 72
5.2 Event Commutativity Analysis 74
5.2.1 Event Precondition 74
5.2.2 Event Commutativity 75
5.3 Learning Precondition Specifications 77
5.3.1 Types of Atomic Symbolic Conditions 78
5.3.2 Symbolic Naming 79
5.3.3 Daikon Trace Generation 80
5.3.4 Obtaining Training Executions 80

5.4 Detecting Non-Commutative Events 81
5.5 Limitations 83
5.6 Evaluation 83
5.6.1 Experimental Setup 84
5.6.2 Sensitivity of Precondition Specification Learning . . 84
5.6.3 Accuracy 85
5.6.4 False Alarms 87
5.6.5 False Negatives 87
5.7 Summary 88

6 Conclusion 89

BIBLIOGRAPHY 91

LIST OF TABLES

3.1 The priority function for Rule PRIORITY. $E_1 \prec E_2$ if $\texttt{send}(E_1) \prec \texttt{send}(E_2)$ and the corresponding cell is true. 30

3.2 Races reported by CAFA. (a) Races that lead to in-thread violations. (b) Races that lead to inter-thread violations. (c) Races that lead to conventional violations. 45

4.1 Summary of trace collection and analysis sorted by looper events. Columns L, B and W under Column *Threads* stand for looper, binder, and worker threads. Column *Analysis* shows the performance of ASYNC-CLOCK with a 2-minute time window and FIFO chain decomposition. Column $-$*Mem%* shows the percentage of reduced memory usage (larger is better). All analyses are run offline on a local machine. . . . 63

4.2 The user-induced race groups reported in 8 applications. Row *Filtered* shows how many race groups were removed by our filter. Rows *Harmful* and *Harmless* are the actual number of race groups reported by our tool. 68

5.1 Reduction in the amount of human effort needed in inspecting the races reported by Licorice, compared to those by a data race detector. . . . 86

LIST OF FIGURES

1.1 Illustration of an Event-Driven Model [27] 2

1.2 A concurrency bug between the onServiceConnected and onDestroy events in Google's MyTracks app. (a) A correct execution. (b) An incorrect execution occurs when onServiceConnected and onDestroy are processed in the reversed order. 3

1.3 Synchronous versus asynchronous executions. 5

1.4 Two non-commutative events in Google's MyTracks app constitute a concurrency bug. The arrows show the causal order between onResume and onUpdateResults. (a) A correct execution where onResume and onUpdateResults are not interleaved with onPause. (b) An incorrect execution occurs when onPause is executed between onResume and onUpdateResults. 7

1.5 Illustration of non-commutative events in Google's MyTracks app. The arrows show the happens-before relation between onResume and onUpdateResults. Since onUpdateResults and onPause are not ordered and onPause violates the "dataHub != null" precondition of onUpdateResults, these two events are non-commutative. 9

2.1 Event queues can be bound with: (a) a single looper thread, which guarantees FIFO order and atomicity between events, or (b) multiple binder threads, which have no ordering and atomicity guarantee. . . . 12

2.2 Illustration of chain decomposition. The optimal chain decomposition partitions the 4 events into 2 chains, while an online greedy approach might partition them into 3 chains. 17

3.1 Causality rules for event-driven programs. $task(\alpha)$ is the worker thread or event that executes operation α. $looper(E)$ is the looper thread that executes event E. _ is a don't-care. 26

3.2 Illustration of asynchronous causality rules. The derived happens-before orders are noted in each figure, and the dotted arrows indicate these derived relation. (a) The event queue rule. The statuses of the event queue are shown at right. (b) The atomicity rule. 27

3.3 Extended causality rules for Android. *priority* is the priority function defined in Table 3.1. 28

3.4 Illustration of Rules PRIORITY and ATFRONT. The happens-before orders that can or cannot be derived are noted in each figure, and the dotted arrows indicate these derived orders. The statuses of the event queue are show at right, and for each event, the earliest time when it can be processed is marked on top of each cell. (a) $A \prec B$ as their `send` operations are ordered and their delays are the same. (b) A is processed after B owing to its higher delay, and thus no happens-before order can be derived between A and B. (c) Since `send`$(B) \prec$ `begin`(A), B is guaranteed to be enqueued in the front of the queue before A can be processed, so we have $B \prec A$. (d) and (e) show two scenarios where `send`$(B) \prec$ `begin`(A) is not true, and thus no happens-before order can be derived between A and B. 29

3.5 A read-write race in the ConnectBot application. The memory write in onPause cannot be executed between the if statement and the succeeding statements that update the size information in onLayout because the two events in the same looper thread are executed atomically with respect to each other. Thus, this read-write race is not a bug. 32

3.6 A use-after-free violation in Google's MyTracks application. (a) A correct execution. (b) An incorrect execution where an use-after-free violation manifests owing to the lack of happens-before order between onServiceConnected and onDestroy. 33

3.7 An example of commutative events that contain uses and frees. 34

3.8 An illustration of the if-guard check. In each case, the branch instruction is located at pc and conditionally jumps forward or backward to pc + offset or pc - offset respectively. ∞ indicates the end of the current function. If a use of pointer in the shadowed area is executed after the branch instruction at runtime, pointer would be guaranteed non-null, so the use would be safe. 36

3.9 CAFA architecture overview. RED represents newly added components, BLUE represents unmodified components, and GREEN represents instrumented components. 38

3.10 The slowdown for CAFA to collect traces on various applications. . . . 45

4.1 Illustration of AsyncClock. The ACs depict the AsyncClock of T_1 and T_2 after processing their preceding operations. ϵ indicates that there is no predecessor in the corresponding chain. 49

4.2 Example of infinitely many non-heirless events. The executed trace is shown in black, and a possible future execution is shown in gray. . . . 53

4.3 (a) A heirless event (A_1) with positive reference count. Only nonempty ACs are shown. (b) Illustration of multi-path reduction. 54

4.4 Illustration of time window approximation. 55

4.5 Illustration async-before lists for `Delayed` events. The causal predecessors of an event with `time` = 3 are shown in gray. 58

4.6 Speculative causality rules for events in binder threads [15]. Functions *process* and *thread* return the process and thread of an operation. Function *binder* returns the binder thread of an event. 60

4.7 Comparison between EventRacer and AsyncClock. The x-axes show the number of looper events. The top row shows the average time spent *per event*. The bottom row shows total memory used during the analyses. The results of 3 configurations of AsyncClock are shown: no event reclaiming (△), reclaiming heirless events (□), and reclaiming events with a 2-minute time window (○). 65

4.8 An event pattern in BarcodeScanner, where I_n are input events. EventRacer traversed the shaded area to find predecessors of B_3. 65

4.9 The percentages of races reported from 8 selected applications versus the total time and memory used for various time windows. The memory axis is in log scale. 67

5.1 A concurrency bug in Google's MyTracks app. (a) The precondition "dataHub != null" of onUpdateResults is always true in a correct execution. (b) The precondition is falsified by onPause in an incorrect execution. 71

5.2 Illustration of non-commutative events. The solid arrows show the happens-before relation between the events. (a) In Google's MyTracks app, events onUpdateResults and onPause contain a harmful data race on variable this.dataHub and these two events are non-commutative as reordering them violates the precondition "this.dataHub != null of onUpdateResults.". (b) In the ConnectBot app, onLayout and onPause contain a harmless data race on variable manager.resizeAllowed but no precondition of any event is violated when they are reordered. 73

5.3 Illustration of minor *GEN* and minor *KILL* events. G_c, G'_c are events that generate condition c, and K_c, K'_c are events that kill condition c. The triangles above and below an event indicates the "area" that happens before and after that event respectively. (a) If all events that require c are in the gray area, then G_c is a minor *GEN* of c. (b) If all events that require c are in the gray area, then K_c is a minor *KILL* of c. 77

5.4 Sensitivity study for the learning phase. (a) The number of learned preconditions versus the size of training executions. The number of learned preconditions at each training size are normalized with respect to that at 50 executions. (b) The percentage of preconditions failed by the testing executions versus the training size. 85

5.5 Regular versus shuffled executions for precondition specification learning. (a) The percentage of learned preconditions from shuffled executions with respect to regular executions. (B) The percentage of preconditions failed by the testing executions. 86

ABSTRACT

Mobile platforms commonly support an event-driven model of concurrent programming. In an event-driven system, the flow of a program is controlled by asynchronous events. Events processed sequentially in the same thread can be *logically concurrent* to each other, as they may not be ordered by any programmer-specified ordering operations. The lack of programmer-defined order between multiple *non-commutative* concurrent events — that is, only certain execution orders between these events yields correct results — leads to a unique class of concurrency bugs in event-driven programs.

Unfortunately, the state of the art for detecting concurrency errors in event-driven systems is significantly weaker than that in traditional thread-based systems. This thesis aims to fill this important gap by developing models, algorithms and tools that aid programmers to analyze and diagnose event-driven programs to improve software reliability. Specifically, this thesis presents the following three techniques to detect concurrency errors in event-driven programs:

1. A new *causality model* for event-driven program is defined to infer ordering invariants between events across different executions.

2. An efficient and scalable single-pass algorithm to identify concurrent asynchronous events that may lead to concurrency errors.

3. A *statistical commutativity analysis* to find likely non-commutative events that contains concurrency bugs.

The techniques we have developed are broadly applicable to many event-driven platforms. To translate our techniques into real-world impact, we develop a set of tools

in the context of Android to help build up a more robust and reliable platform for event-driven computing.

CHAPTER 1

Introduction

Mobile computers are increasingly important computing platforms in the last few years. It is estimated that 10 billion mobile devices are in use globally in 2016, and there would be over 268 billion downloads of mobile applications by 2017 [8]. For many people, phones and tablets are the primary platform for interacting with computer systems and the data they store. Mainstream mobile devices have a rich array of sensors and user input modalities, which provide large asynchronicity to the input stream of mobile applications. As a result, *event-driven* programming models arise naturally among popular mobile platforms, as they are better in handling I/O concurrency [20]. Figure 1.1 illustrates the design of an event-driven programming model, where events are generated from *event sources* (such as user actions, sensor inputs, or messages from other programs or threads) and sent to an *event dispatcher*, and the dispatcher *asynchronously* processes these events and executes corresponding *event handlers*. The flow of the program is determined by the execution of the event handlers.

Compared to a traditional thread-based model, an event-driven model uses asynchronous execution instead of concurrent execution to manage I/O concurrency, makes it easy to integrate input from diverse sources such as touchscreens, accelerometers, microphones, and other sensors to write interactive applications. Programmers are responsible to implement their own event handlers to process specific types of events, and do not need to worry about the orchestration of the program flow or handling

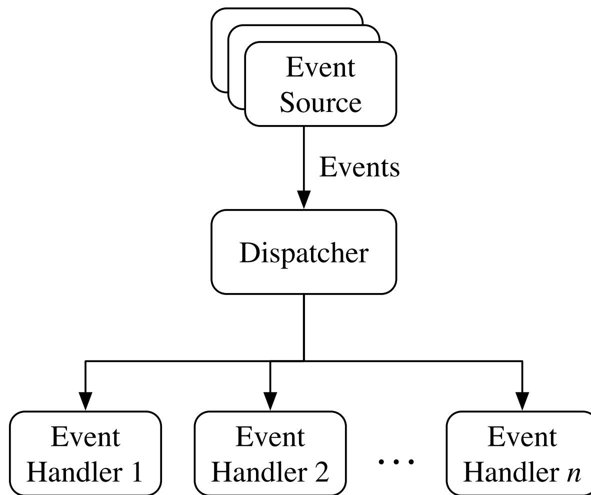


Figure 1.1: Illustration of an Event-Driven Model [27]

synchronization between threads.

1.1 Software Reliability in Event-Driven Programs

Although event-driven programming models provide an ease for programmers to write interactive applications, the wide use of asynchrony in these models leads to a unique type of pernicious concurrency bugs that is hard to find and debug. As events could be generated from multiple concurrent sources, they may not be ordered by any programmer-specified ordering operations, and thus the order of execution of certain event handlers becomes nondeterministic, even if these events are processed sequentially in a single thread. In other words, many events are *logically concurrent* to each other in an event-driven program, which may lead to concurrency errors. Figure 1.2 shows a concurrency bug between two logically concurrent events in Google's MyTracks app. In this example, the onResume and onDestroy events are generated in response to user actions, and onServiceConnected is generated from another thread in the program. Since they are generated concurrently, their processing order is nondeterministic, and a buggy result would manifest when onDestroy is processed before onServiceConnected.

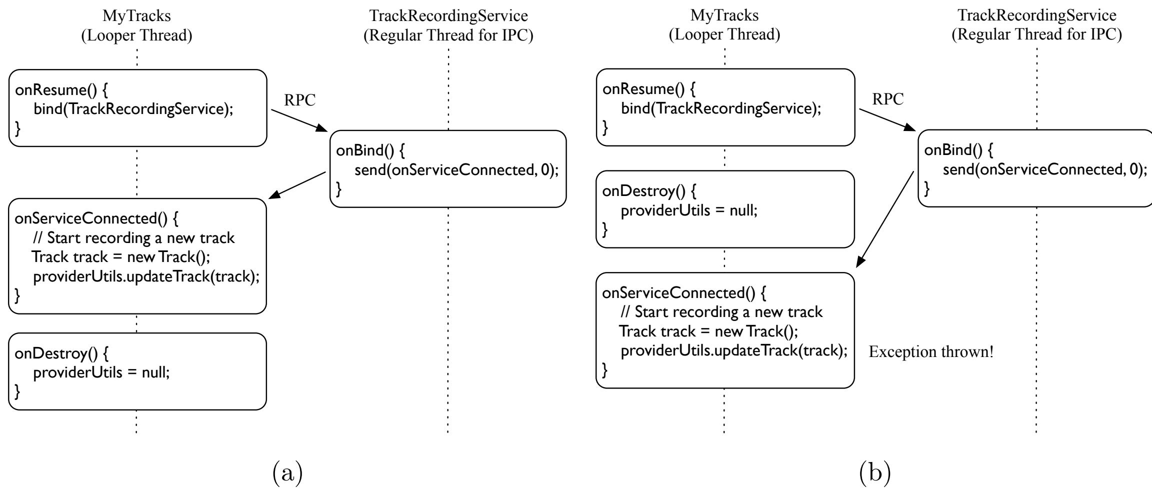


Figure 1.2: A concurrency bug between the onServiceConnected and onDestroy events in Google's MyTracks app. (a) A correct execution. (b) An incorrect execution occurs when onServiceConnected and onDestroy are processed in the reversed order.

Such concurrency bugs between asynchronous events, just like any other concurrency errors, are hard to find and reproduce, so being able to identify these concurrency bugs can help programmers to improve software reliability. Unfortunately, most existing tools for finding concurrency errors focus on detecting data races (conflicting memory accesses between different threads without proper synchronization) in thread-based programs, and are not applicable to event-driven programs. For example, dynamic data race detectors such as FastTrack [28] cannot find race conditions between asynchronous events that are processed in one thread, and usually assume a bounded number of concurrent tasks, which is also not true in event-driven programs. As event-driven program developers become a dominant population in software programming, it is imminent to develop new algorithms and tools to help developers deliver reliable software.

1.2 Challenges in Detecting Concurrency Bugs for Event-Driven Programs

Asynchrony differs event-driven programming models from conventional thread-based models, and brings new challenges in developing new program analyses to find concurrency bugs. This section summarizes the challenges we encountered and addressed in this dissertation.

Inferring Event Ordering Invariants. In event-driven programming models, events generated from concurrent sources are processed in a nondeterministic order. However, a shared memory event-driven system with no ordering guarantee for events generated by the same source would result in a "state explosion" problem for both software programming and verification [22]. To reduce the complexity of event-driven programming, mainstream event-driven platforms, including Android, iOS and HTML5, guarantee partial order of events from the same event source. For example, the HTML5 specification regulates the event loop to process all tasks from the same task source in a timely order [63], and Android also implements a first-in-first-out manner of processing events [3], so in Figure 1.2, onDestroy always comes after onResume. With the partial-order guarantees provided by the runtime systems, it is much easier for programmers to reason about the behavior of their applications. Programmers also rely on such guarantees to orchestrate multiple events to perform certain tasks.

Therefore, understanding the *ordering invariants* between asynchronous events in an execution is an important step to identify concurrency bugs in an event-driven program, since these bugs could not exist between ordered events. Conventional data race detectors [14, 34, 55, 28] takes a similar approach in finding data races in thread-based programs: a data race is redefined as a pair of unordered conflicting memory accesses with respect to the conventional *causality model* for a thread-based program [14, 34]. Unfortunately, naively applying these tools for event-driven systems works poorly, because they implicitly assume that events handled in one thread are

Figure 1.3: Synchronous versus asynchronous executions.

ordered by the program order. Thus, a causality model that accounts for the unique causal relation in a event-driven system, and a race detector based on it has been lacking.

Efficiency and Scalability. We observe that inferring the happens-before relation for an event-driven program is fundamentally harder than that for thread-based program. A thread-based program consists of synchronous calls, and the happens-before relation is defined by pairs of synchronization operations (signal-wait, release-acquire, fork-joins) that operate on common handles (thread IDs and synchronization variables). As can be seen in Figure 1.3a, the ordering invariant "A happens before B" can be trivially inferred by deriving the *logical times* of `begin` and `wait` operations from the times of its *immediate causal predecessors* (`invoke` and `end` operations) through the common handles (A and B) they operate on. Past research [28] has developed efficient algorithms to maintain logical times and happens-before relation between memory accesses to identify data races, based on the fact that there are usually a bounded number of concurrent threads and synchronization handles in a thread-based program.

In an event-driven program consisting of asynchronous events, however, it is more

challenging to infer ordering invariants between two events, such as A and B in Figure 1.3b. Since there is no explicit program handle to relate them, it is hard to identify the immediate causal predecessors of B among past events. A naive approach, as presented in related work [54, 57, 44, 15], is to build a happens-before graph of all events and synchronization operations, and iteratively traverse this graph for every event. Apparently, this approach is not efficient and scalable, especially as the graph size grows with the program execution.

Another challenge in designing a scalable algorithm for inferring ordering invariants for an event-driven program is keeping track of logical times of events such that it scales in terms of performance and space with the program execution length. Unlike a thread-based program that executes only dozens of threads, an event-driven program may execute hundreds to thousands of events every minute. Since the times of past events cannot be simply discarded as they will be used to compute the logical time of their future succeeding events, it is important to identify events that have no future *immediate causal successors* and reclaim their metadata to achieve good scalability.

Identifying Non-Commutative Events. Past research [41] has showed that most non-deadlock concurrency bugs in thread-based programs are either *atomicity violations* (i.e., the programmer-intended serializability among certain operations could be violated during execution) or *order violations* (the programmer-intended execution order among two operations is not enforced during execution). Most concurrency bug detectors for thread-based programs focus on finding data races, as data races are good indicators of these two types of concurrency bugs. However, since events from the same event queue are processed *non-preemptively* by a single looper thread, they are *atomic* with respect to each other. As a result, there are no atomicity violation within a event, and most data races are in *commutative* events and thus harmless, where two events are commutative if they produce correct results irrespective to their execution order. Figure 1.2 shows two non-commutative events, onServiceConnected and onDestroy, in Google's MyTracks app, and they constitute a concurrency bug because they are not causally ordered. Figure 1.4 shows another example of

non-commutative events, onUpdateResults and onPause, in the same app, which also constitute a concurrency bug. To find concurrency bugs in event-driven programs, we need to distinguish harmful data races in non-commutative events from harmless races in commutative events, which is a new and challenging problem.

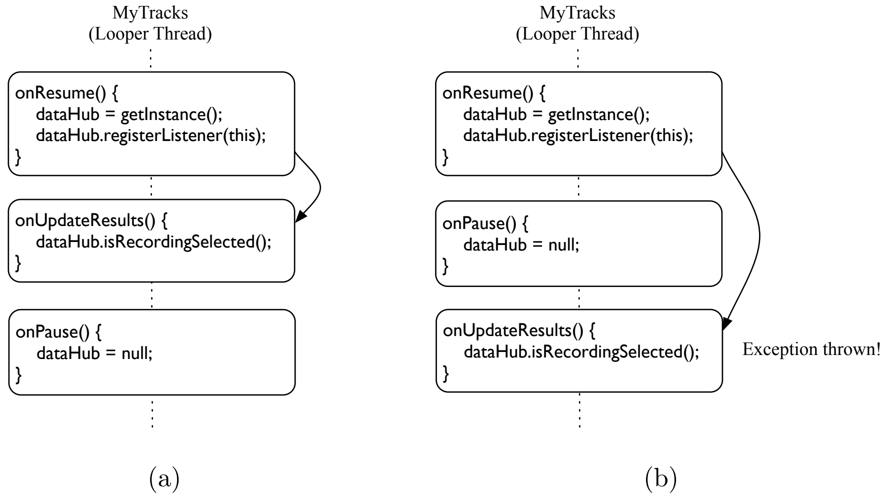


Figure 1.4: Two non-commutative events in Google's MyTracks app constitute a concurrency bug. The arrows show the causal order between onResume and onUpdateResults. (a) A correct execution where onResume and onUpdateResults are not interleaved with onPause. (b) An incorrect execution occurs when onPause is executed between onResume and onUpdateResults.

1.3 Thesis Statement

Despite the high popularity of event-driven programming models, tools for detecting concurrency bugs due to asynchrony for event-driven programming models is still lacking. This dissertation aims to fill this important gap by developing a causality model and an efficient and scalable algorithm to infer ordering invariants for asynchronous events, and a new commutativity analysis for asynchronous events to identify concurrency bugs in event-driven programs.

1.4 Contributions of and Organization

We have developed techniques to help programmers to find concurrency bugs in event-driven applications. The techniques we have developed are broadly applicable many event-driven platforms. To translate our techniques into real-world impact, we have implemented and evaluated a set of tools in the context of Android to help build up a more robust and reliable platform for event-driven computing.

