BARIS KASIKCI

TECHNIQUES FOR DETECTION, ROOT CAUSE DIAGNOSIS AND CLASSIFICATION OF IN-PRODUCTION CONCURRENCY BUGS

TECHNIQUES FOR DETECTION, ROOT CAUSE DIAGNOSIS AND CLASSIFICATION OF IN-PRODUCTION CONCURRENCY BUGS

BARIS KASIKCI

Baris Kasikci: *Techniques for Detection, Root Cause Diagnosis and Classification of In-Production Concurrency Bugs*

Everybody who learns concurrency thinks they understand it, ends up finding mysterious races they thought weren't possible, and discovers that they didn't actually understand it yet after all.

— Herb Sutter

Any fool can know. The point is to understand.

— Albert Einstein

ABSTRACT

Concurrency bugs are at the heart of some of the worst bugs that plague in-production software. Concurrency bugs stymie software development by remaining elusive for weeks or even months before developers can identify and fix them. Consequently, concurrency bugs hinder developers' efforts for building new systems and programs with a faster pace than what is possible today.

Detection, root cause diagnosis, and classification of concurrency bugs is especially challenging for in-production software. Concurrency bugs are hard to detect with low overhead, because common detection mechanisms require heavyweight analyses, and therefore they are not suited for in-production usage. Even when detected, it is hard to determine which concurrency bugs are more likely to cause program failures, because such consequences arise as a result of complicated interactions of program inputs, thread schedules, and the underlying memory model. Exploring such interactions require intensive analyses, which are also not suited for in-production software. Even when concurrency bugs that are likely to lead to failures are detected, developers still need to come up with an explanation of how they caused a given failure. This requires gathering a significant amount of runtime information from in-production program executions and performing further analyses, which is hard to do without undue runtime performance overhead.

This dissertation develops techniques for effective and efficient detection, root cause diagnosis, and classification of concurrency bugs for in-production software. The high level goal of this dissertation is to provide developers with techniques that will allow them to better reason about concurrent programs. This dissertation builds upon the following fundamental tenets:

- It adopts a problem solving approach that spans multiple layers of the system stack. This allows the dissertation to counter the problems related to concurrency, which is ubiquitous in many layers of the system stack from parallel hardware to language memory models to operating system schedulers and multithreaded and multiprocess applications.
- It decouples its solution techniques in order to perform most of the heavyweight analyses in-house and resorts to minimal in-production analysis. This separation of concerns allows this dissertation to strike an appropriate balance between static program analysis and dynamic program analysis.
- It eschews relying on technologies that are not widely deployed, by limiting the scope of the technologies it leverages to what is

available today. This frugal approach pushes the limits of the dissertation to come up with novel techniques that are capable of solving the outlined problems without resorting to custom solutions that may be infeasible to implement in the real world.

- It employs techniques that sometimes oppose conventional wisdom. This heretical approach allows the dissertation to overcome key open problems.

Using the aforementioned key tenets, this dissertation proposes:

1. Techniques to automatically detect concurrency bugs (data races and atomicity violations) in-production by combining in-house static analysis and in-production dynamic analysis.
2. A technique to automatically identify the root causes of in-production failures, with a particular emphasis on failures caused by concurrency bugs.
3. A technique that given a data race, automatically classifies it based on its potential consequence, allowing developers to answer questions regarding the data race such as "can the data race cause a crash or a hang?", "does the data race have any observable effect?".

We build a toolchain that implements all the aforementioned techniques. We show that the tools we develop in this dissertation are effective, incur low runtime performance overhead, have high accuracy and precision.

Keywords: Concurrency bugs, data race, atomicity violation, static analysis, dynamic analysis

RÉSUMÉ

Les bogues de concurrences sont au cœur des pires problèmes que rencontrent les programmes en production. Les bogues de concurrence narguent les développeurs de logiciels en restant élusifs pendant des semaines voir des mois avant que les développeurs ne les trouvent et puissent les fixer. Les bogues de concurrence empêchent ainsi les développeurs de créer de nouveaux systémes et programmes à un rythme plus èlevé que ce qui est possible actuellement.

La détection, le diagnostic de la cause d'un bogue de concurrence et sa classification sont un défi spécialement compliqué pour les logiciels en production. Ces bogues sont difficiles á détecter á faible coût, les mécanismes usuels de détection requérant de lourdes analyses dynamiques, ce qui les rend inutilisables en production. Même une fois détectés, il est compliqué de déterminer quel problème de concurrence a la plus forte probabilité de causer une défaillance du programme, car de telles défaillances sont le résultats d'interactions complexes entre les entrées du programme, l'ordonnancement des processus, et le modèle de mémoire sous-jacent. Explorer de telles interactions demande des analyses demandant beaucoup de temps de calculs, ce qui n'est pas non plus utilisable pour des programmes en production. Même lorsque les bogues de concurrences qui peuvent probablement causer une défaillance du programme sont détectés, les développeurs ont encore besoin de trouver une explication de comment une telle défaillance a été causée. Cela demande de récolter un nombre important d'informations sur l'exécution du programme en production et d'exécuter de plus amples analyses, ce qui est difficile à faire sans impacter les performances à l'exécution.

Cette thèse développe des techniques pour une détection efficace et économique des bogues de concurrence pour des logiciels en production ainsi qu'une analyse de la cause de ceux-ci et leur classification. L'idée générale de cette thèse est de procurer au développeur des techniques qui lui permettront de mieux raisonner et comprendre le comportement des programmes concurrents. Cette thèse se base sur les principes suivant :

- Réduire l'impact des analyses en production au minimum : en décomposant sa solution en deux parties distincte, les analyses les plus lourdes en interne, et un minimum d'analyses en production, cette thèse permet d'obtenir une balance appropriée entre les analyses statiques et dynamiques des programmes.
- Eviter d'utiliser des techniques qui ne sont pas largement déployées, en limitant la portée des technologies qu'elle utilise à celles qui sont disponibles aujourd'hui. Cette approche frugale

pousse cette thèse à développer de nouvelles techniques capable de résoudre les problèmes présentés sans utiliser de solutions sur mesure qui pourrait rendre impossible leur implémentation.

- Utiliser des techniques qui vont parfois à l'encontre de la sagesse populaire. Cette approche hérétique permet de surpasser certains problèmes non résolus autrement.

Utilisant les principes ci-dessus, cette dissertation propose:

1. Des techniques pour détecter des bogues de concurrence en production, en combinant des analyses statiques en interne et une analyse dynamique en production.
2. Une technique pour identifier la cause d'un bogue en production.
3. Une technique qui, pour un accès concurrent, la classifie selon ses conséquences potentielles, permettant aux développeurs de répondre à certaines questions comme "Cet accès concurrent cause-t-il un plantage, ou bloque-t-il le programme ?", "Est-ce que cet accès concurrent peut-il changer la sortie du programme ?", "Est ce que cet accès concurrent n'a aucun effet observable ?".

Nous avons construit une série d'outils qui implémentent toutes les techniques mentionnées ci-dessus. Nous montrons que ces outils que nous avons développés dans cette thèse sont efficace, ont un impact faible sur les performances des logiciels et ont une haute précision.

Mots clés: bogues de concurrences, accès concurrent, violation d'atomicité, analyse statique, analyse dynamique

PUBLICATIONS

This dissertation primarily builds upon the ideas presented in the following publications:

- — Baris Kasikci et al. "Failure Sketching: A Technique for Automated Root Cause Diagnosis of In-Production Failures." In: *Symp. on Operating Systems Principles*. 2015
- — Baris Kasikci et al. "Failure Sketches: A Better Way to Debug." In: *Workshop on Hot Topics in Operating Systems*. 2015
- — Cristian Zamfir et al. "Automated Debugging for Arbitrarily Long Executions." In: *Workshop on Hot Topics in Operating Systems*. 2013
- — Baris Kasikci, Cristian Zamfir, and George Candea. "RaceMob: Crowdsourced Data Race Detection." In: *Symp. on Operating Systems Principles*. 2013
- — Baris Kasikci, Cristian Zamfir, and George Candea. "Data Races vs. Data Race Bugs: Telling the Difference with Portend." In: *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*. 2012
- — Baris Kasikci, Cristian Zamfir, and George Candea. "Automated Classification of Data Races Under Both Strong and Weak Memory Models." In: *ACM Transactions on Programming Languages and Systems* 37.3 (2015)
- — Baris Kasikci et al. "Efficient Tracing of Cold Code Via Bias-Free Sampling." In: *USENIX Annual Technical Conf.* 2014

ACKNOWLEDGMENTS

As I set out to write these acknowledgements, I realized that many people contributed to the formation of this dissertation. I would like to apologize upfront if I forgot to mention the names of people who helped my throughout my PhD; it is an honest mistake.

First and foremost, I would like to thank my advisor George Candea who taught me how to do research that matters. His unwavering perfectionism is omnipresent in this dissertation. To me, George is the embodiment of the ideal advisor: he is resourceful, he always follows-up, and he always supports his students. I try to follow his example with the students I work with.

I would like to thank Emery Berger, Christos Kozyrakis, Madan Musuvathi, and Willy Zwaenepoel for being in my dissertation committee. It is an honor to get the input and criticism of such world-class experts. They helped me greatly improve this dissertation.

I would like to thank all the members and alumni of DSLAB, who have been an amazing company over the past five years. I learned a lot from this amazingly talented and smart team. This dissertation benefited greatly from the honest and unbiased feedback of DSLAB. Many thanks go to Silviu Andrica, Radu Banabic, Alexandre Bique, Stefan Bucur, Amer Chamseddine, Vitaly Chipounov, Alexandru Copot, Francesco Fucci, Loïc Gardiol, Horatiu Jula, Johannes Kinder, Vova Kuznetsov, Georg Schmid, Benjamin Schubert, Ana Sima, Jonas Wagner, Cristian Zamfir, Peter Zankov, Arseniy Zaostrovnykh, and Lisa Zhou. I would like to especially thank Nicoletta Isaac, DSLAB's administrative assistant, for making my life at EPFL easier.

I would like to thank Babak Falsafi and Edouard Bugnion for all the advice and the support they gave me. They are great people and mentors, and I am very lucky to have met them during my PhD. I would also like to thank Nisheeth Vishnoi, for all the information he gave me about life in academia.

I would like to thank my mentors and colleagues at Microsoft Research, VMware and Intel. Many ideas in this dissertation developed from lengthy discussions with them. In particular, many thanks go to Thomas Ball, John Erickson, Chandra Prasad, Wolfram Schulte, and Danny van Velzen of Microsoft; Eric von Bayer, Dilpreet Bindra, Swathi Koundinya, Hiep Ma of VMware; Mohammad Haghighat, Cristiano Pereira, and Gilles Pokam of Intel.

I would like to thank Kerem Kapucu, Eren Can Erdoğan, and Hakan Ertuna for their great friendship. I would like to also thank other close friends Duygu Ceylan, Roy Combe, Berra Erkoşar, Cansu

Kaynak, Onur Kazanç, Onur Koçberber, Cüneyt Songüler, and Pınar Tözün.

I would like to thank Vikram Adve, Gustavo Alonso, Katerina Argyraki, Olivier Crameri, Sotiria Fytraki, Christopher Ming-Yee Iu, Ryan Johnson, James Larus, Petros Maniatis, Yanlei Zhao, and all the anonymous reviewers of my work whose comments helped greatly improve this dissertation.

I would like to thank VMware, Intel and the European Research Council for supporting my research.

Last but not least, I would like to thank my family: Yildiz, Baris, and Tia. They provided their continuous love and support throughout my PhD. Without their company, this dissertation wouldn't have been possible.

CONTENTS

---

i SETTING THE STAGE 1

1 INTRODUCTION 3
  1.1 Problem Definition 3
  1.2 Challenges 5
    1.2.1 The Runtime Performance Overhead Challenge 5
    1.2.2 The Accuracy Challenge 6
    1.2.3 The In-Production Challenge 8
  1.3 Overview of Prior Solution Attempts 8
    1.3.1 Attempts at Addressing the Overhead Challenge 9
    1.3.2 Attempts at Addressing the Accuracy Challenge 9
    1.3.3 Attempts at Addressing the In-Production Challenge 10
  1.4 Solution Overview 10
  1.5 Summary of Contributions 11
  1.6 Summary of Results 13

2 BACKGROUND AND RELATED WORK 15
  2.1 Definitions 15
    2.1.1 Data Race 15
    2.1.2 Atomicity Violation 16
    2.1.3 Root Cause 17
  2.2 Concurrency Bug Surveys 18
  2.3 Data Race Detection Literature 19
    2.3.1 Static Data Race Detection 19
    2.3.2 Dynamic Data Race Detection 19
    2.3.3 Mixed Static-Dynamic Data Race Detection 22
    2.3.4 Detecting Data Races In Production 22
    2.3.5 Data Race Avoidance 22
  2.4 Atomicity Violation Detection 24
    2.4.1 Static Atomicity Violation Detection 24
    2.4.2 Dynamic Atomicity Violation Detection 24
  2.5 Root Cause Diagnosis of In Production Failures 25
  2.6 Concurrency Bug Classification 28

ii ELIMINATING CONCURRENCY BUGS FROM IN-PRODUCTION SYSTEMS 31

3 RACEMOB: DETECTING DATA RACES IN PRODUCTION 33
  3.1 RaceMob's Design Overview 33
  3.2 Static Data Race Detection 34
  3.3 Dynamic Data Race Validation 35
    3.3.1 Dynamic Context Inference 35
    3.3.2 On-Demand Data Race Detection 36
    3.3.3 Schedule Steering 38

- 3.4 Crowdsourcing the Validation 38
- 3.5 Reaching a Verdict 40
- 3.6 RaceMob's Implementation Details 41

4 GIST: ROOT CAUSE DIAGNOSIS OF IN-PRODUCTION FAILURES 43
- 4.1 Gist's Design Overview 44
- 4.2 Static Slice Computation 46
- 4.3 Slice Refinement 48
  - 4.3.1 Adaptive Slice Tracking 49
  - 4.3.2 Tracking Control Flow 50
  - 4.3.3 Tracking Data Flow 51
- 4.4 Identifying the Root Cause 53
- 4.5 Gist's Implementation Details 55

5 PORTEND: CLASSIFYING DATA RACES DURING TESTING 57
- 5.1 A Fine-Grained Way to Classify Data Races 59
- 5.2 Portend's Design Overview 61
- 5.3 Single-Path Analysis 67
- 5.4 Multi-Path Analysis 69
- 5.5 Symbolic Output Comparison 70
- 5.6 Multi-Schedule Analysis 72
- 5.7 Symbolic Memory Consistency Modeling 74
- 5.8 Classification Verdicts 79
- 5.9 Portend's Debugging Aid Output 81
- 5.10 Portend's Implementation Details 81

6 EVALUATION 85
- 6.1 RaceMob's Evaluation 85
  - 6.1.1 Experimental Setup 85
  - 6.1.2 Effectiveness 87
  - 6.1.3 Efficiency 88
  - 6.1.4 Comparison to Other Detectors 90
  - 6.1.5 Comparison to Concurrency Testing Tools 93
  - 6.1.6 Scalability with Application Threads 96
- 6.2 Gist's Evaluation 97
  - 6.2.1 Experimental Setup 97
  - 6.2.2 Automated Generation of Sketches 98
  - 6.2.3 Accuracy of Failure Sketches 101
  - 6.2.4 Efficiency 103
- 6.3 Portend's Evaluation 105
  - 6.3.1 Experimental Setup 106
  - 6.3.2 Effectiveness 107
  - 6.3.3 Accuracy and Precision 109
  - 6.3.4 Efficiency 111
  - 6.3.5 Comparison to Existing Data Race Detectors 112
  - 6.3.6 Efficiency and Effectiveness of Symbolic Memory Consistency Modeling 115

6.3.7 Memory Consumption of Symbolic Memory Consistency Modeling 117

iii WRAPPING UP 121

7 ONGOING AND FUTURE WORK 123
7.1 Enhancing Security through Path Profiling 123
7.2 Privacy Implications of Collaborative Approaches 124
7.3 Exposing Concurrency Bugs 124
7.4 Concurrency In Large-Scale Distributed Systems 125
8 CONCLUSIONS 127

BIBLIOGRAPHY 129

LIST OF FIGURES

Figure 1 Example of a switch statement adapted from [22] 4

Figure 2 False negatives in happens-before (HB) dynamic race detectors: the data race on x is not detected in Execution 1, but it is detected in Execution 2. 8

Figure 3 Two executions from the same program without a data race. Execution 1 has a race condition, because the program's specification defines executions where x is set to 2 in T_2 after it is set to 1 in T_1 as erroneous. 16

Figure 4 Two executions from different programs. Both executions violate the atomicity requirement of writing to x and reading from it atomically in T_1. Execution 1 has data races, whereas execution 2 does not have any data races. 17

Figure 5 RaceMob's crowdsourced architecture: A static detection phase, run on the hive, is followed by a dynamic validation phase on users' machines. 34

Figure 6 Minimal monitoring in DCI: For this example, DCI stops tracking synchronization operations as soon as each thread goes once through the barrier. 37

Figure 7 The state machine used by the hive to reach verdicts based on reports from program instances. Transition edges are labeled with validation results that arrive from instrumented program instances; states are labeled with RaceMob's verdict. 41

Figure 8 The failure sketch of pbzip2 bug. 44

Figure 9 The architecture of Gist 45

Figure 10 Adaptive slice tracking in Gist 49

Figure 11 Example of control (a) and data (b) flow tracking in Gist. Solid horizontal lines are program statements, circles are basic blocks. 52

Figure 12 Four common atomicity violation patterns (RWR, WWR, RWW, WRW). Adapted from [8]. 53

Figure 13 A sample execution failing at the second read in T_1 (a), and three potential concurrency errors: a RWR atomicity violation (b), 2 WR data races (c-d). 54

Figure 14 Portend taxonomy of data races. 60
Figure 15 High-level architecture of Portend. The six shaded boxes indicate new code written for Portend, whereas clear boxes represent reused code from KLEE [31] and Cloud9 [29]. 62
Figure 16 Increasing levels of completeness in terms of paths and schedules: [a. single-pre/single-post] $\ll$ [b. single-pre/multi-post] $\ll$ [c. multi-pre/multi-post]. 64
Figure 17 Simplified example of a harmful data race from Ctrace [132] that would be classified as harmless by classic data race classifiers. 65
Figure 18 Portend prunes paths during symbolic execution. 70
Figure 19 A program to illustrate the benefits of symbolic output comparison 73
Figure 20 Simple multithreaded program 75
Figure 21 Lamport clocks and a happens-before graph 77
Figure 22 Write Buffering 79
Figure 23 Example debugging aid report for Portend. 81
Figure 24 Breakdown of average overhead into instrumentation-induced overhead and detection-induced overhead. 89
Figure 25 Contribution of each technique to lowering the aggregate overhead of RaceMob. Dynamic detection represents detection with TSAN. RaceMob without DCI and on-demand detection just uses static data race detection to prune the number of accesses to monitor. 92
Figure 26 Concurrency testing benchmarks: bench$_1$ is shown in Fig. 2, thus not repeated here. In bench$_2$, the accesses to x in thread T_1 and T_3 can race, but the long sleep in T_3 and T_4 causes the signal-wait and lock-unlock pairs to induce a happens-before edge between T_1 and T_4. bench$_3$ has a similar situation to bench$_2$. In bench$_4$, the accesses to variables x, y, z from T_1 and T_2 are racing if the input is either in_1, in_2, or in_3. 95
Figure 27 Data race detection coverage for RaceMob vs. RaceFuzzer. To do as well as RaceMob, RaceFuzzer must have a priori access to all test cases (the RaceFuzzer$_3$ curve). 96
Figure 28 RaceMob scalability: Induced overhead as a function of the number of application threads. 97
Figure 29 The failure sketch of Curl bug #965. 98

Figure 30 The failure sketch of Apache bug #21287. The grayed-out components are not part of the ideal failure sketch, but they appear in the sketch that Gist automatically computes. 100
Figure 31 Accuracy of Gist, broken down into relevance accuracy and ordering accuracy. 102
Figure 32 Contribution of various techniques to Gist's accuracy. 102
Figure 33 Gist's average runtime performance overhead across all runs as a function of tracked slice size. 104
Figure 34 Tradeoff between slice size and the resulting accuracy and latency. Accuracy is in percentage, latency is in the number of failure recurrences. 104
Figure 35 Comparison of the full tracing overheads of Mozilla rr and Intel PT. 105
Figure 36 Breakdown of the contribution of each technique toward Portend's accuracy. We start from single-path analysis and enable one by one the other techniques: ad-hoc synchronization detection, multi-path analysis, and finally multi-schedule analysis. 110
Figure 37 Simplified examples for each data race class from real systems. (a) and (b) are from ctrace, (c) is from memcached and (d) is from pbzip2. The arrows indicate the pair of racing accesses. 111
Figure 38 Change in classification time with respect to number of preemptions and number of dependent branches for some of the data races in Table 9. Each sample point is labeled with data race id. 113
Figure 39 Portend's accuracy with increasing values of k. 113
Figure 40 Micro-benchmarks used for evaluating SMCM. 116
Figure 41 Running time of Portend-weak and Portend-seq 118

Figure 42 Memory usage of Portend-weak and Portend-seq 118

LIST OF TABLES

Table 1 Data race detection with RaceMob. The static phase reports *Data race candidates* (row 2). The dynamic phase reports verdicts (rows 3-10). *Causes hang* and *Causes crash* are data races that caused the program to hang or crash. *Single order* are true data races for which either the primary or the alternate executed (but not both) with no intervening synchronization; *Both orders* are data races for which both executed without intervening synchronization. 86

Table 2 Runtime overhead of data race detection as a percentage of uninstrumented execution. Average overhead is 2.32%, and maximum overhead is 4.54%. 88

Table 3 Data race detection results with RaceMob, ThreadSanitizer (TSAN), and RELAY. Each cell shows the number of reported data races. The data races reported by RaceMob and TSAN are all true data races. The only true data races among the ones detected by RELAY are the ones in the row "RaceMob". To the best of our knowledge, two of the data races that cause a hang in SQLite were not previously reported. 89

Table 4 RaceMob aggregate overhead vs. TSAN's average overhead, relative to uninstrumented execution. RaceMob's aggregate overhead is across all the executions for all users. For TSAN, we report the average overhead of executing all the available test cases. 91

Table 5 RaceMob vs. concurrency testing tools: Ratio of data races detected in each benchmark to the total number of data races in that benchmark. 94

Table 6 Bugs used to evaluate Gist. Bug IDs come from the corresponding official bug database. Source lines of code are measured using sloccount [200]. We report slice and sketch sizes in both source code lines and LLVM instructions. Time is reported in minutes:seconds. 99
Table 7 Programs analyzed with Portend. Source lines of code are measured with the `cloc` utility. 106
Table 8 "Spec violated" data races and their consequences. 107
Table 9 Summary of Portend's classification results. We consider two data races to be distinct if they involve different accesses to shared variables; the same data race may be encountered multiple times during an execution—these two different aspects are captured by the *Distinct data races* and *Data race instances* columns, respectively. Portend uses the stack traces and the program counters of the threads making the racing accesses to identify distinct data races. The last 5 columns classify the distinct data races. The *states same/differ* columns show for how many data races the primary and alternate states were different after the data race, as computed by the Record/Replay Analyzer [142]. 108
Table 10 Portend's classification time for the 93 data races in Table 9. 112
Table 11 Accuracy for each approach and each classification category, applied to the 93 data races in Table 9. "Not-classified" means that an approach cannot perform classification for a particular class. 114
Table 12 Portend's effectiveness in bug finding and state coverage for two memory model configurations: sequential memory consistency mode and Portend's weak memory consistency mode. 117

Part I

SETTING THE STAGE

In this part, we define the problem tackled in this dissertation along with the associated challenges for solving it, and prior solution attempts. We give a brief overview of the solution we propose, followed by a thorough treatment of related work on detection, root cause diagnosis, and classification of concurrency bugs.