In Chapter 3, we present the first causality model for event-driven programs that infers the happens-before relation between asynchronous events. A unique aspect of this model is that it accounts for the happens-before relation that are enforced by mainstream event-driven runtime systems, for example, the first-in-first-out order of event processing. In addition, our causality model also accounts for the happens-before relation due to conventional synchronization operations, so it is useful in any runtime system that is a mixture of both thread-based and event-based models. We also present a specialization of our causality model for Android to show that our causality model is adequate for real-world applications. We implement the first data race detector for Android based on our new causality model and show that our new model could find real concurrency bugs in popular open-source Android applications.

In Chapter 4, we develop the AsyncClock algorithm to efficiently infer the happens-before order between asynchronous events based on the causality model presented in Chapter 3. Since the logical time of a newly created event is derived from the times of its causally preceding events in the past, we introduce a new primitive called AsyncClock to keep track of these causal predecessors in a succinct data structure, and design an efficient non-iterative algorithm to infer its logical time. We also design optimizations to efficiently identify events whose metadata is no longer needed for driving the logical times of its causal successors in the future, and reclaim their metadata to ensure scalability. The resulting algorithm enables us to build the first single-pass, non-graph-based dynamic data race detector for event-driven programs, with a performance comparable to the state-of-the-art dynamic data race detectors for thread-based programs [28].

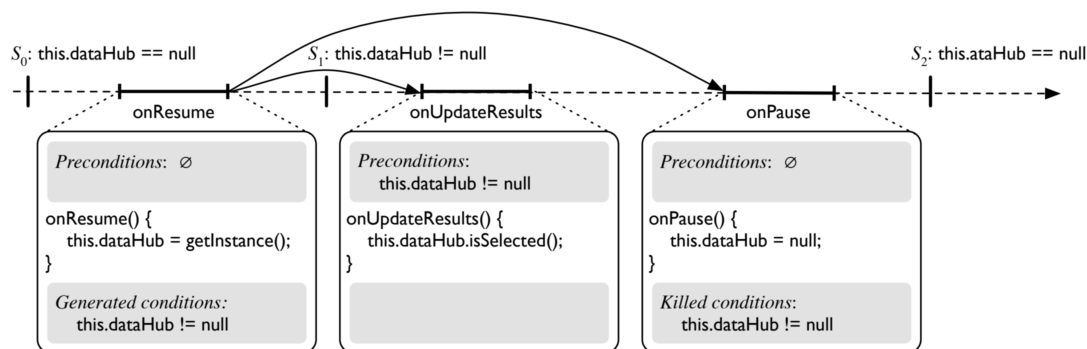


Figure 1.5: Illustration of non-commutative events in Google's MyTracks app. The arrows show the happens-before relation between onResume and onUpdateResults. Since onUpdateResults and onPause are not ordered and onPause violates the "dataHub != null" precondition of onUpdateResults, these two events are non-commutative.

Chapter 5 addresses the problem of false alarms due to benign data races in event-driven programs. In an event-driven program, most data races defined under our new causality model are harmless, because although different execution orders of the racing events might result in different program states, both states follow programmer's intention, and thus these races lead to no concurrency bug. On the other hand, although invariant-based bug detectors have been successful for thread-based programs [26], they have not been applied to find concurrency errors in event-driven systems. Therefore, we combine the ideas of a happens-before race detector and an invariant-based bug detector, and present a new technique — a *statistical commutativity analysis* — to find non-commutative events in event-driven programs and concurrency bugs within them. Figure 1.5 shows an example of non-commutative events. Since onUpdateResults and onPause are not properly ordered, onPause could violate the "dataHub != null" condition, which is required by onUpdateResults, indicating that these two events are non-commutative.

Finally, to translate our techniques into real-world impact, we developed a set of tools in the context of the widely-used Android platform [1], to help build up a more robust and reliable platform for event-driven computing.

CHAPTER 2

Background and Related Work

In this chapter, we give an overview of the event-driven programming models, and discuss closely related work, including concurrency bug detection in thread-based and event-based programs in the literature.

2.1 Event-Driven Programming Models

We first describe introduce a simplified event-driven programming model that captures the salient features of most mainstream event-driven platforms, and then present how a real-world event-driven framework is designed in Android. We also briefly discuss other programming models designed for event-driven systems.

2.1.1 Generic Model for Mobile Platforms

Mainstream event-driven platforms, such as Android, iOS, and HTML5 web apps, provide event-driven runtime frameworks running on shared memory architectures. An execution of an event-driven program developed in such a runtime framework consists of several threads. A subset of these threads are designated to process *events* from *event queues*. Depending on the relationships between these threads and the *event queues*, they are referred to as *looper threads* or *binder threads*. The other threads, which are referred to as *worker threads*, are spawned by application code and behave like conventional threads. We describe each of these components below.

Event. An event is a lightweight asynchronous task, which can be generated by the operating system in response to an external input (e.g., user actions, sensor input, network, or IPC from other applications), or by another worker thread or event in the application through `send` operations. Every event is associated with a *message* when they are generated, which provides information to determine what *event handlers* to call and what are the arguments of the calls when the event is processed. In many event-driven platforms, events may also be associated with other properties such as time constraints that would affect the order they are processed. We will describe a concrete realization in Section 2.1.2.

Event queue. Once an event is generated, it is not processed synchronously. Instead, it is placed in a specific *event queue*, and later processed by the event processing thread. Generally, events from the same queue are processed in the order they are queued, also known as the first-in-first-out (FIFO) order. Although certain event-driven platforms provide mechanisms to change the processing order of events, FIFO order is usually used as a tiebreaker for the sake of fairness among events.

Looper thread. An event queue is bound with a single thread, called the *looper thread* (Figure 2.1a). Events that access shared data are usually placed in such event queues. The role of a looper thread is to continuously check its event queues, select and process one event at a time. A looper thread can have multiple queues, and when it does, the event can be selected from any of its queues, so there is no ordering guarantee between events from different queues, even if they are processed by the same looper thread. To process the event, the looper thread first runs the *event dispatcher* in the runtime framework to determine which *event handlers* to call and what are the arguments of the calls based on the event's message, and then call each event handler with appropriate arguments. All events processed in a looper thread are atomic with respect to each other. This is an important property of event-driven systems that programmers often rely on. However, events processed in a looper thread are not atomic with respect to other events in another looper thread.

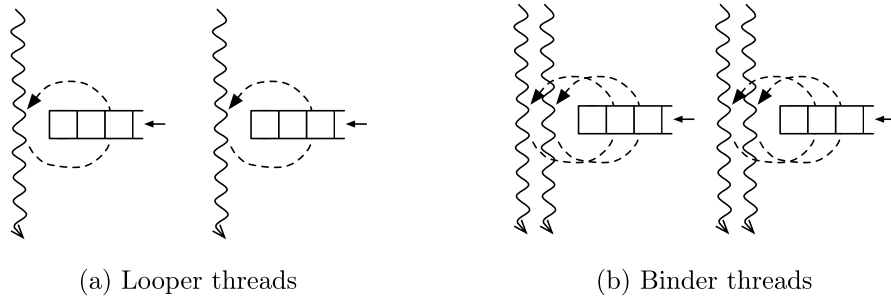


Figure 2.1: Event queues can be bound with: (a) a single looper thread, which guarantees FIFO order and atomicity between events, or (b) multiple binder threads, which have no ordering and atomicity guarantee.

Event handler. When designing an application, the programmer may want to change the states of multiple components in their application when certain events are processed. This can be done by *registering* user-defined event handlers to the system's event dispatcher in advance, such that when these events are processed, the event dispatcher can invoke multiple user-defined event handlers. Each event handler invocation, although triggered by the same event, usually perform its own task for distinct components in the application. The programmer may also *unregister* any user-defined event handlers once they are no longer needed.

2.1.2 Case Study: Android's Event-Driven Framework

In this section, we study a real-world implementation of an event-driven programming model — Google's Android OS [1]. Android's framework is a mixture of thread-based programming (with worker threads) and event-driven programming, and it complicates the event-driven programming model by adding additional features listed below to the generic model.

Event with priority tag. Android provides the ability to overrule FIFO order of processing events by allowing programmers to assign the following *priority tags* to events to change their queuing policy:

- `Delayed` and `AtTime` events: In Android, an event may be associated with a *time constraint* that determines when it can be processed. The time constraint can be either a delay with respect to when the event is generated (`Delayed`) or an absolute time (`AtTime`). An event can only be processed after its time constraint has been met. When there are multiple events from the same queue whose time constraints have been met, the looper thread would select an event according to FIFO order. Other event-driven platforms, such as iOS and HTML5, also support these types of events.

- `AtFront` events: Android allows programmers to place an event at the front of an event queue through a special sendAtFront API. Such an event becomes last-in-first-out (LIFO) and would be processed immediately after the looper thread finish processing its current event. However, as pointed out by the Android's developer manual, these events "*can easily starve the event queue, cause ordering problems, or have other unexpected side-effects*" [10], and thus they are rarely used by programmers.

- `Sync` and `Async` events: To improve responsiveness of an application, programmers can specify events with *asynchronous messages* (`Async`) to precede ordinary events (`Sync`) when a *barrier message* is placed in the event queue [11]. Although these events are processed out of order with respect to ordinary events, all events with asynchronous messages are still processed in FIFO order.

Binder thread. Android supports event queues that are bound with multiple event processing threads called *binder threads* [2] (Figure 2.1b). Each binder thread continuously check its event queue and select one event to process at a time according to FIFO order, just like what a looper thread does. However, although events from the same queue are still selected in FIFO order, they are not atomically processed with respect to each other, because these events may be processed by different threads in parallel. Using event queues with binder threads is more efficient for exploiting concurrency in multi-core processors, but extra synchronizations are required for accessing shared

data. Android programmers usually use event queues with binder threads to handle inter-process communications (IPCs) or process events that perform independent tasks and access no shared data.

2.1.3 Other Programming Models

The generic event-driven programming model described in Section 2.1.1 captures the essence of mainstream event-driven platforms. However, there are other event-driven programming models developed for programming device drivers and service protocols in distributed systems. P [22] is a domain-specific language to write asynchronous event-driven code, and P# [21] is an asynchronous programming language that integrates P into C#. Both P and P# represent event-driven programs as state machines, and programs written in these languages can be fully verified using model checking. However, it is not easy to represent interactive event-driven applications as state machines, as mobile applications usually have complex component structures and lead to state explosion. Therefore, their techniques cannot be easily applied to mobile systems like Android.

2.2 Data Race Detection for Thread-Based Programs

Many studies have been done in the literature to detect data races in thread-based programs, either statically [25, 62], or dynamically based on locksets [18, 58], causality models [14, 34, 28, 24, 53], or hybrid techniques [55, 50, 64]. This section reviews various dynamic techniques based on causality models, as these techniques are most closely related to our work.

Data Races

Conventionally, a data race is defined as a pair of memory access to the same location, at least one of them is a write, and they are not properly synchronized. A thread-based program with data races can yield different program states in different

executions, even with the same program input. Therefore, most modern programming languages assume a data-race-free-0 memory model [13] and provide no semantic guarantee for programs containing data races.

Causality Model and Happens-Before Race Detection

A *causality model*, or Lamport's *happens-before model* [39], defines a partial order relation, called *happens-before relation* ($\prec$), over all operations in a dynamic execution. The causality model used by prior happens-before data race detectors [14, 34, 55, 28] employs the following *causality rules*:

Program order: Two operations executed in the same thread are ordered.

Release-acquire: For a lock variable l, `unlock`(l) happens before its next `lock`(l) in the execution.

Signal-wait: For a condition variable m, `signal`(m) happens before its next `wait`(m) in the execution.

Fork-join: the `fork` operation happens before all operations in the forked thread, which happens before the `join` operation of the thread.

Under the happens-before relation, if a pair of accesses are properly synchronized, these accesses will be ordered. Therefore, past work [14, 34] redefine a data race as a pair of memory accesses to the same location, at least one of them is a write, and they are not ordered under the causality model. In our work, we use this definition of data race.

Happens-before data race detectors have been very successful for finding concurrency bugs in thread-based programs. However, they work poorly for event-based programs. For example, FastTrack [28] assumes that all memory accesses from the same thread are totally ordered, thus no race will be reported for accesses executed in the same thread. This assumption is too strict (missing potentially races) for event-based programs since events executed in the same thread in an event-based program can be logically concurrent.

Vector Clock, Epoch and Chain Decomposition

Vector clock [45] is a succinct data structure to express a happens-before relation, and thus a standard way to maintain logical times of all operations in happens-before data race detectors. A vector clock $VC : \mathit{Threads} \rightarrow \mathbb{Z}^+$ is a vector of logical timestamps, where each timestamp identifies the time of the causally preceding operation in a thread. It supports the following operations:

$$
\begin{aligned}
VC \sqsubseteq VC' \quad & \text{iff} \quad \forall i.\ VC(i) \leq VC'(i) \\
VC \sqcup VC' \quad & = \quad \lambda i.\ \max(VC(i), VC'(i)) \\
\bot_{VC} \quad & = \quad \lambda t.\ 0 \\
\mathrm{inc}_i(VC) \quad & = \quad \lambda j.\ \text{ if } j = i \text{ then } VC(j) + 1 \text{ else } VC(j)
\end{aligned}
$$

The logical time VC_x of an operation x in thread i is maintained in a way such that it is greater than the logical times of any causally preceding operations. This is achieved through *inheriting* the vector clocks of the causally preceding operations of x:

$$VC_x = inc_i(\bigsqcup_{y \prec x} VC_y).$$

For any two operations x and y, $x \preceq y$ if and only if $VC_x \sqsubseteq VC_y$.

To maintain the happens-before relation between operations, past happens-before data race detectors [14, 34, 55, 28] keep track of a vector clock for each thread any synchronization variable. However, it is inefficient to keep track of a vector clock for every data access. FastTrack [28] resolve this problem by tracking *epochs* instead of full vector clocks for data accesses. An epoch $c@t$ records the timestamp c for an access executed in thread t, and $c@t$ happens before a vector clock VC ($c@t \preceq VC$) if and only if $c \leq VC(t)$. Our work also use this optimization for race detection.

Naively adapting vector clocks for event-driven programs, however, requires dedicating a distinct timestamp for each event in a vector clock. Since there could be an unbounded number of events in an execution, this approach clearly does not scale. This scalability problem could be partly alleviated using *chain decomposition* [35, 57], where events are partitioned into *chains*, each consisting of a sequence of causally ordered events. Each chain can be thought of as a logical thread, so only one timestamp

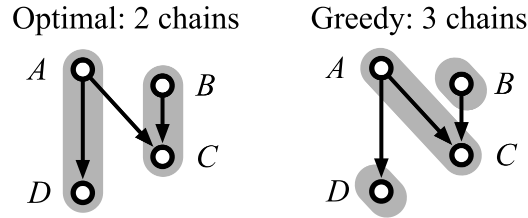


Figure 2.2: Illustration of chain decomposition. The optimal chain decomposition partitions the 4 events into 2 chains, while an online greedy approach might partition them into 3 chains.

is needed for each chain of events in a vector clock. Figure 2.2 shows an optimal chain decomposition partitioning 4 events into 2 chains, so it is sufficient to use 4 timestamps in a vector clock to track the logical times in all threads and chains. Unfortunately, computing an optimal chain decomposition requires the knowledge of the entire happens-before relation, and thus it cannot be done in an online manner. Instead, past work [35, 57] takes an online greedy approach to assign each event to an existing chain ending with one of its immediate causal predecessors, or to a new chain if there exists no such chain.

Our work adapts the above approach and uses *vector clocks with chain decomposition* to track the logical time of an operation, where a vector clock $VC : \mathit{Chains} \rightarrow \mathbb{Z}^+$ is now defined over all chains. Here we make no distinction between a chain of events or a worker thread, and refer to both as "chains."

Predictive Race Detection

The problem with happens-before race detection is that races can be missed due to accidental happens-before ordering. For instance, the ordering critical sections may transitively order conflicting accesses that are not properly synchronized. Another example is that in event-driven programs, the program order is too strict as it order all events executed in the same thread. To increase the coverage, prior work [37, 59, 17, 61] takes a *predictive* approach to detect data races by relaxing the causality model.

In this thesis, we also take this approach and develop a relaxed causality model for event-driven programs. As a result, predictive analyses can find more concurrency bugs, but at the cost of reporting benign races, unlike a sound happens-before race detector such as FastTrack [28]. Smaragdakis *et al.* [61] discussed a sound predictive race detection technique, but it is for thread-based programs and not applicable to event-based programs.

2.2.1 Addressing Benign Races

Data races are not necessarily harmful. Many data races are *benign* races and do not compromise program's correctness. Since predictive analyses report data races in a relaxed causality model, they are likely to report more benign races, hence the usability of these tools is reduced. However, it is hard to figure out if a data race is harmful, even for those with domain expertise. Some prior work tries to address this problem through record and reply [49]. In this section, we focus on techniques that are closely related to our work.

Commutativity Analysis

Two operations are said to be *commutative* if executing them in different orders leads to the same program state. Benign races defined under a relaxed causality model usually consists of commutative memory accesses. Therefore, to improve the accuracy of a predictive race detector, techniques for commutativity analysis are desirable.

Huang *et al.* [33] discussed a few heuristics in the past to check if two critical sections are commutative or not. However, their heuristics are designed for thread-based programs, and are neither sound nor complete. Dimitrov [23] introduced the concept of commutativity races, and developed a framework to systematically describe high-level operation commutativity for library functions. We adapt their idea and come up with a simplified commutativity specification that helps us to remove benign races between events in the same thread in Chapter 4.

Effect-Oriented Race Detection

Instead of detecting concurrency bugs through identifying common bug triggers such as data races, effect-oriented approaches [67, 66] focus on certain operations that can lead to severe program errors (e.g., memory errors or assertion violations), and find the triggers that cause the errors. For example, ConMem [67] looks for races between dereference and nullification of a pointer in thread-based programs. We also take effect-oriented approaches in our predictive analyses for event-driven programs to avoid a large volume of benign races: In Chapter 3, we focus on *use-after-free* violations; and Chapter 5 presents a new technique by combining predictive race detection and invariant-based bug detection, which is also an effect-oriented approach.

2.3 Bug Detection for Event-Driven Programs

Since event-driven computing has become increasingly popular recently, a few researchers have shifted their focuses, and start looking at race detection techniques as well as other techniques to detect various kinds of bugs in event-driven programs. This section reviews these recent techniques, which are independently developed from our work.

2.3.1 Data Race Detection

WebRacer [54] and EventRacer [57] are two recent studies focusing on detecting races for one type of event-based programs: web applications. A web application is typically executed by the browser in a single thread in an event-driven style. These authors have shown that even if there is only one thread executing, races are still possible; and they have presented a causality model for web applications and successfully found races in many popular web sites. Though closely related, our work differs with theirs in the following aspects. First, our causality model is more comprehensive, as it captures ordering constraints enforced by the event queue, and can handle programs

that are mixture of thread-based and event-driven model. Second, the types of bugs they targeted are also web application specific. In contrast, we focus on a more general type of concurrency bugs. DroidRacer [44] independently proposes a causality model for Android similar to the one described in Chapter 3. Our causality model is more complete since it handles AtFront events and provides a more generic rule for event atomicity.

EventRacer for Android [15] (referred to as EventRacer in this thesis) is one of the most closely related work. It presents a refined causality model specialized for Android based on our causality model presented in Chapter 3 to model the causal order provided by specific Android APIs; in contrast, we propose a generalized model in Chapter 3 that is not specific to any system, and our model can be easily specialized to model a wide variety of event-driven systems. The authors also have developed a new data race detection algorithm based on both happens-before graph and vector clock with chain decomposition: EventRacer finds immediate causal predecessor of an event by tracking the entire happens-before graph of all past events with their logical time. When an event is about to begin, it traverses the happens-before graph backward from its `send` operation to find causally preceding `send` operations. The events sent by these operations are its causal predecessors. To speed up the search of predecessors, EventRacer uses *graph traversal pruning* to stop the search along certain paths when a `send` is encountered. However, in the worst case, the entire graph may need to be traversed. We empirically show that EventRacer solution does not scale as the number of events increases with the program trace length in Chapter 4.

2.3.2 Bug Detection for Mobile Applications

Performance bugs are a class of bugs that received wild interests lately as mobile applications are usually user facing and latency critical. Performance bug detection tools like AppInsight [56] and Panappticon [65] analyze critical paths in the system to identify performance bottlenecks in the applications that could potentially cause

user perceived delays. Energy bugs are another type of bugs that have been studied recently. Pathak *et al.* have performed a series of studies [51, 52] on classifying and characterizing energy bugs in Android applications. Lide *et al.* have developed a technique that uses taint-tracking to find computations that never influence applications' outputs and thus are unnecessary and cause energy leaks.

CHAPTER 3

Race Detection for Event-Driven Mobile Applications

In this chapter, we present a new generic causality model for event-driven programs, which we use to infer the happens-before relation between events. This model abstracts away the details of different event-driven platforms, and focuses on the causal relation between events due to the event queues. We also present a specialized model for the widely-used Android platform [1], as the system and many of its applications are open-sourced. Based on our new causality model, we implemented the first race detection tool named CAFA for event-driven applications.

We find that naively reporting races between assembly-level read and write accesses to a memory location in concurrent events leads to thousands of benign races. The reason is that concurrent events processed in the same looper thread could be *commutative* with respect to each other. Two events are commutative if they produce a correct result irrespective of the order in which they are executed by the looper thread. If two concurrent events processed in the same looper thread are commutative, then any conflicting memory accesses executed within them are not indicative of a race error. Thus, only the conflicting operations in non-commutative events are indicative of race errors.

Automatically determining if two events are commutative is a challenging problem, because it depends on high-level semantics of those events. Therefore, in this study,

we limit our focus to finding race errors that lead to *use-after-free* violations, where pointers are dereferenced (used) after they no longer point to any object (freed). To improve the accuracy of CAFA, we employ two simple heuristics, if-guard and intra-event-allocation, to remove benign races.