1

INTRODUCTION

In this chapter, we elaborate on the definition of the problem addressed in this dissertation (§1.1); we describe the challenges of detection, root cause diagnosis and classification of concurrency bugs for in-production software (§1.2); we summarize prior attempts at solving the problem (§1.3). We then give an overview of the solution we propose in this dissertation (§1.4); we summarize our contributions (§1.5) and finally our results (§1.6).

1.1 PROBLEM DEFINITION

Concurrency bugs such as data races, atomicity violations, and deadlocks are at the root of many software problems [124]. These problems have lead to losses of human lives [115], caused massive material losses [184], and triggered various security vulnerabilities [45, 72]. Perhaps more subtly, concurrency bugs increase the difficulty of reasoning about concurrent programs because of their sporadic occurrence and unpredictable effects.

Concurrency bugs proliferated in modern software after the advent of multicore processors. As hardware became increasingly parallel, developers wrote more programs that tried to leverage such parallelism by relying on concurrency. Since then, multithreading and parallel programming became widespread. Concurrency is desirable for getting more performance out of parallel hardware, but it comes with a cost: concurrent programs are hard to write correctly, and therefore it is easy to make mistakes when writing such programs (e.g., data races).

During the transition to the multicore era (early 2000s), mainstream programming languages were not designed to support concurrent programming natively, which contributed to the proliferation of concurrency bugs. For example, C and C++, which were among the most popular programming languages when this transition happened [187], were specified as single-threaded languages [25], without reference to the semantics of threads.

Concurrency was added to these mainstream languages through libraries (e.g. Pthreads [80] and Windows threads [203]), which added informal constructs that developers could use to restrict access to shared variables (e.g., `pthread_mutex_lock`). These constructs were informal, because they did not change the nature of the C/C++ compilers that were inherently oblivious to concurrency.

```
1  unsigned x;
2  ...
3  if (x < 4) {
4    ... code that doesn't change x ...
5    switch (x) {
6      case 0:
7         ...
8      case 1:
9         ...
10     case 2:
11        ...
12     case 3:
13   }
14 }
```

Figure 1 – Example of a switch statement adapted from [22]

Despite the presence of libraries attempting to add concurrency support to C/C++, associated compilers would generate code as if the programs were single-threaded, thereby occasionally violating the intended semantics of concurrent programs. For instance, consider the program snippet in Fig. 1, where a compiler could compile the program to emit a branch table for the switch statement and omit bounds check for the branch table because it already knows that `x < 4`. If the resulting program loads `x` twice, once on line 3, and once on line 5, and `x` is modified between the two loads by another thread (i.e., there is a data race on `x`), the program may take a wild branch and will most probably crash.

Atomicity violations, data races, and deadlocks can all cause similar subtle behavior and cause software to fail in hard-to-predict circumstances [23]. Moreover, the subtle behavior such bugs complicate reasoning about concurrent programs.

It is challenging to fix concurrency bugs as it is, however if such failures only occur in production, the problem is exacerbated. This is because developers traditionally rely on reproducing failures in order to understand the associated bugs and fix them. However, if such bugs only recur in production and cannot be reproduced in-house, diagnosing the root cause and fixing the bugs is truly hard. In [166], developers noted: *"We don't have tools for the once every 24 hours bugs in a 100 machine cluster."* An informal poll on Quora [160] asked *"What is a coder's worst nightmare,"* and the most popular answer was *"The bug only occurs in production and can't be replicated locally,"*.

Intuitively, a root cause is the real reason behind a failure, we talk about root causes in detail in (§2).

To address these problems, this dissertation introduces techniques for the detection, root cause diagnosis, and classification of concurrency bugs that occur in production. We introduce techniques that are applicable to concurrency bugs that occur both in house and in production. However we focus on concurrency bugs that occur in production, because such bugs present additional challenges as we describe in the next section (§1.2).

1.2 CHALLENGES

Researchers and practitioners have observed that concurrency bugs are hard to detect and fix [69, 91, 98, 99, 166]. In this section, we first explain the fundamental challenges, of the detection, root cause diagnosis, and classification of concurrency bugs, namely the runtime performance overhead challenge (§1.2.1) and the accuracy challenge (§1.2.2). We then elaborate on why performing these tasks is more challenging in production (§1.2.3).

1.2.1 *The Runtime Performance Overhead Challenge*

The runtime tracing that is required for the detection, root cause diagnosis, and classification of concurrency bugs incurs high runtime performance overhead. In this section, we discuss the challenges that arise from runtime overheads of techniques and tools that perform dynamic program analysis, because purely static analysis has no runtime overhead.

Although static analysis tools do not impose runtime overhead, they suffer from false positives, which is related to another challenge we discuss in (§1.2.2)

Dynamic concurrency bug detection, whether it is the detection of data races, atomicity violations, or deadlocks, is expensive. This is because concurrency bug detection requires monitoring memory accesses and synchronization operations, and performing intensive computations at runtime (e.g., building a happens-before relationship [110] graph for data race detection).

For instance, dynamic data race detection needs to monitor many memory accesses and synchronization operations, therefore it incurs high runtime overhead (as high as 200× in industrial-strength tools like Intel Parallel Studio [82]). The lion's share of instrumentation overhead is due to monitoring memory reads and writes, reported to account for as much as 96% of all monitored operations [59].

Similarly, atomicity violation detectors incur high overheads (up to 45× in the case of state-of-the-art detector AVIO-S [123] and up to 65× in the case of SVD [207]). The overhead of atomicity violation detection stems from tracking updates to each monitored memory access and performing the necessary checks for determining whether a given access constitutes an atomicity violation or not.

With regards to the classification of concurrency bugs, prior work mostly focused on data race classification [86, 91, 97, 98, 142, 186], because data race detectors tend to report many data races. The abundance of data races pushes developers to understand which data races have higher impact, in order to prioritize their fixing.

By classification, we mean the classification of true positives (i.e., real bugs). Identification of false positives (i.e., reports that do not correspond to real bugs) is considered separately in this dissertation

Classifying data races according to their potential consequences requires more computationally-intensive analyses than mere data race detection, and therefore imposes significant runtime overhead. In order to classify data races based on their consequences, not only data races need to be detected, but further analyses need to be enabled to monitor data races' effects on the program state and output. Moreover, accurate classification of data races requires exploring multiple program paths and schedules to gain sufficient confidence in the classification results, and this further increases the runtime performance overhead. For example, a state of the art data race classification tool, Record/Replay analyzer [142], incurs 45× runtime overhead when performing data race classification.

Finally, root cause diagnosis of concurrency bugs requires tracking memory accesses and certain relations among memory accesses (e.g., their execution order), and therefore incurs large runtime overhead.

The overhead of root cause diagnosis of concurrency bugs is further exacerbated because root cause diagnosis techniques typically require gathering execution information from multiple program executions in order to isolate the failing thread schedules and inputs [127]. For example the *Delta Debugging* technique [40, 216]—a state of the art technique for isolating bug inducing inputs and thread schedules—requires gathering execution information from several dozens (50 to 100) of runs before homing in on bugs' root causes. Another state of the art concurrency bug isolation technique CBI [89] also relies on gathering execution information from multiple inputs, and it incurs overheads as high as 460×.

1.2.2 *The Accuracy Challenge*

Static detection of concurrency bugs works without actually running programs, therefore it does not incur any runtime performance overhead, however this comes at the expense of false positives (i.e., bug reports that do not correspond to actual bugs). False positives arise, because static analysis does not reason about the program's full runtime execution context.

False positives in static analysis of concurrency bugs arise because of four main reasons: first, static detectors perform some approximations such as conflating program paths during analysis or constraining the analysis to be intraprocedural (as opposed to interprocedural) in order to scale to large code bases. Second, static analyzers cannot always accurately infer which program contexts are multithreaded. Third, static analyzers typically model the semantics of lock/unlock synchronization primitives but not other primitives, such as barriers, semaphores, or wait/notify constructs. Finally, static analyzers cannot accurately determine whether two memory accesses alias or not.

Static classification of concurrency bugs typically relies on heuristics, and therefore inherently has false positives as is the case with most heuristic-based approaches. For instance, DataCollider [91] prunes data race reports that appear to correspond to updates of statistics counters and to read-write conflicts involving different bits of the same memory word, or that involve variables known to developers to have intentional data races (e.g., a "current time" variable is read by many threads while being updated by the timer interrupt). Updates on a statistics counter might be considered harmless for the cases investigated by DataCollider, but if a counter gathers critical statistics related to resource consumption in a language runtime, classifying a race on such a counter as harmless may be incorrect. More importantly, even data races that developers consider harmless may become harmful (e.g., cause a crash or a hang) when the code is compiled with a different compiler or when the program executes on some hardware with a different memory model [23, 26].

Dynamic detectors and classifiers [75, 82, 174] tend to report fewer false positives. Developers prefer tools that have fewer false positives, because they do not have the time to cherry-pick true positives (i.e., reports corresponding to real bugs) in the presence of false positives [17].

Dynamic root cause diagnosis techniques [7, 8, 89, 102, 118, 168, 190] typically rely on statistical analysis for isolating the root causes of bugs, and therefore they are susceptible to false positives. These techniques gather execution information from multiple failing and successful executions to determine the key differences between those executions. The accuracy of statistical analysis hinges on the number of samples gleaned, therefore dynamic root cause diagnosis techniques can have false positives if they cannot monitor a sufficiently large sample of executions.

We are not aware of static root cause diagnosis schemes. However, we cautiously speculate that static root cause diagnosis will suffer similarly from false positives.

On the other hand of the spectrum are false negatives (i.e., real bug reports that are missed). False negatives can be an artifact of the approximations used in static analysis, or they may occur because a certain analysis (static or dynamic) is unable to analyze a certain portion of the code.

False negatives are typical of dynamic detection, root cause diagnosis, and classification of concurrency bugs, because dynamic analysis can only operate on executions it witnesses, which is typically only a tiny subset of a program's possible executions.

Somewhat an exception to this is data race detection using causal precedence [178], which can accurately predict some data races that do not occur during actual program executions

False negatives also arise because of fortuitous events. For example, while monitoring a subset of executions, dynamic data race detectors may incorrectly infer happens-before relationships that are mere artifacts of the witnessed thread interleaving. To illustrate this point, consider Fig. 2. In execution 1, the accesses to the shared variable x are ordered by an accidental happens-before relationship (due to a fortuitous ordering of the acquire/release order of locks) that masks the true data race. Therefore, a precise dynamic detector would not flag this as a data race. However, this program does have a data race, which manifests itself under a different thread schedule. This is shown in execution 2, where there is no happens-before relationship between accesses to x; a precise dynamic detector would have reported a data race only if it witnessed this latter thread schedule.

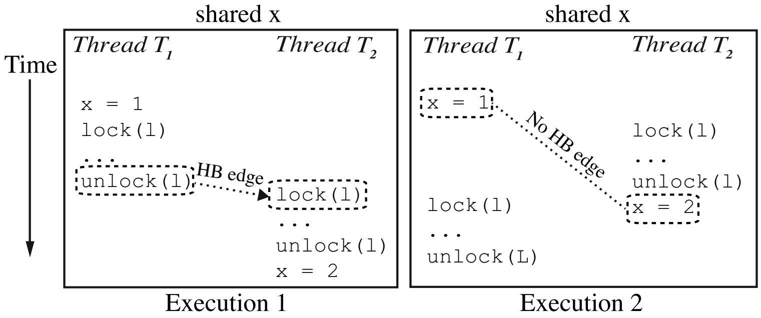


Figure 2 – False negatives in happens-before (HB) dynamic race detectors: the data race on x is not detected in Execution 1, but it is detected in Execution 2.

1.2.3 *The In-Production Challenge*

Any in-production detection, classification, and root cause diagnosis technique needs to incur very low performance overhead and minimally perturb real-user executions. The overhead challenge (§1.2.1) is exacerbated in production, because users will not tolerate performance degradation—even if it comes with increased reliability. Solutions that perturb the actual behavior of production runs nondeterministically may mask the bug frequently but not always, and thus make it harder to detect the bug and remove the potential for (even occasional) failure [137].

Moreover, a great challenge is posed by bugs that only recur in production and cannot be reproduced in-house. The ability to reproduce failures is essential for detecting, classifying and diagnosing the root causes of bugs. A recent study at Google [166] revealed that developers' ability to reproduce bugs is crucial to fixing them. However, in practice, it is not always possible to reproduce bugs, and practitioners report that it takes weeks to fix hard-to-reproduce concurrency bugs [69].

To summarize this section, detection, classification and root cause diagnosis of concurrency bugs pose significant challenges. In particular, it is hard to efficiently and accurately performs these tasks. These challenges are further exacerbated in production.

1.3 OVERVIEW OF PRIOR SOLUTION ATTEMPTS

In this section, we present an overview of prior attempts at addressing the challenges of detection, root cause diagnosis and classification of concurrency bugs. We briefly summarize how existing techniques attempts to address an aforementioned challenges We also elaborate on the shortcomings of prior attempts.

1.3.1 *Attempts at Addressing the Overhead Challenge*

In order to reduce the runtime performance overhead, certain dynamic concurrency bug detectors combine static analysis with dynamic analysis. For example, Goldilocks [51] uses thread escape analysis [140] to reduce the set of memory accesses that needs to be monitored at runtime. A similar approach was proposed earlier by Choi. et al. [41], using a variant of escape analysis. Certain approaches [3, 169] introduce a type system to reduce the overhead of data race and atomicity violation detection. Despite these assisting static analyses, existing concurrency bug detectors still incur overheads that make them impractical for in-production use.

Another way in which existing dynamic detectors and root cause diagnosis techniques attempt to address the overhead challenge is sampling. Sampling-based concurrency bug detection and root cause diagnosis tracks synchronization operations whenever sampling is enabled. For instance, sampling-based data race detectors [27, 130] reduce runtime performance overhead. Although sampling reduces runtime overhead, this comes at the expense of false negatives: since these detectors do not monitor all runtime events, they inevitably miss certain bugs.

Another common way prior work tries to cope with the overhead challenge of detection and root cause diagnosis of concurrency bugs is through the usage of customized hardware. HARD [218] uses special hardware support for data race detection. LBRA/LCRA [7] uses hardware extensions to diagnose root causes of bugs. These techniques indeed alleviate the overhead challenge, alas the hardware support they introduce is not implemented and deployed in the real world.

We haven't come across attempts to overcome the overhead challenge for classification of concurrency bugs.

1.3.2 *Attempts at Addressing the Accuracy Challenge*

In order to deal with false positives, dynamic tools employ filtering, which is typically unsound. Unsound filtering can filter out true positives along with false positives. Although this type of filtering reduces false positives, it cannot eliminate all of them. For example, even after attempting to filter out false data race reports, RacerX still has 37% - 46% false positives.

False negatives in concurrency bug detection can be trivially reduced (or even eliminated) by flagging more bug reports, but this will come at the expense of increased false positives. For instance, a data race detector could report a data race for every pair of memory accesses in a program. This strategy will eliminate all false negatives, but it will likely introduce a lot of false positives. Static concurrency bug detection tools (e.g., RELAY [192]) do not go to such extremes, nevertheless they rely on unsound techniques such as using inaccurate but fast alias analysis to flag as many bugs as possible (i.e., they reduce false negatives), and consequently suffer from false positives (84%).

We explain hybrid data race detectors in detail in §2.3.2.3

We haven't come across attempts to overcome the accuracy challenge for classification of concurrency bugs.

Hybrid data race detectors [146] overcome the false negatives due to fortuitous happens-before relationships by combining two of the primary dynamic data race detection algorithms, namely happened-be-fore based data race detection and lockset-based data race detection. Although this combination reduces false negatives, it can introduce false positives due to the imprecise nature of lockset-based data race detection.

1.3.3 *Attempts at Addressing the In-Production Challenge*

Recall from §1.2.3 that the *in production challenge* exacerbates the *overhead challenge*, therefore prior work used similar methods to cope with the in-production challenge as it did for the overhead challenge. Below, we outline a few of the techniques that prior work used to deal with the in production challenge in addition to the techniques used to cope with the overhead challenge (which we talked about in §1.3.1).

To alleviate the aggravated overhead challenge, prior work employs a variant of sampling. In particular, a common way prior work addresses the in-production challenge is through collaborative approaches like CCI [89] and CBI [118] that rely on monitoring executions at multiple user endpoints. There are two outstanding issues with collaborative approaches: although they reduce runtime overhead per user endpoint for which they perform detection or root cause diagnosis, the reduced overhead is still not suitable (up to 9×) for most in-production environments. Second, because these collaborative approaches sample a subset of the executions—in order to not impose overhead for every execution they monitor—, they may miss rare failures that only recur in production. This happens because sampling further reduces the probability of encountering failures that rarely recur in the first place.

1.4 SOLUTION OVERVIEW

In this section, we present an overview of the solution we propose to the problem we defined in §1.1.

In this dissertation, we address the challenge of in-production detection and root cause diagnosis of concurrency bugs by first employing deep static program analysis offline, and subsequently performing lightweight dynamic analysis online at user endpoints. Static analysis and dynamic analysis work synergistically in a feedback loop: static analysis reduces the overhead of ensuing dynamic analysis and dynamic analysis improves the accuracy of static analysis.

More specifically, with regards to data race detection, our key objective is to have a good data race detector that can be used (1) always-on and (2) in production. This is why we use static analysis to reduce the number of memory accesses that need to be monitored at runtime, thereby reducing overhead by up to two orders of magnitude compared to existing sampling-based techniques. Because we don't rely on sampling an execution during data race detection, our data race detection ends up being more accurate.

We attack the problem of detection of atomicity violations and root cause diagnosis of failures due to concurrency bugs using a technique we call failure sketching. Failure sketching is a technique that automatically produces a high level execution trace called the *failure sketch* that includes the statements that lead to a failure and the differences between the properties of failing and successful program executions. We show in this dissertation that these differences, which are commonly accepted as pointing to root causes [118, 168, 216], indeed point to the root causes of the failures we evaluated (§6). Identifying the root causes of failures also allows detecting the bugs that are associated with those failures.

Addressing the challenge of data race classification requires first addressing the challenge of in-production data race detection, which we do via hybrid static-dynamic analysis as we just mentioned.

We then do the classification entirely offline, because classification is a computationally-expensive process: multiple program paths and schedules need to be explored in order to understand the consequences of a data race, and it is not possible to do such analyses in production without incurring prohibitive runtime performance overheads or utilizing many more resources.

In this dissertation, we do not introduce new hardware mechanisms that would conveniently solve the aforementioned challenges. There are two key reasons for this: (1) not inventing our own custom hardware solution that solves a challenge we are facing, allows us to come up with novel contributions (detailed below in §1.5); (2) the techniques we develop are broadly applicable, because they do not depend on a hardware feature that is not deployed in the real world.

1.5 SUMMARY OF CONTRIBUTIONS

This dissertation introduces **the first data race detector that can both be used always-on in production and provides good accuracy.**

Data race detection with low overhead has been a longstanding problem. Because data race detection is very costly, to our knowledge, prior work has not attempted to explore data race detection in-production. In this dissertation we tackle the problem of in-production data race detection via:

- A two-phase static-dynamic approach for detecting data races in real world software in a way that is more accurate than the state of the art.
- A new algorithm for dynamically detecting data races on-demand, which has lower overhead than state-of-the-art dynamic detectors, including those based on sampling.
- A crowdsourcing framework that, unlike traditional testing, taps directly into real user executions to detect data races.

The second contribution of this dissertation is **failure sketching, a low overhead technique to automatically build failure sketches, which succinctly represent a failure's root cause.**

Root cause diagnosis of in-production failures—especially failures due to concurrency bugs—has long been explored. To our knowledge, there is no prior work that can perform root cause diagnosis of in-production failures without resorting to custom hardware or system state checkpointing infrastructure. In this dissertation, we achieve root cause diagnosis of in-production failures via:

- A hybrid static-dynamic approach that combines in-house static program analysis with in-production collaborative and adaptive dynamic analysis.
- A first and practical demonstration of how Intel Processor Trace, a new technology that started shipping in early 2015 Broadwell processors [83], can be used to perform root cause diagnosis.

The third and final contribution of this dissertation is **a technique to automatically classify data races based on their potential consequences.**

Prior work on data race classification has not been accurate, either because it relied on heuristics or because the abstraction-level of the classification criteria was not correctly identified. In this dissertation, we solve the classification problem via:

- A four-category taxonomy of data races that is finer grain, more precise and, we believe, more useful than what has been employed by the state of the art.
- A technique for predicting the consequences of data races that combines multi-path and multi-schedule analysis with symbolic program-output comparison to achieve high accuracy in consequence prediction, and thus classification of data races according to their severity.
- Symbolic memory consistency modeling, a technique that can be used to model various architectural memory models in a principled way in order to perform data race classification under those memory models.

1.6 SUMMARY OF RESULTS

We built prototypes of all the techniques we present in this dissertation, and we evaluated them. In this section, we give an overview of our evaluation results. Later in §6, we detail these evaluation results.

We evaluated RaceMob, our in-production data race detector on ten different systems, including Apache, SQLite, and Memcached. It found 106 real data races while incurring an average runtime overhead of 2.32% and a maximum overhead of 4.54%. Three of the data races hang SQLite, four data races races crash Pbzip2, and one data race in Aget causes data corruption. Of all the 841 data race candidates found during the static detection phase, RaceMob labeled 77% as likely false positives. Compared to three state-of-the-art data race detectors [27, 174, 192] and two concurrency testing tools [98, 172], RaceMob has lower overhead and better accuracy than all of them.

We evaluated, Gist, our root cause diagnosis prototype using 11 failures from 7 different programs including Apache, SQLite, and Memcached. The Gist prototype managed to automatically build failure sketches, which point developers to failures' root causes, with an average accuracy of 96% for all the failures, while incurring an average performance overhead of 3.74%. On average, Gist incurs 166× less runtime performance overhead than a state-of-the art record/replay system.

We evaluated our data race classification prototype Portend, by applying Portend to 93 data race reports from 7 real-world applications: it classified 99% of the detected data races accurately in less than 5 minutes per race on average. Compared to state-of-the-art data race classifiers, Portend is up to 89% more accurate in predicting the consequences of data races (§6.3.7). This improvement comes from Portend's ability to perform multi-path and multi-thread schedule analysis, as well as Portend's fine grained classification scheme. We found not only that multi-path multi-schedule analysis is critical for high accuracy, but also that the "post-race state comparison" approach used in state-of-the-art classifiers does not work well on real-world programs, despite being perfect on simple microbenchmarks (§6.3.3).

2

BACKGROUND AND RELATED WORK

In this chapter, we first define important terms used in this dissertation (§2.1). Then, we briefly review surveys that examine concurrency bug characteristics (§2.2). Finally, we talk about the literature on the detection (§2.3, §2.4), root cause diagnosis (§2.5), and classification (§2.6) of concurrency bugs. Throughout this chapter, we explain how prior work relates to this dissertation whenever applicable.

2.1 DEFINITIONS

In this section, we give definitions for key concepts used in this dissertation.

2.1.1 *Data Race*

Two memory accesses are conflicting if they access a shared memory location and at least one of the two accesses is a write. A data race occurs when two threads make a conflicting access, and these accesses are not ordered by a *happens-before* relationship [110]— if memory effects of an operation O_1 in a process P_1 becomes visible to to a process P_2 before P_2 performs O_2, we say that O_1 happened before O_2.

A happens-before relationship can only be established using non-ad-hoc synchronization. Ad-hoc synchronization is custom synchronization devised by a developer that relies on loops to synchronize shared variables. By ad-hoc synchronization, we do not refer to custom correct implementations of synchronization constructs, but rather to the incorrect synchronization operations that are widespread in real-world code [206], and that lead to concurrency bugs.

The terms *data race* and *race condition* are often incorrectly used interchangeably. There is a subtle yet important distinction between these terms that has garnered attention from both the academic [92] and the practical [13] community. A data race is a condition, which can be precisely defined as we did above. This precise definition allows the accurate detection of a data race to be automated. A race condition on the other hand is a flaw that occurs in the timing or the ordering of events that leads to erroneous program behavior. It is not always possible to precisely define different types of race conditions, therefore accurate detection of race conditions may not always be possible.

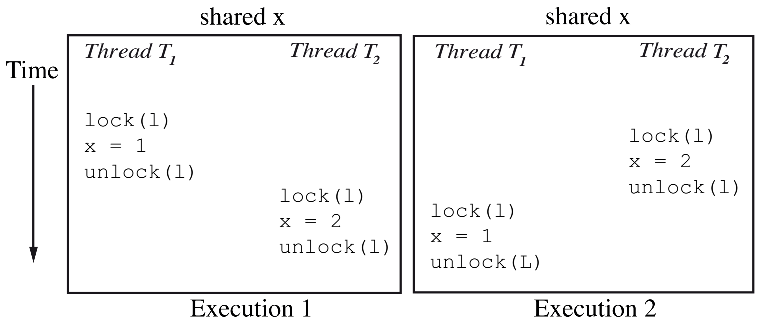


Figure 3 – Two executions from the same program without a data race. Execution 1 has a race condition, because the program's specification defines executions where x is set to 2 in T_2 after it is set to 1 in T_1 as erroneous.

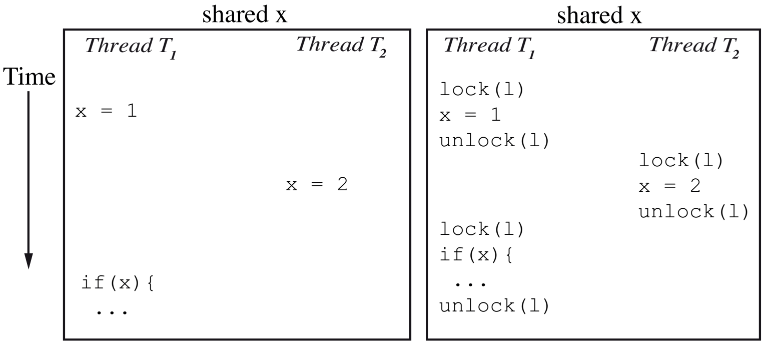
We note that data races and race conditions are neither a necessary nor a sufficient condition for the occurrence of one another. To see why this is the case, consider the example in Fig. 3. In this example, the writes to the shared variable x in threads T_1 and T_2 are protected by locks, therefore they are always happening in some order enforced by the order with which the locks are acquired at runtime (either as in execution 1 or as in execution 2). That is, writes' atomicity cannot be violated; there is always a happens-before relationship between the two writes in any execution. It is not possible to determine which write happened before the other until after the program executes. The reason why there is no fixed ordering between the writes is because locks cannot provide such ordering. If the programs' correctness is compromised, say when the write to x in T_2 is followed by the write to x in T_1 (execution 1), we say that there is a race condition, although technically, there is no data race.

2.1.2 *Atomicity Violation*

Atomicity is a property of a multithreaded program segment that allows the segment to appear as if it occurred instantaneously to the rest of the system. In that regard, atomicity is similar to the linearizability [77] property for concurrent objects and the serializability property of database transactions [147].

An atomicity violation occurs when operations that are supposed to be executed atomically do not, because the operations do not reside in the same critical section. This happens, because developers make incorrect assumptions about which operations should execute atomically and thus fail to enclose such operations in a critical section (e.g., via using locks) [125].

In this dissertation, we define atomicity violations as bugs. In that regard, we do not employ the same definition of atomicity violation as some prior work [61] that treats any violation of serializability as



```
x = 1
                x = 2
 if(x){
  ...
```

```
lock(l)
x = 1
unlock(l)
                lock(l)
                x = 2
                unlock(l)
lock(l)
if(x){
 ...
unlock(l)
```

Execution 1

Execution 2

Figure 4 – Two executions from different programs. Both executions violate the atomicity requirement of writing to x and reading from it atomically in T_1. Execution 1 has data races, whereas execution 2 does not have any data races.

an atomicity violation. While this may be technically true, programs can have many violations of serializability that are not indicative of a bug. In other words in the context of this dissertation, an atomicity violation is a violation of serializability that leads to a failure (e.g., crash or hang).

In that regard, we consider atomicity violations as race conditions. In other words, atomicity violations are flaws in the timing or the ordering of events. It is generally not possible to automatically and accurately detect atomicity violations unless portions of the code that ought to execute atomically is pre-specified.

Pre-specification of atomic regions is unlikely. If atomic regions could be pre-specified, developers could make sure to enclose them in critical sections.

Atomicity violations may or may not involve a data race. Fig. 4 demonstrates this. Let's assume that for both execution 1 and execution 2, the write to x and the read from x in thread T_1 should occur atomically. Note that in execution 1, the accesses in thread 1 are involved in a data race with the access in T_2, whereas in execution 2, there are no data races. However, both executions violate the atomicity requirement of the program.

2.1.3 *Root Cause*

Intuitively, a root cause is the gist of the failure; it is a cause, or a combination of causes, which when removed from the program, prevents the failure associated with the root cause from recurring [201].

A more precise attempt describes a root cause in terms of its relation to a failure. In particular, a failure occurs when a program produces wrong output according to a specification. Then, the root cause of a failure is the negation of the predicate that needs to be enforced so that the execution is constrained to not encounter the failure [215].

Despite the aforementioned precise definition attempt, it is difficult to formally define a failure's root cause. This is because, in general, there are multiple fixes that could fix a given failure. Different developers may choose to eliminate a failure in different ways (e.g., by enforcing different predicates on the execution as per the above definition). In that regard, developers' perception of a root cause may vary.

Due to the difficulty of providing a precise definition, in the context of this dissertation, we resort to a purely statistical definition of a root cause. In particular, we define root causes as events that are primarily correlated with failures. We later show empirically that such events indeed point to root causes that developers end up eliminating in order to fix bugs (§6.2).

2.2 CONCURRENCY BUG SURVEYS

As concurrent programming gained more momentum after the early 2000s, concurrency bugs proliferated, and researchers studied concurrency bugs. One of the first bug studies to include concurrency bugs collected only 12 concurrency bugs [33] from three systems applications, namely MySQL, GNOME and Apache. This study validated the hypothesis that generic recovery techniques such as process pairs [71] can be used to mask concurrency bugs.

The first comprehensive concurrency bug study is due to [122, 124]. This study examined 105 randomly selected bugs from 4 real world systems, namely MySQL, Apache, Mozilla, and OpenOffice. A key finding of this study was that almost all of the non-deadlock concurrency bugs were due to developers' violation of an ordering or atomicity assumption. This key result highlights the importance of developing techniques to detect and fix data races, atomicity violations and order violations in order to deal with concurrency bugs.

Another study conducted at Microsoft [69] further revealed results highlighting the importance of concurrency bugs. %72 of all the participants to this study considered that detecting and fixing concurrency bugs is hard. Participants said that fixing concurrency bugs can take days (63.4%) to weeks (8.3%) to months (0.9%). 66% of participants said that they have to deal with concurrency bugs as part of their daily routine .

A more recent study [167] in the context of root cause diagnosis of bugs determined that although most of the bugs can be reproduced in-production by running the program with the same set of inputs (82%), the remainder of the bugs had non-deterministic behavior. One of the conclusions of this study was that determining fault-triggering inputs for concurrency bugs and reproducing failures due to concurrency bugs is significantly harder than for other bugs.

Although not a survey per se, the common vulnerability and exposure database contains a comprehensive list of common vulnerabilities and exposures (CVEs) related to concurrency bugs [45]. At the time of the writing of this dissertation, common vulnerability database contains 336 vulnerabilities related to concurrency bugs. These CVEs impact kernels (Linux, Windows), browsers (Chrome, Internet Explorer), firewalls (in Cisco IOS) among other software.

2.3 DATA RACE DETECTION LITERATURE

Detection of data races garnered a lot of attention, because data races are one of the most notorious concurrency bugs. They can cause other bugs such as atomicity violations and deadlocks; their occurrence is typically sporadic, and their effects subtle.

Data race detection can be broadly classified into three classes: static data race detection, dynamic data race detection, and mixed static-dynamic data race detection.

2.3.1 *Static Data Race Detection*

Static data race detectors [53, 60, 95, 96, 141, 155, 192, 139, 157, 178, 79] analyze the program source code without executing it. They reason about multiple program paths at once, and thus typically miss few data races (i.e., have a low rate of false negatives) [146]. Static detectors can also run fast, and they can scale to large code bases if they employ necessary approximations (e.g., conflating program paths). The problem is that static data race detectors tend to have many false positives (i.e., produce reports that do not correspond to real data races). For instance, 84% of data races reported by RELAY are not true data races [192]. This can send developers on a wild goose chase, making the use of static detectors potentially frustrating and expensive.

In this dissertation, we use static data race detection to help reduce the overhead of our mixed static-dynamic data race detection technique (§3).

2.3.2 *Dynamic Data Race Detection*

Dynamic data race detectors [48, 75, 82, 170, 174, 27, 41, 59, 99, 130, 47, 144, 156, 165, 50, 52, 91, 153, 154, 211, 138] typically monitor memory accesses and synchronization operations at runtime, and determine if the monitored accesses race with each other. Such detectors can achieve low rates of false positives. Alas, dynamic detectors miss all the data races that are not seen in the directly observed execution (i.e., they have false negatives), and these can be numerous.

Moreover, the instrumentation required to monitor all executed memory accesses makes dynamic detectors incur high runtime overheads (200× for Intel Thread Checker [82], 30× for Google Thread-Sanitizer [174]). As a result, dynamic detectors are not practical for in-production use, rather only during testing—this deprives them of the opportunity to observe real-user executions, thus missing data races that only occur in real-user environments. Some dynamic data race detectors employ sampling [27, 130] to decrease runtime overhead, but this comes with further false negatives. Sampling causes false positives, because most of the time, sampled events are not useful for the purposes of bug detection. For instance the study in [116] found that fewer than 1 in $25,000$ randomly sampled events were indicative of failures, and that over 99.996% of the sampled execution profile was discarded as not being relevant for bug detection. In this dissertation, we rely on static program analysis to make more informed decisions for gathering execution information.

Below, we present a policy-centric classification of dynamic data race detection algorithms. Whenever applicable, we also talk about various mechanisms with which these data race detection policies are implemented.

2.3.2.1 *Happens-Before-Based Data Race Detection*

Happens-before relationship [110] based detectors [75, 97, 98, 171, 174, 27, 41, 59, 99, 130] track the happens-before relationships between memory accesses of a program during execution, and detect data races based on those relationships.

More specifically, if two memory accesses access the same memory location, at least one of the accesses is a write, and there is no happens-before relationship between the two accesses, these detectors flag a data race.

Dynamic detectors that solely use happens-before relationships do not have false positives as long as they are aware of all the synchronization mechanisms employed by the developer. happens-before based dynamic data race detectors can have false positives if developers use custom synchronization primitives to which the detectors are oblivious.

As we discussed in the limitations of prior work (§1.2.2), happens-before-based data race detection is susceptible to false negatives because of fortuitous happens-before edges that get created merely as an artifact of an arbitrary execution. In some other executions, these edges may not get created and happens-before based data race detection could flag a data race. In the next section (§2.3.2.2), we describe another dynamic data race detection algorithm that avoids false positives due to fortuitous happens-before edges.

In this dissertation, we present a hap-pened-before-based data race detection algorithm that aggressively starts and stops tracking happens-before relationships based on the events at runtime in order to reduce the runtime performance overhead (§3).

2.3.2.2 *Lockset-Based Data Race Detection*

Locksets describe the locks held by a program at any given point in the execution.

Lockset-based data race detection [170, 218, 144, 47, 156, 165] checks whether all shared memory accesses follow a consistent *locking discipline*. A locking discipline is a policy that ensures the absence of data races. A trivial locking discipline would require all memory accesses in a program to be protected by the same lock.

The simple locking discipline that Eraser [170], namely the first lockset-based data race detector uses, states that every shared variable access should be protected by some common lock. In other words, Eraser requires any thread accessing a given shared variable to hold a common lock while it is performing the access in order to consider that access as non-racing. If Eraser determines that the program violates this locking discipline at runtime, it will flag a data race.

Eraser's simple locking discipline is overly strict, and as a result, it would cause it to report many false positives. To lower the number of false positives, Eraser employs several refinements. For example, Eraser will not report data races due to the initialization of shared variables, which is frequently done without holding a lock. Eraser employs other similar heuristics to lower false positives, however Eraser cannot completely eliminate false positives. Furthermore, these heuristics can potentially introduce false negatives.

HARD [218] is a hardware implementation of the lockset-based data race detection. HARD uses bloom filters [20] to store locksets and uses bitwise logic operations on the locksets. HARD was able to detect 54 data races out of 60 randomly-injected data races in six SPLASH-2 applications (20% more than happens-before based data race detection) with overheads ranging between 0.1% to 2.6%.

In the next section (§2.3.2.3), we present a data race detection technique that reduces the false positive rates of lockset based data race detection.

2.3.2.3 *Hybrid Data Race Detection*

Perhaps unfortunately, in the data race detection literature, hybrid data race detection implies data race detection that combines the two major dynamic data race detection algorithms, namely happens-before-based data race detection and lockset-based data race detection. This is because the first piece of work that called a data race detection algorithm "hybrid" [146] combined these two dynamic data race detection algorithms.

That is as opposed to calling the combination of static and dynamic data race detection as hybrid.

Hybrid data race detection works in two stages: the hybrid data race detector has an always-on lockset-based data race detector, which when flags a potential data race, verifies whether the potential data race is indeed a true data race or not by using a happens-before data race detector.

Hybrid data race detection improves the accuracy of data race detection by reporting fewer false positives than lockset-based data race detection [146].

2.3.3 *Mixed Static-Dynamic Data Race Detection*

Some data race detectors combine static analysis and dynamic analysis in order to reduce the runtime overhead of data race detection.

Goldilocks [51] and the mixed static-dynamic detector from [41] used a static thread escape analysis phase to eliminate the need to track thread-local variables. This dissertation takes a similar approach to these techniques, but uses a complete static data race detector—which is more accurate than just using thread escape analysis—to detect *most* data races (i.e., to have few false negatives). It then uses a novel dynamic data race detection algorithm to achieve lower runtime overhead and higher accuracy than existing mixed-static dynamic data race detectors.

2.3.4 *Detecting Data Races In Production*

To our knowledge, this dissertation and the RaceMob system is the first to explore always-on in-production detection of data races.

A recent system called Litecollider [18] also explores in-production detection of data races. LiteCollider has a two stage in-house data race detection scheme (as opposed to a single stage in-house static analysis in RaceMob): first LiteCollider detects data races statically and then uses *in-house* alias analysis and lockset-based data race detection to further prune the set of potential data races to detect at production time. Then, in production, LiteCollider uses collision analysis (similar to DataCollider [91]) to detect data races. Although LiteCollider's in-house dynamic analysis reduces the number of candidate data races to dynamically detect in production, it also introduces false negatives[18].

Collision analysis uses various techniques to try to make conflicting accesses occur simultaneously. More details can be found in [91].

2.3.5 *Data Race Avoidance*

In addition to the previously-described data race detection techniques, there are a number of techniques that rely on system or language support to avoid data races at execution time or by construction.

To our knowledge, the first principled approach to avoiding data races using language support was due to Lamport's work on monitors [78]. Monitors bundle a number of variables and procedures together with a lock that is automatically acquired at entry to each procedure in the bundle and released at the exit from the procedure. The shared variables in the monitor can only be accessed by procedures in the monitor when the monitor lock is held.

Monitors provide a static (e.g., compile-time) guarantee that accesses to static shared global variables are data race free, but if shared variables are allocated dynamically, monitors don't work well. Lampson and Redell note [111] in their experiences on using monitors in the Mesa programming language that the limited applicability of monitors was a significant drawback when designing systems such as the Pilot operating system [163].

Research on deterministic execution systems have gained tremendous popularity in recent years [121, 44, 16, 46, 11, 185, 21]. Deterministic execution requires making the program merely a function of its inputs [188], thereby eliminating the unpredictable behavior due to data races. These systems allow executing arbitrary concurrent software deterministically using hardware, runtime, or operating system support. Alas, deterministic execution systems typically incur high overheads. Therefore, they are not adopted in production software yet. DMP [46] proposes hardware extensions that allow arbitrary software to be executed deterministically with low overhead. However, hardware support required by DMP is not readily available.

According to some experts, deterministic execution systems do not inherently simplify parallel programming [73].

Determinator [11] is a novel operating system kernel that aims to deterministically execute arbitrary programs. Determinator allocates each thread of execution a private working space that is a copy of the global state. Threads reconcile their view of the global state at well-defined points in a program's execution. The use of private workspaces eliminates all read/write conflicts in Determinator, and write/write conflicts are transformed into runtime exceptions. Determinator allows running a number coarse-grain parallel benchmarks with comparable performance to a nondeterministic kernel. Current operating system kernels are not built with Determinator's determinism guarantees, and it is unclear if they will be in the future.

Some deterministic execution techniques rely on language support. StreamIt [185] is a stream-based programming language that allows threads to communicate only via explicitly defined streams, and therefore provides determinism for stream-based programs. DPJ [21] is a type and effect system for Java that is deterministic by default and only allows explicit non-determinism. These techniques that rely on language support allow the developer to build deterministic systems by construction, however they are not widely adopted yet.

In order to achieve deterministic execution in the general case, data races must be eliminated from programs in some way. However, eliminating all data races leads to high overheads due to excessive increase in synchronization operations. We believe that this overhead has been an important obstacle for the widespread adoption of deterministic execution systems in production software. Combined with the techniques developed in this dissertation, it may be possible to relax determinism guarantees and eliminate data races that really matter from the point of view of a developer or user, and make deterministic execution more practical.

2.4 ATOMICITY VIOLATION DETECTION

It is challenging to build an atomicity violation detector that does not have false positives. The reason behind this is that as opposed to data races, atomicity violations are high-level semantic bugs. Atomicity violations occur because developers' assumptions regarding atomicity properties of program statements (or segments) is incorrect. Alas, in the absence of a formal specification (or annotations [58]) of correct atomicity requirements for a program, it is challenging to detect the violations of atomicity.

Atomicity violation detectors mainly come in two flavors: static [158, 149] and dynamic [58, 61, 123, 19, 19].

2.4.1 *Static Atomicity Violation Detection*

Static atomicity violation detectors operate similarly to static data race detectors in that they reason about atomicity violations using the source code and without executing the program.

Von Praun et al. [158] developed a technique that relies on an abstract model of threads and data to detect potential atomicity violations. While this technique has few false negatives when run on programs with previously-known synchronization problems, it suffers from false positives. Another heuristic-based approach [149] statically searches for a pattern that is typically indicative of an atomicity violation. Similarly, this system has a low number of false negatives, but it suffers from false positives.

2.4.2 *Dynamic Atomicity Violation Detection*

Atomizer [58] relies on annotations to denote blocks that are supposed to execute atomically and checks whether such blocks indeed execute atomically at runtime or not. Atomizer can also use heuristics to automatically annotate certain blocks as atomic (e.g., all *synchronized* blocks.). Atomizer performs atomicity checking by combining a lockset-based analysis with Lipton's theory of reduction for parallel programs [120]. If the program uses synchronization mechanisms other than locks, Atomizer can report false positives.

We are referring to Java's synchronized primitive [87].

Recall that in this dissertation, we define atomicity violations as bugs and we do not consider all serializability violations as bugs.

Velodrome [61] soundly and completely checks for violations of serializability in a program by recording a trace of the execution and reasoning about the dependencies between operations in the trace. Although Velodrome's strategy ensures that it detects all atomicity violations in a program, not all serializability violations are necessarily bugs. Therefore, according to our atomicity violation definition (§2.1.2), Velodrome reports false positives.

AVIO [123] automatically extracts invariants that aim to capture developers' assumptions about atomic code regions. Then at runtime, AVIO checks whether these invariants are violated in order to detect atomicity violations. AVIO is effective at detecting atomicity violations, nevertheless AVIO's atomicity invariant extraction is imperfect and can lead to false positives.

Atom-Aid [125] uses architectural support to arbitrarily group consecutive memory operations to reduce the probability of atomicity violations. Atom-Aid reduces the probability that an atomicity violation will lead the program to a failure by 98.7% to 100%. Systems such as Atom-Aid can partially use hardware transactional memory [161, 76] support. Nevertheless, Atom-Aid requires dynamically selecting program segments to execute in a transaction, whereas current transactional memory support requires explicitly defining transactions in the code.

In summary, false positives in atomicity violation detection are hard to avoid for both static and dynamic techniques that attempt to detect patterns for atomicity violations. In contrast, in this dissertation, we adopt a statistics-based approach to detect atomicity violations. In short, to statistically detect atomicity violations, we correlate failures with events that look like violations of atomicity across many executions at user endpoints. Prior work uses similar techniques for finding the root cause of failures due to atomicity violations and other bugs as well, which we talk about in the next section (§2.5).

2.5 ROOT CAUSE DIAGNOSIS OF IN PRODUCTION FAILURES

In this section, we review a variety of techniques that have been developed to date to understand the root causes of failures and to help developers with debugging. We review general techniques for root cause diagnosis as well as special techniques targeted towards failures due to concurrency bugs. We talk about techniques that are geared towards both testing time root cause diagnosis as well as in-production root cause diagnosis.

Delta debugging [216] isolates program inputs and variable values that cause a failure by systematically narrowing the state difference between a failing and a successful run. Delta debugging achieves this by repeatedly reproducing the failing and successful run, and altering variable values. Delta debugging has also been extended to isolate failure-inducing control flow information [40]. As opposed to delta debugging, in this dissertation, we target bugs that are hard to reproduce and aim to generate a (potentially imperfect) explanation of the root cause of a failure even with a single failing execution.

Cooperative approaches such as cooperative bug isolation (CBI) [118], cooperative concurrency bug isolation (CCI) [89], PBI [8], LBRA/LCRA [7] utilize statistical techniques to isolate failure root causes. CBI, CCI and PBI rely on sampling executions in order to reduce the runtime performance overhead. LBRA/LCRA does not resort to sampling, because it introduces custom hardware extensions to do root cause diagnosis with low overhead. LBRA/LCRA relies on observing a failure multiple times to statistically isolate the root cause. However, LBRA/LCRA only works well for bugs with short root cause to failure distances, because the hardware support that it relies on has limited capacity to record events such as branches and cache coherency messages [148]. LBRA/LCRA preserves the privacy of users to some extent, because it does not track the data flow of a program. In this dissertation, we use different failure predicting events for multithreaded bugs (e.g., atomicity violations) than these systems, to allow developers to differentiate between different types of concurrency bugs.