We studied several Android applications, which included applications such as FireFox and MyTracks (Google's GPS tracker). We found 67 previously unknown harmful races (races that may lead to incorrect outputs when their accesses are executed in certain orders). We also detected 2 known harmful races. 60% of the races detected were harmful.

3.1 Modeling Causality

The event-driven programming model described in Chapter 2 differs substantially from the thread-based concurrency model that is assumed by most concurrency tools. Unlike in conventional thread-based causality models, we cannot assume that all the events executed in a thread are ordered by program order. We present a generic causality model for the event-driven programs that relaxes this order, but accounts for additional happens-before relation due to the event queues.

3.1.1 Overview

An execution of an event-driven program involves looper threads that process many events, as well as worker threads orchestrated by programmer-specified synchronizations such as locks, thread forks, and joins. Therefore, the causality model should account for the following 2 types of causal orders.

- *Synchronous* causal orders including program order in a worker thread or an event, and the orders due to programmer-specified synchronization operations, thread forks and joins, and event-related operations (`send` operations and event handler registrations). This type of causal orders is similar to what we have in the conventional thread-based causality model described in Section 2.2, ex-

cept for two major differences. First, *there is no program order between events processed by a looper thread*, because programmers do not intend to provide orders between events that are generated concurrently. Second, *no causal order is assumed between unlocks to succeeding locks*. That is because programmers do not intend to provide orders using locks in most scenarios, so assuming orders between critical sections protected by the same lock may introduce false happens-before relation.

- *Asynchronous* causal orders due to the *event queues*. Events sent to the same event queue are atomically processed in FIFO order by one looper thread, so it is incorrect to assume that there is never any order between events.

3.1.2 Event-driven Program Trace

We start with a definition of an execution trace of an event-driven program. A program trace is a sequence of operations listed below, where T is a thread, E is an event, S and S' is either a thread or an event. If an operation is part of an event, it is attributed to that event instead of its looper thread.

- `begin`(S), `end`(S): start or end of S.
- `rd`(S, x), `wr`(S, x): read or write to data variable x in S.
- `fork`(S, T), `join`(S, T): S spawns or waits for T.
- `signal`(S, m) and `wait`(S', m) S' waits till S signals through handle m. Registering event handlers and their invocations are also modeled as `signal`s and `wait`s, respectively. We omit S and S' when there is no ambiguity.
- `send`(S, q, E): S enqueues event E to queue q. We omit S and/or q when there is no ambiguity.

While we account for mutual exclusion between the critical sections protected by the same lock, we do not assume a happens-before relation due to locks, as discussed in 3.1.1. Instead, we check the locksets for mutual exclusion, assuming that the critical sections are race-free since programmers handle them explicitly through locks.

3.1.3 Generic Causality Model

A program trace defines a sequential order for it operations. For two operations α and β in a trace, we say $\alpha < \beta$ if α is executed before β. This order is not preserved across executions, however, due to nondeterminism of concurrent programs. Given an execution trace of an event-driven program, to identify potential concurrency bugs in other executions, we define a new *causality model* for event-driven programs.

The causality model defines a *happens-before* relation ($\prec$), which is the smallest transitive closure over the operations in the trace based on the causality rules in Figure 3.1, which we explain as follows. In the following text, we use the terms *causally precedes* and *happens before* interchangeably. And for two events E and E' from the same queue, we shorten $\texttt{end}(E) \prec \texttt{begin}(E')$ as $E \prec E'$.

The first set of rules account for the synchronous causal orders in an event-driven program:

Program order rule (PO): In an event-driven program, program order is assumed only for operations within a worker thread or an event, but not between events executed in the same looper thread. This is because the order of events in a looper thread may differ across executions.

Fork-join rules (Fork, Join): A thread begins after its `fork` operation and ends before its `join` operation.

Signal-wait rule (Signal): For a condition variable m, $\texttt{signal}(m)$ happens before its next $\texttt{wait}(m)$ in the trace.

Event send rule (Send): The `send` operation of an event happens before its execution.

Event loop rules (LoopBegin, LoopEnd): Any event is executed after its looper thread begins and before its looper thread ends.

The second set of rules account for the asynchronous causal orders due to the event queues:

$$\frac{task(\alpha) = task(\beta) \quad \alpha < \beta}{\alpha \prec \beta}\ (\text{PO}) \qquad \frac{\alpha = \texttt{end}(E_1) \quad \beta = \texttt{begin}(E_2) \quad \texttt{send}(_, q, E_1) \prec \texttt{send}(_, q, E_2)}{\alpha \prec \beta}\ (\text{FIFO})$$

$$\frac{\alpha = \texttt{begin}(T) \quad \beta = \texttt{begin}(E) \quad looper(E) = T}{\alpha \prec \beta}\ (\textsc{LoopBegin}) \qquad \frac{\alpha = \texttt{end}(E) \quad \beta = \texttt{end}(T) \quad looper(E) = T}{\alpha \prec \beta}\ (\textsc{LoopEnd})$$

$$\frac{\alpha = \texttt{fork}(_, T) \quad \beta = \texttt{begin}(T)}{\alpha \prec \beta}\ (\textsc{Fork}) \qquad \frac{\alpha = \texttt{end}(T) \quad \beta = \texttt{join}(_, T)}{\alpha \prec \beta}\ (\textsc{Join})$$

$$\frac{\alpha = \texttt{signal}(_, m) \quad \beta = \texttt{wait}(_, m) \quad \alpha < \beta}{\alpha \prec \beta}\ (\textsc{Signal}) \qquad \frac{\alpha = \texttt{send}(_, _, E) \quad \beta = \texttt{begin}(E)}{\alpha \prec \beta}\ (\textsc{Send})$$

$$\frac{\alpha = \texttt{end}(E_1) \quad \texttt{begin}(E_1) \prec \beta \quad \exists E_2.\ E_2 \in Events \wedge E_2 = task(\beta) \wedge looper(E_2) = looper(E_1)}{\alpha \prec \beta}\ (\textsc{Atomic})$$

Figure 3.1: Causality rules for event-driven programs. $task(\alpha)$ is the worker thread or event that executes operation α. $looper(E)$ is the looper thread that executes event E. _ is a don't-care.

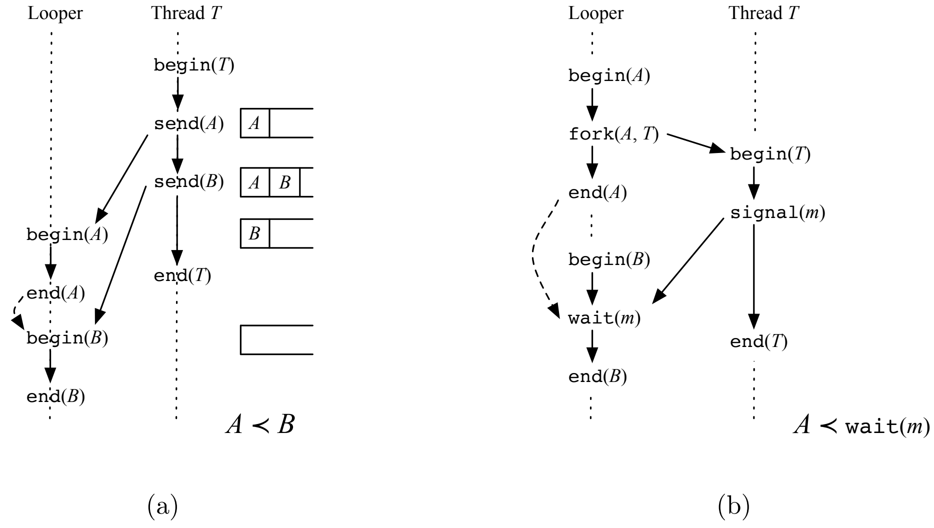


Figure 3.2: Illustration of asynchronous causality rules. The derived happens-before orders are noted in each figure, and the dotted arrows indicate these derived relation. (a) The event queue rule. The statuses of the event queue are shown at right. (b) The atomicity rule.

Event queue rule (FIFO): Two events must be causally ordered, if they are from the same queue, and their `send`s are causally ordered. An example is shown in Figure 3.2a.

Atomicity rule (Atomic): Two events in a looper thread are atomic with respect to each other, and thus are causally ordered by the synchronization operations they perform. For example, in Figure 3.2b, we only order E_1 with the part *after* `wait`(m) in E_2, since the synchronization operations provide no ordering semantics before `wait`(m).

In addition to the above rules, we also assume a closed system and thus a total order between input events:

External input rule: If events E_1 and E_2 are generated from the external world and $E_1 < E_2$, we have $E_1 \prec E_2$.

$$\frac{\begin{array}{c}\alpha = \texttt{end}(E_1)\\ \beta = \texttt{begin}(E_2)\\ \texttt{send}(_, q, E_1) \prec \texttt{send}(_, q, E_2)\\ \textit{priority}(E_1, E_2)\end{array}}{\alpha \prec \beta}\ (\textsc{Priority})$$

$$\frac{\begin{array}{c}\alpha = \texttt{end}(E_1)\\ \beta = \texttt{begin}(E_2)\\ \texttt{send}(_, q, E_2) \prec \texttt{send}(_, q, E_1, \texttt{AtFront}) \prec \beta\end{array}}{\alpha \prec \beta}\ (\textsc{AtFront})$$

Figure 3.3: Extended causality rules for Android. *priority* is the priority function defined in Table 3.1.

3.1.4 Case Study: Android's Causality Model

In this section, we study how to adapt our generic causality model to Android. As described in Section 2.1.2, Android provides the ability to overrule FIFO order of processing events by allowing programmers to assign *priority tags* and *time constraints* to events to change their queuing policy. To precisely model the happens-before relation in Android, we add the causality rules in Figure 3.3, which we explain below.

Rule PRIORITY orders two events A and B, if their `send` operations are ordered, provided that $\textit{priority}(A, B)$ is true (Table 3.1). For example, in Figure 3.4a, a worker thread sends two `Delayed` events A, B in program order. Enqueued events with nondecreasing delays are processed in the FIFO order, guaranteeing that $A \prec B$. However, as can be seen in Figure 3.4b, even when events A and B are sent in order, if the delay of the earlier event is even slightly greater than the delay of the latter event, then we cannot infer any happens-before order between the two events, because there is a chance that the later event can execute before the earlier event.

Rule ATFRONT accounts for the happens-before orders between ordinary events and `AtFront` events. As shown in Figures 3.4d and 3.4e, although $\texttt{send}(A) \prec \texttt{send}(B)$, the ordinary event A and the `AtFront` event B are not ordered, because both execution orders are possible as shown in the figures. However, there is one

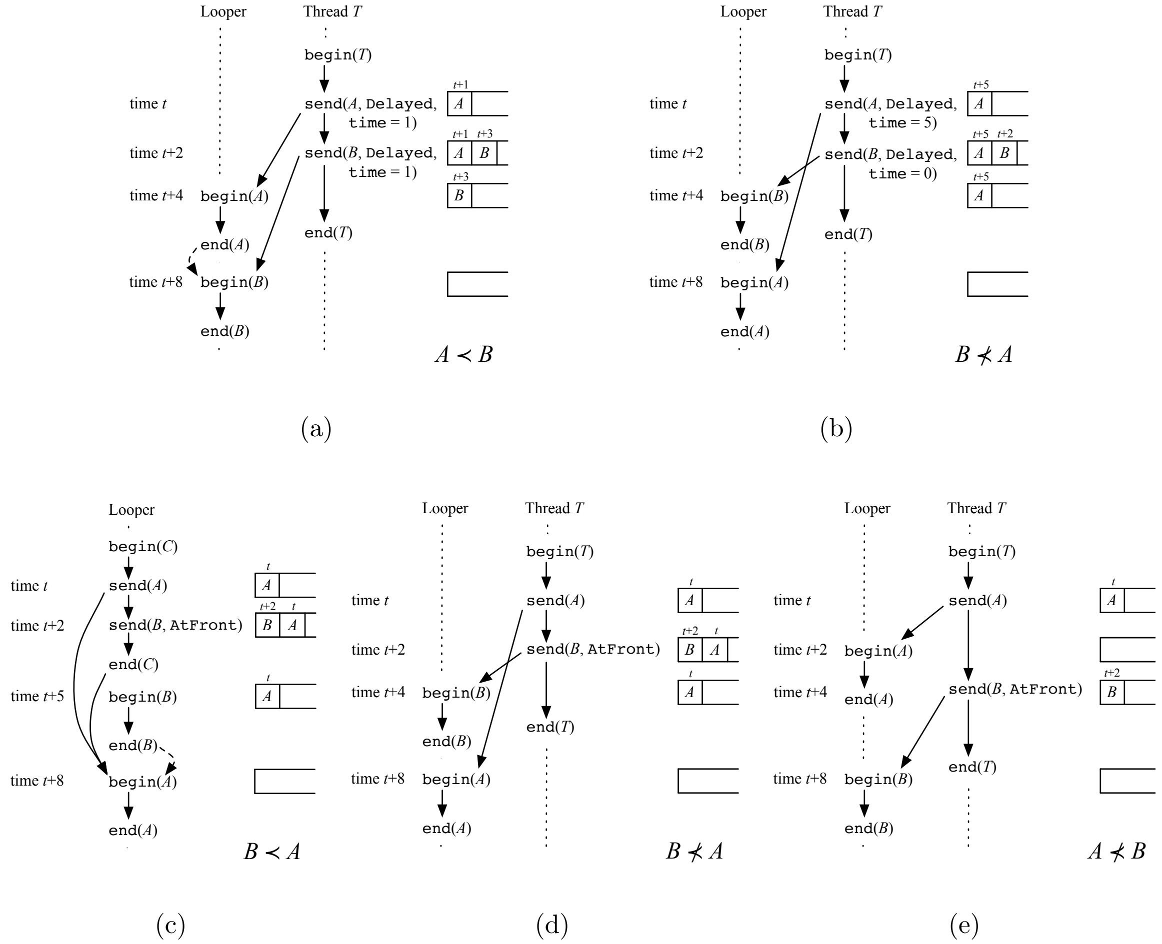


Figure 3.4: Illustration of Rules PRIORITY and ATFRONT. The happens-before orders that can or cannot be derived are noted in each figure, and the dotted arrows indicate these derived orders. The statuses of the event queue are show at right, and for each event, the earliest time when it can be processed is marked on top of each cell. (a) $A \prec B$ as their `send` operations are ordered and their delays are the same. (b) A is processed after B owing to its higher delay, and thus no happens-before order can be derived between A and B. (c) Since $\texttt{send}(B) \prec \texttt{begin}(A)$, B is guaranteed to be enqueued in the front of the queue before A can be processed, so we have $B \prec A$. (d) and (e) show two scenarios where $\texttt{send}(B) \prec \texttt{begin}(A)$ is not true, and thus no happens-before order can be derived between A and B.

E_1 \ E_2	`Delayed, Async`	`Delayed, Sync`	`AtTime, Async`
`Delayed, Async`	E_1.`time` $\leq E_2$.`time`	E_1.`time` $\leq E_2$.`time`	false
`Delayed, Sync`	false	E_1.`time` $\leq E_2$.`time`	false
`AtTime, Async`	false	false	E_1.`time` $\leq E_2$.`time`
`AtTime, Sync`	false	false	false
`AtFront, Async`	true	true	true
`AtFront, Sync`	false	true	false

E_1 \ E_2	`AtTime, Sync`	`AtFront, Async`	`AtFront, Sync`
`Delayed, Async`	false	false	false
`Delayed, Sync`	false	false	false
`AtTime, Async`	E_1.`time` $\leq E_2$.`time`	false	false
`AtTime, Sync`	E_1.`time` $\leq E_2$.`time`	false	false
`AtFront, Async`	true	false	false
`AtFront, Sync`	true	false	false

Table 3.1: The priority function for Rule PRIORITY. $E_1 \prec E_2$ if `send`$(E_1) \prec$ `send`(E_2) and the corresponding cell is true.

special condition under which we can derive a happens-before order between an ordinary event A and an `AtFront` event B. This is when it is guaranteed that `send`$(B) \prec$ `begin`(A). Figure 3.4c shows one such instance where such a guarantee can be made. In this example, `send`(A) and `send`(B) are both executed within event C. Also, event C is executed by the same looper thread as the one that processes events A and B. Since it is guaranteed that C ends before any other event can be processed, it in turn guarantees that `send`$(B) \prec$ `begin`(A). This rule is the only one showing that *a causally later* `send` *could create a causal predecessor of an event sent earlier.*

For events executed in binder threads in Android, since there is no ordering guarantee between these events, they are modeled as short-lived threads in our causality model.

3.1.5 Generalized Asynchronous Causality Rule

Rules FIFO, Atomic and Priority all follow the following form of *generalized asynchronous causality rule*:

$$\frac{\alpha \in \Lambda \quad \gamma(\alpha) \prec \eta(\beta) \quad \rho(\alpha, \beta)}{\alpha \prec \beta}$$

defined on the set of operations Op in a trace with parameters $\Lambda \subseteq Op, \gamma : \Lambda \rightarrow Op, \eta : Op \rightarrow Op, \rho : Op \times Op \rightarrow \{\text{true}, \text{false}\}$ such that $\forall \alpha \in \Lambda. \gamma(\alpha) \preceq \eta(\alpha)$ and $\forall \beta \in Op. \eta(\beta) \preceq \beta$. For example, for Rule Priority, Λ is the set of `end` operations of all events from a queue, γ, η return the `send` operation of the containing event, and ρ is the priority function; for Rule Atomic, Λ is the set of event `end`s in a looper thread, γ returns the `begin` operation of the containing event, η is the identity function, and ρ checks if both operations are in the same looper thread.

Intuitively speaking, this generalized asynchronous causality rule says that, any *asynchronous causality rule* takes the following form: the order between a pair of earlier operations ($\gamma(\alpha)$ and $\eta(\beta)$) can *asynchronously* decide the order between a pair of later operations (α and β). This generalized form describes causalities in a wide range well-behaved shared-memory event-driven systems, and thus techniques developed in this thesis are broadly applicable to these systems. However, for ill-behaved APIs (such as sendAtFront), special treatments such as Rule AtFront are required.

3.2 Race Detection

Data races are common concurrency bugs in multithreaded programs. Our initial study of bugs reported for open-source Android applications indicates that race-related bugs are prevalent in event-driven programs as well. However, conventional data races are poor indicators of concurrency bugs in event-driven mobile applications. In this section, we briefly discuss the problem of conventional data races, and present an offline algorithm to find use-after-free violations. Also, to reduce false alarms, we present two effective heuristics, if-guard and intra-event-allocation. They check for the

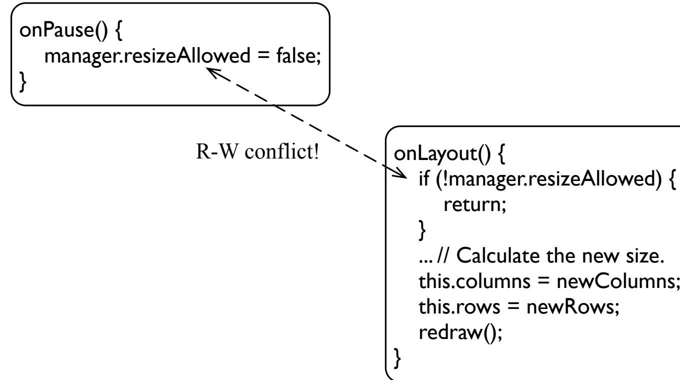


Figure 3.5: A read-write race in the ConnectBot application. The memory write in onPause cannot be executed between the if statement and the succeeding statements that update the size information in onLayout because the two events in the same looper thread are executed atomically with respect to each other. Thus, this read-write race is not a bug.

common cases of benign races and filter the race warnings.

3.2.1 Use-Free Races

Conventionally, a data race is defined as a pair of memory accesses, of which at least one is a write, and are not ordered by a happens-before relation. This definition, however, is not useful for detecting bugs in a mobile application. For example, there are 1,664 such races in a 30-second trace of ConnectBot, and most of them are benign races. One major reason is that we find many read-write and write-write races between concurrent events that are *commutative*. Two concurrent events are commutative if they produce correct results irrespective of the order in which they are executed. Figure 3.5 shows a false positive example in ConnectBot.

We tackle this problem by limiting our focus on finding *use-after-free* violations due to races between concurrent events, and devising two heuristics to check if the racing events are commutative or not. A *free* is a write operation that sets an object pointer to null. A *use* is a read operation to an object pointer that would be dereferenced later. There is a *use-free* race if a use and a free are not ordered by the happens-before

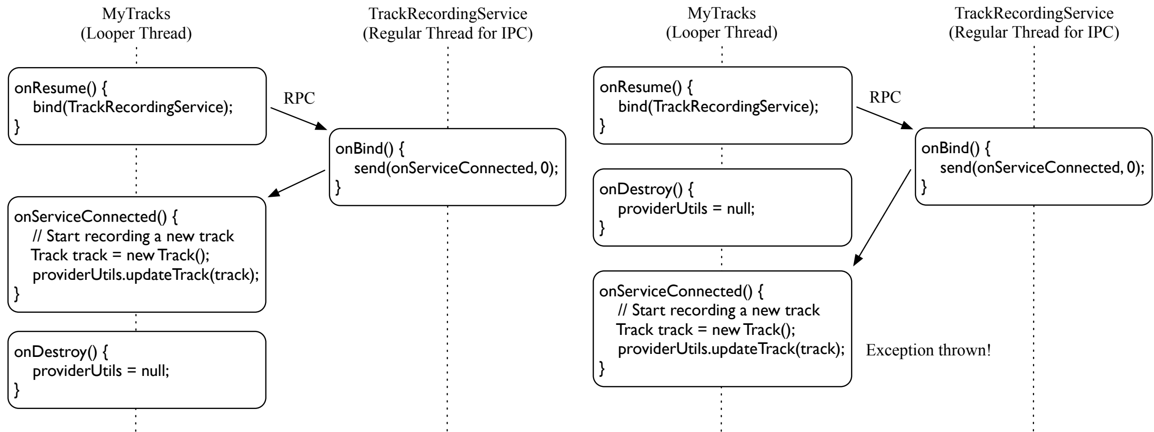


Figure 3.6: A use-after-free violation in Google's MyTracks application. (a) A correct execution. (b) An incorrect execution where an use-after-free violation manifests owing to the lack of happens-before order between onServiceConnected and onDestroy.

relation according to the causality model. Note that a use-free race is a special kind of read-write race, and may trigger a *use-after-free* violation if the free is executed before the use. By limiting our focus on this special kind of data race, we can find meaningful bugs without introducing lots of false positives.