Windows Error Reporting (WER) [67], is a large-scale cooperative error reporting system operating at Microsoft. After a failure, WER collects snapshots of memory and processes them using a number of heuristics (e.g., classification based on call stacks and error codes) to cluster reports that likely point to the same bug. Systems like WER can use root cause diagnosis techniques we develop in this dissertation to improve their clustering of bugs, and help developers fix the bugs faster.

Symbiosis [127] uses a technique called differential schedule projections that displays the set of data flows and memory operations that are responsible for a failure in a multithreaded program. Symbiosis profiles a failing program's schedule and generates non-failing alternate schedules. Symbiosis then determines the data flow differences between the failing schedule and the non-failing schedule in order to help developers identify root causes of failures. Unlike Symbiosis, in this dissertation, we do not assume that we have access to a failing program execution that can be reproduced in-house for the purposes of root cause diagnosis.

PRES [151] records execution sketches, which are abstractions of real executions (e.g., just an execution log of functions), and performs state space exploration on those sketches to reproduce failures. The sketches PRES builds can be used to reason about how a certain failure occurred and do root cause diagnosis.

Previous work also explored adaptive monitoring schemes for gathering execution information from end users. SWAT [74] adaptively samples program segments at a rate that is inversely proportional to their execution frequency. RaceTrack [211] adaptively monitors parts of a program that are more likely to harbor data races. Bias free sampling [100] allows a developer to provide an adaptive scheme for monitoring a program's behavior. Adaptive bug isolation [9] uses heuristics to adaptively estimate and track program behaviors that are likely predictors of failures. In this dissertation, we rely on static analysis to bootstrap and guide runtime monitoring, thereby achieving low latency and low overhead in root cause diagnosis.

HOLMES [35] uses path profiles to perform root cause diagnosis. The main motivation behind HOLMES is that path information is richer and more expressive than other execution information such as return values or scalar relations among variables that prior work [117] resorted to. HOLMES does not track any data values when performing root cause diagnosis, whereas techniques we develop in this dissertation rely on tracking data values for performing root cause diagnosis of concurrency bugs. Tracking data values provides richer debugging information to developers.

SherLog [212] uses a combination of program analysis and execution logs from a failed production run in order to automatically generate control and data flow information that aims to help developers diagnose the root causes of errors. Unlike Sherlog, in this dissertation, we do not assume that logging is always enabled at execution time.

ConSeq [217] computes static slices to identify shared memory reads starting from potentially failing statements (e.g., `assert`). It then records correct runs and, during replay, it perturbs schedules around shared memory reads to try to uncover bugs. In this dissertation, we use static slicing to identify all control and data dependencies to the failure point and do root cause diagnosis of a given failure, without relying on record and replay.

Triage [190], Giri [168], and DrDebug [195] use dynamic slicing for root cause diagnosis. Triage works for systems running on a single processor and uses custom checkpointing support [159]. DrDebug and Giri assume that failures can be reproduced in-house by record/replay and that one knows the inputs that lead to the failure, respectively. In this dissertation, we do not assume that failures can be reproduced in-house.

Tarantula [93] and Ochiai [1] record all program statements that get executed during failing and successful runs, to perform statistical analysis of the recorded statements for root cause diagnosis. In this dissertation, we do not rely on an infrastructure to record all program statements during an execution.

Exterminator [145] and Clearview [152] automatically detect and generate patches for certain types of bugs (e.g., memory errors and security exploits). The techniques we develop in this dissertation can help these tools diagnose failures for which they can generate patches.

2.6 CONCURRENCY BUG CLASSIFICATION

In this section, we will review techniques for classifying concurrency bugs based on their consequences.

As mentioned previously, we are mainly aware of classification schemes for data races. The reason why classification schemes mainly exist for data races is because data race detectors (both dynamic and static), report a lot of data races. Therefore, in practice, developers need to understand the consequences of data races in order to prioritize their fixing.

Prior work on data race classification employs record/replay analysis [142], heuristics [91], detection of ad-hoc synchronization patterns [86, 186] or simulation of the memory model [57].

Record/replay analysis [142] records a program execution and tries to enforce a thread schedule in which the racing threads access a memory location in the reverse order of the original data race. Then, it compares the contents of memory and registers, and uses a difference as and indication of potential harmfulness. In this dissertation, we do not attempt an exact comparison, but rather use a symbolic comparison technique for program outputs (as opposed to the low level internal memory state), and we explore multiple program paths and schedules to increase classification accuracy.

DataCollider [91] uses heuristics to prune predefined classes of likely-to-be harmless data races, thus reporting fewer harmless races overall. DataCollider will consider data races on statistics counters and on variables known to developers to have intentional data races as harmless data races. DataCollider's heuristic-based pruning can introduce false negatives.

Helgrind^{+} [86] and Ad-Hoc Detector [186] eliminate data race reports due to ad-hoc synchronization. Detecting ad-hoc synchronizations or happens-before relationships that are generally not recognized by data race detectors can help further prune harmless data race reports, as demonstrated recently by ATDetector [88].

Adversarial memory [57] finds data races that occur in systems with memory consistency models that are more relaxed than sequential consistency, such as the Java memory model [128]. This approach uses a model of memory that returns stale yet valid values for memory reads (using various heuristics), in an attempt to crash target programs. If a data race causes the program to crash as a result of using the adversarial memory approach, that data race will be classified as harmful. In this dissertation, we follow a similar approach when exploring the consequences of data races. We use a special model of the memory, which buffers all prior writes to memory to be able to later return them based on the semantics of a memory model (§5.7). In this way, we can systematically explore all possible memory model behaviors.

Prior work employed bounded model checking to formally verify concurrent algorithms for simple data types under weak memory models [30]. Formal verification of programs under relaxed memory models (using bounded model checking or any other technique) is a difficult problem, and is undecidable in the general case [10]. In this dissertation, we do not intend to formally verify a program under weak memory consistency. Instead, we focus on determining the combined effects of data races and weak memory models.

RaceFuzzer [172] generates random schedules from a pair of racing accesses to determine whether the race is harmful or not. RaceFuzzer performs schedule fuzzing with the goal of finding bugs, not classifying data races.

Output comparison was used by Pike to find concurrency bugs while fuzzing thread schedules [63]. Pike users can also write state summaries to expose latent semantic bugs that may not always manifest in the program output.

Frost [191] follows a similar approach to Record/Replay Analyzer and Pike in that it explores complementary schedules and detects and avoids potentially harmful races by comparing the program states after program execute with these schedules. This detection is based on state comparison and is therefore, prone to false positives as we later show (§6).

Part II

ELIMINATING CONCURRENCY BUGS FROM IN-PRODUCTION SYSTEMS

In this part, we describe the design, implementation, and evaluation of the techniques and tools we developed for the detection, root cause diagnosis, and classification of in-production concurrency bugs.

First, we present a technique to accurately detect data races in-production. We then present a general technique for diagnosing the root causes of in-production failures that may be caused by concurrency bugs, failing inputs, or rare program paths—although, we primarily focus on failures caused by concurrency bugs. Our technique for root cause diagnosis allows both detecting bugs and providing an explanation of how the failure happened. Then, we discuss how we can perform data race classification under arbitrary memory models. Finally, we present a comprehensive evaluation of all the prototypes we developed.

3

RACEMOB: DETECTING DATA RACES IN PRODUCTION

In this section, we present RaceMob, a new data race detector that combines static and dynamic data race detection to obtain both good accuracy and low runtime overhead. For a given program P, RaceMob first uses a static detection phase with few false negatives to find potential data races; in a subsequent dynamic phase, RaceMob crowdsources the validation of these alleged data races to user machines that are anyway running P. RaceMob provides developers with a dynamically updated list of data races, split into "confirmed true races", "likely false positives", and "unknown"—developers can use this list to prioritize their debugging attention. To minimize runtime overhead experienced by users of P, RaceMob adjusts the complexity of data race validation on-demand to balance accuracy and cost. By crowdsourcing validation, RaceMob amortizes the cost of validation and (unlike traditional testing) gains access to real user executions. RaceMob also helps discovering user-visible failures like crashes or hangs, and therefore helps developers to reason about the consequences of data races. To the best of our knowledge, RaceMob is the first data race detector that combines sufficiently low overhead to be always-on with sufficiently good accuracy to improve developer productivity.

3.1 RACEMOB'S DESIGN OVERVIEW

RaceMob is a crowdsourced, two-phase static–dynamic data race detector. It first statically detects potential data races in a program, then crowdsources the dynamic task of validating these potential data races to users' sites. This validation is done using an on-demand data race detection algorithm. The benefits of crowdsourcing are twofold: first, data race validation occurs in the context of real user executions; second, crowdsourcing amortizes the per-user validation cost. Data race validation confirms true data races, thereby increasing the data race detection coverage.

We define data race detection coverage as the ratio of true data races found in a program by a detector to the total number of true data races in that program.

The usage model is presented in Fig. 5. First, developers set up a "hive" service for their program P; this hive can run centralized or distributed. The hive performs static data race detection on P and finds potential data races (§3.2); these go onto P's list of data races maintained by the hive, and initially each one is marked as "Unknown". Then the hive generates an instrumented binary P′, which users download ① and use instead of the original P. The instrumen-

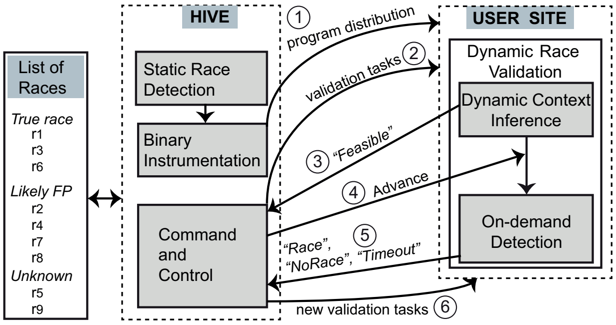

List of Races
True race
r1
r3
r6
Likely FP
r2
r4
r7
r8
Unknown
r5
r9
HIVE
Static Race Detection
Binary Instrumentation
Command and Control
① program distribution
validation tasks ②
③ "Feasible"
④ Advance
"Race", ⑤ "NoRace", "Timeout"
new validation tasks ⑥
USER SITE
Dynamic Race Validation
Dynamic Context Inference
On-demand Detection

Figure 5 – RaceMob's crowdsourced architecture: A static detection phase, run on the hive, is followed by a dynamic validation phase on users' machines.

tation in P' is commanded by the hive, to activate the validation of specific data races in P ②. Different users will typically be validating different, albeit potentially overlapping, sets of data races from P (§3.3). The first phase of validation, called dynamic context inference (§3.3.1), may decide that a particular racing interleaving for data race r is feasible, at which point it informs the hive ③. At this point, the hive instructs all copies of P' that are validating r to advance r to the second validation phase ④. This second phase runs RaceMob's on-demand detection algorithm (§3.3.2), whose result can be one of Race, NoRace, or Timeout ⑤. As results come in to the hive, it updates the list of data races: if a "Race" result came in from the field for data race r, the hive promotes r from "Unknown" to "True Race"; the other answers are used to decide whether to promote r from "Unknown" to "Likely False Positive" or not (§3.5). For data races with status "Unknown" or "Likely False Positive," the hive redistributes "validation tasks" ⑥ among the available users (§3.4). We now describe each step in further detail.

3.2 STATIC DATA RACE DETECTION

RaceMob can use any static data race detector, regardless of whether it is complete or not. We chose Relay, a lockset-based data race detector [192]. Locksets describe the locks held by a program at any given point in the program (as we explained in §2.3.2.2). Relay performs its analysis bottom-up through the program's control flow graph while computing function summaries that summarize which variables are accessed and which locks are held in each function. Relay then composes these summaries to perform data race detection: it flags a data race whenever it sees at least two accesses to memory locations that are the same or may alias, and at least one of the accesses is a write, and the accesses are not protected by at least one common lock.

Relay is complete (i.e., does not miss data races) if the program does not have inline assembly and does not use pointer arithmetic.

Relay may become incomplete if configured to perform file-based alias analysis or aggressive filtering, but we disable these options in RaceMob. As suggested in [114], it might be possible to make Relay complete by integrating program analysis techniques for assembly code [12] and by handling pointer arithmetic [202].

Based on the data race reports from Relay, RaceMob instruments the suspected-racing memory accesses as well as all synchronization operations in the program. This instrumentation will later be commanded (in production) by RaceMob to perform on-demand data race detection.

The hive activates parts of the instrumentation on-demand when the program runs, in different ways for different users. The activation mechanism aims to validate as many data races as possible by uniformly distributing the validation tasks across the user population.

3.3 DYNAMIC DATA RACE VALIDATION

The hive instructs the instrumented programs for which memory accesses to perform data race validation. The validation task sent by the hive to the instrumented program consists of a data race candidate to validate and one desired order (of two possible) of the racing accesses. We call these possible orders the *primary* and the *alternate*.

The dynamic data race validation phase has three stages: dynamic context inference (§3.3.1), on-demand data race detection (§3.3.2), and schedule steering (§3.3.3). Instrumentation for each stage is present in all the programs, however stages 2 and 3 are toggled on/off separately from stage 1, which is always on. Next, we explain each stage in detail.

3.3.1 *Dynamic Context Inference*

Dynamic context inference (DCI) is a lightweight analysis that partly compensates for the inaccuracies of the static data race detection phase. RaceMob performs DCI to figure out whether the statically detected data races can occur in a dynamic program execution context.

DCI validates two assumptions made by the static data race detector about a race candidate. First, the static detector's abstract analysis hypothesizes aliasing as the basis for some of its race candidates, and DCI looks for concrete instances that can validate this hypothesis. Second, the static detector hypothesizes that racing accesses are made by different threads, and DCI aims to validate this as well. Once these two hypotheses are confirmed, the user site communicates this to the hive, and the hive promotes the race candidate to the next phase.

Without a confirmation from DCI, the race remains in the "Unknown" state.

The motivation for DCI comes from our observation that the majority of the potential data races detected by static data race detection (53% in our evaluation) are false positives due to only alias analysis inaccuracies and the inability of static data race detection to infer multithreaded program contexts.

For every potential data race r with racing instructions r_1 and r_2, made by threads T_1 and T_2, respectively, DCI determines whether the potentially racing memory accesses to addresses a_1 and a_2 made by r_1 and r_2, respectively, may alias with each other (i.e., $a_1 = a_2$), and whether these accesses are indeed made by different threads (i.e., $T_1 \neq T_2$). To do this, DCI keeps track of the address that each potentially racing instruction accesses, along with the accessing thread's ID at runtime. Then, the instrumentation checks to see if *at least one* pair of accesses is executed. If yes, the instrumented program notifies the hive, which promotes r to the next stages of validation (on-demand data race detection and schedule steering) on all user machines where r is being watched. If no access is executed by any instrumented instance of the program, DCI continues watching r's potentially racing memory accesses until further notice.

DCI has negligible runtime overhead (0.01%) on top of the binary instrumentation overhead (0.77%); therefore, it is feasible to have DCI always-on. DCI's memory footprint is small: it requires maintaining 12 bytes of information per potential racing instruction (8 bytes for the address, 4 bytes for the thread ID). DCI is sound because, for every access pair that it reports as being made from different threads and to the same address, DCI has clear concrete evidence from an actual execution. DCI is of course not guaranteed to be complete.

3.3.2 *On-Demand Data Race Detection*

In this section, we explain how on-demand data race detection works; for clarity, we restrict the discussion to a single potential data race.

On-demand data race detection starts tracking happens-before relationships once the first potentially racing access is made, and it stops tracking once a happens-before relationship is established between the first accessing thread and all the other threads in the program (in which case a "NoRace" result is sent to the hive). Tracking also stops if the second access is made before such a happens-before relationship is found (in which case a "Race" result is sent to the hive).

Intuitively, RaceMob tracks a minimal number of accesses and synchronization operations. RaceMob needs to track both racing accesses to validate a potential data race. However, RaceMob does not need

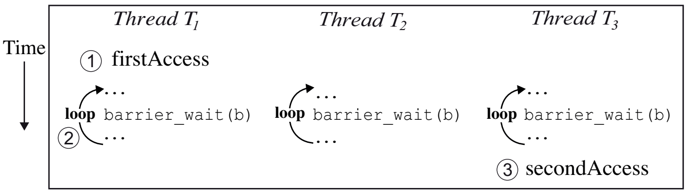


Figure 6 – Minimal monitoring in DCI: For this example, DCI stops tracking synchronization operations as soon as each thread goes once through the barrier.

to track any memory accesses other than the target racing accesses, because any other access is irrelevant to this data race.

Sampling-based data race detection (e.g., PACER [27]) adopts a similar approach to on-demand data race detection by tracking synchronization operations whenever sampling is enabled. The drawback of PACER's approach is that it may start tracking synchronization operations too soon, even if the program is not about to execute a racing access. RaceMob avoids this by turning on tracking synchronization operations on-demand, when an access reported by the static data race detection phase is executed.

RaceMob tracks synchronization operations—and thus, happens-before relationships—using an efficient, dynamic, vector-clock algorithm similar to DJIT$^+$ [153]. We maintain vector clocks for each thread and synchronization operation, and the clocks are partially ordered with respect to each other.

We illustrate in Fig. 6 how the on-demand data race detection stops tracking synchronization operations, using a simple example derived from the `fmm` program [175]: *firstAccess* executes in the beginning of the program in T_1 ①, and the program goes through a few thousand iterations of synchronization-intensive code ②. Finally, T_3 executes *secondAccess* ③. It is sufficient to keep track of the vector clocks of all threads only up until the first time they go through the `barrier_wait` statement, as this establishes a happens-before relationship between *firstAccess* in T_1 and any subsequent access in T_2 and T_3. Therefore, on-demand data race detection stops keeping track of the vector clocks of threads T_1, T_2, and T_3 after they each go through `barrier_wait` once.

RaceMob distinguishes between a static racing instruction in the program and its dynamic instance at runtime, and it can enable on-demand data race detection for any dynamic instance of a potential racing access. However, practice shows that going beyond the first dynamic instances adds little value (§6.1).

Our experimental evaluation shows that on-demand data race detection reduces the overhead of dynamic race detection (§6).

3.3.3 *Schedule Steering*

The schedule steering phase further improves RaceMob's data race detection coverage by exploring both the primary and the alternate executions of potentially racing accesses. This has the benefit of detecting data races that may be hidden by accidental happens-before relationships (as discussed in §1.2.2 and Fig. 2).

Schedule steering tries to enforce the order of the racing memory accesses provided by the hive, i.e., either the primary or the alternate. Whenever the intended order is about to be violated (i.e., the undesired order is about to occur), RaceMob pauses the thread that is about to make the access, by using a wait operation with a timeout τ, to enforce the desired order. Every time the hive receives a "Timeout" from a user, it increments τ for that user (up to a maximum value), to more aggressively steer it toward the desired order, as described in the next section.

Prior work [142, 172, 98] used techniques similar to schedule steering to detect whether a *known* data race can cause a failure or not. RaceMob, however, uses schedule steering to increase the likelihood of encountering a suspected race and to improve data race detection coverage.

Our evaluation shows that schedule steering helps RaceMob to find two data races (one in Memcached and the other one in Pfscan) that would otherwise be missed. It also helps RaceMob uncover failures (e.g., data corruption, hangs, crashes) that occur as a result of data races, thereby enabling developers to reason about the consequences of data races and fix the important ones early, before they affect more users. However, users who do not wish to help in determining the consequences of data races can easily turn off schedule steering.

3.4 CROWDSOURCING THE VALIDATION

Crowdsourcing is defined as the practice of obtaining needed services, ideas, or content by soliciting contributions from a large group of people and especially from the online community. RaceMob gathers validation results from a large base of users and merges them to come up with verdicts for potential data races.

RaceMob's main motivation for crowdsourcing data race detection is to access real user executions. This enables RaceMob, for instance, to detect the important but often overlooked class of input-dependent data races [98], i.e., races that occur only when a program is run with a particular input. RaceMob found two such races in Aget, and we detail them in §6.1.2. Crowdsourcing also enables RaceMob to leverage many executions to establish statistical confidence in the detection verdict. We also believe crowdsourcing is more applicable today than ever: collectively, software users have overwhelmingly more hardware than any single software development organization, so leveraging end-users for data race detection is particularly advantageous. Furthermore, such distribution helps reduce the per-user overhead to barely noticeable levels.

Validation is distributed across a population of users, with each user receiving only a small number of data races to validate. The hive distributes validation tasks, which contain the locations in the program of two memory accesses suspected to be racing, along with a particular order of these accesses. Completing the validation task consists of confirming, in the end-user's instance of the program, whether the indicated ordering of the memory accesses is possible. If a user site receives more than one data race to validate, it will first validate the data race whose racing instruction is first reached during execution.

There exists a wide variety of possible assignment policies that can be implemented in RaceMob. By default, if there are more users than races, RaceMob initially randomly assigns a single data race to each user for validation. Assigned validation tasks that fail to complete within a time bound are then distributed to additional users as well, in order to increase the probability of completing them. Such multiple assignment could be done from the very beginning, in order to reach a verdict sooner. The number of users asked to validate a data race r could be based, for example, on the expected severity of r, as inferred based on heuristics or the static analysis phase. Conversely, in the unlikely case that there are more data races to validate than users, the default policy is to initially distribute a single validation task to each user, thereby leaving a subset of the data races unassigned. As users complete their validation tasks, RaceMob assigns new tasks from among the unassigned data races. Once a data race is confirmed as a true data race, it is removed from the list of data races being validated, for all users.

During schedule steering, whenever a data race candidate is "stubborn" and does not exercise the desired order despite the wait introduced by the instrumentation, a "Timeout" is sent to the hive. The hive then instructs the site to increase the timeout τ, to more aggressively favor the alternate order; every subsequent timeout triggers a new "Timeout" response to the hive and a further increase in τ. Once τ reaches a configured upper bound, the hive instructs the site to abandon the validation task. At this point, or even in parallel with increasing τ, the hive could assign the same task to additional users.

There are two important steps in achieving low overhead at user sites. First, the timeout τ must be kept low. For example, to preserve the low latency of interactive applications, RaceMob uses an upper bound $\tau \leqslant 200$ ms; for I/O bound applications, higher timeouts can be configured. Second, the instrumentation at the user site disables schedule steering for a given data race after a first steering attempt for a given race, regardless of whether it succeeded or not; this is particularly useful when the racing accesses are in a tight loop. Steering is resumed when a new value of τ comes in from the hive. It is certainly possible that a later dynamic instance of the potentially racing instruction might be able to exercise the desired order, had steering not been disabled; nevertheless, in our evaluation we found that RaceMob achieves higher accuracy than state-of-the art data race detectors using a single steering attempt.

RaceMob monitors each user's validation workload and aims to balance the global load across the user population. Rebalancing does not require users to download a new version of the program, but rather the hive simply toggles validation on/off at the relevant user sites. In other words, each instance of the instrumented program P' is capable of validating, on demand, any of the data races found during the static phase—the hive instructs it which data race(s) is/are of interest to that instance of P'.

If an instance of P' spends more time doing data race validation than the overall average, then the hive redistributes some of that instance's validation tasks to other instances. Additionally, RaceMob reshuffles tasks whenever one program experiences more timeouts than the average. In this way, we reduce the number of outliers, in terms of runtime overhead, during the dynamic phase.

Crowdsourcing offers RaceMob the opportunity to tap into a large number of executions, which makes it possible to only perform a small amount of monitoring per user site without harming the precision of detection. This in turn reduces RaceMob's runtime overhead, making it more palatable to users and easier to adopt.