Figure 3.6 shows a typical example of a use-free race: the use of providerUtils in onServiceConnected is racy with the free to provideUtils in onDestroy. In this case, reversing their execution order would cause a harmful program behavior and thus it is a use-after-free violation.

3.2.2 Finding Use-Free Races

We describe the following offline algorithm to find use-free races. Given an execution trace of an event-driven program, the algorithm first builds a *happens-before graph* for the collected trace based on the causality model described in Section 3.1. The happens-before graph is a directed acyclic graph consisting of all operations in the trace as its vertices. Any two operations are causally ordered if and only if there is a path between them in the graph. For any use operation, the algorithm then checks if it is ordered with its previous and next free operations to the same pointer by per-

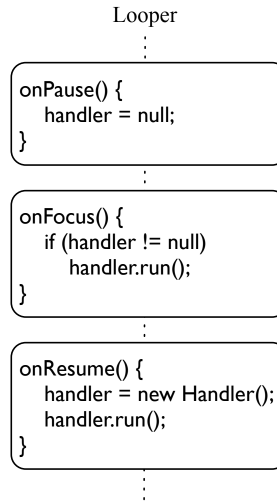


Figure 3.7: An example of commutative events that contain uses and frees.

forming reachability tests on the happens-before graph, and reports any concurrent use-free pairs. This algorithm guarantees to report at least the first use-free race on each pointer.

3.2.3 Pruning False Positives

Even if two events contain use-free races, it is possible for the racing events to be commutative for two reasons. The first reason is that if the use is not executed when the event containing the free is processed first, then no use-after-free violation would occur. For example, as can be seen in Figure 3.7, onPause and onFocus are commutative, because it is guaranteed that when onFocus is processed before onPause, the use in onPause will not be executed owing to the if-condition guarding the use.

The second reason is that, if the pointer accessed by the use is assigned to a valid object address before the use is executed, then no use-after-free violation would be triggered. We call such an assignment an *allocation* to the pointer. For example, in Figure 3.7, onPause and onResume are commutative, because within onResume, there is always an allocation before the use.

To reduce the number of false alarms due to the above reasons, we present two simple heuristics: if-guard and intra-event-allocation. These heuristics recognize two

common programming patterns that make concurrent events containing use-free races commutative. *Both heuristics are only applicable to events that are sent to the same event queue and processed by the same looper thread.*

If-Guard Check Programmers often check if a pointer is null before using it. Branch instructions used to perform this check can be leveraged to check if a use is safe or not. Therefore, in addition to log the operations described in Section 3.1.2, we also log the following branch instructions that test on object pointers: if-eqz (jump if a pointer is null), if-nez (jump if a pointer is not null), and if-eq (jump if two pointers are equal).

The if-guard check for if-eqz and if-nez is illustrated in Figure 3.8. The heuristic works as follows. Suppose there is an if-eqz instruction at address pc, and it performs a forward jump to address pc + offset if pointer is null. Then we assume that *the code between* pc *and* pc + offset *is executed only if the branch is not taken at runtime*, which guarantees that pointer would be non-null, and any use of pointer in this code region would thus be safe. Therefore, any use-free races involving such uses are ignored. Similar arguments can be applied to backward jumps, and to if-eqz instructions.

Note that the if-guard check is neither sound nor complete. Harmless use-free races can elude the check if the nullity check is not done through if-eqz and if-nez instructions, but this is rare in compiler-generated code. Harmful use-fee races might be mistakenly classified as harmless because we have assumed that the code between pc and pc + offset is executed only if the branch is not taken, but this assumption is not always true. However, for compiler-generated code in most applications, this assumption is usually true and hence we design the if-guard check based on this assumption.

In addition, we have found that the if-eq instruction that tests on two object pointers is often used to check if an object pointer is equal to this object in Java. Therefore, the if-eq instruction provides the same safety guarantee as if-nez does, and it is included in the if-guard check as well.

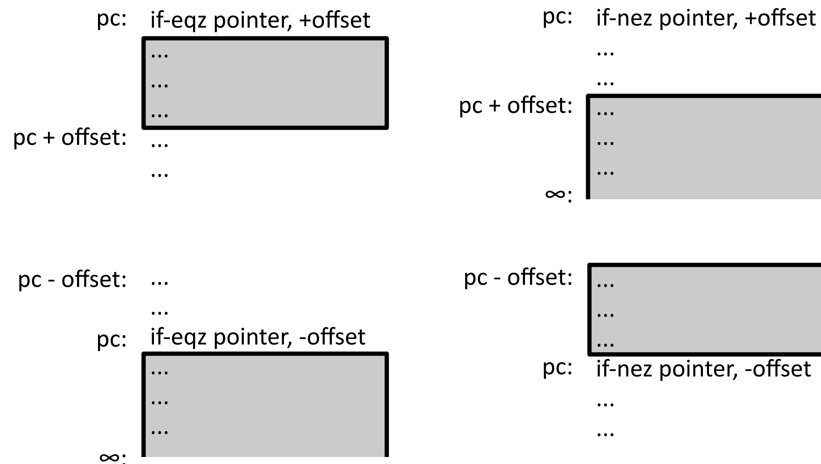


Figure 3.8: An illustration of the if-guard check. In each case, the branch instruction is located at pc and conditionally jumps forward or backward to pc + offset or pc - offset respectively. ∞ indicates the end of the current function. If a use of pointer in the shadowed area is executed after the branch instruction at runtime, pointer would be guaranteed non-null, so the use would be safe.

Intra-event-allocation If there is an allocation after a free in an event, then the null value written by the free will never become visible to any other event executed in the same looper thread. Therefore, use-free races involving such a free are ignored. Similarly, if there is an allocation before a use within the same event, then it is guaranteed that the use cannot read any null value written by a free outside the event. Any use-free races involving such a use are also filtered.

3.3 Implementation

We built the first use-free races detector for mobile applications called CAFA. We chose to implement CAFA for Android [1], as the system and many of its applications are open-source. CAFA consists of a customized Android ROM and an offline analysis tool. The customized ROM instruments several key components in Android (e.g., the Dalvik Virtual Machine (DVM), and the core framework libraries) to collect execution traces for target applications and system services. The collected traces are later used

by the offline analysis tool to reconstruct the happens-before graphs and detect use-free races. The customized ROM can be directly installed on some of the Android devices such as Google Nexus 4. CAFA is completely transparent, and thereby can trace and analyze unmodified Android applications.

In this section, we mainly discuss our instrumentation framework. We omit the details of the offline analysis tool as its implementation is straightforward. This section is organized as follows. We first provide an overview for our instrumentation framework. Then, we provide details about how we instrument various components in Android that allows us to capture the causalities described in Section 3.1. Finally, we discuss how we find and instrument potentially racy operations to capture use-free races.

3.3.1 Overview

Figure 3.9 shows the architecture of CAFA. We introduce a new logger device in Android kernel. All execution traces are sent to this logger device. The CAFA offline analyzer, which may reside in a remote server, can directly read traces from this logger device through the Android Debug Bridge (ADB). One can also choose to dump traces into a flash storage and process them later. We add a new native library called CAFA which provides interfaces for writing traces to the logger device. Each component in Android that needs instrumentation is linked with this native library (e.g., the Dalvik Virtual Machine and the core native utility library). We also add a Java binding for the CAFA native library to provide interfaces for Java programs (e.g., Java core library and Android core framework).

Inter-Process Communications (IPC) are heavily used in Android. For example, whenever an application wants to access system services like the GPS and camera services, it should initiate Remote Procedure Calls (RPC) with a remote process called system_server. If we only collect traces for the target application just like most conventional race detectors do, we may miss many causalities caused by IPCs. For instance, an application may initiate an RPC call with the GPS service asking for the

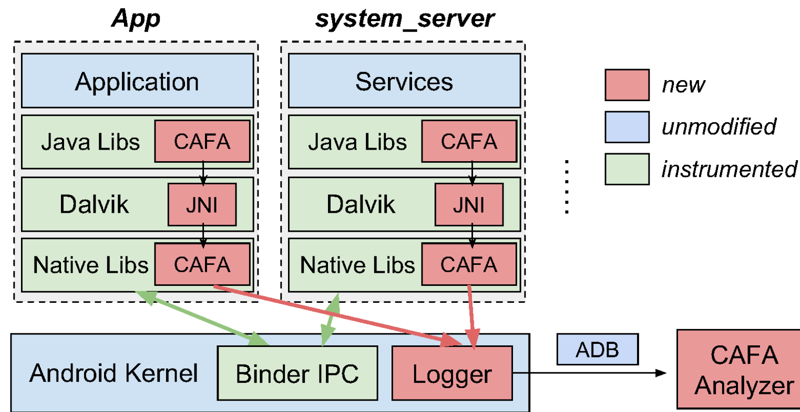


Figure 3.9: CAFA architecture overview. RED represents newly added components, BLUE represents unmodified components, and GREEN represents instrumented components.

current location. Later, it receives a message from the GPS service containing the coordinates of the current location. If we do not collect traces for the GPS service, we may miss the causality between the RPC call and the receipt of the message in this case. Therefore, we collect traces not only for target applications, but for system services as well. We also instrument the IPC framework in Android (called Binder) which enables us to establish causalities across process boundaries.

3.3.2 Instrumentation for Capturing Causalities

As we described above, we need to instrument various components in Android to collect execution traces so that the offline analysis tool can later reconstruct the happens-before graph based on these traces. The following lists the major components that we have instrumented.

- **Java Core Library**. To track thread forks and joins, we instrument the Java core library in Android (i.e., we modify the java.lang.Thread class) to emit a trace entry every time a thread is forked or joined.

- **Dalvik Virtual Machine**. We also need to instrument all the thread begins and ends, as well as all synchronization primitives used by Java programs

(e.g., java.lang.Object.wait and synchronized{...}). We achieve that by modifying the Dalvik Virtual Machine in Android. We assign a unique object ID for each object created by the virtual machine. This unique ID will later be used by the offline analyzer to capture those causalities caused by threading and synchronizations.

- **Android Core Library**. We also instrument the Android core framework library at both the native and the Java layer to capture traces like event begins and ends as well as sending of events (e.g., in android.os.Handler and android.os.Looper). These traces are crucial for us to analyze the causalities caused by event queues as we discussed in Section 3.1. In addition, the causalities caused by event listeners are done by instrumenting the listener registration functions and the internal functions that invoke the listeners. Currently we instrument for all event listeners in the android.app, android.view, android.widget, and android.content packages. Although we have accounted for most causalities due to event listeners, the packages listed above do not contain all event listeners and thus some causalities would be missed by CAFA.

- **Binder IPC Framework**. For the offline analyzer to capture the causalities caused by IPCs, we instrument the Binder IPC framework in Android. In fact, all the Remote Procedure Calls in Android are handled by the Binder IPC framework. A unique transaction ID is generated each time a process initiates a RPC call. The Binder transaction data is piggybacked with this transaction ID and sent to the remote process that handles this RPC call. All transaction related operations are also recorded and tagged with the corresponding transaction IDs. Later, the offline analyzer can capture the causalities caused by those IPCs by correlating transaction operations with the same transaction ID.

- **Other IPC Channels**. In Android, some latency critical IPCs, such as the display events and the input events, are performed through pipes (or Unix domain sockets), instead of the Binder IPC framework. CAFA handles these IPCs similarly by tagging those messages sent through pipes (or Unix domain

sockets) with uniquely generated IDs.

3.3.3 Logging Potentially Racy Operations

Logging the low-level read and write operations as well as certain branch instructions is mainly done by instrumenting the DVM bytecode interpreter to log all related Dalvik bytecode instructions. In addition, method invocation/return and certain branch instructions are also instrumented for logging. All the instrumentation is done in the portable interpreter mode, and we are also porting CAFA to the fast interpreter mode for better efficiency. Currently we don't support CAFA in JIT interpreter mode due to the complexity of tracing accesses in native code. The detailed instrumentation in the DVM bytecode interpreter for use-free race detection is described as follows.

- **Frees**. The Dalvik instruction set provides a set of instructions to write values to object pointers (e.g., i-put-object, s-put-object, and a-put-object). We instrument the DVM bytecode interpreter to emit a trace entry when such an instruction is executed. The log includes the ID of the object being dereferenced (if any), the address of the object pointer, and the written value. If the written value is null, then the instruction is a free; otherwise it is an allocation.

- **Uses**. Unlike frees, uses are harder to detect. A use involves a read from an object pointer (e.g. i-get-object, s-get-object, and a-get-object) to get an object, followed by an instruction that dereferences the object. The dereference instruction can be either an access to a field of the object, or a method invocation on the object. We instrument the DVM bytecode interpreter to emit a trace entry for the former read instruction to log the address of the pointer and the ID of the object it gets, and another entry that logs the ID of the dereferenced object for the latter field access or method invocation. The difficulty is that without a dynamic data flow analysis (which is expensive to perform at runtime so we choose not to do it), when we see a dereference instruction, we only know the ID of the dereferenced object, but have no idea which pointer it dereferences.

Therefore, we assume that a dereference instruction reads the object ID from its closest previous pointer read that gets the same object ID, which is usually true for compiler-generated code.

- **Calling Context Stack**. The calling context stack is traced for 3 purposes: (1) To provide context information for reasoning about races. (2) To compute the relative address of each instruction so they can be mapped to the static Java code. (3) To log method invocations that dereference objects. For each method invocation, its method and return addresses are logged. We only log the name of a function upon its first invocation to reduce the size of a trace. For a method return, the method and return addresses are logged again. In addition, method exits through exception throwing are also logged.

- **If-Guard Check**. We instrument the DVM bytecode interpreter to log the if-eqz, if-nez, or if-eq instructions that test on object pointers. For an if-eqz instruction, a trace entry is emitted only when the branch is not taken; for an if-nez or if-eq instruction, a trace entry is emitted only when the branch is taken. The entry contains the current and target addresses of the branch instruction, as well as the ID of the testing object. Since we only have the object ID but no pointer address, we use a heuristic similar to the one that recognizes uses: a branch instruction is matched with its nearest previous pointer read to decide which pointer it tests on.

3.4 Evaluation

In this section, we describe our experience on using CAFA to find use-after-free violations, and then provide the accuracy and performance evaluation of CAFA.

3.4.1 Experimental Setup

We built CAFA on the Android Open Source Project 4.3 r1.1. and applied CAFA on 10 open-source Android applications picked from the built-in applications of the

Android Open Source Project, the list of free and open-source Android applications in Wikipedia [9] and the F-Droid repository [7]. All our experiments were conducted on the latest 16GB model of Google Nexus 4, which is equipped with a Qualcomm Snapdragon S4 Pro quad-core ARMv7 processor. Each trace was collected through an execution of 10–30 seconds on the instrumented system.

3.4.2 A Survey of Use-After-Free Violations

We have found several use-after-free violations among the tested applications. Most of these violations might be triggered when the application switches to the paused state. Typically, a clean-up procedure (e.g., freeing pointers) is called at this moment. As a result, any event that is scheduled after the pause event, which might be sent from another thread or process, would crash the application if it tries to use the freed pointers. For example, BarcodeScanner contains a bug of this kind.

Usually use-after-free violations would cause exceptions when they are triggered. Some of these exceptions would be caught to prevent the applications from crashing. However, sometimes these exceptions are not handled properly such that the behaviors of the applications do not meet a user's expectation. We consider such races buggy and believe that programmers should take care of such races more carefully. For example, CAFA reported that MyTracks contains use-after-free violations in the following method in MyTracks.java:

```
public void onServiceConnected(...) {
  ...
  try {
    // TODO: Send a start service intent and broadcast
    // service started message to avoid the hack below
    // and a race condition.
    ...
    startRecordingNewTrack(...);
  } finally {
    ...
  }
}
```

The startRecordingNewTrack method contains the racy code illustrated in Figure 3.5.

As described in the TODO comment, instead of fixing the program state in the finally block to avoid a crash, a more appropriate way is to enforce a happens-before order between this event and the racing event.

Another example of improperly handled exceptions can be found in ToDoList, where the author simply resolved the exception with the following code:

```
try {
  ...
  db.updateNote(...);
} catch (NullPointerException npe) { /* do nothing */ }
```

Although the above code prevents the application from crashing, the latest user input would not be written to the database and the data would be lost.

3.4.3 Accuracy and Performance

Table 3.2 shows the use-free races detected by CAFA in the tested applications. CAFA reported 115 use-free races, among which, 69 of them could lead to use-after-free violations and thus are *harmful*. These violations are further classified into 3 categories: (a) *in-thread* violations due to races that happen between events in a looper thread; (b) *inter-thread* violations due to races that happen between threads but cannot be detected by a conventional data race detector; (c) *conventional* violations due to races that happen between threads and can be detected by a conventional detector. Here the "conventional" detector assumes a total order for all events in the same looper thread, but no causal order between `unlock` operations and their succeeding `lock` operations. Because we relax the event order within the looper thread, we can capture more inter-thread races than a conventional detector. In summary, 60% of the reported races are harmful. Note that the in-thread violations are bugs, and the inter-thread and conventional violations are considered bugs in a DRF0 memory model [13], but may not necessarily be bugs in a stronger memory model such as SC [40].

We also analyzed the causes of false positives, including false races and benign races, and classified them into three major categories.

Type I false positives are false races due to missing happens-before orders for event listeners. As described in Section 3.3, currently we only instrumented the event listeners in specific packages of the Android library. Having a more thorough instrumentation, we could infer the happens-before orders more accurately and remove this kind of false positives.

Type II false positives are benign races that happen primarily because the if-guard and intra-event-allocation heuristics we used are not able to precisely check if two events are commutative. For example, the if-guard check infers that a use is safe only if there is a pointer test, but if the programmer uses a boolean flag to indicate that the pointer is safe for dereference, if-guard would not infer this information.

Type III false positives are false races that happen because CAFA mistakenly matched dereference instructions to incorrect pointer reads. When such mismatches happen, the pointer reads would be incorrectly recognized as uses, and false races would be reported if there are racing frees. Currently CAFA only uses the traces to recognize use operations. It can be improved by performing a static data flow analysis on the Dalvik bytecode of the applications to accurately match the dereference instructions to the corresponding pointer reads.

Although the if-guard check may miss harmful use-free races, we did not find any such instance among the applications we studied. Since this study focuses on use-free races, we did not investigate how many harmful races there are in the large volume of non-use-free races.

Figure 3.10 shows the slowdown of CAFA when collecting traces for each application. The slowdown is between 2x to 6x compared to their uninstrumented executions. The running time of the offline analysis is slow, since we did not focus on optimizing the offline analysis. In Chapter 4, we will present a new algorithm to speed up the offline analysis drastically.

Application	Events	Races reported	True races			False positives		
			(a)	(b)	(c)	I	II	III
ConnectBot	3,058	3	0	2	0	1	0	0
MyTracks	6,628	8	1	3	0	0	4	0
BarcodeScanner	4,554	5	0	2	0	1	1	1
ToDoList	7,122	9	8	0	0	0	1	0
Browser	3,965	35	0	8	19	1	7	0
Firefox	5,467	25	0	6	10	4	5	0
VLCPlayer	2,805	7	0	0	1	0	5	1
FBReader	3,528	9	1	3	1	2	2	0
Camera	7,287	9	1	1	0	0	5	2
Music	6,684	5	2	0	0	0	2	1
Overall		115	13	25	31	9	32	5

Table 3.2: Races reported by CAFA. (a) Races that lead to in-thread violations. (b) Races that lead to inter-thread violations. (c) Races that lead to conventional violations.

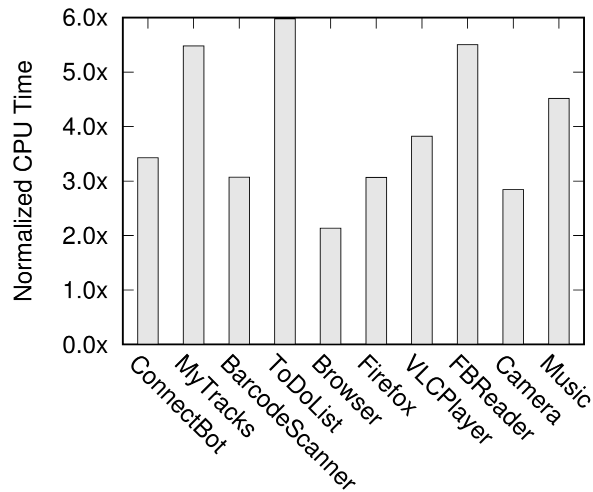


Figure 3.10: The slowdown for CAFA to collect traces on various applications.

3.5 Summary

Mobile applications are increasingly popular and are written by common programmers. These applications are written in an asynchronous event-driven model, which is prone to concurrency errors. Unfortunately, currently we do not have adequate tools to help programmers find races in these event-driven mobile systems. This chapter presented the first causality model for event-driven programs, and a tool that detects use-after-free races for Android applications. Our study showed that a significant number of harmful races could be found with adequate accuracy.

CHAPTER 4

AsyncClock: Scalable Inference of Asynchronous Event Causality

In this chapter, we introduce a primitive called AsyncClock to realize an efficient non-iterative algorithm to infer the happens-before relation between events on the fly. An AsyncClock keeps track of causally preceding events for a newly created event, so we can compute its logical time efficiently by inheriting the end times of all events in the AsyncClock. We show that AsyncClock can be generalized to handle a wide variety of causality rules for event-driven programs.