3.5 REACHING A VERDICT

TRUE RACE RaceMob decides a data race candidate is a true data race whenever either the primary or the alternate orders are executed at a user site with no intervening happens-before relationship between the corresponding memory accesses. Among the true data races, some can be specification-violating data races in the sense of [98] (e.g., that cause crash or deadlock). In the case of a crash, the RaceMob instrumentation catches the SIGSEGV signal and submits a crash report to the hive. In the case of an unhandled SIGINT (e.g., the user pressed Ctrl-C), RaceMob prompts the user with a dialog asking whether the program has failed to meet expectations. If yes, the hive is informed that the enforced schedule leads to a specification violation. Of course, users who find this kind of "consequence reporting" too intrusive can disable schedule steering altogether.

LIKELY FALSE POSITIVE RaceMob concludes that a potential data race is likely a false positive if at least one user site reported a No-

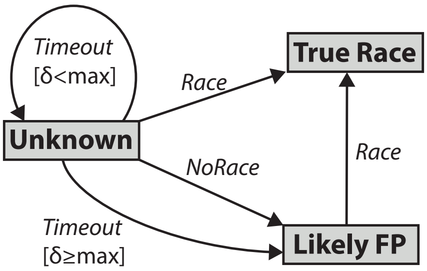


Figure 7 – The state machine used by the hive to reach verdicts based on reports from program instances. Transition edges are labeled with validation results that arrive from instrumented program instances; states are labeled with RaceMob's verdict.

Race result to the hive (i.e., on-demand race detection discovered a happens-before relationship between the accesses in the primary or alternate). RaceMob cannot provide a definitive verdict on whether the race is a false positive or not, because there might exist some other execution in which the purported false positive proves to be a real data race (e.g., due to an unobserved input dependency). The "Likely False Positive" verdict, especially if augmented with the number of corresponding NoRace results received at the hive, can help developers decide whether to prioritize for fixing this particular data race over others. RaceMob continues validation for "Likely False Positive" data races for as long as the developers wishes.

UNKNOWN As long as the hive receives no results from the validation of a potential data race r, the hive keeps the status of the data race "Unknown". Similarly, if none of the program instances report that they reached the maximum timeout value, r's status remains "Unknown". However, if at least one instance reaches the maximum timeout value for r, the corresponding report is promoted to "Likely False Positive".

The "True Race" verdict is definite: RaceMob has proof of the data race occurring in a real execution of the program. The "Likely False Positive" verdict is probabilistic: the more NoRace or Timeout reports are received by the hive as a result of distinct executions, the higher the probability that a data race report is indeed a false positive, even though there is no precise probability value that RaceMob assigns to this outcome.

3.6 RACEMOB'S IMPLEMENTATION DETAILS

RaceMob can use any data race detector that outputs data race candidates; preferably it should be complete (i.e., not miss data races). We use RELAY, which analyzes programs that are turned into CIL, an intermediate language for C programs [143]. The instrumentation engine at the hive is based on LLVM [113]. We wrote a 500-LOC plugin that converts Relay reports to the format required by our instrumentation engine.

The instrumentation engine is an LLVM static analysis pass. It avoids instrumenting empty loop bodies that have a data race on a variable in the loop condition (e.g., of the form `while(notDone){}`). These loops occur often in ad-hoc synchronization [206]. Not instrumenting such loops avoids excessive overhead that results from running the instrumentation frequently. When such loops involve a data race candidate, they are reported by the hive directly to developers. We encountered this situation in two of the programs we evaluated, and both cases were true data races (thus validating prior work that advocates against ad-hoc synchronization [206]), so this optimization did not effect RaceMob's accuracy.

Whereas Fig. 7 indicates three possible results from user sites (Race, NoRace, and Timeout), our prototype also implements a fourth one (NotSeen), to indicate that a user site has not witnessed the race it was expected to monitor. Technically, NotSeen can be inferred by the hive from the absence of any other results. However, for efficiency purposes, we have a hook at the exit of `main`, as well as in the signal handlers, that send a NotSeen message to the hive whenever the program terminates without having made progress on the validation task.

RaceMob uses compiler-based instrumentation, but other techniques are also possible. For example, we plan to use dynamic binary rewriting, which would also allow us to dynamically remove instrumentation for data races that are not enabled in a given instance of the program. The instrumentation is for the most part inactive; the hive activates part of the instrumentation on-demand, when the program runs.

4

GIST: ROOT CAUSE DIAGNOSIS OF IN-PRODUCTION FAILURES

In this chapter, we present failure sketching, a technique that automatically produces a high-level execution trace called the *failure sketch* that includes the statements that lead to a failure and the differences between the properties of failing and successful program executions.

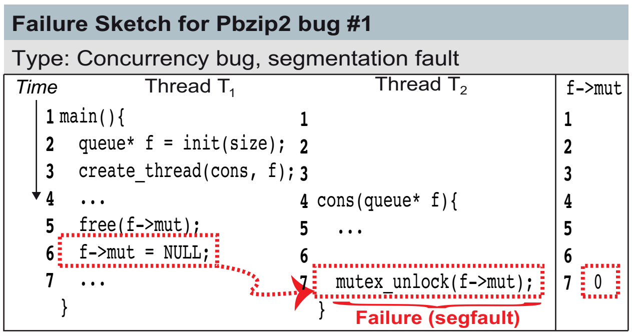
We show in our evaluation (§6) that the differences between failing and successful executions which are displayed on the failure sketch, point to root causes [118, 168, 216], of failures for the bugs we evaluated (§6.2). As mentioned earlier in §2.1.3 In the context of our work, we are talking about a statistical definition of a root cause: we consider events that are primarily correlated with a failure as the root causes of that failure.

Root cause diagnosis allows detecting bugs as well as providing an explanation of how the failure associated with the bug occurred: which branches were taken, which data values were computed, which thread schedule led to the failure, etc. After all, understanding how a failure occurred requires detecting what the failure causing bug is. Therefore, in the rest of this chapter, when we refer to root cause diagnosis, we imply both detection of a bug and an explanation of how a given bug led to a failure.

Root cause diagnosis of a failure subsumes the detection of the bug causing the failure.

Failure sketching is a general technique for root cause diagnosis of concurrency bugs as well as some sequential bugs due to failing input values or rare paths that a program takes. Despite this generality, in the context of this dissertation, and especially when evaluating failure sketching, we focus on concurrency bugs.

Fig. 8 shows an example failure sketch for a bug in Pbzip2, a multithreaded compression tool [66]. Time flows downward, and execution steps are enumerated along the flow of time. The failure sketch shows the statements (in this case statements from two threads) that affect the failure and their order of execution with respect to the enumerated steps (i.e., the control flow). The arrow between the two statements in dotted rectangles indicates the difference between failing executions and successful ones. In particular, in failing executions, the statement `f->mut` from T_1 is executed before the statement `mutex_unlock(f->mut)` in T_2. In non-failing executions, `cons` returns before `main` sets `f->mut` to `NULL`. The failure sketch also shows the value of the variable `f->mut` (i.e., the data flow) in a dotted rectangle in step 7, indicating that this value is 0 in step 7 only in failing runs. A developer can use the failure sketch to fix the bug by introducing proper synchronization that eliminates the offending thread schedule. This


Failure Sketch for Pbzip2 bug #1
Type: Concurrency bug, segmentation fault
Time
Thread T_1
Thread T_2
f->mut
0
Failure (segfault)

```
main(){
  queue* f = init(size);
  create_thread(cons, f);
  ...
  free(f->mut);
  f->mut = NULL;
  ...
}
```

```
cons(queue* f){
  ...
  mutex_unlock(f->mut);
}
```

Figure 8 – The failure sketch of pbzip2 bug.

is exactly how pbzip2 developers fixed this bug, albeit four months after it was reported [66].

The insight behind the work presented in the chapter is that failure sketches can be built automatically, by using a combination of static program analysis and cooperative dynamic analysis. The use of a brand new hardware feature in Intel CPUs helps keep runtime performance overhead low (3.74% in our evaluation).

We built a prototype, Gist, that automatically generates a failure sketch for a given failure. Given a failure, Gist first statically computes a program slice that contains program statements that can potentially affect the program statement where the failure manifests itself. Then, Gist performs data and control flow tracking in a cooperative setup by targeting either multiple executions of the same program in a data center or users who execute the same program. Gist uses hardware watchpoints to track the values of data items in the slice, and uses Intel Processor Trace [83] to track the control flow.

Although Gist relies on Intel Processor Trace and hardware watchpoints for practical and low-overhead control and data flow tracking, Gist's novelty is in the combination of static program analysis and dynamic runtime tracing to judiciously select how and when to trace program execution in order to best extract the information for failure sketches.

In the rest of this chapter, we describe the overview of Gist's design (§4.1). We then explain each component of Gist's design in detail: we first describe static slicing in Gist (§4.2), we then explain how Gist performs slice refinement (§4.3), and we finally describe how Gist identifies the root cause of a failure (§4.4).

4.1 GIST'S DESIGN OVERVIEW

Gist, our system for building failure sketches has three main components: the static analyzer, the failure sketch computation engine, and the runtime that tracks production runs. The static analyzer and the failure sketch computation engine constitutes the server side of

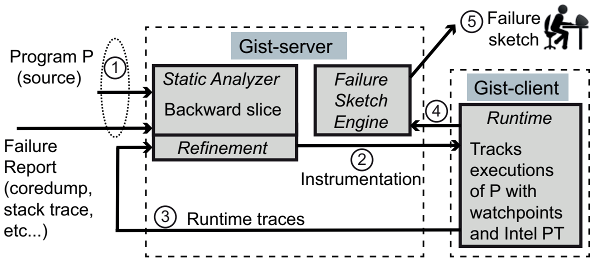

Program P (source)
①
Failure Report (coredump, stack trace, etc...)
Gist-server
Static Analyzer
Backward slice
Refinement
Failure Sketch Engine
② Instrumentation
③ Runtime traces
④
⑤ Failure sketch
Gist-client
Runtime
Tracks executions of P with watchpoints and Intel PT

Figure 9 – The architecture of Gist

Gist, and they can be centralized or distributed, as needed. The runtime constitutes the client-side of Gist, and it runs in a cooperative setting such as in a data center or in multiple users' machines, similar to RaceMob [99].

The usage model of Gist is shown in Fig. 15. Gist takes as input a program (source code and binary) and a failure report ① (e.g., stack trace, the statement where the failure manifests itself, etc). Gist, being a developer tool, has access to the source code. Using these inputs, Gist computes a backward slice [199] by computing the set of program statements that potentially affect the statement where the failure occurs. Gist uses an interprocedural, path-insensitive and flow-sensitive backward slicing algorithm. Then, Gist instructs its runtime, running in a data center or at user endpoints, ② to instrument the programs and gather more traces (e.g., branches taken and values computed at runtime). Gist then uses these traces to refine the slice ③: refinement removes from the slice the statements that don't get executed during the executions that Gist monitors, and it adds to the slice statements that were not identified as being part of the slice initially. Refinement also determines the inter-thread execution order of statements accessing shared variables and the values that the program computes. Refinement is done using hardware watchpoints for data flow and Intel Processor Trace (Intel PT) for control flow. Gist's failure sketch engine gathers execution information from failing and successful runs ④. Then, Gist determines the differences between failing and successful runs and builds a failure sketch ⑤ for the developer to use.

Gist operates in a cooperative setting [99, 118] where multiple instances of the same software execute in a data center or in multiple users' machines. Gist's server side (e.g., a master node) performs offline analysis and distributes instrumentation to its client side (e.g., a node in a data center). Gist incurs low overhead, so it can be kept always-on and does not need to resort to sampling an execution [118], thus avoiding missing information that can increase root cause diagnosis latency.

Gist operates iteratively: the instrumentation and refinement continues as long as developers see fit, continuously improving the accuracy of failure sketches. Gist generates a failure sketch after a first failure using static slicing. Our evaluation shows that, in some cases, this initial sketch is sufficient for root cause diagnosis, whereas in other cases refinement is necessary (§4.3).

We now describe how each component of Gist works and explain how they solve the challenges presented in §1.2. We first describe how Gist computes the static slice followed by slice refinement via adaptive tracking of control and data flow information. Then, we describe how Gist identifies the root cause of a failure using multiple failing and successful runs.

4.2 STATIC SLICE COMPUTATION

Gist uses an interprocedural, path-insensitive and flow-sensitive backward slicing algorithm to identify the program statements that may affect the statement where the failure manifests itself at runtime. We chose to make Gist's slicing algorithm interprocedural because failure sketches can span the boundaries of functions. The algorithm is path-insensitive in order to avoid the cost of path-sensitive analyses that do not scale well [5, 162]. However, this is not a shortcoming, because Gist can recover precise path information at runtime using low-cost control flow tracking (§4.3.2). Finally, Gist's slicing algorithm is flow-sensitive because it traverses statements in a specific order (backwards) from the failure location. Flow-sensitivity generates static slices with statements in the order they appear in the program text (except some out-of-order statements due to path-insensitivity, which are fixed using runtime tracking), thereby helping the developer to understand the flow of statements that lead to a failure.

Algorithm 1 describes Gist's static backward slicing: it takes as input a failure report (e.g., a coredump, a stack trace) and the program's source code, and it outputs a static backward slice. For clarity, we define several terms we use in the algorithm. CFG refers to the control flow graph of the program (Gist computes a whole-program CFG as we explain shortly). An item (line 7) is an arbitrary program element. A source (line 8, 16) is an item that is either a global variable, a function argument, a call, or a memory access. Items that are not sources are compiler intrinsics, debug information, and inline assembly. The definitions for a source and an item are specific to LLVM [113], which is what we use for the prototype (§4.5). The function `getItems` (line 1) returns all the items in a given statement (e.g., the operands of an arithmetic operation). The function `getRetValues` (line 11) performs an intraprocedural analysis to compute and return the set of items that can be returned from a given function call. The function `getArgValues` (line 14) computes and returns the set of arguments that can be used when calling a given function. The function `getReadOperand` (line 20) returns the item that is read, and the function `getWrittenOperand` (line 23) returns the item that is written.

```
Input : Failure report report, program source code program
Output : Static backward slice slice
workSet ← getItems(failingStmt)
function init ()
  failingStmt ← extractFailingStatement(report)
function computeBackwardSlice (failingStmt, program)
  cfg ← extractCFG(program)
  while !workSet.empty() do
      item ← workSet.pop()
      if isSource(item) then
          slice.push(item)
          if isCall(item) then
              retValues ← getRetValues(item, cfg)
              workSet ← workSet ∪ retValues
          else if isArgument(item) then
              argValues ← getArgValues(item, cfg)
              workSet ← workSet ∪ argValues
function isSource (item)
if item is (global||argument||call||memory access) then
    return true
else if item is read then
    workSet ← workSet ∪ item.getReadOperand()
    return true
else if item is write then
    workSet ← workSet ∪ item.getWrittenOperand()
    return true
  return false
```

Algorithm 1 : Backward slice computation (Simplified)

Gist's static slicing algorithm differs from classic static slicing [199] in two key ways:

First, Gist addresses a challenge that arises for multithreaded programs because of the implicit control flow edges that get created due to thread creations and joins. For this, Gist uses a compiler pass to build the *thread interprocedural control flow graph* (TICFG) of the program [205]. An *interprocedural control flow graph* (ICFG) of a program connects each function's CFG with function call and return edges. TICFG then augments ICFG to contain edges that represent thread creation and join statements (e.g., a thread creation edge is akin to a callsite with the thread start routine as the target function). TICFG represents an overapproximation of all the possible dynamic control flow behaviors that a program can exhibit at runtime. TICFG is useful for Gist to track control flow that is implicitly created via thread creation and join operations (§4.3.2).

Second, unlike other slicing algorithms [177], Gist does not use static alias analysis. Alias analysis could determine an overapproximate set of program statements that may affect the computation of a given value and augment the slice with this information. Gist does not employ static alias analysis because, in practice, it can be over 50% inaccurate [99], which would increase the static slice size that Gist would have to monitor at runtime, thereby increasing its performance overhead. Gist compensates for the lack of alias analysis with runtime data flow tracking, which adds the statements that Gist misses to the static slice (§4.3.3).

The static slice that Gist computes has some extraneous items that do not pertain to the failure, because the slicing algorithm lacks actual execution information. Gist weeds out this information using accurate control flow tracking at runtime (§4.3.2).

4.3 SLICE REFINEMENT

Slice refinement removes the extraneous statements from the slice and adds to the slice the statements that could not be statically identified. Together with root cause identification (§4.4), the goal of slice refinement is to build what we call the ideal failure sketch.

We define *an ideal failure sketch to be one that*: 1) contains only statements that have data and/or control dependencies to the statement where the failure occurs; 2) shows the failure predicting events that have the highest positive correlation with the occurrence of failures.

Different developers may have different standards as to what is the “necessary” information for root cause diagnosis; nevertheless, we believe that including all the statements that are related to a failure and identifying the failure predicting events, constitute a reasonable and practical set of requirements for root cause diagnosis. Failure predictors are identified by determining the difference of key properties between failing and successful runs.

For example, failure sketches display the partial order of statements involved in data races and atomicity violations, however certain developers may want to know the total order of all the statements in an ideal failure sketch. In our experience, focusing on the partial order of statements that matter from the point of view of root cause diagnosis is more useful than having a total order of all statements. Moreover, obtaining the total order of all the statements in a failure sketch would be difficult without undue runtime performance overhead using today’s technology.

We now describe Gist’s slice refinement strategy in detail. We first describe adaptive tracking of a static slice to reduce the overhead of refinement (§4.3.1), then we describe how Gist tracks the control flow (§4.3.2) and the data flow (§4.3.3) to 1) add to the slice statements that get executed in production but are missing from the slice, and 2) remove from the slice statements that don’t get executed in production.

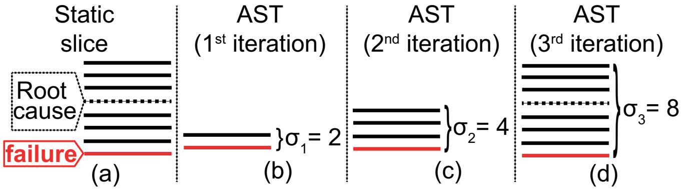


Figure 10 – Adaptive slice tracking in Gist

4.3.1 *Adaptive Slice Tracking*

Gist employs Adaptive Slice-Tracking (AsT) to track increasingly larger portions of the static slice, until it builds a failure sketch that contains the root cause of the failure that it targets. Gist performs AsT by dynamically tracking control and data flow while the software runs in production. AsT does not track all the control and data elements in the static slice at once in order to avoid introducing performance overhead.

It is challenging to pick the size of the slice, σ, to monitor at runtime, because 1) a too large σ would cause Gist to do excessive runtime tracking and increase overhead; 2) a too small σ may cause Gist to track too many runs before identifying the root cause, and so increase the latency of root cause diagnosis.

Based on previous observations that root causes of most bugs are close to the failure locations [159, 198, 217], Gist initially enables runtime tracking for a small number of statements ($\sigma = 2$ in our experiments) backward from the failure point. We use this heuristic because even a simple concurrency bug is likely to be caused by two statements from different threads. This also allows Gist to avoid excessive runtime tracking if the root cause is close to the failure (i.e., the common case). Nonetheless, to reduce the latency of root cause diagnosis, Gist employs a multiplicative increase strategy for further tracking the slice in other production runs. More specifically, Gist doubles σ for subsequent AsT iterations, until a developer decides that the failure sketch contains the root cause and instructs Gist to stop AsT.

Consider the static slice for a hypothetical program in Fig. 10.(a), which displays the failure point (bottom-most solid line) and the root cause (dashed line). In the first iteration (Fig. 10.(b)), AsT starts tracking $\sigma_1 = 2$ statements back from the failure location. Gist cannot build a failure sketch that contains the root cause of this failure by tracking 2 statements, as the root cause lies further backwards in the slice. Therefore, in the second and third iterations (Fig. 10.(c-d)), AsT tracks $\sigma_2 = 4$ and $\sigma_3 = 8$ statements, respectively. Gist can build a failure sketch by tracking 8 statements.

In summary, AsT is a heuristic to resolve the tension between performance overhead, root cause diagnosis latency, and failure sketch accuracy. We elaborate on this tension in our evaluation (§6.2). AsT does not limit Gist's ability to track larger slices and build failure sketches for bugs with greater root-cause-to-failure distances, although it may increase the latency of root cause diagnosis.

4.3.2 *Tracking Control Flow*

Gist tracks control flow to increase the accuracy of failure sketches by identifying which statements from the slice get executed during the monitored production runs. Static slicing lacks real execution information such as dynamically computed call targets, therefore tracking the dynamic control flow is necessary for high accuracy failure sketches.

Static slicing and control flow tracking jointly improve the accuracy of failure sketches: control flow traces identify statements that get executed during production runs that Gist monitors, whereas static slicing identifies statements that have a control or data dependency to the failure. The intersection of these statements represents the statements that relate to the failure and that actually get executed in production runs. Gist statically determines the locations where control flow tracking should start and stop at runtime in order to identify which statements from the slice get executed.

Although control flow can be tracked in a relatively straightforward manner using software instrumentation [126], hardware facilities offer an opportunity for a design with lower overhead. Our design employs Intel PT, a set of new hardware monitoring features for debugging. In particular, Intel PT records the execution flow of a program and outputs a highly-compressed trace (~0.5 bits per retired assembly instruction) that describes the outcome of all branches executed by a program. Intel PT can be programmed to trace only user-level code and can be restricted to certain address spaces. Additionally, with the appropriate software support, Intel PT can be turned on and off by writing to processor-specific registers. Intel PT is currently available in Broadwell processors, and we control it using our custom kernel driver (§4.5). Future families of Intel processors are also expected to provide Intel PT functionality.

We explain how Gist tracks the statements that get executed via control flow tracking using the example shown in Fig. 11.(a). The example shows a static slice composed of three statements (stmt_1, stmt_2, stmt_3). The failure point is stmt_3. Let us assume that, as part of AsT, Gist tracks these three statements. At the high level, Gist identifies all entry points and exit points to each statement and starts and stops control-flow tracking at each entry point and at each exit point, respectively. Tracing is started to capture control flow if the statements in the static slice get executed, and is stopped once those statements complete execution. We use postdominator analysis to optimize out unnecessary tracking.

In this example, Gist starts its analysis with $stmt_1$. Gist converts the branch decision information to statement execution information using the technique shown in box I in Fig. 11.(a). It first determines bb_1, the basic block in which $stmt_1$ resides, and then determines the predecessor basic blocks $p_{11}...p_{1n}$ of bb_1. The predecessor basic blocks of bb_1 are blocks from which control can flow to bb_1 via branches. As a result, Gist starts control flow tracking in each predecessor basic block $p_{11}...p_{1n}$ (i.e., entry points). If Gist's control flow tracking determines at runtime that any of the branches from these predecessor blocks to bb_1 was taken, Gist deduces that $stmt_1$ was executed.

Gist uses an optimization when a statement it already processed strictly dominates the next statement in the static slice. A statement d strictly dominates a statement n (written d *sdom* n) if every path from the entry node of the control flow graph to n goes through d, and $d \neq n$. In our example, $stmt_1$ *sdom* $stmt_2$, therefore, Gist will have already started control flow tracking for $stmt_1$ when the execution reaches $stmt_2$, and so it won't need special handling to start control flow tracking for $stmt_2$.

However, if a statement that Gist processed does not strictly dominate the next statement in the slice, Gist stops control flow tracking. In our example, after executing $stmt_2$, since the execution may never reach $stmt_3$, Gist stops control flow tracking after $stmt_2$ gets executed. Otherwise tracking could continue indefinitely and impose unnecessary overhead. Intuitively, Gist stops control flow tracking *right after* $stmt_2$ gets executed as shown in box II of Fig. 11.(a). More precisely, Gist stops control flow tracking after $stmt_2$ and before $stmt_2$'s immediate postdominator. A node p is said to strictly postdominate a node n if all the paths from n to the exit node of the control flow graph pass through p, and $n \neq p$. The immediate postdominator of a node n (*ipdom*(n)) is a unique node that strictly postdominates n and does not strictly postdominate any other strict postdominators of n.

Finally, as shown in box III in Fig. 11.(a), Gist processes $stmt_3$ using the combination of techniques it used for $stmt_1$ and $stmt_2$. Because control flow tracking was stopped after $stmt_2$, Gist first restarts it at each predecessor basic block $p_{31}...p_{3n}$ of the basic block bb_3 that contains $stmt_3$, then Gist stops it after the execution of $stmt_3$.

4.3.3 *Tracking Data Flow*

Similar to control flow, data flow can also be tracked in software, however this can be prohibitively expensive [190]. Existing hardware support can be used for low overhead data flow tracking. In this section, we describe why and how Gist tracks data flow.

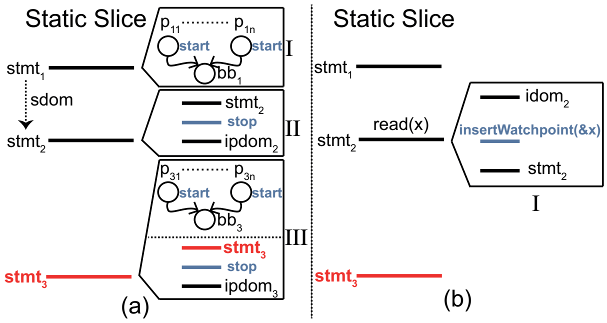


Figure 11 – Example of control (a) and data (b) flow tracking in Gist. Solid horizontal lines are program statements, circles are basic blocks.

Determining the data flow in a program increases the accuracy of failure sketches in two ways:

First, Gist tracks the total order of memory accesses that it monitors to increase the accuracy of the control flow shown in the failure sketch. Tracking the total order is important mainly for shared memory accesses from different threads, for which Intel PT does not provide order information. Gist uses this order information in failure sketches to help developers reason about concurrency bugs.

Second, while tracking data flow, Gist discovers statements that access the data items in the monitored portion of the slice that were missing from that portion of the slice. Such statements exist because Gist's static slicing does not use alias analysis (due to alias analysis' inaccuracy) for determining all statements that can access a given data item.

Gist uses hardware watchpoints present in modern processors to track the data flow (e.g., x86 has 4 hardware watchpoints [81]). They enable tracking the values written to and read from memory locations with low runtime overhead.

For a given memory access, Gist inserts a hardware watchpoint for the address of the accessed variable at a point *right before* the access instruction. More precisely, the inserted hardware watchpoint must be located before the access and after the immediate dominator of that access. Fig. 11.(b) shows an example, where Gist places a hardware watchpoint for the address of variable `x`, just before stmt2 (`read(x)`).

Gist employs several optimizations to economically use its budget of limited hardware watchpoints when tracking the data flow. First, Gist only tracks accesses to shared variables, it does not place a hardware watchpoint for the variables allocated on the stack. Gist maintains a set of active hardware watchpoints to make sure to not place a second hardware watchpoint at an address that it is already watching.

If the statements in the slice portion that AsT monitors access more memory locations than the available hardware watchpoints on a user machine, Gist uses a cooperative approach to track the memory locations across multiple production runs. In a nutshell, Gist's collabora-

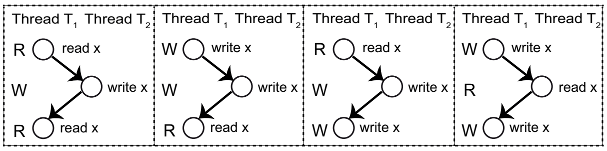


Figure 12 – Four common atomicity violation patterns (RWR, WWR, RWW, WRW). Adapted from [8].

tive approach instructs different production runs to monitor different sets of memory locations in order to monitor all the memory locations that are in the slice portion that Gist monitors. However, in practice, we did not encounter this situation (§6.2).

4.4 IDENTIFYING THE ROOT CAUSE

In this section, we describe how Gist determines the differences of key execution properties (i.e., control and data flow) between failing and successful executions in order to do root cause diagnosis and statistically detect concurrency bugs (e.g., atomicity violations).

For root cause diagnosis, Gist follows a similar approach to cooperative bug isolation [8, 89, 118], which uses statistical methods to correlate failure predictors to failures in programs. A failure predictor is a predicate that, when true, predicts that a failure will occur [117]. Carefully crafted failure predictors point to failure root causes [118].

Gist-generated failure sketches contain a set of failure predictors that are both informative and good indicators of failures. A failure predictor is informative if it contains enough information regarding the failure (e.g., thread schedules, critical data values). A failure predictor is a good indicator of a failure if it has high positive correlation primarily with the occurrence of the failure.

Gist defines failure predictors for both sequential and multithreaded programs. For sequential programs, Gist uses branches taken and data values computed as failure predictors. For multithreaded programs, Gist uses the same predicates it uses for sequential programs, as well as special combinations of memory accesses that portend concurrency failures. In particular, Gist considers the common single-variable atomicity violation patterns shown in Fig. 12 (i.e., RWR (Read, Write, Read), WWR, RWW, WRW) and data race patterns (WW, WR, RW) as concurrency failure predictors.

For both failing and successful runs, Gist logs the order of accesses and the value updates to shared variables that are part of the slice it tracks at runtime. Then, using an offline analysis, Gist searches the aforementioned failure-predicting memory access patterns in these access logs. Gist associates each match with either a successful run or a failing run. Gist is not a bug detection tool, but it can understand common failures, such as crashes, assertion violations, and hangs.

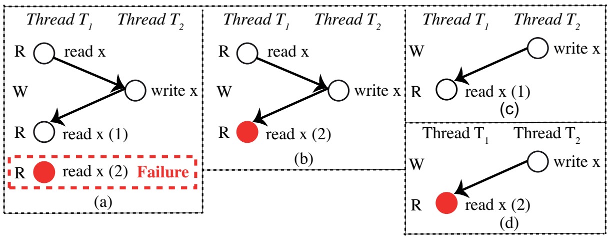


Figure 13 – A sample execution failing at the second read in T_1 (a), and three potential concurrency errors: a RWR atomicity violation (b), 2 WR data races (c-d).

Other types of failures can either be manually given to Gist, or Gist can be used in conjunction with a bug finding tool.

Once Gist has gathered failure predictors from failing and successful runs, it uses a statistical analysis to determine the correlation of these predictors with the failures. Gist computes the precision P (how many runs fail among those that are predicted to fail by the predictor?), and the recall R (how many runs are predicted to fail by the predictor among those that fail?). Gist then ranks all the events by their F-measure, which is the weighted harmonic mean of their precision and recall $F_\beta = (1+\beta^2)\frac{P.R}{\beta^2.P+R}$ to determine the best failure predictor. Gist favors precision by setting β to 0.5 (a common strategy in information retrieval [164]), because its primary aim is to not confuse the developers with potentially erroneous failure predictors (i.e., false positives).

The failure sketch presents the developer with the highest-ranked failure predictors for each type of failure predictor (i.e., branches, data values, and statement orders). An example of this is shown in Fig. 8, where the dotted rectangles show the highest-ranked failure predictor. Gist's root cause detection process also enables it to statistically detect concurrency bugs such as atomicity violations.

As an example, consider the execution trace shown in Fig. 13.(a). Thread T_1 reads x, after which thread T_2 gets scheduled and writes to x. Then T_1 gets scheduled back and reads x twice in a row, and the program fails (e.g., the second read could be made as part of an assertion that causes the failure). This execution trace has three memory access patterns that can potentially be involved in a concurrency bug: a RWR atomicity violation in Fig. 13.(b) and two data races (or order violations) in Fig. 13.(c) and 13.(d). For this execution, Gist logs these patterns and their outcome (i.e., failure and success: 13.(b) and 13.(d) fail, whereas the pattern in 13.c succeeds. Gist keeps track of the outcome of future access patterns and computes their F-measure to identify the highest ranked failure predictors.

There are two key differences between Gist and cooperative bug isolation (CBI). First, Gist tracks all data values that are part of the slice that it monitors at runtime, allowing it to diagnose the root cause of failures caused by a certain input, as opposed to CBI, which tracks ranges of some variables. Second, Gist uses different failure predictors than CCI [89] and PBI [8], which allow developers to distinguish between different kinds of concurrency bugs, whereas PBI and CCI use the same predictors for failures with different root causes (e.g., invalid MESI [148] state for all of RWR, WWR, RWW atomicity violations).

4.5 GIST'S IMPLEMENTATION DETAILS

Gist's static slicing algorithm is built on the LLVM framework [113]. As part of this algorithm, Gist first augments the intraprocedural control flow graphs of each function with function call and return edges to build the interprocedural control flow graph (ICFG) of the program. Then, Gist processes thread creation and join functions (e.g., `pthread_create`, `pthread_join`) to determine which start routines the thread creation functions may call at runtime and where those routines will join back to their callers, using data structure analysis [112]. Gist augments the edges in the ICFGs of the programs using this information about thread creation/join in order to build the thread interprocedural control flow graph (TICFG) of the program. Gist uses the LLVM information flow tracker [90] as a starting point for its slicing algorithm.

Gist currently inserts a small amount of instrumentation into the programs it runs, mainly to start/stop Intel PT tracking and place a hardware watchpoint. To distribute the instrumentation, Gist uses `bsdiff` to create a binary patch file that it ships off to user endpoints or to a data center. We plan to investigate more advanced live update systems such as POLUS [34] or Courgette [70]. Another alternative is to use binary rewriting frameworks such as DynamoRio [28] or Paradyn [135].

Trace collection is implemented via a Linux kernel module which we refer to as the Intel PT kernel driver [107, 106]. The kernel driver configures and controls the hardware using the documented MSR (Machine Specific Register) interface. The driver allows filtering of what code is traced using the privilege level (i.e. kernel vs. userspace) and CR3 values, thus allowing tracing of individual processes. The driver uses a memory buffer sized at 2 MB, which is sufficient to hold traces for all the applications we have tested. The driver relies on the Intel PT trace decoding library [43] to decode control flow traces. Finally, Gist-instrumented programs use an `ioctl` interface that our driver provides to turn tracing on/off.

Gist's hardware watchpoint use is based on the `ptrace` system call. Once Gist sets the desired hardware watchpoints, it detaches from the program (using the PTRACE_DETACH), thereby not incurring any performance overhead. Gist's instrumentation handles hardware watchpoint triggers atomically in order to maintain a total order of accesses among memory operations. Gist logs the program counter when a hardware watchpoint is hit, which it later translates into source line information at developer site. Gist does not need debug information to do this mapping: it uses the program counter and the offset at which the program binary is loaded to compute where in the actual program this address corresponds to.

5

PORTEND: CLASSIFYING DATA RACES DURING TESTING

Ideally, programs would have no data races at all. In this way, programs would avoid possible catastrophic effects due to data races. This either requires programs to be data race-free by design, or it requires finding and fixing all data races in a program. However, modern software still has data races either because it was written carelessly, or the complexity of the software made it very difficult to properly synchronize threads, or the benefits of fixing all data races using expensive synchronization did not justify its costs.

From a programming languages standpoint, attempting to classify data races is only meaningful for some languages. One such language is the Java programming language. The Java memory model [129] defines semantics for programs with data races, because Java must support the execution of untrusted sandboxed code, and such code could contain data races. Therefore, attempting to classify data races in Java is a meaningful endeavor from a programming languages point of view. Similar arguments apply to assembly languages, where data races are permitted.

On the other hand, recent C [85] and C++ [84] standards do not provide meaningful semantics for programs involving data races. In other words, data races in those languages constitute undefined behavior. As a consequence, C and C++ compilers are allowed to perform optimizations on code with data races that may transform seemingly benign data races into harmful ones [23, 24, 2].

However, as noted in [23], compilers do not always employ transformations that will break code with data races.

From a practical standpoint, developers may choose to prioritize the fixing of data races (and they do so) regardless of the implications of language standards. This happens primarily because modern multithreaded software tends to have a large number of data races, and it may be impractical to try to fix all the data races in a given program at once. For example, Google's Thread Sanitizer [174] reports over 1,000 unique data races in Firefox (written in C++) when the browser starts up and loads http://bbc.co.uk.

Another reason why developers sometimes choose to not fix all data races is because synchronizing all racing memory accesses would introduce performance overheads that may be considered unacceptable. For example, developers have not fixed a data race that can lead to lost updates in memcached for a year—ultimately finding an alternate solution—because, it leads to a 7% drop in throughput [133]. Performance implications led to 23 data races in Internet Explorer and Windows Vista being purposely left unfixed [142]. Similarly, several data races have been left unfixed in the Windows 7 kernel, because fixing those races did not justify the associated costs [91].

Another reason why data races go unfixed is that 76%–90% of data races are actually considered to be harmless [91, 142, 53, 192]—*harmless races* are assumed to not affect program correctness, either fortuitously or by design, while *harmful races* lead to crashes, hangs, resource leaks, even memory corruption or silent data loss. Deciding whether a race is harmful or not involves a lot of human labor (with industrial practitioners reporting that it can take days, even weeks [69]), so time-pressed developers may not even attempt this high-investment/low-return activity.

In order to construct programs that are free of data races by design, novel languages and language extensions that provide a deterministic programming model have been proposed [185, 21]. Deterministic programs are data race-free, and therefore, their behavior is not timing dependent. Even though these models may be an appropriate solution for the long term, the majority of modern concurrent software is written in mainstream languages such as C, C++, and Java, and these languages don't provide any data race-freedom guarantees.

In order to eliminate all data races in current mainstream software, developers need to first find them. This can be achieved using data race detectors such as RaceMob that we discussed in the previous section. However, given the large number of data race reports in modern software, we argue that data race detectors should also triage reported data races based on the consequences they could have in future executions. This way, developers are better informed and can fix the critical bugs first. A data race detector should be capable of inferring the possible consequences of a reported data race: is it a false positive, a harmful data race, or a data race that has no observable harmful effects and left in the code perhaps for performance reasons?

Alas, automated classifiers [86, 91, 142, 186] are often inaccurate (e.g., [142] reports a 74% false positive rate in classifying harmful races). To our knowledge, no data race detector/classifier can do this without false positives.

In this chapter, we describe Portend, a technique and tool that, given a data race (e.g., detected using RaceMob), analyzes the code, infers each data race's potential consequences and automatically classifies them into four categories: "specification violated", "output differs", "k-witness harmless" and "single ordering". In Portend, harmlessness is circumstantial rather than absolute; it implies that Portend did not witness a harmful effect of the data race in question for k different executions. For the first two categories, Portend produces a replayable trace that demonstrates the effect, making it easy on the developer to fix the race.

Portend, has support for classifying data races under different memory models (e.g., weak memory model [49]) using a technique called symbolic memory consistency modeling (SMCM).

Portend works in-house, because its analysis is computationally intensive, and therefore not suited to in-production use.

Portend operates on binaries, not on source code (more specifically on LLVM [113] bitcode obtained from a compiler or from a machine-code-to-LLVM translator like RevGen [36]). Therefore, it can effectively classify both source-code-level data races and assembly-level data races that are not forbidden by any language-specific memory model (e.g., C [85] and C++ [84]).

In the rest of this chapter, we first introduce our classification scheme (§5.1); give an overview of Portend's design (§5.2); describe how Portend performs single path analysis (§5.3), multipath analysis (§5.4), symbolic output comparison (§5.5), multi-schedule analysis (§5.6); describe symbolic memory consistency modeling (§5.7); Portend's classification verdicts (§5.8); its debugging aid output (§5.9); and its implementation details (§5.10).

5.1 A FINE-GRAINED WAY TO CLASSIFY DATA RACES

A simple harmless vs. harmful classification scheme is undecidable in general (as will be explained below), so prior work typically resorts to "likely harmless" and/or "likely harmful." Alas, in practice, this is less helpful than it seems (§6.3). We therefore propose a new scheme that is more precise.

Note that there is a distinction between false positives and harmless data races: when a purported data race is not a true data race, we say it is a *false positive*. When a true data race's consequences are deemed to be harmless for all the witnessed executions, we refer to it as a harmless data race [1].

A false positive is harmless in an absolute sense (since it is not a data race to begin with), but not the other way around, harmless data races are still true data races. Static [53] and lockset [170] data race detectors typically report false positives.

If a data race does not have any observable effect (crash, hang, data corruption) for all the witnessed executions, we say that the data race is harmless with respect to those executions. Harmless data races are still true data races. Note that our definition of harmlessness is circumstantial; it is not absolute. It is entirely possible that a harmless data race for some executions can become harmful in another execution.

---

[1] In the rest of the text, whenever we mention harmless data races, we refer to this definition. If we refer to another definition adopted by prior work, we make the distinction clear.

Our proposed scheme classifies the true data races into four categories: "spec violated", "output differs", "k-witness harmless", and "single ordering". We illustrate this taxonomy in Fig. 14.

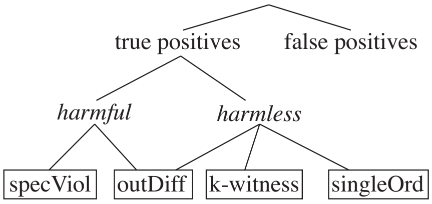


Figure 14 – Portend taxonomy of data races.

"Spec violated" corresponds to data races for which at least one ordering of the racing accesses leads to a violation of the program's specification. These are, by definition, harmful. For example, data races that lead to crashes or deadlocks are generally accepted to violate the specification of any program; we refer to these as "basic" specification violations. Higher level program semantics could also be violated, such as the number of objects in a heap exceeding some bound, or a checksum being inconsistent with the checksummed data. Such semantic properties must be provided as explicit predicates to Portend, or be embedded as assert statements in the code.

"Output differs" is the set of data races for which the two orderings of the racing accesses can cause the program to generate different outputs, thus making the output depend on scheduling that is beyond the application's control. Such data races are often considered harmful: one of those outputs is likely "the incorrect" one. However, "output differs" data races can also be considered as harmless, whether intentional or not. For example, a debug statement that prints the ordering of the racing memory accesses is intentionally order-dependent, thus an intentional and harmless data race. An example of an unintentional harmless data race is one in which one ordering of the accesses may result in a duplicated syslog entry—while technically a violation of any reasonable logging specification, a developer may decide that such a benign consequence makes the data race not worth fixing, especially if she faces the risk of introducing new bugs or degrading performance when fixing the bug.

As with all high level program semantics, automated Portends *cannot* decide on their own whether an output difference violates some non-explicit specification or not. Moreover, whether the specification has been violated or not might even be subjective, depending on which developer is asked. It is for this reason that we created the "output differs" class of data races: we provide developers a clear characterization of the output difference and let them decide using the provided evidence whether that difference matters.

"K-witness harmless" are data races for which the harmless classification is performed with some quantitative level of confidence: the higher the k, the higher the confidence. Such data races are guaranteed to be harmless for at least k combinations of paths and schedules; this guarantee can be as strong as covering a virtually infinite input space (e.g., a developer may be interested in whether the data race is harmless for all positive inputs, not caring about what happens for zero or negative inputs). Portend achieves this using a symbolic execution engine [31, 29] to analyze entire equivalence classes of inputs. Depending on the time and resources available, developers can choose k according to their needs—in our experiments we found $k = 5$ to be sufficient to achieve 99% accuracy (manually verified) for all the tested programs. The value of this category will become obvious in this chapter. We also evaluate the individual contributions of exploring paths versus schedules in §6.3.

"Single ordering" are data races for which only a single ordering of the accesses is possible, typically enforced via ad-hoc synchronization [206]. In such cases, although no explicit synchronization primitives are used, the shared memory could be protected using busy-wait loops that synchronize on a flag. Considering these to be non-data races is inconsistent with our definition (§2.1.1) because the ordering of the accesses is not enforced using non-ad-hoc synchronization primitives, even though it may not actually be possible to exercise both interleavings of the memory accesses (hence the name of the category). Such ad-hoc synchronization, even if bad practice, is frequent in real-world software [206]. Previous data race detectors generally cannot tell that only a single order is possible for the memory accesses, and thus report this as an ordinary data race. Such data races can turn out to be both harmful [206] or they can be a major source of harmless data races [86, 186]. That is why we have a dedicated class for such data races.

5.2 PORTEND'S DESIGN OVERVIEW

Portend feeds the target program through its own data race detector or through RaceMob (or even a third party one, if preferred), analyzes the program and the report automatically, and determines the potential consequences of the reported data race. The report is then classified, based on these predicted consequences, into one of the four categories in Fig. 14. To achieve the classification, Portend performs targeted analysis of multiple schedules of interest, while at the same time using symbolic execution [31, 104, 105] to simultaneously explore multiple paths through the program; we call this technique *multi-path multi-schedule data race analysis*. Portend can thus reason about the consequences of the two orderings of racing memory accesses in a richer execution context than prior work. When comparing program states or program outputs, Portend employs *symbolic output comparison*, meaning it compares constraints on program output as well as path constraints when these outputs are made, in addition to comparing the concrete values of the output, in order to generalize the comparison to more possible inputs that would bring the program to the specific data race and to determine if the data race affects the constraints on program output. Unlike prior work, Portend can accurately classify even data races that, given a fixed ordering of the original racing accesses, are harmless along some execution paths, yet harmful along others. In §5.2 we go over one such data race (Fig. 17) and explain how Portend handles it.

Fig. 15 illustrates Portend's architecture. Portend is based on Cloud9 [29], a parallel symbolic execution engine that supports running multi-threaded C/C++ programs. Cloud9 is in turn based on KLEE [31], which is a single-threaded symbolic execution engine. Cloud9 has a number of built-in checkers for memory errors, overflows and division-by-0 errors; on top of which, Portend adds an additional deadlock detector. Portend has a built-in data race detector that implements a dynamic happens-before algorithm [110]. This detector relies on a component that tracks Lamport clocks [110] at runtime (details are in §5.7). Portend's analysis and classification engine performs multi-path multi-schedule data race analysis and symbolic output comparison. This engine also works together with the Lamport clock tracker and the symbolic memory consistency modeling (SMCM) plugin to perform classification. The SMCM plugin defines the rules according to which a memory read operation from a shared memory location can return previously written values to that location. SMCM plugin is crucial for classifying data races under different memory consistency models.

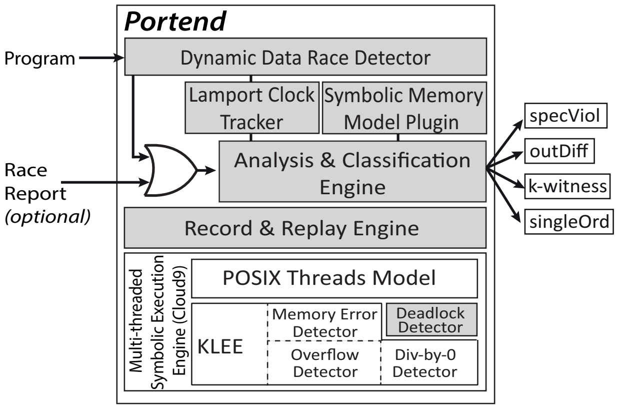

Portend
Program
Dynamic Data Race Detector
Lamport Clock Tracker
Symbolic Memory Model Plugin
Race Report (optional)
Analysis & Classification Engine
specViol
outDiff
k-witness
singleOrd
Record & Replay Engine
Multi-threaded Symbolic Execution Engine (Cloud9)
POSIX Threads Model
KLEE
Memory Error Detector
Deadlock Detector
Overflow Detector
Div-by-0 Detector

Figure 15 – High-level architecture of Portend. The six shaded boxes indicate new code written for Portend, whereas clear boxes represent reused code from KLEE [31] and Cloud9 [29].