We also provide several solutions to improve scalability of AsyncClock. We propose methods to efficiently identify *heirless* events that can no longer have immediate successors, and free their metadata to achieve good scalability. Unfortunately, not all heirless events can be identified efficiently. To address this issue, we exploit the intuition that an old event and a recent event are unlikely to be re-ordered in an alternative execution. Any concurrency error between two such events is unlikely to manifest, and thus low priority for developers to fix. Based on this assumption, we choose to free old events. We conservatively set the time threshold for classifying old events so high that this optimization does not result in any false negatives in our empirical evaluation. Finally, to further reduce the space overhead, we propose a sparse representation for an AsyncClock to take advantage of our observation that an event often have few predecessors.

We built the first single-pass, non-graph-based data race detector for event-driven programs. Compared to a recent solution, EventRacer [15], we show that AsyncClock is 8x faster, used 87% less memory, and scales well in terms of both performance and space. The average performance overhead of collecting and analyzing the traces of 20 common Android applications is about 6x. Thus, we also meet our goal in realizing a tool for asynchronous programs that is as efficient as conventional data race detectors [28].

4.1 AsyncClock Design

In this section, we describe the AsyncClock algorithm that efficiently establishes happens-before relations between asynchronous events for the *generic causality model* for event-driven programs described in Section 3.1.3. For simplicity, we omit Rule Atomic in this section. In Section 4.3, we will generalize our solutions to handle Rule Atomic and the *generalized asynchronous causality rule* described in Section 3.1.5.

4.1.1 Overview

To establish happens-before relations between any operations, we could maintain the logical time at each operation via vector clocks. The vector clock (VC) at an operation inherits the vector clocks at its *immediate causal predecessors*. According to Rule FIFO, the immediate causal predecessors of an event E must be posted to the same queue by the immediately causally preceding `send` operations of `send`(E). Therefore, at a `send` operation, we seek to determine the set of events posted by its immediately causally preceding `send`s to the same queue. AsyncClock helps track this information in each chain. The vector clock of an event is then computed at its `begin` operation, where the times of its immediate causal predecessors are guaranteed to have been resolved, by inheriting the end times of the events tracked in the AsyncClock at its `send`.

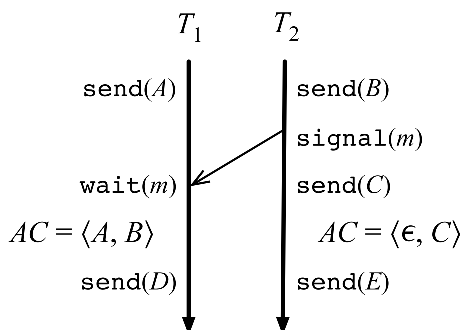


Figure 4.1: Illustration of AsyncClock. The ACs depict the AsyncClock of T_1 and T_2 after processing their preceding operations. ϵ indicates that there is no predecessor in the corresponding chain.

4.1.2 AsyncClock Primitive

For any `send` operation, *there is at most one immediately causally preceding* `send` *to the same queue in each chain.* Therefore, an AsyncClock (AC) for a queue can be represented as a vector of events posted by the latest causally preceding `send` to the queue in each chain.

Figure 4.1 illustrates AsyncClock using an example with a single queue. AC at `send`(D) consists of A and B as they are posted by the immediately causally preceding `send`s in the two threads. AC at `send`(E) contains only C, as `send`(B) does not immediately precedes `send`(E) in T_2, and there is no causally preceding `send` in T_1.

Formal Definition For an event queue q, the AsyncClock AC_q : *Chains* $\rightarrow$ *Events* at operation x is formally defined as a vector of events, one for each chain, such that

$$AC_q(i) = \begin{cases} E, & \text{if } \texttt{send}(_, q, E) \text{ is the latest } \texttt{send} \text{ operation in chain } i \\ & \text{s.t. } \texttt{send}(_, q, E) \prec x; \\ \epsilon, & \text{otherwise.} \end{cases}$$

Note that for an event E, AC at `send`(E) may track more than the immediate causal predecessors of E, since not all the *latest* causally preceding `send`s in every

chain are *immediately* causally preceding `send`s of $\texttt{send}(E)$. For example, in Figure 4.1, if $\texttt{send}(A)$ happens after $\texttt{wait}(m)$ in T_1, then $\texttt{send}(B)$ would no longer be an immediately causally preceding `send` of $\texttt{send}(D)$, but B is still tracked in AC at $\texttt{send}(D)$. In Section 4.1.3, we will describe a simple optimization to avoid tracking such events.

Resolving Event Time The logical time of an event E from queue q is resolved at $\texttt{begin}(E)$, where all immediate causal predecessors of E are guaranteed to have finished. *VC* at $\texttt{begin}(E)$ must inherit: 1. *VC* at $\texttt{send}(E)$, and 2. *VC* at $\texttt{end}(E')$ for each E' in AC_q at $\texttt{send}(E)$.

4.1.3 Maintaining AsyncClocks

For each chain, we maintain a AsyncClock for each queue to keep track of the events posted by the immediately causally preceding `send`s in all chains. The following defines the join operation ($\sqcup$) to inherit another AsyncClock, and the identity function ($\mathcal{I}_{AC}$) which constructs an AsyncClock containing only one event. They are used to maintain AsyncClocks at synchronization operations, `send`s and `begin`s. Here, *sender*(E) denotes the chain that $\texttt{send}(E)$ is in.

$$\begin{aligned} AC \sqcup AC' &= \lambda i.\ \text{if } \texttt{send}(AC(i)) \prec \texttt{send}(AC'(i)) \\ &\qquad \text{then } AC'(i) \text{ else } AC(i) \\ \mathcal{I}_{AC}(E) &= \lambda i.\ \text{if } i = \mathit{sender}(E) \text{ then } E \text{ else } \epsilon \end{aligned}$$

Synchronization Operations For each event queue q, AC_q of chain i at a `wait` operation is joined with AC_q at the corresponding `signal` operation. Note that since the join operation performs happens-before queries only between respective `send` operations in the same chain, each query can be implemented as a simple integer comparison between their logical times in that chain. So, the join takes only $O(n)$ time, where n is the number of chains.

Event Creation The AsyncClock data structure described above sometimes tracks more than immediate causal predecessors. For example, in Figure 4.1, any event sent from T_1 after `send`(D) would have only one immediate causal predecessor, D. So it is redundant and inefficient to keep B in AC of T_1 after `send`(D). Instead of maintaining the full AsyncClock, our algorithm maintains a "reduced" AC, which becomes $\langle D, \epsilon \rangle$, for T_1 after `send`(D). More generally, after an event E from queue q is sent from chain i, our algorithm reduces its AC_q to $\mathcal{I}_{AC}(E)$.

Event Begin At the beginning of an event E from queue q, its AsyncClock needs to be computed to track the immediate causal predecessors of any event sent from E. For each queue q', $AC_{q'}$ at `begin`(E) must inherit: 1. $AC_{q'}$ at `send`(E), and 2. $AC_{q'}$ at `end`(E') for each E' in AC_q at `send`(E). Especially, our algorithm removes all causal predecessors of E from AC_q, as their logical times have been already inherited by VC of E. For a system having n chains and l queues, the computation takes $O(n^2 l)$ time.

4.1.4 Race Detection

Our race detector maintains a vector clock and a set of AsyncClocks, one for each queue, for each chain, past event, and synchronization variable, to resolve the logical time of an event. Once the time of an event is resolved, our algorithm uses *greedy chain decomposition* [57] to choose the chain that the event should be part of according to its time. Read and write operations inherit the logical time of their attributed chains. We use the *epochs* (Section 2.2) to optimize metadata stored for data variables and find races between their accesses.

4.2 Improving Scalability

The AsyncClock algorithm described in Section 4.1 maintains VCs and ACs for all past events, as these metadata might be used to compute the logical times of immediate causal successors of past events in future. However, this algorithm has the

same problem as previous work [15]: it does not scale since the memory profile grows with the program execution length.

To address this problem, we present several solutions to identify *heirless events* — events that have no immediate causal successors in future — and reclaim their metadata. We also describe optimizations to make the metadata slim, thus improving memory efficiency of our algorithm.

4.2.1 Reclaiming Heirless Events

An heirless event cannot have any immediate causal successor in future. That means, its `send` operation cannot be an immediately causally preceding `send` of: 1. any ongoing thread or event; 2. any future event. Since this condition is hard to check, we present two efficient solutions, *reference counting* and *multi-path reduction*, to find most heirless events based on two sufficient but not necessary conditions.

Unfortunately, these solutions do not fully address the problem of memory scalability because: 1. they cannot find all heirless events; 2. there could be unbounded number of events that are not heirless. Figure 4.2 shows a such example, where $A_2, B_2, C_2, ...$ are not heirless, as a possible future execution (shown in gray) creates immediate causal successors $(A_3, B_3, C_3, ...)$ for each of them.

We propose *time window approximation* to resolve this issue. It exploits the following intuition: *events that are far apart in time are unlikely to be re-ordered in alternative executions.* Concurrency bugs between such events are unlikely to manifest, and hence low priority for developers to fix. We assume happens-before relations between such events to make *old events* heirless and reclaim their metadata, thus ensuring constant memory usage.

Reference Counting If an event is not heirless, it would be in the AC of some chain or future event, whose AC is also computed from the ACs of past events. Hence *events that are not in any AC must be heirless.* So, an AC can be implemented as a vector of *reference counting pointers* to the metadata of events. For a newly created event, the reference count of its metadata is 1, as it is in the AC of the sending

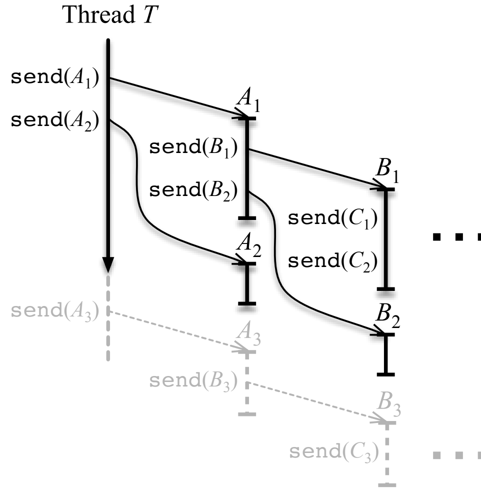


Figure 4.2: Example of infinitely many non-heirless events. The executed trace is shown in black, and a possible future execution is shown in gray.

chain. This count may increment when a containing AC is inherited by another AC, and may decrement when a containing AC inherits another AC, or is reclaimed, or an immediate causal successor is created. When the reference count becomes 0, the event is heirless and its metadata is reclaimed.

Multi-path Reduction Not all heirless events can be reclaimed by reference counting, as can be seen in Figure 4.3a. In this example, A_1 is in AC_q of B_1 and thus has a positive reference count. However, since B_1 does not send any event, its immediate causal successor can only be sent from either T, A_2, or some causal successor of A_2 in future. This means that `send`(A_2) causally precedes any future event sent to queue q. Therefore, A_1 becomes heirless once B_1 finishes.

Such heirless events could be identified as follows. If an event E is not heirless, it would be in the AC of a chain (this case is already handled by reference counting), or in the AC of a future event F. In the latter case, E is either in the AC at `send`(F), or in the AC at the end of some immediate causal predecessor of F (say, F'). Especially, when `send`(E) $\prec$ `send`(F') (Figure 4.3b), E is in the AC of F only if there is no causal successor of E created along *both* paths from `send`(F') to

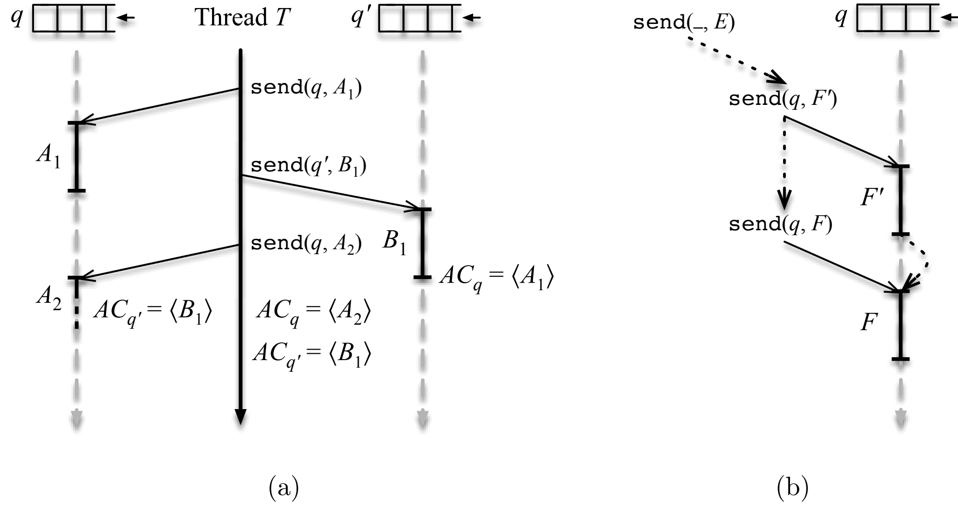


Figure 4.3: (a) A heirless event (A_1) with positive reference count. Only nonempty ACs are shown. (b) Illustration of multi-path reduction.

begin(F), and thus the reference count of E's metadata is at least 2, one for each path. Therefore, for any two events E and F' (not necessarily from the same queue) such that send$(E) \prec$ send(F') and E is in the AC at end(F'), if E's reference count is 1, then E is heirless, and thus we can remove E from the AC to relinquish E's metadata.

Time Window Approximation Since an old event and a recent event are unlikely to be executed in reversed order in alternative executions, we assume a happens-before relation between these two events, making the old event heirless. Figure 4.4 illustrates this idea. For each looper thread, we maintain a sliding time window of recently finished events, up to a predefined time threshold t, and a *time window clock* (TC). Happens-before relations between events in the time window are precisely maintained. When an event is moved out from the time window (E_2) it becomes *old*, so TC is updated to TC' by inheriting the end time and AC of the old event. The newly started event (E_{n+1}) then inherits TC'. Since TC' is a causal successor of every old event and a causal predecessor of every new event, an old event no longer has any immediate causal successor in future and thus becomes heirless.

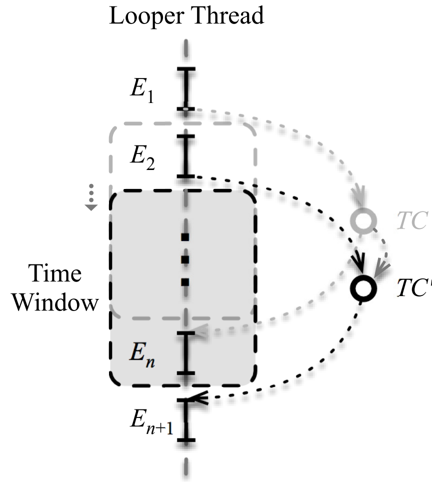


Figure 4.4: Illustration of time window approximation.

To actively reclaim the metadata of old events, the reference counting pointers to the metadata are implemented with an *invalidate* operation: when an event becomes old, we invalidate an arbitrary pointer to its metadata, so that the metadata is immediately relinquished, and all other pointers to the same metadata become null pointers.

Time window approximation ensures memory scalability in twofold. First, it limits the number of events tracked by the algorithm: if a looper thread processes events with rate r, at most $(rt + 1)$ events are tracked for this looper thread. Second, the number of chains required to place all events in the looper thread (and hence the size of VCs and ACs) is also bounded by $(rt + 1)$.

4.2.2 Reducing AsyncClock Size

Another important factor that affects both time and space efficiency of the AsyncClock algorithm is the size of AsyncClocks. In the following, we present optimizations to reduce the size of the metadata.

Sparse Vectors The number of chains, which affects the dimension of VC and AC, is unbounded, as the example in Figure 4.2 also shows that. However, as described

in Section 4.1.3, the size of an AC of a chain is reset to 1, once the chain sent a new event to the corresponding queue. Therefore, *most ACs are rather sparse*. So, we implemented a sparse vector representation [19] with hash tables for ACs to significantly reduce the space, and also the time of join operations.

Garbage Collection Time window approximation and removal of events (discussed in Section 4.3) leave residual null pointers in the AsyncClocks and thus reduce their sparsity. Therefore, we perform a *garbage collection* process periodically to scan through every AsyncClock maintained in the system and remove all null pointers.

FIFO Chain Decomposition Different chain decompositions may affect both the total number of chains and the sparsity of AsyncClocks, and thus time and memory performance. We propose a new chain decomposition method, *FIFO chain decomposition*, that improves AsyncClock's performance. It is based on the following two observations: 1. our empirical study shows that many events in mobile applications are *ordinary events* (events without any priority tag) that are either children or grandchildren of worker threads or input events; 2. All ordinary events sent from a chain to an event queue are sequentially ordered.

The chain decomposition works as follows. For an event queue, all ordinary events sent from a worker thread are placed in a *level-1 FIFO chain*. All input events also form a level-1 FIFO chain. Ordinary events sent from a level-1 FIFO chain are placed in a *level-2 FIFO chain*, and so on. Among all events, about 54% are in level-1 FIFO chains, 4.8% are in level-2 FIFO chains, and 1.7% are in level-3 FIFO chains. Since there are very few ordinary events sent from level-3 chains, we fall back to greedy chain decomposition for other events.

4.3 Generalizing AsyncClock

In this section, we present a generalization of AsyncClock that can handle *any* causality rule that follows the generalized asynchronous causality rule (Section 3.1.5). In the end, we describe a realization of AsyncClock to support events with priority tags that overrule FIFO ordering and other features such as event removal and events executed in binder threads in Android's causality model (Section 3.1.4).

4.3.1 Generalized AsyncClock

Section 3.1.5 describes the *generalized asynchronous causality rule*, which we list below:

$$\frac{\alpha \in \Lambda \quad \gamma(\alpha) \prec \eta(\beta) \quad \rho(\alpha, \beta)}{\alpha \prec \beta}$$

defined on the set of operations Op in a trace with parameters $\Lambda \subseteq Op, \gamma : \Lambda \to Op, \eta : Op \to Op, \rho : Op \times Op \to \{\text{true}, \text{false}\}$ such that $\forall \alpha \in \Lambda.\, \gamma(\alpha) \preceq \eta(\alpha)$ and $\forall \beta \in Op.\, \eta(\beta) \preceq \beta$.

For any causality rule following the above generalized form, we can generalize AsyncClock to track the immediately causally preceding γ-operations of $\eta(\beta)$ to find the immediate causal predecessors of β as follows: at operation x, the generalized AsyncClock $AC_\Lambda : \mathit{Chains} \to \Lambda$ is a vector over Λ, one for each chain, such that

$$AC_\Lambda(i) = \begin{cases} \alpha, & \text{if } \gamma(\alpha) \text{ is in chain } i \text{ s.t. } \gamma(\alpha) \prec x \text{ and } \gamma' \prec \gamma(\alpha) \text{ for all} \\ & \gamma' \in \gamma(\Lambda) \text{ in chain } i; \\ \epsilon, & \text{otherwise.} \end{cases}$$

If ρ is a function satisfying that: 1. ρ is commutative and transitive, and 2. $\forall \alpha, \alpha' \in \Lambda.\, \gamma(\alpha) \prec \gamma(\alpha') \implies \rho(\alpha, \alpha')$, then one can show that for any operations α, α' and β, if $\gamma(\alpha) \prec \gamma(\alpha')$, then either both of α, α' or none of them are β's causal predecessors. Intuitively speaking, the above property indicates that *there is at most one immediately causally preceding γ-operations of $\eta(\beta)$ in each chain.* So one can easily generalize the algorithm described in Section 4.1 to compute the logical time of β. Moreover, since $\gamma(\alpha), \eta(\beta), \alpha$ and β form a "diamond-structure" similar to the one in Figure 4.3b, multi-path reduction can also be generalized.

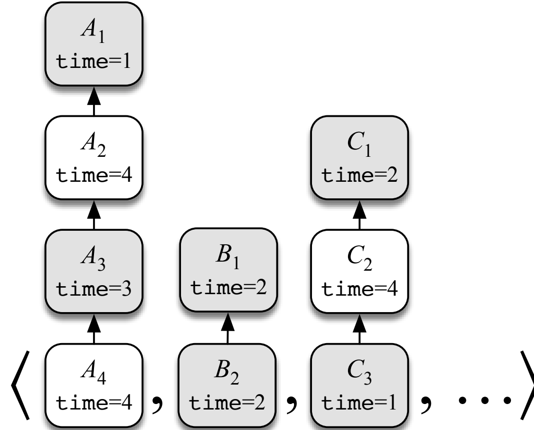


Figure 4.5: Illustration async-before lists for `Delayed` events. The causal predecessors of an event with `time` = 3 are shown in gray.

If ρ does not satisfy the above properties, we can still use AC_Λ with a helper data structure, *async-before lists*, to find immediate causal predecessors. We illustrate async-before lists with an example for `Delayed` events shown in Figure 4.5. We store all `Delayed` events sent from each chain in a list ordered by their `send`s (e.g., events A_i form a list, and B_i form another). To find the immediate causal predecessors of an event E with `time` = 3, we traverse each list, starting from the events in the AC at `send`(E) ($A_4, B_2, C_3, ...$), to visit all events whose `send`s happens-before `send`(E). The set of visited events satisfying the priority function (shown in gray) would contain all immediate causal predecessors of E.

4.3.2 Case Study: Realizing AsyncClock for Android

Here we describe how to use generalized AsyncClocks to handle Rule Atomic, and Rule Priority for Android described in Section 3.1.4. We use the rule evaluation order described in [15] to obtain the minimal closure of all causality rules. We also describe solutions for Rule AtFront, event removal, and events in binder threads.

Event Atomicity For an operation β in an event, we use generalized Async-Clocks to track its *immediately causally preceding* `begin` operations, thus their cor-

responding `end` operations are the immediate causal predecessors of β. Then, the logical time of β is resolved by inheriting the time of these `end`s.