When Portend determines that a data race is of "spec violated" kind, it provides the corresponding evidence in the form of program inputs (including system call return values) and thread schedule that reproduce the harmful consequences deterministically. Developers can replay this "evidence" in a debugger, to fix the data race.

We now give an overview of our approach and illustrate it with an example, describe the first step, single-path/single-schedule analysis (§5.3), followed by the second step, multi-path analysis (§5.4) and symbolic output comparison (§5.5) augmented with multi-schedule analysis (§5.6). We introduce SMCM and describe how it can be used to model various memory models while performing data race classification (§5.7). We describe Portend's data race classification (§5.8) and the generated report that helps developers debug the data race (§5.9).

Portend's data race analysis starts by executing the target program and dynamically detecting data races (e.g., developers could run their existing test suites under Portend). Portend detects data races using a dynamic happens-before algorithm [110] or using RaceMob. Alternatively, if another detector is used, Portend can start from an existing execution trace; this trace must contain the thread schedule and an indication of where in the trace the suspected data race occurred. We developed a plugin for Thread Sanitizer [174] to create a Portend-compatible trace; we believe such plugins can be easily developed for other dynamic data race detectors [75].

Portend has a record/replay infrastructure for orchestrating the execution of a multi-threaded program; it can preempt and schedule threads before/after synchronization operations and/or racing accesses. Portend uses Cloud9 to enumerate program paths and to collect symbolic constraints.

A trace consists of a schedule trace and a log of system call inputs. The schedule trace contains the thread id and the program counter at each preemption point. Portend treats all POSIX threads synchronization primitives as possible preemption points and uses a single-processor cooperative thread scheduler. Portend can also preempt threads before and after any racing memory access. We use the following notation for the trace: $(T_1 : pc_0) \rightarrow (T_2 \rightarrow RaceyAccess_{T_2} : pc_1)$ $\rightarrow (T_3 \rightarrow RaceyAccess_{T_3} : pc_2)$ means that thread T_1 is preempted after it performs a synchronization call at program counter pc_0; then thread T_2 is scheduled and performs a memory access at program counter pc_1, after which thread T_3 is scheduled and performs a memory access at pc_2 that is racing with the previous memory access of T_1. The schedule trace also contains the absolute count of instructions executed by the program up to each preemption point. This is needed in order to perform precise replays when an instruction executes multiple times (e.g., a loop) before being involved in a data race; this is not shown as part of the schedule trace, for brevity. The log of system call inputs contains the non-deterministic program inputs (e.g., *gettimeofday*).

In a first analysis step (illustrated in Fig. 16.a), Portend replays the schedule in the trace up to the point where the data race occurs. Then it explores two different executions: one in which the original schedule is followed (the *primary*) and one in which the alternate ordering

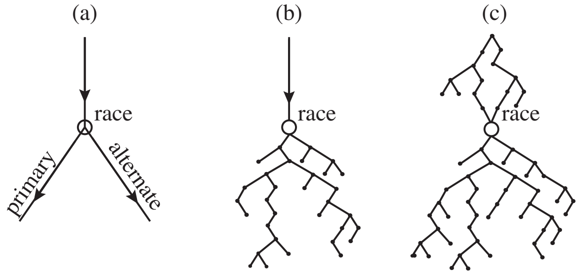
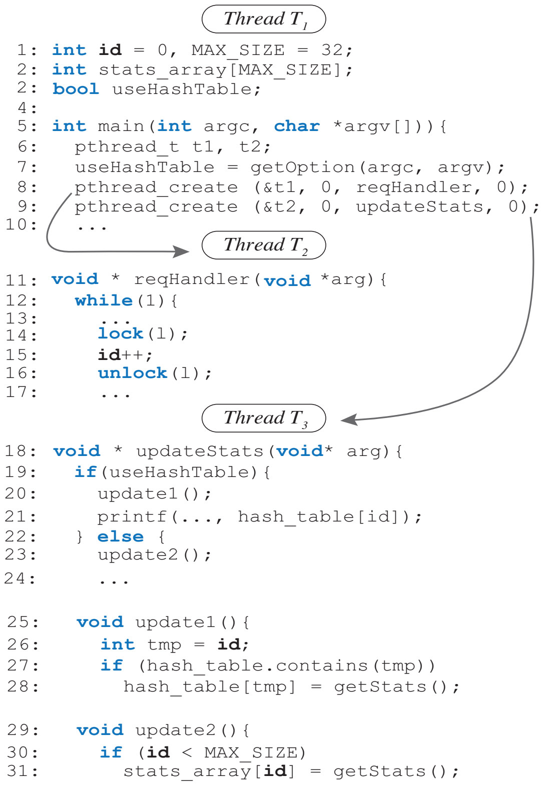


Figure 16 – Increasing levels of completeness in terms of paths and schedules: [a. single-pre/single-post] ≪ [b. single-pre/multi-post] ≪ [c. multi-pre/multi-post].

of the racing accesses is enforced (the *alternate*). As described in §2.6, some classifiers compare the primary and alternate program state immediately after the data race, and, if different, flag the data race as potentially harmful, and, if same, flag the data race as potentially harmless. Even if program outputs are compared rather than states, "single-pre/single-post" analysis (Fig. 16.a) may not be accurate, as we will show below. Portend uses "single-pre/single-post" analysis mainly to determine whether the alternate schedule is possible at all. In other words, this stage identifies any ad-hoc synchronization that might prevent the alternate schedule from occurring.

If there is a difference between the primary and alternate post-data race states, we do not consider the data race as necessarily harmful. Instead, we allow the primary and alternate executions to run, independently of each other, and we observe the consequences. If, for instance, the alternate execution crashes, the data race is harmful. Of course, even if the primary and alternate executions behave identically, it is still not certain that the data race is harmless: there may be some unexplored pair of primary and alternate paths with the same pre-data race prefix as the analyzed pair, but which does not behave the same. This is why single-pre/single-post analysis is insufficient, and we need to explore *multiple* post-data race paths. This motivates "single-pre/multi-post" analysis (Fig. 16.b), in which multiple post-data race execution possibilities are explored—if any primary/alternate mismatch is found, the developer must be notified.

Even if all feasible post-data race paths are explored exhaustively and no mismatch is found, one still cannot conclude that the data race is harmless: it is possible that the absence of a mismatch is an artifact of the specific pre-data race execution prefix, and that some different prefix would lead to a mismatch. Therefore, to achieve higher confidence in the classification, Portend explores multiple feasible paths even in the pre-data race stage, not just the one path witnessed by the data race detector. This is illustrated as "multi-pre/multi-post" analysis in Fig. 16c. The advantage of doing this vs. considering



```
int id = 0, MAX_SIZE = 32;
int stats_array[MAX_SIZE];
bool useHashTable;

int main(int argc, char *argv[]){
  pthread_t t1, t2;
  useHashTable = getOption(argc, argv);
  pthread_create (&t1, 0, reqHandler, 0);
  pthread_create (&t2, 0, updateStats, 0);
  ...

void * reqHandler(void *arg){
  while(1){
    ...
    lock(l);
    id++;
    unlock(l);
    ...

void * updateStats(void* arg){
  if(useHashTable){
    update1();
    printf(..., hash_table[id]);
  } else {
    update2();
    ...

  void update1(){
    int tmp = id;
    if (hash_table.contains(tmp))
      hash_table[tmp] = getStats();

  void update2(){
    if (id < MAX_SIZE)
      stats_array[id] = getStats();
```

Figure 17 – Simplified example of a harmful data race from Ctrace [132] that would be classified as harmless by classic data race classifiers.

these as different data races is the ability to systematically explore these paths.

Finally, we combine multi-path analysis with *multi-schedule* analysis, since the same path through a program may generate different outputs depending on how its execution segments from different threads are interleaved. The branches of the execution tree in the post-race execution in Fig. 16.c correspond to different paths that stem from both multiple inputs and schedules, as we detail in §5.6.

Of course, exploring all possible paths and schedules that experience the data race is impractical, because their number typically grows exponentially with the number of threads, branches, and preemption points in the program. Instead, we provide developers a "dial" to control the number k of path/schedule alternatives explored during analysis, allowing them to control the "volume" of paths and schedules in Fig. 16. If Portend classifies a data race as "k-witness harmless", then a higher value of k offers higher confidence that the data race is harmless for *all* executions (i.e., including the unexplored ones), but it entails longer analysis time. We found $k = 5$ to be sufficient for achieving 99% accuracy in our experiments in less than 5 minutes per data race on average.

To illustrate the benefit of multi-path multi-schedule analysis over "single-pre/single-post" analysis, consider the code snippet in Fig. 17, adapted from a real data race bug. This code has racing accesses to the global variable *id*. Thread T_1 spawns threads T_2 and T_3; thread T_2 updates *id* (line 15) in a loop and acquires a lock each time. However, thread T_3, which maintains statistics, reads *id* without acquiring the lock—this is because acquiring a lock at this location would hurt performance, and statistics need not be precise. Depending on program input, T_3 can update the statistics using either the *update1* or *update2* functions (lines 20-23).

Say the program runs in Portend with input *–use-hash-table*, which makes *useHashTable=true*. Portend records the primary trace $(T_1 : pc_9) \to ... (T_2 \to RaceyAccess_{T_1} : pc_{15}) \to (T_3 \to RaceyAccess_{T_3} : pc_{26}) \to ... T_1$. This trace is fed to the first analysis step, which replays the trace with the same program input, except it enforces the alternate schedule $(T_1 : pc_9) \to ...(T_3 \to RaceyAccess_{T_3} : pc_{26}) \to (T_2 \to RaceyAccess_{T_2} : pc_{15}) \to ... T_1$. Since the printed value of *hash_table[id]* at line 21 would be the same for the primary and alternate schedules, a "single-pre/single-post" classifier would deem the data race harmless.

However, in the multi-path multi-schedule step, Portend explores additional paths through the code by marking program input as symbolic, i.e., allowing it to take on any permitted value. When the trace is replayed and Portend reaches line 19 in T_3 in the alternate schedule, *useHashTable* could be both true and false, so Portend splits the execution into two executions, one in which *useHashTable* is set to *true* and one in which it is *false*. Assume, for example, that *id* = 31 when checking the if condition at line 30. Due to the data race, *id* is incremented by T_2 to 32, which overflows the statically allocated buffer (line 31). Note that in this alternate path, there are two racing accesses on *id*, and we are referring to the access at line 31.

Portend detects the overflow (via Cloud9), which leads to a crashed execution, flags the data race as "spec violated", and provides the developer the execution trace in which the input is *–no-hash-table*, and the schedule is $(T_1 : pc_9) \to ...(T_3 \to RaceyAccess_{T_3} : pc_{30}) \to (T_2 \to RaceyAccess_{T_2} : pc_{15}) \to (T_3 : pc_{31})$. The developer can replay this trace in a debugger and fix the race.

Note that this data race is harmful only if the program input is *–no-hash-table*, the given thread schedule occurs, and the value of *id* is 31; therefore the crash is likely to be missed by a traditional single-path/single-schedule data race detector.

We now describe Portend's data race analysis in detail: Sections §5.3–§5.6 focus on the *exploration* part of the analysis, in which Portend looks for paths and schedules that reveal the nature of the data race, and §5.8 focuses on the *classification* part.

```
Input : Primary execution trace primary
Output : Classification result $\in \{specViol, outDiff, outSame, singleOrd\}$
1  current ← execUntilFirstThreadRacyAccess(primary)
2  preDataRaceCkpt ← checkpoint(current)
3  execUntilSecondThreadRacyAccess(current)
4  postDataRaceCkpt ← checkpoint(current)
5  current ← preDataRaceCkpt
6  preemptCurrentThread(current)
7  alternate ← execWithTimeout(current)
8  if alternate.timedOut then
9      if detectInfiniteLoop(alternate) then
10         return specViol
11     else
12         return singleOrd
13 else
14     if detectDeadlock(alternate) then
15         return specViol
16 primary ← exec(postDataRaceCkpt)
17 if detectSpecViol(primary) ∨ detectSpecViol(alternate) then
18     return specViol
19 if primary.output ≠ alternate.output then
20     return outDiff
21 else
22     return outSame
```

Algorithm 2 : Single-Pre/Single-Post Analysis (singleClassify)

5.3 SINGLE-PATH ANALYSIS

The goal of this first analysis step is to identify cases in which the alternate schedule of a data race cannot be pursued, and to make a first classification attempt based on a single alternate execution. Algorithm 2 describes the approach.

Portend starts from a trace of an execution of the target program, containing one or more data races, along with the program inputs that generated the trace. For example, in the case of the pbzip2 file compressor used in our evaluation, Portend needs a file to compress and a trace of the thread schedule.

As mentioned earlier, such traces are obtained from running, for instance, the developers' test suites (as done in CHESS [137]) with a dynamic data race detector enabled.

Portend takes the primary trace and plays it back (line 1). Note that *current* represents the system state of the current execution. Just before the first racing access, Portend takes a checkpoint of system state; we call this the *pre-data race checkpoint* (line 2). The replay is then allowed to continue until immediately after the second racing access of the data race we are interested in (line 3), and the primary execution is suspended in this post-data race state (line 4).

Portend then primes a new execution with the pre-data race9 checkpoint (line 5) and attempts to enforce the alternate ordering of the racing accesses. To enforce this alternate order, Portend preempts the thread that did the first racing access (T_i) in the primary execution and allows the other thread (T_j) involved in the data race to be scheduled (line 6). In other words, an execution with the trace ...$(T_i \rightarrow \mathit{RaceyAccess}_{T_i} : pc_1) \rightarrow (T_j \rightarrow \mathit{RaceyAccess}_{T_j} : pc_2)$... is steered toward the execution ...$(T_j \rightarrow \mathit{RaceyAccess}_{T_j} : pc_2) \rightarrow (T_i \rightarrow \mathit{RaceyAccess}_{T_i} : pc_1)$...

This attempt could fail for one of three reasons: (a) T_j gets scheduled, but T_i cannot be scheduled again; or (b) T_j gets scheduled but $\mathit{RaceyAccess}_{T_j}$ cannot be reached because of a complex locking scheme that requires a more sophisticated algorithm [94] than Algorithm 2 to perform careful scheduling of threads; or (c) T_j cannot be scheduled, because it is blocked by T_i. Case (a) is detected by Portend via a timeout (line 8) and is classified either as "spec violated", corresponding to an infinite loop (i.e., a loop with a loop-invariant exit condition) in line 10 or as ad-hoc synchronization in line 12. Portend does not implement the more complex algorithm mentioned in (b), and this may cause it to have false positives in data race classification. However, we have not seen this limitation impact the accuracy of data race classification for the programs in our evaluation. Case (c) can correspond to a deadlock (line 15) and is detected by Portend by keeping track of the lock graph. Both the infinite loop and the deadlock case cause the data race to be classified as "spec violated", while the ad-hoc synchronization case classifies the data race as "single ordering" (more details in §5.8). While it may make sense to not stop if the alternate execution cannot be enforced, under the expectation that other paths with other inputs might permit the alternate ordering, our evaluation suggests that continuing adds little value (§6).

If the alternate schedule succeeds, Portend executes it until it completes, and then records its outputs. Then, Portend allows the primary to continue (while replaying the input trace) and also records its outputs. During this process, Portend watches for "basic" specification violations (crashes, deadlocks, memory errors, etc.) as well as "high level" properties given to Portend as predicates—if any of these properties are violated, Portend immediately classifies (line 18) the data race as "spec violated". If the alternate execution completes with no specification violation, Portend compares the outputs of the primary and the alternate; if they differ, the data race is classified as "output differs" (line 20), otherwise the analysis moves to the next step. This is in contrast to replay-based classification [142], which compares the program state immediately after the data race in the primary and alternate interleavings.

5.4 MULTI-PATH ANALYSIS

The goal of this step is to explore variations of the single paths found in the previous step (i.e., the primary and the alternate) in order to expose Portend to a wider range of execution alternatives.

First, Portend finds multiple primary paths that satisfy the input trace, i.e., they (a) all experience the same thread schedule (up to the data race) as the input trace, and (b) all experience the target data race condition. These paths correspond to *different* inputs from the ones in the initial race report. Second, Portend uses Cloud9 to record the "symbolic" outputs of these paths—that is, the constraints on the output, rather than the concrete output values themselves—as well as path constraints when these outputs are made, and compares them to the outputs and path constraints of the corresponding alternate paths; we explain this below. Algorithm 3 describes the functions invoked by Portend during this analysis in the following order: 1) on initialization, 2) when encountering a thread preemption, 3) on a branch that depends on symbolic data, and 4) on finishing an execution.

Unlike in the single-pre/single-post step, Portend now executes the primary *symbolically*. This means that the target program is given symbolic inputs instead of regular concrete inputs. Cloud9 relies in large part on KLEE [31] to interpret the program and propagate these symbolic values to other variables, corresponding to how they are read and operated upon. When an expression with symbolic content is involved in the condition of a branch, *both* options of the branch are explored, if they are feasible. The resulting path(s) are annotated with a constraint indicating that the branch condition holds true (respectively false). Thus, instead of a regular single-path execution, we get a tree of execution paths, similar to the one in Fig. 18. Conceptually, at each such branch, program state is duplicated and constraints on the symbolic parameters are updated to reflect the decision taken at that branch (line 11). Describing the various techniques for performing symbolic execution efficiently [31, 29] is beyond the scope of this article.

An important concern in symbolic execution is "path explosion," i.e., that the number of possible paths is large. Portend offers two parameters to control this growth: (a) an upper bound M_p on the number of primary paths explored; and (b) the number and size of symbolic inputs. These two parameters allow developers to trade performance vs. classification confidence. For parameter (b), the fewer inputs are symbolic, the fewer branches will depend on symbolic input, so less branching will occur in the execution tree.

Determining the optimal values for these parameters may require knowledge of the target system as well as a good sense of how much confidence is required by the system's users. Reasonable (i.e., good but not necessarily optimal) values can be found through trial and

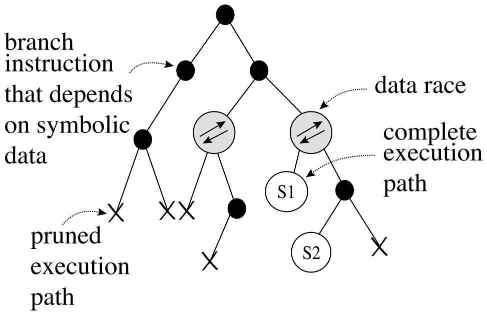


Figure 18 – Portend prunes paths during symbolic execution.

error relatively easily—we expect development teams using Portend to converge onto values that are a good fit for their code and user community, and then make these values the defaults for their testing and triage processes. We empirically study in §6.3 the impact of these parameters on classification accuracy on a diverse set of programs and find that relatively small values achieve high accuracy for a broad range of programs.

During symbolic execution, Portend prunes (Fig. 18) the paths that do not obey the thread schedule in the trace (line 8), thus excluding the (many) paths that do not enable the target data race. Moreover, Portend attempts to follow the original trace only until the second racing access is encountered; afterward, it allows execution to diverge from the original schedule trace. This enables Portend to find more executions that partially match the original schedule trace (e.g., cases in which the second racing access occurs at a different program counter, as in Fig. 17). Tolerating these divergences significantly increases Portend's accuracy over the state of the art [142], as will be explained in §6.3.5.

Once the desired paths are obtained (at most M_p, line 14), the conjunction of branch constraints accumulated along each path is solved by KLEE using an SMT solver [65] in order to find concrete inputs that drive the program down the corresponding path. For example, in the case of Fig. 18, two successful leaf states S_1 and S_2 are reached, and the solver provides the inputs corresponding to the path from the root of the tree to S_1, respectively S_2. Thus, we now have $M_p = 2$ different primary executions that experience the data race.

5.5 SYMBOLIC OUTPUT COMPARISON

Portend now records the output of each of the M_p executions, like in the single-pre/single-post case, and it also records the path constraints when these outputs are made. However, this time, in addition to simply recording concrete outputs, Portend propagates the constraints on symbolic state all the way to the outputs, i.e., the outputs of each primary execution contain a mix of concrete values and

```
   Input : Schedule trace trace, initial program state $S_0$, set of states
           $S = \emptyset$, upper bound $M_p$ on the number of primary paths
   Output : Classification result $\in \{specViol, outDiff, singleOrd\ k\text{-}witness\}$
1  function init ()
2    $S \leftarrow S \cup S_0$
3    $current \leftarrow S.head()$
4    $pathsExplored \leftarrow 0$
5  function onPreemption ()
6    $t_i \leftarrow$ scheduleNextThread$(current)$
7    if $t_i \neq$ nextThreadInTrace$(trace, current)$ then
8        $S \leftarrow S.remove(current)$
9        $current \leftarrow S.head()$
10 function onSymbolicBranch ()
11   $S \leftarrow S \cup current.fork()$
12 function onFinish ()
13   $classification \leftarrow classification \cup$ classify$(current)$
14   if $pathsExplored < M_p$ then
15       $pathsExplored \leftarrow pathsExplored + 1$
16 else
17     return classification
18 function classify (primary)
19   $result \leftarrow$ singleClassify$(primary)$
20   if $result = outSame$ then
21       $alternate \leftarrow$ getAlternate$(primary)$
22       if symbolicMatch$(primary.symState, alternate.symState)$ then
23           return k-witness
24       else
25           return outDiff
26 else
27     return result
```

Algorithm 3 : Multi-path Data Race Analysis (Simplified)

symbolic constraints (i.e., symbolic formulae). Note that by output, we mean all arguments passed to output system calls.

Next, for each of the M_p executions, Portend produces a corresponding alternate (analogously to the single-pre/single-post case). The alternate executions are fully concrete, but Portend records constraints on the alternate's outputs (lines 19-21) as well as the path constraints when these outputs are made. The function *singleClassify* in Algorithm 3 performs the analysis described in Algorithm 2. Portend then checks whether the constraints on outputs of each alternate and the path constraints when these outputs are made match the constraints of the corresponding primary's outputs and the path constraints when primary's outputs are made. This is what we refer to as *symbolic output comparison* (line 22). The purpose behind comparing symbolic outputs is that Portend tries to figure out if the data race caused the constraints on the output to be modified or not, and the purpose behind comparing path constraints is to be able generalize the output comparison to more possible executions with different inputs.

This symbolic comparison enables Portend's analysis to extend over more possible primary executions. To see why this is the case, consider the example in Figure 19. In this example, the $Main$ thread reads input to the shared variable i and then spawns two threads T_1 and T_2 which perform racing writes to $globalx$. T_1 prints 10 if the input is positive. Let us assume that during the symbolic execution of the primary, the write to $globalx$ in T_1 is performed before the write to $globalx$ in T_2. Portend records that the output at line 6 is 10 if the path constraint is $i \geqslant 0$. Let us further assume that Portend runs the program while enforcing the alternate schedule with input 1. The output of the program will still be 10 (since $i \geqslant 0$) and the path constraint when the output will be made will still be $i \geqslant 0$. The output and the path constraint when the output is made is therefore the same regardless of the order with which the accesses to $globalx$ are performed (i.e., the primary or the alternate order). Therefore, Portend can assert that the program output for $i \geqslant 0$ will be 10 regardless of the way the data race goes even though it only explored a single alternate ordering with input 1.