Events with Priority Tags For each tag combination in Table 3.1, we use generalized AsyncClocks to track *immediately causally preceding* `send` operations along with async-before lists to find immediate causal predecessors. A `Sync` event should inherit the time from both its `Sync` and `Async` predecessors. For a `Delayed` or `AtTime` event E, we can speed up the search of predecessors by early-stopping the traversal of each list after visiting some event E' in the list such that 1. $E'.\texttt{time} = E.\texttt{time}$, or 2. $E''.\texttt{time} \leq E'.\texttt{time}$ for all preceding events E'' in the list. For example, in Figure 4.5, we can prune A_1 (Case 1) and B_1 (Case 2), since they happen-before A_3 and B_2 respectively and thus not immediate causal predecessors of the new event. The search of `AtFront` predecessors can also be avoided by caching the join of the logical times of all `AtFront` events sent along a chain.

Sent-at-Front Events Rule AtFront does not follow the form of generalized asynchronous causality rule, and thus cannot be addressed with AsyncClock. Instead, we maintain a *sent-at-front list* for each event in a queue. When an `AtFront` event is executed, it is added to all sent-at-front lists. When an event is dequeued, we iterate through its sent-at-front list to find its causal predecessors.

Event Removal An event removed explicitly by the programmer cannot be relinquished, since it might have an immediate causal successor. In such case, we first resolve the time of this removed event, and use it to resolve the time of the successor. Removed events are reclaimed through reference counting, or along with garbage collection when all its predecessors are old.

Events in Binder Threads We apply the speculative causality rules in [15] (summarized in Figure 4.6) to order events executed in binder threads. These speculative rules are heuristics observed from empirical program runs. Since they also follow the form of generalized asynchronous causality rule (Rule IpcHandle checks the sequen-

$$\frac{\begin{array}{c}\alpha = \texttt{end}(E_1) \\ \beta = \texttt{begin}(E_2) \\ \texttt{send}(_, q, E_1) < \texttt{send}(_, q, E_2) \\ \mathit{process}(\texttt{send}(_, q, E_1)) = \mathit{process}(\texttt{send}(_, q, E_2)) \\ E_1.\texttt{Sync} \vee E_2.\texttt{Async} \\ \mathit{binder}(E_1) = \mathit{binder}(E_2)\end{array}}{\alpha \prec \beta} \ (\textsc{IpcHandle})$$

$$\frac{\begin{array}{c}\alpha = \texttt{end}(E_1) \\ \beta = \texttt{begin}(E_2) \\ \texttt{send}(_, q, E_1) \prec \texttt{send}(_, q, E_2) \\ \mathit{thread}(\texttt{send}(_, q, E_1)) = \mathit{thread}(\texttt{send}(_, q, E_2)) \\ E_1.\texttt{Async} \wedge E_2.\texttt{Async} \\ \mathit{binder}(E_1) = \mathit{binder}(E_2)\end{array}}{\alpha \prec \beta} \ (\textsc{IpcAsync})$$

Figure 4.6: Speculative causality rules for events in binder threads [15]. Functions *process* and *thread* return the process and thread of an operation. Function *binder* returns the binder thread of an event.

tial order ($<$) between `send` operations, but this can be easily modeled in the causal relation), we can also use generalized AsyncClocks for these causality rules.

4.4 Race Detection

To detect data races under our causality model, we use the AsyncClock algorithm to derive the logical time for each memory access, and keep track of the epochs of recent read and write accesses, similar to FastTrack [28], to find races between a memory access and its previous and next write accesses to the same location. This guarantees us to find at least the first data race for each memory location.

As discussed in Section 3.2.1, data races reported by AsyncClock are either *harmful races* that lead to *order violation* bugs, or *harmless races* that produce "correct" results irrespective of their execution order and do not lead to any concurrency

bug. Android applications usually incur many harmless races, which significantly reduce the usability of a race detector.

However, without a formal correctness specification for any Android program, we have to rely on manual reasoning of a race and its events to determine if it is harmful or harmless, which is a common approach used in similar tools such as EventRacer [15]. We manually investigated the races reported by our tool, and observed that the following races are usually harmless:

1. Races in Android's framework that are not induced by the user code.
2. Races in certain OS-generated events that are *commutative* with other events (e.g., screen refresh events).
3. Race in libraries that provides *commutative operations* (e.g., increments to size when adding two items into a list).

Therefore, our tool only reports races between *user-induced* accesses (accesses in user code, or in libraries called by user code), and uses a *race filter* that whitelists pairs of commutative accesses and events described above to remove races that are highly likely to be harmless. We built the whitelist conservatively such that the filter did not remove harmful races in our experience. Our whitelist currently contains around 400 entries that mark commutative operations and events in the Android framework, OS, and libraries, but not in the user code. So, the manual effort required to create this whitelist is not necessary for every application.

To further improve usability, our tool reports races in *race groups*, similar to EventRacer: races that are induced by the same library invocations in the user code are grouped together. This reduce a fair amount of work for manual investigations on race reports.

4.5 Evaluation

In this section, we evaluate the performance AsyncClock and the effectiveness of the proposed solutions.

4.5.1 Experimental Setup

We used a Google Nexus 4 device with an instrumented Dalvik runtime of Android OS v4.3 to record the execution traces, then offloaded the traces to a machine equipped with 2.67G Intel Xeon X5650 CPU and 48GB of memory for race detection. We evaluated AsyncClock on 20 popular Android applications picked from [31, 44, 5, 15, 6]. We used *Android Monkey* [12] to generate traces that run in 10–30 minutes on an uninstrumented system (longer traces were collected for applications that generated fewer events per minute.)

4.5.2 Overall Performance

Trace Collection The statistics of collected traces and the overheads of trace recording for all 20 applications are shown in Table 4.1. The overhead is primarily incurred by recording all read and write accesses to heap objects. On average, instrumented runs were about 5x slower than uninstrumented runs with respect to CPU time.

Offline Analysis Column *Analysis* in Table 4.1 shows the time and memory of our race detector based on AsyncClock, with a 2-minute time window and FIFO chain decomposition. Note that we run the offline analyses on a local machine. The analysis overhead is computed as the analysis time (on local machine) normalized by the application CPU time (on phone), indicating how much slowdown a user would experience to use our tool to analyze an application. All analyses finished in 9 minutes, and averaged 3 minutes and around 700MB of memory.

4.5.3 Comparison with EventRacer

EventRacer [15, 6] is the state-of-the-art race detector for Android applications. Since it is close-sourced, we could not use its publicly available binary executable for various reasons. First, we were unable to adapt our causality model (Section 3.1.4) to EventRacer's implementation. Second, we wanted to compare AsyncClock and

Application	CPU Time	Operations		Threads			Events		Trace	Analysis		vs. EventRacer	
		Sync	Mem	L	B	W	Looper	Binder	Overhead	Overhead	Mem	Speedup	−Mem%
AnyMemo	145s	760k	67M	24	5	158	244584	1110	5.49x	2.41x	2095M	28.33x	81%
ConnectBot	279s	722k	166M	3	6	60	86056	4819	4.58x	0.69x	474M	17.03x	97%
Firefox	1448s	720k	195M	7	4	269	78719	2673	2.31x	0.16x	248M	10.38x	97%
NPRNews	283s	556k	198M	8	5	87	77619	50011	5.48x	0.79x	672M	9.38x	85%
K9Mail	369s	507k	164M	6	5	46	48493	8136	2.95x	0.44x	785M	5.85x	87%
OpenSudoku	216s	651k	121M	1	4	20	47062	2810	4.39x	0.55x	478M	5.17x	87%
SGTPuzzles	188s	702k	115M	3	5	65	42110	1938	4.95x	0.74x	592M	5.49x	90%
AardDict	148s	382k	80M	3	4	117	37345	4331	4.10x	1.04x	963M	4.78x	82%
BarcodeScanner	114s	208k	127M	2	3	17	34792	949	7.61x	1.24x	197M	5.49x	98%
FlymNews	251s	338k	122M	4	6	151	31690	1579	4.21x	0.81x	982M	4.08x	81%
RemindMe	157s	280k	56M	10	6	57	31637	1391	2.83x	0.34x	310M	2.49x	87%
AdobeReader	216s	418k	75M	14	4	307	31301	1751	3.47x	0.61x	831M	3.95x	79%
FlipKart	387s	811k	242M	34	4	233	31054	1264	10.78x	0.92x	1438M	2.92x	71%
OIFileManager	205s	593k	124M	76	5	227	30841	6694	5.05x	0.84x	723M	2.70x	76%
VLCPlayer	170s	1045k	90M	31	25	360	26241	28133	5.34x	0.98x	616M	3.44x	87%
ASQLiteManager	78s	421k	64M	1	4	34	25597	1529	6.27x	0.87x	455M	4.43x	88%
Twitter	273s	355k	159M	128	9	149	24333	2615	8.34x	0.61x	1134M	2.92x	70%
Tomdroid	143s	465k	70M	2	6	104	22121	3441	4.01x	0.72x	429M	2.21x	82%
FBReader	223s	632k	107M	14	5	56	21300	4064	3.85x	0.61x	548M	2.32x	73%
ATimeTracker	112s	291k	52M	1	6	22	19620	1880	3.35x	0.57x	462M	2.48x	67%
Average	270s	543k	120M	–			49626	6556	4.97x	0.80x	722M	7.99x	87%

Table 4.1: Summary of trace collection and analysis sorted by looper events. Columns *L*, *B* and *W* under Column *Threads* stand for looper, binder, and worker threads. Column *Analysis* shows the performance of AsyncClock with a 2-minute time window and FIFO chain decomposition. Column −*Mem%* shows the percentage of reduced memory usage (larger is better). All analyses are run offline on a local machine.

EventRacer for the exact same trace, and this was not possible without access to their source code. Also, we couldn't get to run or obtain long enough traces for several applications used in our analysis. Therefore, we re-implemented its happens-before graph construction algorithm in our framework, including graph traversal pruning and other optimizations presented in [15]. This enabled us to do an end-to-end comparison between the two algorithms with the same trace and the same framework overheads.

Column *vs. EventRacer* compares the performance in time and memory between our tool and EventRacer. Since EventRacer scales super-linearly (Figure 4.7), its overhead is not fixed, but grows with the number of events, so our tool achieved greater speedup when more events were analyzed. For the collected traces, our tool was at least twice as fast as EventRacer, and averaged 8x faster. The memory performance was also significant: with heirless events reclaimed and a 2-minute time window, our tool used only 0.7GB of memory on average, where EventRacer used 5.5GB.

We further studied the scalability of EventRacer and AsyncClock in Figure 4.7. As can be seen, EventRacer scaled super-linearly in time. We observed that its graph traversal pruning did not handle the scenario in Figure 4.8 well because:

1. It nearly pruned nothing for `AtTime` events since their `time`s are usually different.
2. The entire parent chain of B_3 (i.e., I_1, I_2, I_3) in Figure 4.8 was traversed to find its predecessors, which is proportional to the trace length.

In contrast, the time spent in finding predecessors of an event in our tool remained low due to the sparsity of AsyncClocks and early-stopping in the async-before lists of `Delayed`, `AtTime`, and `AtFront` events. EventRacer also does not scale in memory with number of events because it maintains the whole happens-before graph, where our tool shows better memory scalability due to storing only AsyncClocks for events and aggressively reclaiming events with our scalability optimizations described in Section 4.2.

Since both EventRacer's graph traversal algorithm and AsyncClock (with

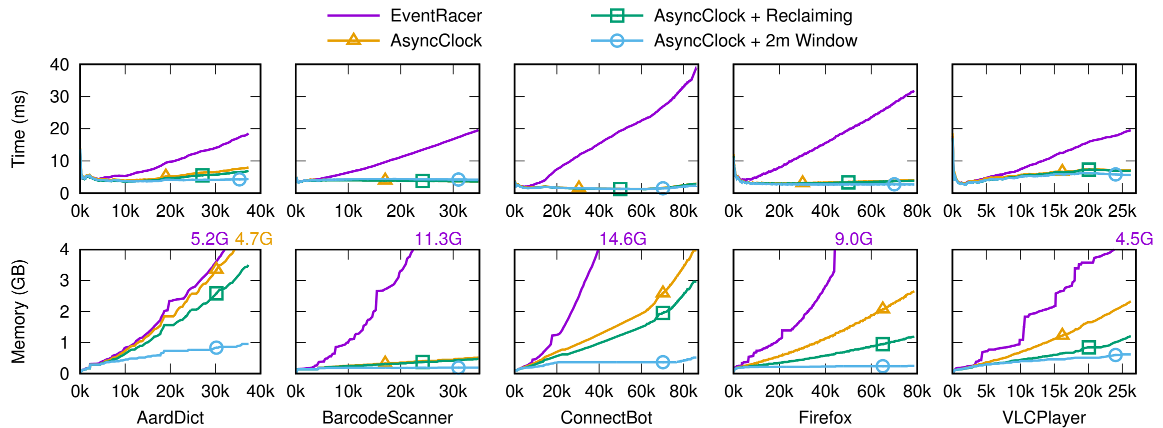


Figure 4.7: Comparison between EVENTRACER and ASYNCCLOCK. The x-axes show the number of looper events. The top row shows the average time spent *per event*. The bottom row shows total memory used during the analyses. The results of 3 configurations of ASYNCCLOCK are shown: no event reclaiming (△), reclaiming heirless events (□), and reclaiming events with a 2-minute time window (○).

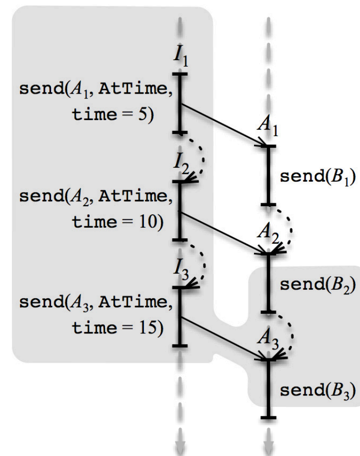


Figure 4.8: An event pattern in BarcodeScanner, where I_n are input events. EVENTRACER traversed the shaded area to find predecessors of B_3.

no time window) are sound, the races reported by both methods are the same under our causality model. However, both methods use an additional step that employs different heuristics to classify the found races into harmless and harmful races, which is orthogonal to finding races under a causality model and thus not part of the performance analysis.

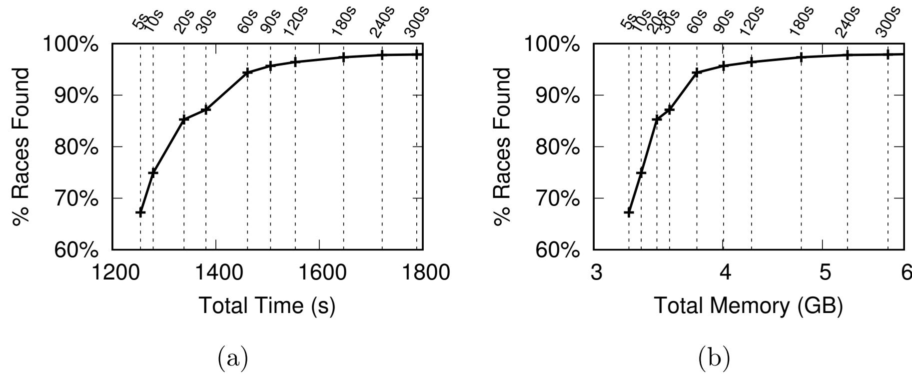
4.5.4 Scalability Improvements

Effectiveness of Reclaiming Events

Figure 4.7 also shows how effective the event reclaiming optimizations are in 5 applications. As can be seen, AsyncClock had good time performance even without any scalability optimization described in Section 4.2, but the memory usage grew with the number of events being processed. The two sound optimizations for reclaiming heirless events (reference counting and multi-path reduction) reduced half of the memory usage for Firefox and VLCPlayer. But for AardDict and ConnectBot, there were too many non-heirless concurrent events, which made the number of chains and metadata kept in memory increase rapidly. These two applications showed the need of time window approximation. With a 2-minute time window, our tool scaled reasonably well in memory.

Recall of Time Window Approximation

We evaluated how many false negatives were introduced by assuming happens-before relations between old and recent events in time window approximation. Figure 4.9 shows the trade-off between its recall and resource usage with different window sizes for 8 applications (listed in Table 4.2). As can be seen, only 4% of the races were missing when we used a 2-minutes or larger window. Most of these missing races are between the onCreate events of the benchmark applications and some later events, such as onPause or onDestroy events. After manual investigations, all the missing races in the benchmark applications are identified as harmless races. Therefore, we used a 2-minute time window in all other experiments.



(a) (b)

Figure 4.9: The percentages of races reported from 8 selected applications versus the total time and memory used for various time windows. The memory axis is in log scale.

FIFO Chain Decomposition

We compared FIFO chain decomposition with a naive greedy approach [57]. Our method achieved 5% improvement in memory, and 10% in time (due to finding chains with table-lookups for ordinary events).

4.5.5 Reported Races

Our tool found 1437 user-induced race groups in 8 selected applications. All of them are true races under the causality model, but many are harmless races and removed by our race filter. The results are shown in Table 4.2. As can be seen, 1106 race groups are removed by the filter (Row *Filtered*), and of the remaining race groups (Rows *Harmful* and *Harmless*), 44% of them are harmful. Compared to EventRacer [15], our tool reported more harmless races. We speculate the following reason: EventRacer reported only *uncovered races* and assumed that those races are potentially *custom synchronization operations* that order *covered races*. Since this heuristic does not guarantee soundness and could lead to false negatives, we decided not to apply this heuristic but only employed a conservative race filter. Therefore, our tool reported more true and false positives.

Application		Aard Dict	Barcode Scanner	Connect Bot	FB Reader	Firefox	OIFile Manager	Tom droid	VLC Player
All Races Groups		184	33	218	190	282	74	79	377
Filtered		99	25	175	164	217	51	65	310
Harmful		67	5	22	9	28	6	2	8
Harmless	Type I	6	0	8	0	10	2	1	7
	Type II	4	1	8	3	20	1	4	30
	Other	8	2	5	14	7	14	7	22

Table 4.2: The user-induced race groups reported in 8 applications. Row *Filtered* shows how many race groups were removed by our filter. Rows *Harmful* and *Harmless* are the actual number of race groups reported by our tool.

Harmful Races After manual inspection, we found 147 harmful races from the 8 applications listed in Table 4.2, including one that was confirmed by the developers of Firefox. Here are some harmful races we discovered:

- Firefox uses the main *UI thread* to process user input, and another *compositor thread* to render web pages. While rendering a page, the compositor thread would read the locale settings, which might be been updating by a UI event. So, the browser might render locale-specific contents incorrectly till the next UI refresh.

- BarcodeScanner initializes the CameraManager in the onResume event. The CameraManager is then used in the surfaceCreated event, which usually comes after onResume, but the order is not guaranteed by the Android system, and it might use a wrong CameraManager object, which is not cleaned up correctly in a previous onPause event, to initialize the camera.

- VLCPlayer switches from the audio player mode to the video player mode when the next item in the playlist is a video without checking if the next item has been nullified because of loading a new playlist, which would lead to a NullPointerException.

Harmless Races Although the problem of identifying harmless races is orthogonal to this work, we manually investigated the types of harmless races to understand

the causes to give us some insights to develop better heuristics in the future. In our experience, over 50% of the harmless fall in the following two categories:

Type I (*Delayed update*): Many races are between an event that modified the UI components or internal data structures in reaction to user input, and an event independently generated by Android to update the UI. The changes made in the earlier events would not be observed until the later event executed.

Type II (*Control-dependent races*): Since events are executed atomically with respected to each other, programmers usually write to a flag variable in an earlier event to change the execution of a later event. Some of the races in this categories might be able to remove by the *If-Guard check* proposed in CAFA [31].

4.6 Summary

In this chapter, we presented a new primitive, AsyncClock, to infer happens-before relations between asynchronous events for a wide variety of causality models for event-driven systems. We also addressed an important scalability problem: reclaiming heirless events. We built the first single-pass, non-graph-based Android data race detector, and show that our algorithm is scalable in time and space. We used it to find 147 previously unknown harmful races in popular Android applications.

CHAPTER 5

Statistical Commutativity Analysis for Asynchronous Events

Previous studies [58, 64, 14, 34, 28] detect concurrency errors in thread-based programs through finding *data races* under the conventional causality model [14, 34], where a data race is a pair of memory accesses at a shared data, where at least one of them is a write, and they are not ordered by the causality model. Although data races have been good indicators of concurrency errors in conventional thread-based programs, this is not true for data races defined by a causality model for event-driven programs. For thread-based programs, many data races can easily lead to incorrect results in some thread interleavings, and are prohibited in the data-race-free-0 (DRF0) memory model [13]. In contrast, in event-driven programs, since events from the same queue are *non-preemptively* processed by the same looper thread, these events are *atomic* with respect to each other. As a result, data races in these events are allowed in the DRF0 model, and data races in *commutative* events are *harmless*, in the sense that they produce programmer-intended results, no matter what order they are executed. Therefore, the major challenge in identifying harmful data races is to determine if two events are commutative.

In this chapter, we present a way to use *invariant-based bug detection* to address this challenge and identify concurrency bugs in event-driven programs. Although invariant-based bug detectors have been successful for thread-based programs [26],

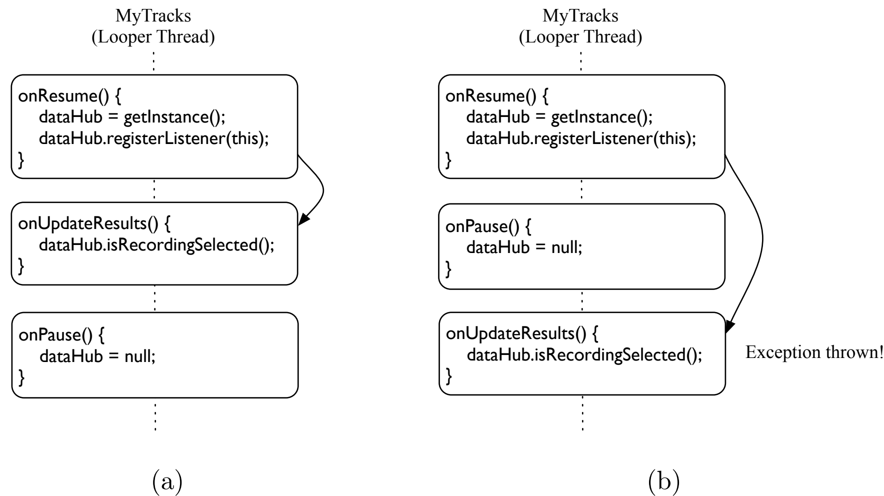


Figure 5.1: A concurrency bug in Google's MyTracks app. (a) The precondition "dataHub != null" of onUpdateResults is always true in a correct execution. (b) The precondition is falsified by onPause in an incorrect execution.

they have not been applied to find concurrency errors in event-driven programs. We built a new concurrency bug detector for event-driven programs named Licorice, which combines the ideas of invariant-based bug detection and happens-before race detection based on the following observation: programmers usually assume certain *preconditions* for events to be correctly executed, but some of these preconditions are violated when concurrency bugs manifest. For example, "dataHub != null" is assumed for event onUpdateResults in Figure 5.1a, and is violated by event onPause in Figure 5.1b. For any pair of events, if executing them in a certain order would violate any precondition of some event, then this pair of events are non-commutative and contain a concurrency bug.

Licorice is a *statistical commutativity analysis*. It works in two phases: *learning* and *detection*. In the learning phase, Licorice analyzes a large number of correct executions to statistically learn likely preconditions of each event handler. These likely preconditions usually indicate programmers' assumptions for the event handlers to be executed correctly. In the detection phase, Licorice analyzes a new execution and identifies if there are two concurrent events such that they could be reordered to falsify

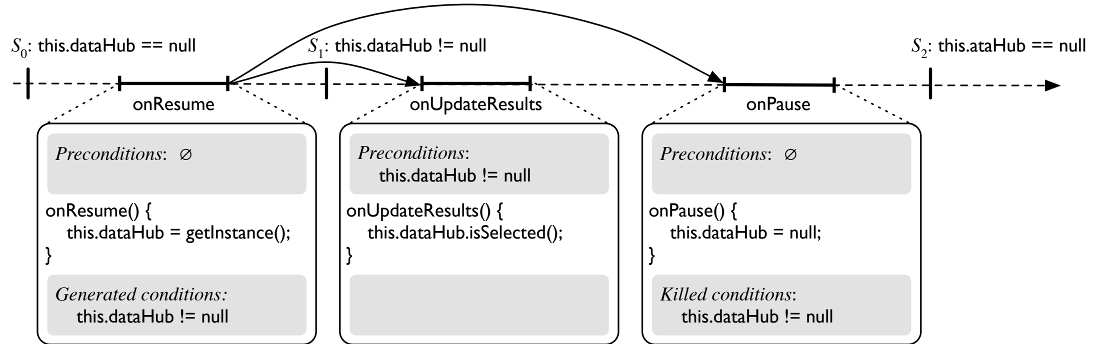
a precondition of any event. These two events are therefore likely non-commutative and may constitute a concurrency bug.

5.1 Running Examples

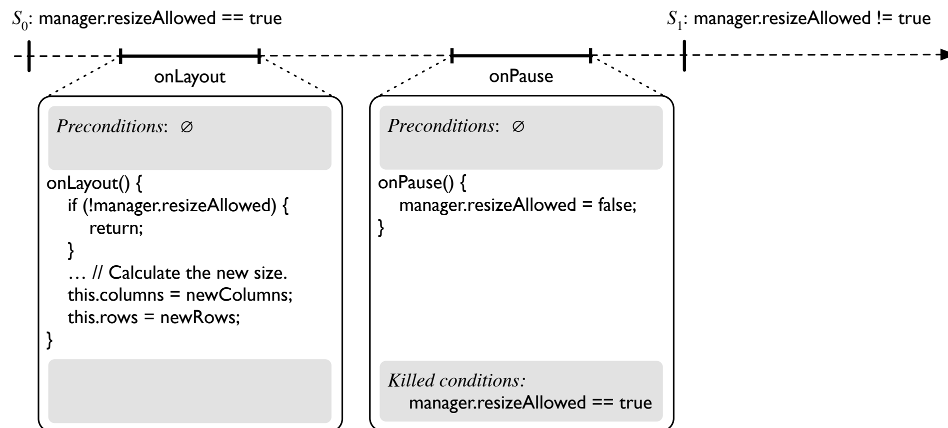
We show how to identify non-commutative events via their preconditions with two examples in Figure 5.2. In the figure, the arrows indicate the happens-before relation between the events. As can be seen, there is a data race on variable `this.dataHub` between events `onUpdateResults` and `onPause` in Figure 5.2a, and another data race on variable `manager.resizeAllowed` between events `onLayout` and `onPause` in Figure 5.2b. However, although these data races lead to different program states when the racing events are reordered, it is hard to reason if they lead to *incorrect* program states.

If we have the knowledge of the *preconditions* of each event (which we will discuss how to find later), we could use it to reason if reordering two concurrent events generates incorrect program states. In Figure 5.2a, `onUpdateResults` is executed on program state S_1. S_1 satisfies "`this.dataHub != null`," which is a precondition of `onUpdateResults`. If we reorder the two events `onUpdateResults` and `onPause`, `onUpdateResults` will be executed on program state S_2 where "`this.dataHub != null`" is violated, so `onUpdateResults` cannot be correctly executed. Therefore, by observing how `onPause` affects the precondition of `onUpdateResults`, we can tell that these two events are non-commutative and thus contain a concurrency bug. If we reorder the two events `onLayout` and `onPause` in Figure 5.2b, however, there is no precondition violation since the preconditions of `onLayout` agree with both program states S_0 and S_1, and thus no error.

As can be seen in the two examples, we usually do not care if an event is executed on the same program state in different executions, as long as the program states do not violate any of its preconditions. Therefore, we can think the preconditions of an event as a *high-level description* that summarizes all valid program states this event can be executed on. If two events are non-commutative, there is an execution order such that one event is executed on an invalid program state, hence an error would manifest.



(a)



(b)

Figure 5.2: Illustration of non-commutative events. The solid arrows show the happens-before relation between the events. (a) In Google's MyTracks app, events onUpdateResults and onPause contain a harmful data race on variable this.dataHub and these two events are non-commutative as reordering them violates the precondition "this.dataHub != null of onUpdateResults.". (b) In the ConnectBot app, onLayout and onPause contain a harmless data race on variable manager.resizeAllowed but no precondition of any event is violated when they are reordered.

5.2 Event Commutativity Analysis

In this section, we formally describe *event preconditions* and how to use them to identify non-commutative events. Conceptually, if two events are commutative, then none of their execution order would affect the correctness of any event, so no preconditions should be violated. In other words, if two events can be scheduled in a way such that a precondition of some event is likely to be violated, then these two events are non-commutative, and there would be a concurrency bug between these two events.

5.2.1 Event Precondition

An event handler is usually implemented as a function in a high-level programming language. When implementing an event handler, programmers access shared data through *symbolic names*, such as manager.resizeAllowed in Figure 5.2b. A **symbolic condition** is a valid Boolean expression in the programming language that involves one or more shared data. For example, in the same figure, "manager.resizeAllowed == true" is a symbolic condition involving one shared datum, and "this.columns <= this.rows" is another symbolic condition involving two shared data. To simplify the design, we consider only *atomic* symbolic conditions, which is defined in Section 5.3.1 The **precondition specification** $P(h)$ of an event handler h is a set of atomic symbolic conditions that must be satisfied to ensure a correct execution of the event handler h.

The execution of an event consists of one or more event handler invocations. When executing each event handler, their symbolic names are bound to memory locations at runtime. A **condition** is a runtime realization of a symbolic condition in which the symbolic names are replaced by their memory locations. For example, suppose variable manager in event onLayout in Figure 5.2b is bound to a dynamic object instance *obj1*, then "*obj1*.resizeAllowed == true" is a condition that realizes "manager.resizeAllowed == true." Given a program state S that contains the values of all shared data, we can evaluate if it satisfies any condition c. We say that $S \models c$ if S satisfies c. In the

same figure, $S_0 \models$ "$obj1$.resizeAllowed == true" but $S_1 \not\models$ "$obj1$.resizeAllowed == true".

With a proper *symbolic naming mechanism* (which will be discussed in Section 5.3.2), the dynamic binding between symbolic names and memory locations in an event handler is fixed during its dynamic execution. Given a dynamic execution η of event handler h, denote the condition that realizes a symbolic condition ϕ as $\phi[\eta]$, then its set of **preconditions** is defined as the set of conditions that realize $P(h)$:

$$P[\eta] = \{\phi[\eta] : \phi \in P(h)\}$$

$P[\eta]$ is the set of conditions that must be satisfied to ensure that η is a correct execution.

To ensure the correct execution of an event, however, not all the preconditions of its event handler invocations need to be satisfied at the beginning of the event, because a former event handler invocation might modify shared data to satisfy some preconditions of a latter event handler invocation. Therefore, only preconditions that involves *live-in* shared data need to be satisfied at the beginning of the event, where a live-in shared datum is a shared datum such that the event read its value from a write in another event. Therefore, for an event e, we can define its set of preconditions $P[e]$ as follows:

$$P[e] = \bigcup_{\eta \in e} \{\phi[\eta] \in P[\eta] : \phi[\eta] \text{ contains only live-in shared data of } e\}$$

For e to be correctly executed on program state S, every condition in $P[e]$ must be satisfied, which we mathematically write as $S \models P[e]$.

5.2.2 Event Commutativity

Given an event e, suppose that the program states before and after its execution are S and S' respectively. For any condition c, we say that e *requires* c, or e is a *REQ* event of c, if $c \in P[e]$. Also, e *generates* condition c, or e is a *GEN* event of c, if $S \not\models c$ but $S' \models c$; e *kills* c, or e is a *KILL* event of c if $S \models c$ but $S' \not\models c$. e is a *NOP* event of c if it is neither a *REQ*, *GEN* nor a *KILL* event of c. Note that an event can be a *REQ* and *KILL* event of c at the same time.

Consider two concurrent events. If for all precondition c of all events, both events are *GEN* events of c, then these two events would result in a program state S such that $S \models c$ irrespective to their execution order, so their order does not make any difference to any later *REQ* events. Thus, they are commutative with respect to those *REQ* events. Similarly, if they are both *KILL* events of c, or one of the two events is a *NOP* event of c, then these two events are also commutative with respect to later *REQ* events. If there is a precondition c of some event such that these two events are a *GEN-REQ*, *GEN-KILL* or *REQ-KILL* pair for c, then these two events are *likely non-commutative*, and there is a high chance that they would contain a concurrency error.

For a condition c, however, not all *GEN-KILL* event pairs are non-commutative. For example, if every event requiring condition c happens before an event e that kills c, then there is no way that reordering e makes an event requiring c executed on a program state that does not satisfy c. So, any *GEN-KILL* pairs involving e are commutative. Similarly, some *GEN* events of c are commutative to all *KILL* events of c. We call such events **minor *GEN*** and **minor *KILL*** events of c. An event e is a minor *GEN* of condition c if every event e' requiring c satisfies one of the following statements:

1. $e' \preceq e$.

2. $e \preceq e'$ and $\exists$ event e^* such that e^* kills c and $e \preceq e^* \preceq e'$.

Likewise, an event e is a minor *KILL* of condition c if every event e' requiring c satisfies one of the following statements:

1. $e' \preceq e$.

2. $e \preceq e'$ and $\exists$ event e^* such that e^* generates c and $e \preceq e^* \preceq e'$.

Figure 5.3 illustrates the idea of minor *GEN* events. In Figure 5.3a, G_c is an event that generates condition c. If all events requiring c are either happens before G_c, or happens after G_c with some K_c that kills c in between, then G_c has no effect on whether these events are executed on valid program states. Similarly, as shown

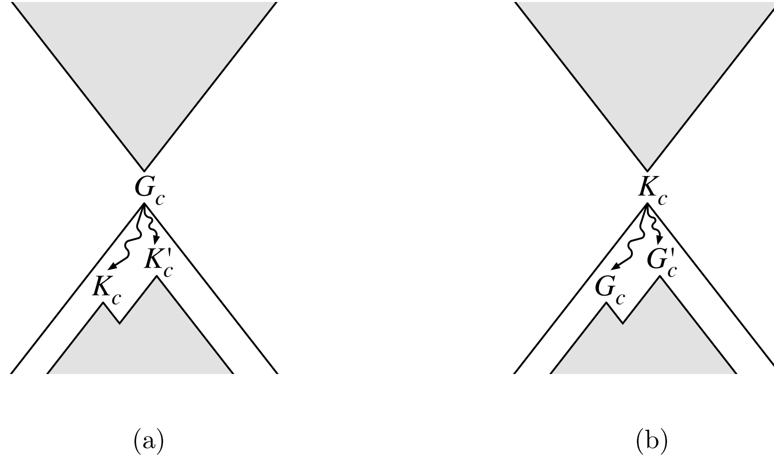


Figure 5.3: Illustration of minor *GEN* and minor *KILL* events. G_c, G'_c are events that generate condition c, and K_c, K'_c are events that kill condition c. The triangles above and below an event indicates the "area" that happens before and after that event respectively. (a) If all events that require c are in the gray area, then G_c is a minor *GEN* of c. (b) If all events that require c are in the gray area, then K_c is a minor *KILL* of c.

in Figure 5.3b, a minor *KILL* event K_c of condition c has no effect on whether any event requiring c is executed on a valid program state. Since minor *GEN* and minor *KILL* events have no effect on the correctness of the execution they are commutative to all other events for condition c.

5.3 Learning Precondition Specifications

A challenge of the event commutativity analysis described in Section 5.2 is how to obtain the precondition specification of each event handler in an event-driven program. Asking programmers to provide such specification is not practical, since it requires substantial manual effort, is prone to human errors, and is not possible for existing programs. Obtaining through symbolic execution [38] is viable, but this approach has two problems: 1. it suffers from the path explosion problem and does not scale when analyzing complex functions in real-world applications; 2. it can only

find preconditions that prevent obvious errors (e.g., no exceptions or assertion errors), and cannot be used to detect other types of errors.

Our idea to obtain the precondition specification for each event handler is to use a *dynamic likely invariant detector* [26] to *statistically* learn the precondition specification from multiple correct executions of an event handler. In each correct execution, programmers' assumptions for the event handler must be true, and it is likely that if we analyze a sufficient amount of executions, preconditions that are always true are likely assumed to be so by programmers.

Therefore, Licorice first runs a *learning phase* that uses the Daikon invariant detector [26] to learn likely precondition specifications for all event handlers from *training executions*, and then use the learned specifications to analyze the commutativity of events when analyzing a new execution in its *detection phase*. In this section, we describe how we use Daikon to implement Licorice's learning phase.

5.3.1 Types of Atomic Symbolic Conditions

Since we use Daikon to learn the atomic symbolic conditions in a precondition specification, the types of atomic symbolic conditions are limited by Daikon's predefined types of likely invariants. We summarize the types of atomic symbolic conditions we consider below, where x and y are symbolic names, and a, b and c are constants.

- Pointer conditions: For pointer variables, the conditions we consider includes being null ($x ==$ null) and equivalence ($x == y$), and their negations.
- Numerical conditions: For numerical variables (integers and float-point numbers), we consider the following conditions: being constant ($x == a$), being in a range ($a <= \mathsf{x} <= b$), being nonzero ($x \mathrel{!=} 0$), being one of specific values ($x \in \{a, b, c\}$), equivalence ($x == y$) and its negation, ordering ($x < y$) and its negation, linear ($y = a * x + b$), being divisible ($x \% y == 0$), and being square ($y = x * x$).

- Integral conditions: For integer variables, in additional to the numerical conditions, we consider these extra conditions, since they are commonly used for bitwise flags: bitwise exclusion (x & y == 0) and being a bitwise subset (x & y == x).

5.3.2 Symbolic Naming

A symbolic condition uses symbolic names, similar to what programmers do in the source code, to refer to shared data it conditions on. A good symbolic naming mechanism should satisfy the following two properties:

- *Execution independence*: A shared variable in the program source code could be bound to different memory locations across executions. To successfully learn symbolic conditions of a shared variable from multiple training executions, its symbolic name should remain the same, independent from dynamic executions.

- *Event consistency*: Suppose the program states before and after the execution of event e are S and S' respectively. To check if e generates or kills condition c, Licorice requires that every symbolic name in c is bound to the same shared data at both S and S'. For example, in Figure 5.2b, if the object referred to by variable manager is changed from *obj1* to *obj2* during the execution of onLayout, then "manager.resizeAllowed == true" would refer to "*obj1*.resizeAllowed == true" at S but "*obj2*.resizeAllowed == true" at S', thus we cannot check if either condition is generated or killed by onLayout. Therefore, we cannot use the variable names in the source code as their symbolic names.

Therefore, Licorice names a heap object by its *allocation context*, as suggested by Smaragdakis, *et al.* [60], to fulfill the above two requirements. If multiple heap objects sharing the same allocation are accessed in the event, they are further numbered according to the order of their creation.

5.3.3 Daikon Trace Generation

Licorice uses Daikon as a backend engine to learn the precondition specifications of all event handlers. For each event handler, Daikon requires multiple samples, each consisting of symbolic names and their live-in values. It first enumerates all possible atomic symbolic conditions, and remove any symbolic condition that cannot be satisfied by at least one sample. Since the atomic symbolic conditions we consider involve at most 2 shared data, the time complexity of Daikon's learning process is quadratic with respect to the size of each sample. Clearly, dumping a complete snapshot of all shared data in the program for a sample is impractical. To resolve this problem, Licorice dynamically infers *abstract types* for each shared data, where two shared data are of the same abstract type if they are involved in a comparison or arithmetic operation during execution [30]. For each sample, Licorice only output the following data:

(1) Shared data that are accessed during the execution of the event handler.

(2) For each shared datum in (1), the other fields in the same heap object that are of the same abstract type are also outputted.

5.3.4 Obtaining Training Executions

To accurately learn the precondition specifications of all event handlers, we need a sufficient number of *correct* and *different* training executions: the training executions need to be correct so that we do not miss any symbolic conditions; they need to be sufficiently different so that we do not learn false symbolic conditions. To reduce human effort on collecting correct trace, we suppose that *concurrency bugs rarely manifest even with bug-exposing inputs*. This assumption is generally believed for thread-based programs, and is used in past work [42]. We believe this is also true for event-based programs, and assume that executions with no self-evident errors are correct.

To obtain sufficiently different executions, we use the following two methods:

1. Each training execution is generated by running the application with random user actions.
2. For half of the training executions, we randomly shuffle the event schedule.

We shuffle the event schedule by inserting delays when generating half of the events, where the delays are drawn from an exponential distribution with a predefined mean. The reason is that, for two concurrent events e_1 and e_2, if e_1 is always executed before e_2 in every training execution, then the conditions generated by e_1 may be falsely treated as preconditions of e_2. To avoid this, we need executions such that e_2 is executed before e_1. Since they are concurrent, we simply assume that they are independently generated, so the time gap between the generations of these two events would be an exponential distribution. By delaying e_1 with sufficiently, we can create the desired executions to remove false preconditions.

To avoid reducing the responsiveness and performance of the application, when e_1 is being delayed and its event queue is empty, the delay would be canceled. However, we will not be able to shuffle events with large time gaps, and thus it is hard to remove false preconditions generated by those events.

5.4 Detecting Non-Commutative Events

This section describes Licorice's *detection phase* for finding event likely concurrent non-commutative events in a new execution. The detection algorithm is a modified version of FastTrack [28]. but instead of finding data races on shared datum, we treat every condition c as a *logical boolean variable*, then *REQ* events of c can be thought of as *reads* to the logical variable, and non-minor *GEN* and *KILL* events can be considered as *writes* to the logical variable. Then, we find *GEN-REQ*, *GEN-KILL*, and *REQ-KILL* races between these logical operations as follows.

The algorithm first use the AsyncClock algorithm described in Chapter 4 to compute a vector clock with chain decomposition for each asynchronous event. For each condition c, the algorithm maintain its *satisfactory state* S_c, a list of *requiring*

epoch $R_c^{(i)}$, one for each chain, that records the epoch of the last *REQ* event of c in chain i, and a *modification epoch* M_c that records the epoch of the last *GEN* or *KILL* event of c. For each event e with vector clock $\mathbb{V}_e$, Licorice performs the following checks to find likely non-commutative events:

1. When e is a *REQ* event of c, Licorice first checks if S_c is true. If not, it reports that there is a precondition violation in the execution. Otherwise, we know that M_c records the epoch of some *GEN* event e' of c. If $M_c \not\preceq \mathbb{V}_e$, there is a *GEN-REQ* race between the e' and e, and e' and e are likely non-commutative.

2. When e is a non-minor *KILL* event of c, S_c must be true prior to the execution of e (otherwise there must be a data race between threads), so we know that M_c records the epoch of some *GEN* event e' of c. Licorice first checks if $R_c^{(i)} \preceq \mathbb{V}_e$ for each chain i. If not, a *REQ-KILL* race is reported between the corresponding event and e, meaning that these two events are likely non-commutative. Then it checks if $M_c \preceq \mathbb{V}_e$. If not, there is a *GEN-KILL* race between e' and e, so they are likely non-commutative.

3. When e is a non-minor *GEN* event of c, S_c must be false prior to the execution of e, so we know that M_c records the epoch of some *KILL* event e' of c. If $M_c \not\preceq \mathbb{V}_e$, there is a *GEN-KILL* race between e' and e and they are likely non-commutative.

Note that it is not necessary to check for *REQ-KILL* races in Step 1, because, if there is a race between an earlier event killing c and a later event requiring c, there must be another event in between that generates c, and that event must be concurrent with at least one of the two events. Similarly, it is not necessary to check for *GEN-REQ* races in Step 3.

To avoid reporting false positives, Licorice also uses the whitelist of *commutative* operations and events in system libraries described in Section 4.4 to check if the events requiring, generating, or killing a condition due to these commutative operations, and remove these known false positives.

5.5 Limitations

Licorice finds concurrency bugs in event-drive programs by detecting likely non-commutative events. A pair of likely non-commutative events (e_1, e_2) only indicates that in the current execution, some precondition of an event e' is satisfied because the two concurrent events are executed in the observed order, so either a causal order is suggested between e_1 and e_2, or e' should not rely on the precondition. However, we cannot tell if the precondition of e' would be falsified when e_1 and e_2 are actually reordered. Therefore, there is no guarantee that e_1 and e_2 are actually non-commutative, so they may incur no concurrency bug.

Licorice assumes that concurrency bugs always involve some conditions that would be broken in buggy executions. While it is largely true, this assumption is a heuristic, so it cannot detect any concurrency bug that does not share this property. Furthermore, if the involving conditions cannot be detected by the backend dynamic invariant detector, then Licorice also cannot detect such bugs. However, Licorice provides a framework that can be easily extended to include new types of conditions.

As described in Section 5.3.4, if two concurrent events are always executed in the same order in all executions, Licorice may learn false preconditions for the event handlers called in the later event, and thus mistakenly consider two commutative events as non-commutative. Although Licorice alleviates this problem by shuffling events, concurrent events that are far apart in time are still hard to be reordered, so they can still cause this problem.

5.6 Evaluation

In this section, we evaluate the effectiveness of Licorice using the following measures. First, we evaluate the *sensitivity* of the learning phase to learn the amount of training executions required. Second, we show the improvement in accuracy of race detection with Licorice, compared to the AsyncClock data race detector for event-driven programs described in Chapter 4. We also discuss the experience of using Licorice to

find concurrency bugs, and the reasons for false positives and false negatives.

5.6.1 Experimental Setup

We picked 7 popular Android applications to evaluate Licorice. Each application was run on an instrumented Android 4.3 ROM on a Google Nexus 4 device for trace collection, with random user actions generated by Android Monkey [12]. The training execution traces were generated with a random amount of user actions, averaged at 1000 actions, and the testing execution traces were generated with exactly 2000 random user actions. We modified Daikon v5.4 [26, 4] to learn the precondition specifications of the event handlers in each application.

5.6.2 Sensitivity of Precondition Specification Learning

We evaluated the sensitivity of the learning phase as follows. For each application, we varied the number of training executions to learn different versions of likely precondition specifications, then use another set of *correct* executions (the testing executions) to verify if how many likely preconditions are falsified by the testing executions.

Figure 5.4 shows the results of the sensitivity study. As can be seen in Figure 5.4a, the number of likely preconditions may either increase or decrease when we have more training executions. On one hand, more executions expose more program behaviors and more event handler invocations, enabling us to learn more preconditions; on the other hand, false preconditions can also be invalidated when more program behaviors are observed. For most applications, the precondition specifications become stable when we have more than 40 training executions.

Figure 5.4b shows how many preconditions were failed on the testing executions. Unsurprisingly, the failure rate of preconditions decreases as more training executions are used for most applications. OIFileManager shows an increment in the failure rate between 40 and 50 training executions, because some false preconditions are learned from the last few training executions, and there were not enough samples to invalidate

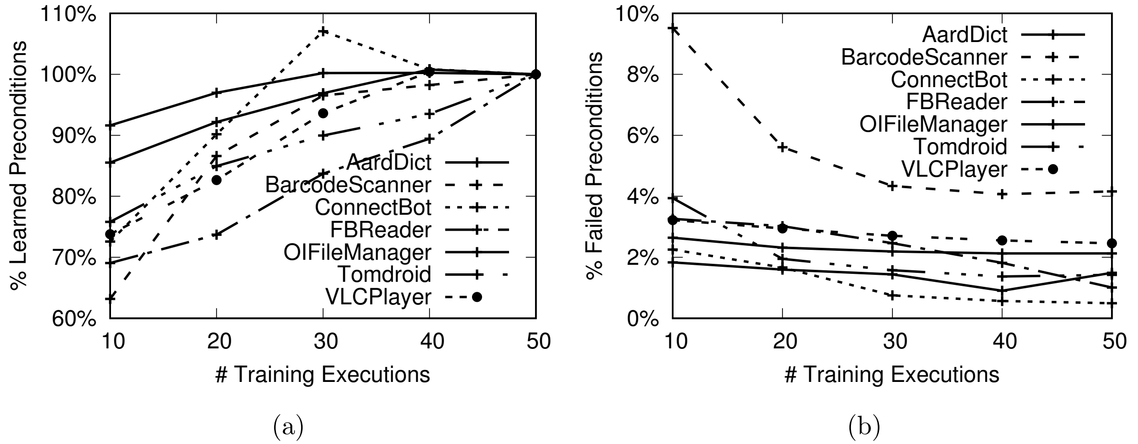


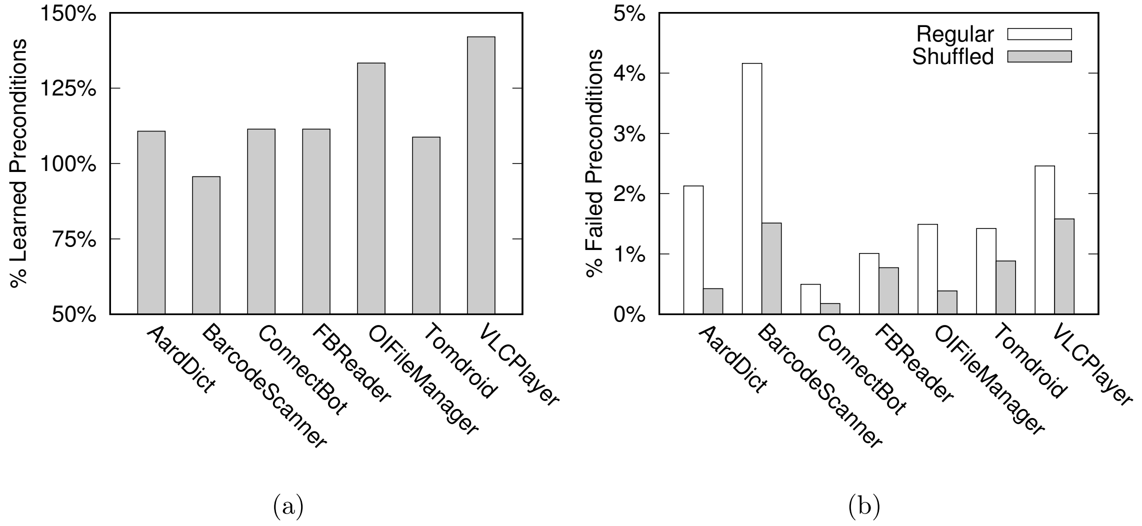
Figure 5.4: Sensitivity study for the learning phase. (a) The number of learned preconditions versus the size of training executions. The number of learned preconditions at each training size are normalized with respect to that at 50 executions. (b) The percentage of preconditions failed by the testing executions versus the training size.

these preconditions.

To expose more program behaviors, we replaced half of the 50 training executions with *shuffled* executions (Section 5.3.4), and compared the resulting precondition specifications with those trained from 50 regular executions. Figure 5.5 shows the comparison. In Figure 5.5a, we can see that for most applications, Licorice actually learned more preconditions from the shuffled executions, because more program behaviors that rarely manifest in regular executions could occur when events are shuffled. And despite more preconditions are learned, the quality of the learned preconditions does not drop, as can be seen in Figure 5.5b. Not only the percentages, but also the actual numbers of failed preconditions are reduced for all applications, showing that event shuffling is a useful technique.

5.6.3 Accuracy

We compare the improvement in accuracy of Licorice with the AsyncClock data race detector for event-driven programs described in Chapter 4 in Table 5.1. AsyncClock reported a total of 569 data races in all applications, including 328 in-thread data



(a) (b)

Figure 5.5: Regular versus shuffled executions for precondition specification learning. (a) The percentage of learned preconditions from shuffled executions with respect to regular executions. (B) The percentage of preconditions failed by the testing executions.

races between concurrent events. In contrast, Licorice only reported 94 races, therefore reduced the human effort by 71%. The overall accuracy, including both reported inter-thread and in-thread races, is improved from 44% to 75%.

When inspecting the race reports, we found that many races detected by Licorice point out critical conditions that affect the correctness of the applications. For example, in the 13 races in BarcodeScanner, 11 of them are critical for correctness, including one that leads to a warning when violated. Although most of them are

Application		Aard Dict	Barcode Scanner	Connect Bot	FB Reader	OIFile Manager	Tom droid	VLC Player	Total
AsyncClock	Reported	8	48	32	93	21	28	98	328
	Harmful	0	3	0	0	2	1	4	10
Licorice	Reported	3	13	8	13	5	1	51	94
	Harmful	0	0	0	0	2	1	4	7

Table 5.1: Reduction in the amount of human effort needed in inspecting the races reported by Licorice, compared to those by a data race detector.

correctly handled due to the well-written application code, the races still provide useful information about where in the code a bug may be easily introduced in future development.

5.6.4 False Alarms

We also investigated the causes of false alarms. Most of the false alarms are due to the following reasons:

- *Insufficient event shuffling*: We observed that many concurrent events that are far apart in time creates many false preconditions in the learning phase. And it is hard to reorder these events through our random event shuffling technique. Therefore, a systematic approach for probabilistic testing of event-driven programs is required to address this issue in the future.

- *Control-flow dependency*: Some event handlers are only invoked by the event dispatcher under certain conditions, and Licorice would mistakenly learn those conditions as the preconditions of the event handlers, generating false alarms. This could be resolved by learning *predicates* of event handlers: under what the conditions the event handlers will only be invoked. By differentiating predicates from preconditions, we could hopefully remove such false alarms.

5.6.5 False Negatives

We have also observed that 3 harmful races are missing by Licorice. After further investigation, we found that these races are missed due to a type of preconditions that cannot be learned by the current Daikon implementation, and we can easily extend Daikon to recognize new types of preconditions and hopefully report all harmful races.

5.7 Summary

In this chapter, we presented a new statistical commutativity analysis for asynchronous events in event-driven programs to find concurrency bugs. The analysis combines the ideas of invariant-based bug detection and happens-before race detection: it statistically learn likely preconditions of all events from many correct executions, and identify likely non-commutative events that are not causally ordered and thus could lead to concurrency errors. Our study reduced a significant amount of human effort needed in finding concurrency bugs through race detection, thus improved its usability.

CHAPTER 6

Conclusion

Event-driven programming models are commonly used today for building computing systems that range from mobile and web applications to distributed systems to sensor networks, yet there is a short in software reliability tools for these systems. The research in this dissertation focuses on filling this gap by developing algorithms and tools to find order violation bugs for event-driven applications. It addresses three important problems in this area: inferring ordering invariants between asynchronous events, efficient and scalable race detection based on the ordering invariants, and improving the usability of race detection for event-driven programs. The contribution of this dissertation includes: a new causality model that accounts the asynchronous causal order between events, the first efficient dynamic race detection algorithm that can infer the happens-before relation based on the new causality model, and a new concept of high-level races that can more accurately identify atomicity violations and order violations in event-driven programs. While we developed our tools and conducted the evaluations on the Android platform, the techniques we have developed can be easily adapted to include other event-driven platforms, as well as web applications and other event-driven systems.

There are many challenges arising from this research. The most fundamental one is: what is the best programming model for a shared-memory event-driven system? Through empirical study of mainstream event-driven platforms, we have developed the concept of asynchronous causality in Chapter 3, which provides good ordering

guarantees between events for programmers to reason about the behavior of event-driven programs. It is desirable to understand what general guarantees an event-driven platform should offer in the programming model for programmers to develop reliable software more easily. For example, we have observed that control-flow dependencies between events has been abused in event-driven applications, which constitute many false alarms in our race detector. Programming and debugging techniques might benefit from formulating such a paradigm in an event-driven programming model.

Due to the asynchronous nature of event-driven systems, many techniques for debugging thread-based programs cannot be easily adapted for event-driven programs. We have developed an efficient race detection algorithm for event-driven programs based on asynchronous causality in Chapter 4, but there are more to be done along the line, such as systematic testing [48, 46, 47] and probabilistic concurrency testing [16], which are powerful testing techniques for thread-based programs, but there is only limited development for event-driven programs [36, 43]. Deterministic replay is another important technique that shows recent development [29, 32]. The techniques we have developed in this dissertation could benefit from future development in these areas. Our work is just a small step in filling the gap of software reliability between thread-based and event-driven programming, and we hope the contribution of this work could enable a broader development for event-driven programming.

BIBLIOGRAPHY

[1] Android Open Source Project. `http://source.android.com/`.

[2] Class Binder in Android. `http://developer.android.com/reference/android/os/Binder.html`.

[3] Class Handler in Android. `https://developer.android.com/reference/android/os/Handler.html`. [Online; accessed 2017-01-27].

[4] The Daikon invariant detector user manual. `https://plse.cs.washington.edu/daikon/download/doc/daikon/`. [Online; accessed 2017-02-08].

[5] Droidracer. `http://www.iisc-seal.net/droidracer`. [Online; accessed 2016-08-15].

[6] EventRacer for Android. `http://eventracer.org/android/`. [Online; accessed 2016-08-15].

[7] F-Droid. `https://f-droid.org`.

[8] The future of mobile application. `http://businessdegrees.uab.edu/resources/infographics/the-future-of-mobile-application/`. [Online; accessed 2017-01-25].

[9] List of free and open-source Android applications. `http://en.wikipedia.org/wiki/List_of_free_and_open-source_Android_applications`.

[10] Method sendMessageAtFrontOfQueue in Android. `https://developer.android.com/reference/android/os/Handler.html#sendMessageAtFrontOfQueue(android.os.Message)`. [Online; accessed 2017-02-01].

[11] Method setAsynchronous in Android. `https://developer.android.com/reference/android/os/Message.html#setAsynchronous(boolean)`. [Online; accessed 2016-08-15].

[12] UI/application exerciser monkey | Android studio. `https://developer.android.com/studio/test/monkey.html`. [Online; accessed 2016-08-15].

[13] S. V. Adve and M. D. Hill. Weak ordering—a new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, pages 2–14, New York, NY, USA, 1990. ACM.

[14] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer. Detecting data races on weak memory systems. *SIGARCH Comput. Archit. News*, 19(3):234–243, Apr. 1991.

[15] P. Bielik, V. Raychev, and M. Vechev. Scalable race detection for android applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 332–348, New York, NY, USA, 2015. ACM.

[16] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs. In *ASPLOS*, pages 167–178, 2010.

[17] F. Chen and G. Roşu. *Parametric and Sliced Causality*, pages 240–253. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[18] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. *SIGPLAN Not.*, 37(5):258–269, May 2002.

[19] M. Christiaens and K. Bosschere. *Euro-Par 2001 Parallel Processing: 7th International Euro-Par Conference Manchester, UK, August 28–31, 2001 Proceedings*, chapter Accordion Clocks: Logical Clocks for Data Race Detection, pages 494–503. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[20] F. Dabek, N. Zeldovich, F. Kaashoek, D. Mazières, and R. Morris. Event-driven programming for robust software. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, EW 10, pages 186–189, New York, NY, USA, 2002. ACM.

[21] P. Deligiannis, A. F. Donaldson, J. Ketema, A. Lal, and P. Thomson. Asynchronous programming, analysis and testing with state machines. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 154–164, New York, NY, USA, 2015. ACM.

[22] A. Desai, V. Gupta, E. K. Jackson, S. Qadeer, S. K. Rajamani, and D. Zufferey. P: safe asynchronous event-driven programming. In *PLDI*, pages 321–332, 2013.

[23] D. Dimitrov, V. Raychev, M. Vechev, and E. Koskinen. Commutativity race detection. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 305–315, New York, NY, USA, 2014. ACM.

[24] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles &Amp; Practice of Parallel Programming*, PPOPP '90, pages 1–10, New York, NY, USA, 1990. ACM.

[25] D. R. Engler and K. Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *SOSP*, pages 237–252, 2003.

[26] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1âĂŞ3):35 – 45, 2007. Special issue on Experimental Software and Toolkits.

[27] S. Ferg. Event-driven programming: Introduction, tutorial, history. `http://http://eventdrivenpgm.sourceforge.net/`, 2006. [Online; accessed 2017-01-25].

[28] C. Flanagan and S. N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *PLDI*, pages 121–133, 2009.

[29] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. RERAN: Timing- and touch-sensitive record and replay for android. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 72–81, Piscataway, NJ, USA, 2013. IEEE Press.

[30] P. J. Guo, J. H. Perkins, S. McCamant, and M. D. Ernst. Dynamic inference of abstract types. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ISSTA '06, pages 255–265, New York, NY, USA, 2006. ACM.

[31] C.-H. Hsiao, J. Yu, S. Narayanasamy, Z. Kong, C. L. Pereira, G. A. Pokam, P. M. Chen, and J. Flinn. Race detection for event-driven mobile applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 326–336, New York, NY, USA, 2014. ACM.

[32] Y. Hu, T. Azim, and I. Neamtiu. Versatile yet lightweight record-and-replay for android. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 349–366, New York, NY, USA, 2015. ACM.

[33] R. C. Huang, E. Halberg, and G. E. Suh. Non-race concurrency bug detection through order-sensitive critical sections. In *ISCA*, pages 655–666, 2013.

[34] A. Itzkovitz, A. Schuster, and O. Zeev-Ben-Mordehai. Toward integration of data race detection in DSM systems. *J. Parallel Distrib. Comput.*, 59(2):180–203, Nov. 1999.

[35] H. V. Jagadish. A compression technique to materialize transitive closure. *ACM Trans. Database Syst.*, 15(4):558–598, Dec. 1990.

[36] C. S. Jensen, A. Møller, V. Raychev, D. Dimitrov, and M. Vechev. Stateless model checking of event-driven applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 57–73, New York, NY, USA, 2015. ACM.

[37] V. Kahlon and C. Wang. Universal Causality Graphs: A Precise Happens-Before Model for Detecting Bugs in Concurrent Programs. In *CAV*, pages 434–449, 2010.

[38] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[39] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, 1978.

[40] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on*, C-28(9):690–691, Sept 1979.

[41] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from Mistakes: a Comprehensive Study on Real World Concurrency Bug Characteristics. In *ASPLOS*, pages 329–339, 2008.

[42] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *ASPLOS*, pages 37–48, 2006.

[43] P. Maiya, R. Gupta, A. Kanade, and R. Majumdar. *Partial Order Reduction for Event-Driven Multi-threaded Programs*, pages 680–697. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[44] P. Maiya, A. Kanade, and R. Majumdar. Race detection for android applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 316–325, New York, NY, USA, 2014. ACM.

[45] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1989.

[46] M. Musuvathi and S. Qadeer. Partial-Order Reduction for Context-Bounded State Exploration. Technical Report MSR-TR-2007-12, Microsoft Research, 2007.

[47] M. Musuvathi and S. Qadeer. Fair Stateless Model Checking. In *PLDI*, pages 362–371, 2008.

[48] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. In *OSDI*, pages 267–280, 2008.

[49] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically Classifying Benign and Harmful Data Races Using Replay Analysis. In *PLDI*, pages 22–31, 2007.

[50] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '03, pages 167–178, New York, NY, USA, 2003. ACM.

[51] A. Pathak, Y. C. Hu, and M. Zhang. Bootstrapping energy debugging on smartphones: a first look at energy bugs in mobile devices. In *HotNets*, page 5, 2011.

[52] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff. What is keeping my phone awake? Characterizing and detecting no-sleep energy bugs in smartphone apps. In *MobiSys*, pages 267–280, 2012.

[53] D. Perkovic and P. J. Keleher. Online data-race detection via coherency guarantees. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, pages 47–57, New York, NY, USA, 1996. ACM.

[54] B. Petrov, M. T. Vechev, M. Sridharan, and J. Dolby. Race detection for web applications. In *PLDI*, pages 251–262, 2012.

[55] E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs. *SIGPLAN Not.*, 38(10):179–190, June 2003.

[56] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. AppInsight: mobile app performance monitoring in the wild. In *OSDI*, pages 107–120, 2012.

[57] V. Raychev, M. T. Vechev, and M. Sridharan. Effective race detection for event-driven programs. In *OOPSLA*, pages 151–166, 2013.

[58] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, Nov. 1997.

[59] K. Sen, G. Roşu, and G. Agha. *Detecting Errors in Multithreaded Programs by Generalized Predictive Analysis of Executions*, pages 211–226. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[60] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: Understanding object-sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 17–30, New York, NY, USA, 2011. ACM.

[61] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan. Sound predictive race detection in polynomial time. In *POPL*, pages 387–400, 2012.

[62] J. W. Voung, R. Jhala, and S. Lerner. RELAY: Static Race Detection on Millions of Lines of Code. In *ESEC/SIGSOFT FSE*, pages 205–214, 2007.

[63] W3C. HTML5: A vocabulary and associated APIs for HTML and XHTML. `https://www.w3.org/TR/html5/webappapis.html#processing-model-3`, 2014. [Online; accessed 2017-01-27].

[64] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: efficient detection of data race conditions via adaptive tracking. *SIGOPS Oper. Syst. Rev.*, 39(5):221–234, Oct. 2005.

[65] L. Zhang, D. R. Bild, R. P. Dick, Z. M. Mao, and P. A. Dinda. Panappticon: Event-based tracing to measure mobile application and platform performance. In *CODES+ISSS*, pages 1–10, 2013.

[66] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. W. Reps. ConSeq: Detecting Concurrency Bugs through Sequential Errors. In *ASPLOS*, pages 251–264, 2011.

[67] W. Zhang, C. Sun, J. Lim, S. Lu, and T. W. Reps. ConMem: Detecting crash-triggering concurrency bugs through an effect-oriented approach. *ACM Trans. Softw. Eng. Methodol.*, 22(2):10, 2013.