This comes at the price of potential false negatives because path constraints can be modified due to a different thread schedule; despite this theoretical shortcoming, we have not encountered such a case in practice, but we plan to investigate this further in future work.

False negatives can also arise because determining semantic equivalence of output is undecidable, and our comparison may still wrongly classify as "output differs" a sequence of outputs that are equivalent at some level (e.g., *<print ab; print c> vs. <print abc>*).

When executing the primaries and recording their outputs and the path constraints, Portend relies on Cloud9 to track all symbolic constraints on variables. To determine if the path constraints and constraints on outputs match for the primary and the alternates, Portend directly employs an SMT solver [65].

As will be seen in §6.3, using symbolic comparison in conjunction with multi-path multi-schedule analysis leads to substantial improvements in classification accuracy.

We do not detail here the case when the program reads input *after* the data race—it is a natural extension of the algorithm above.

5.6 MULTI-SCHEDULE ANALYSIS

The goal of multi-schedule analysis is to further augment the set of analyzed executions by diversifying the thread schedule.

We mentioned earlier that, for each of the M_p primary executions, Portend obtains an alternate execution. Once the alternate ordering

```
int globalx = 0;
int i = 0;
                    Thread T_1
void * work0 (void *arg) {
  globalx = 1;
  if(i >= 0)
     printf(”10\n”);
  return 0;
}
                    Thread T_2
void * work1 (void *arg) {
  globalx = 2;
  return 0;
}
                  Main Thread
int main (int argc, char *argv[]){
  pthread_t t0, t1;
  int rc;
  i = getInput(argc, argv)
  rc = pthread_create (&t0, 0, work0, 0);
  rc = pthread_create (&t1, 0, work1, 0);
  pthread_join(t0, 0);
  pthread_join(t1, 0);
  return 0;
}
```

Figure 19 – A program to illustrate the benefits of symbolic output comparison

of the racing accesses is enforced, Portend randomizes the schedule of the *post-race* alternate execution: at every preemption point in the alternate, Portend randomly decides which of the runnable threads to schedule next. This means that every alternate execution will most likely have a different schedule from the original input trace (and thus from the primary).

Consequently, for every primary execution P_i, we obtain *multiple* alternate executions $A_i^1, A_i^2, ...$ by running up to M_a multiple instances of the alternate execution. Since the scheduler is random, we expect practically every alternate execution to have a schedule that differs from all others. Recently proposed techniques [136] can be used to quantify the probability of these alternate schedules discovering the harmful effects of a data race.

Portend then uses the same symbolic comparison technique as in §5.5 to establish equivalence between the constraint on outputs and path constraints of $A_i^1, A_i^2, ...A_i^{M_a}$ and the symbolic outputs and path constraints of P_i.

Schedule randomization can be employed also in the pre-data race stage of the alternate-execution generation as well as in the generation of the primary executions. We did not implement these options, because the level of multiplicity we obtain with the current design appears to be sufficient in practice to achieve high accuracy. Note however that, as we show in §6.3, multi-path multi-schedule analysis is indeed crucial to attaining high classification accuracy.

In summary, multi-path multi-schedule analysis explores M_p primary executions and, for each such execution, M_a alternate executions with different schedules, for a total of $M_p \times M_a$ path-schedule combinations. For data races that end up being classified as "k-witness harmless", we say that $k = M_p \times M_a$ is the lower bound on the number of concrete path-schedule combinations under which this data race is harmless.

Note that the k executions can be simultaneously explored in parallel: if a developer has p machines with q cores each, she could explore $p \times q$ parallel executions in the same amount of time as a single execution. Given that Portend is "embarrassingly parallel," it is appealing for cluster-based automated bug triage systems.

5.7 SYMBOLIC MEMORY CONSISTENCY MODELING

Modern processor architectures rarely assume sequential consistency as this would hurt program performance. Instead, they adopt relaxed memory consistency models like weak ordering [49] and rely on the programmers to explicitly specify orderings among program statements using synchronization primitives.

Previous work has shown that subtle bugs may arise in code with data races because programmers make assumptions based on sequential consistency. despite the fact that none of the modern processors provide sequential consistency [57]. Such assumptions may be violated under relaxed consistency models, and bugs that are deemed unlikely may appear when the program is running on various CPUs causing programs to crash, hang or violate some given specification of a program.

Therefore, a program analysis tool should ideally have the capability to reason about different memory models and their effects on the performed analysis. The effect of the memory model on the consequences of a data race are serious: code written with the assumption of a particular memory model may end up computing wrong results; or worse, it can crash or cause data loss [57].

Why Does the Memory Model Matter?

In order to better show why reasoning about relaxed memory consistency models matters while performing program analysis and testing, let us consider the example in Fig 20. There are two shared variables `globalx` and `globaly` that both have an initial value of 0. There is a $Main$ thread that spawns two threads T_1 and T_2. T_1 writes 2 to a global variable `globalx` and 1 to another global variable `globaly`. T_2 writes 2 to `globalx`. Then, the $Main$ thread reads the value of the global variables. If the read value of `globalx` and `globaly` are 0 and 1 respectively, the program crashes on line 18.

Programmers naturally expect the program statements to be executed in the order as they appear in the program text. A programmer making that assumption expects that the value of `globaly` being 1 implies the value of `globalx` being 2. This assumption is equivalent to assuming sequential consistency as the underlying memory model: if sequential consistency is assumed as the underlying memory model for the execution of this program, the value of `globalx` cannot be 0 when the value of `globaly` is 1. This is simply because the order of the program text would require `globalx` to be 2.

```
1:  int volatile globalx = 0;
2:  int volatile globaly = 0;
                   Thread T1
3:  void * work0 (void *arg) {
4:    globalx = 2;
5:    globaly = 1;
6:    return 0;
7:  }
                   Thread T2
8:  void * work1 (void *arg) {
9:    globalx = 2;
10:   return 0;
11: }
                  Main Thread
12: int main (int argc, char *argv[]){
13:   pthread_t t0, t1;
14:   int rc;
15:   rc = pthread_create (&t0, 0, work0, 0);
16:   rc = pthread_create (&t1, 0, work1, 0);
17:   if(globalx == 0 && globaly == 1)
18:     ; //crash!
19:   pthread_join(t0, 0);
20:   pthread_join(t1, 0);
21:   return 0;
22: }
```

Figure 20 – Simple multithreaded program

Under a different memory model such as weak ordering [49], nothing prevents the write to `globalx` on line 4 and the write to `globaly` on line 5 to swap places. This stems from the fact that, under weak consistency, if instructions are not conflicting, and they are not ordered by synchronization operations, then any reordering is allowed. In such a scenario, it is possible for T_1 to write 1 to `globaly` while the value of `globalx` is still 0. Furthermore, there is a data race between the write to `globalx` in T_1 and the read from it in $Main$. This means that T_1 can be preempted right after setting `globaly` to 1 and `globaly` and `globalx` can be equal to 1 and 0 respectively. This can cause the program to crash on line 18.

Limitations of Cloud9 as a Multithreaded Testing Platform

Portend is built on Cloud9, which is a multithreaded parallel symbolic execution engine [29]. Cloud9 is essentially an LLVM interpreter which can concretely interpret programs compiled to LLVM, and during this interpretation, it can also keep track of the symbolic values and constraints.

Cloud9 makes the following sequential consistency assumptions: 1) Uniprocessor scheduler: Cloud9 scheduler picks threads in a round robin fashion and runs them by interpreting their text until an opportunity for scheduling arises (such as a sleep or a synchronization operation). 2) Immediate updates to shared memory: Shared memory is modeled as a flat structure with no cache model; therefore, any update to a shared memory location is immediately visible to all other threads. 3) No instruction reordering: Cloud9 interpretation engine works by fetching instructions from the LLVM binary and executing them sequentially without any instruction reordering.

Since shared memory updates are not reordered, and they are directly visible to all threads, and threads are scheduled one after the other, one at a time, any analysis that builds on Cloud9 is bound to perform the analysis within the confines of sequential consistency. However, as it was previously demonstrated, such an analysis may be unable to expose insidious bugs.

Symbolic Memory Consistency Modeling in Portend

Previously, we showed how Portend explores multiple paths and schedules in order to observe the consequences of a data race. The goal of SMCM is to further augment multi-path multi-schedule analysis to factor in the effects of the underlying memory model. SMCM is in essence similar to multi-path analysis: multi-path analysis explores multiple execution paths that the execution could take due to different program input values; SMCM explores multiple paths that the execution could take due to different values that could be read from the shared memory.

SMCM has two main components: the Lamport clock tracker and the SMCM plugin.

The first component is the Lamport clock tracker. Lamport clocks are logical counters that maintain a partial order among synchronization operations in order to determine the relative occurrence sequence of events in a concurrent program [110]. This order is partial because an order is only present among related synchronization operations (e.g., a lock and an unlock on the same lock).

Lamport clocks are maintained per synchronization operation. A thread's Lamport clock is equal to the greatest of the clocks of all the events that occur in the thread. Lamport clocks are incremented under the following conditions:

- Each thread increments the clock of an event before the occurrence of that event in that thread.
- When threads communicate, they also communicate their clocks (upon fork/join or wait/signal)
- The thread that receives a clock sets its own clock to be greater than the maximum of its own clock or that of the received message.

The graph that captures the relative order of events in a concurrent program using Lamport clocks is called a happens-before graph. Figure 21 shows an example happens-before graph and the Lamport clocks associated with a given execution. Note that `locks` and `unlocks` on lock `l` induce a happens-before edge denoted by the arrow between Thread 1 and Thread 2. On the other hand, the lock/unlock block on lock `k` does not have any partial order with any of the events in Thread 2; therefore, although the current timeline shows it as occurring before the lock/unlock block in Thread 2, in some other execution it can occur after that lock/unlock block.

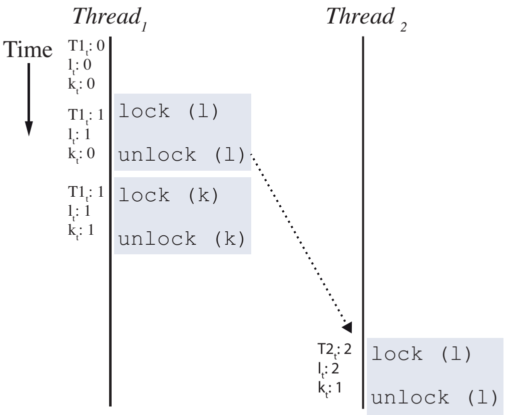


Figure 21 – Lamport clocks and a happens-before graph

Lamport clock tracker needs to monitor the synchronization operations performed by each thread in order to construct the happens-before graph. All synchronization events are intercepted, and the happens-before graph is constructed behind the scenes according to the rules that were previously stated, while the program is being executed.

Lamport clock tracker is a critical component of Portend since it actually forms a well defined ordering among events during program execution that stem from synchronization operations. This is important because different memory models and reordering constraints are defined using synchronization operations in programs.

The second component, namely the SMCM plugin, defines the memory model protocol according to which a read returns previously-written values.

The memory model protocol encodes the rules of the particular memory model that Portend uses. In our case, we define two such protocols: one default protocol for sequential consistency and another one for weak consistency. We previously described the semantics of sequential consistency. Under Portend's weak memory consistency, a read R may see a previous write A, provided that there is no other write B such that B happened before R and A happened before B, with the exception that within the same thread, a sequence of reads from the same variable with no intervening writes to that variable will read the same value as the first read. We call this exception in Portend's weak memory model *write buffering*. Write buffering is responsible for keeping a write history for each shared memory location. Write buffering enables Portend to compute a subset of the values written to a memory location, when that location is read. That subset is computed considering the happens-before graph that is generated during program execution by the Lamport clock tracker.

Similar forms of weak memory consistency has been implemented in architectures such as SPARC [196] and Alpha [176].

To see how write buffering and the memory model protocol works, consider the example given in Figure 22. Again, vertical order of events imply the order of events in time. In this execution, Thread 2 writes 0 to `globalx`, then execution switches to Thread 3, which writes 2 to `globalx` and 1 to `globaly` before the execution switches to Thread 1. Then, Thread 1 writes 3 to `globalx` after a lock/unlock region on `l` and finally execution switches back to Thread 2 which reads both `globalx` and `globaly` while holding the same lock `l`.

So what values do Thread 2 read? Note that since both `globalx` and `globaly` are shared variables, the CPU can buffer all the values that were written to `globalx` $(0, 2, 3)$ and `globaly` (1). For `globaly`, the only value that can be read is 1. Now, when the value `globalx` is read, Portend knows that, under its weak memory consistency model, the values that can be read are $0, 2$ and 3. This is because there is no ordering constraint (a happens-before edge) that prevents from making those three write values readable at the point of the read.

Then, Portend will use these multiple possible reads to augment multi-path analysis: Portend will split the execution to as many possible "read" values there are, by checkpointing the execution state prior to the read and binding each one of those possible "read"s to one such checkpointed state's thread. By binding a read value, we mean copying the value in question into the checkpointed state's memory. Therefore, in this case there will be three such forked states: One with values $(0, 1)$, one with $(2, 1)$ and the other with values $(3, 1)$ corresponding to (`globalx`, `globaly`). Portend will continue exploring the forked states, forking further if the threads in the states read global variables that can potentially return multiple values.

If an already-bound global value is read by the same thread in a state without being altered after the last time it had been read, Portend makes sure to return the already-bound value. This is a necessary mechanism to avoid false positives (a thread reading two different values in a row with no intervening writes) due to write buffering. This is achieved by maintaining a last reader thread ID field per write buffer.

Write buffering and the optimization we use to bind a read value to a thread are performed for a given thread schedule that Portend explores at a time. For example, in Figure 22, if Thread 2 were to read `globalx` twice, it would have been possible for the first read to return 2 and the second read to return 3 (or vice versa) if there had been an intervening write between the two reads. Portend relies on multi-schedule data race analysis to handle this case, rather than relying on SMCM to reason about potential prior thread schedules that would lead to such a behavior.

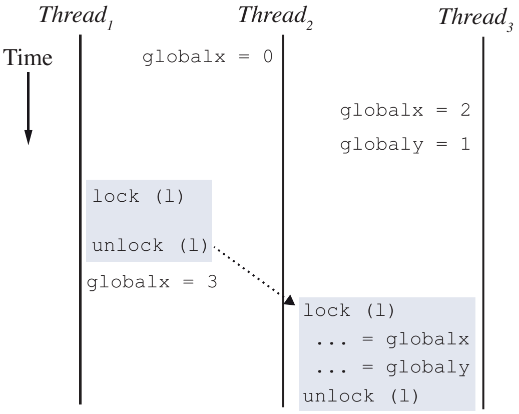


Figure 22 – Write Buffering

This example demonstrates the power of SMCM in reasoning about weak ordering. Note that, if sequential consistency was assumed for the given sequence of events, there would not have been a scenario where the value of `globaly` is 1 whereas the value of `globalx` is 0. This is because the given sequence of events would imply that writing 2 to `globalx` in Thread 3 occurs before writing 1 to `globaly` in the same thread. However, this is not the case under weak consistency. Since there is no synchronization enforcing the ordering of the write to `globalx` and `globaly` in Thread 3, these accesses can be reordered. Therefore it is perfectly possible for Thread 2 to see the value of `globaly` as 1 and `globalx` as 0.

5.8 CLASSIFICATION VERDICTS

We showed how Portend explores paths and schedules to give the classifier an opportunity to observe the effects of a data race. We now provide details on how the classifier makes its decisions.

"Spec violated" data races cause a program's explicit specification to be violated; they are guaranteed to be harmful and thus should have highest priority for developers. To detect violations, Portend watches for them during exploration.

First, Portend watches for "basic" properties that can be safely assumed to violate any program's specification: crashes, deadlocks, infinite loops, and memory errors. Since Portend already controls the program's schedule, it also keeps track of all uses of synchronization primitives (i.e., POSIX threads calls); based on this, it determines when threads are deadlocked. Infinite loops are diagnosed as in [206], by detecting loops for which the exit condition cannot be modified. For memory errors, Portend relies on the mechanism already provided by KLEE inside Cloud9. Even when Portend runs the program concretely, it still interprets it in Cloud9.

Second, Portend watches for "semantic" properties, which are provided to it by developers in the form of assert-like predicates. Developers can also place these assertions inside the code.

Whenever an alternate execution violates a basic or a semantic property (even though the primary may not), Portend classifies the corresponding data race as "spec violated".

"Output differs" data races cause a program's output to depend on the ordering of the racing accesses. As explained previously, a difference between the post-data race memory or register states of the primary and the alternate is not necessarily indicative of a harmful race (e.g., the difference may just be due to dynamic memory allocation). Instead, Portend compares the outputs of the primary and the alternate, and it does so symbolically, as described earlier. In case of a mismatch, Portend classifies the race as "output differs" and gives the developer detailed information to decide whether the difference is harmful or not.

"K-witness harmless" data races: If, for every primary execution P_i, the constraints on the outputs of alternate executions $A_i^1, A_i^2 ... A_i^{M_a}$ and the path constraints when these outputs are made, match P_i's output and path constraints, then Portend classifies the data race as "k-witness harmless", where $k = M_p \times M_a$, because there exist k executions witnessing the conjectured harmlessness. The value of k is often an underestimate of the number of different executions for which the data race is guaranteed to be harmless; as suggested earlier in §5.1, symbolic execution can even reason about a virtually infinite number of executions.

Theoretical insights into how k relates to the confidence a developer can have that a "k-witness harmless" race will not cause harm in practice are beyond the scope of this article. One can think of k in ways similar to code coverage in testing: 80% coverage is better than 60%, but does not exactly predict the likelihood of bugs not being present. For all our experiments, $k = 5$ was shown to be sufficient for achieving 99% accuracy. We consider "k-witness harmless" analyses to be an intriguing topic for future work, in a line of research akin to [136]. Note that Portend explores many more executions before finding the required k path-schedule combinations that match the trace, but the paths that do not match the trace are pruned early during the analysis.

"Single ordering" data races may be harmless data races if the ad-hoc synchronization is properly implemented. In that case, one might even argue they are not data races at all. Yet, dynamic data race detectors are not aware of the implicit happens-before relationship and do report a data race, and our definition of a data race (§2.1.1) considers these reports as data races.

When Portend cannot enforce an alternate interleaving in the single-pre/single-post phase, this can either be due to ad-hoc synchronization that prevents the alternate ordering, or the other thread in question cannot make progress due to a deadlock or an infinite loop. If none of the previously described infinite-loop and deadlock detection mechanisms trigger, Portend simply waits for a configurable amount of time and, upon timeout, classifies the data race as "single ordering." Note that it is possible to improve this design with a heuristic-based static analysis that can in some cases identify ad-hoc synchronization [206, 186].

5.9 PORTEND'S DEBUGGING AID OUTPUT

To help developers decide what to do about an "output differs" data race, Portend dumps the output values and the program locations where the output differs. Portend also aims to help in fixing harmful data races by providing for each data race two items: a textual report and a pair of execution traces that evidence the effects of the data race and can be played back in a debugger, using Portend's runtime replay environment. A simplified report is shown in Fig. 23.

```
Data Race during access to: 0x2860b30
current thread id: 3: READ
racing thread id: 0: WRITE
Current thread at:
  /home/eval/pbzip/pbzip2.cpp:702
Previous at:
  /home/eval/pbzip/pbzip2.cpp:389
size of the accessed field: 4 offset: 0
```

Figure 23 – Example debugging aid report for Portend.

In the case of an "output differs" data race, Portend reports the stack traces of system calls where the program produced different output, as well as the differing outputs. This simplifies the debugging effort (e.g., if the difference occurs while printing a debug message, the data race could be classified as benign with no further analysis).

5.10 PORTEND'S IMPLEMENTATION DETAILS

Portend works on programs compiled to LLVM [113] bitcode and can run C/C++ programs for which there exists a sufficiently complete symbolic POSIX environment [29]. We have tested Portend on

C programs as well as C++ programs that do not link to libstdc++; we leave linking programs against an implementation of a standard C++ library for LLVM [42] as future work. Portend uses Cloud9 [29] to interpret and symbolically execute LLVM bitcode; we suspect any path exploration tool will do (e.g., CUTE [173], SAGE [68], EXE [32], ESD [213], S2E [37, 38, 39]), as long as it supports multi-threaded programs.

Portend intercepts various system calls, such as *write*, under the assumption that they are the primary means by which a program communicates changes in its state to the environment. A separate Portend module is responsible for keeping track of symbolic outputs in the form of constraints, as well as of concrete outputs. Portend hashes program outputs (when they are concrete) and can either maintain hashes of all concrete outputs or compute a hash chain of all outputs to derive a single hash code per execution. This way, Portend can deal with programs that have a large amount of output.

Portend keeps track of Lamport clocks per execution state it explores on the fly. Note that it is essential to maintain the happens-before graph per execution state because threads may get scheduled differently depending on the flow of execution in each state state and therefore synchronization operations may end up being performed in a different order.

The state space exploration in Portend is exponential in the number of values that "read"s can return in a program. Therefore the implementation needs to handle bookkeeping as efficiently as possible. There are several optimizations that are in place to enable a more scalable exploration. The most important ones are: 1) Copy-on-write for keeping the happens-before graph and 2) Write buffer compression. Portend can use other techniques for taming state space explosion such as State Merging [108] in the future.

Portend employs copy-on-write for tracking the happens-before graphs in various states. Initially, there is a single happens-before graph that gets constructed during program execution before any state is forked due to a read with multiple possible return values. Then, when a state is forked, the happens-before graph is not duplicated. The forking state rather maintains a reference to the old graph. Then, when a new synchronization operation is recorded in either one of the forked states, this event is recorded as an incremental update to previously saved happens-before graph. In this way maximum sharing of the happens-before graph is achieved among forked states.

The copy-on-write scheme for states can be further improved if one checks whether two different states perform the same updates to the happens-before graph. If that is the case, these updates can be merged and saved as part of the common happens-before graph.

This feature is not implemented in the current prototype, but it is a potential future optimization.

The second optimization is write buffer compression. This is performed whenever the same value is written to the same shared variable's buffer and the constraints imposed by the happens-before relationship allow these same values to be returned upon a read. Then, in such a case, these two writes are compressed into one, as returning two of them would be redundant from the point of view of state exploration. For example, if a thread writes 1 to a shared variable `globalx` twice before this value is read by another thread, the write buffer will be compressed to behave as if the initial thread has written 1 once.

Portend clusters the data races it detects, in order to filter out similar races; the clustering criterion is whether the racing accesses are made to the same shared memory location by the same threads, and the stack traces of the accesses are the same. Portend provides developers with a single representative data race from each cluster.

The timeout used in discovering ad-hoc synchronization is conservatively defined as 5 times what it took Portend to replay the primary execution, assuming that reversing the access sequence of the racing accesses should not cause the program to run for longer than that.

In order to run multi-threaded programs in Portend, we extended the POSIX threads support found in Cloud9 to cover almost the entire POSIX threads API, including barriers, mutexes and condition variables, as well as thread-local storage. Portend intercepts calls into the POSIX threads library to maintain the necessary internal data structures (e.g., to detect data races and deadlocks) and to control thread scheduling.

6

EVALUATION

In this section, we evaluate all the prototypes we built for all the techniques we presented in the three previous chapters. For each prototype evaluation, we first describe the experimental setup followed with a description of prototype-specific experiments. We first present the evaluation results of RaceMob (§6.1). Next we present the evaluation results of Gist (§6.2), followed by the evaluation results of Portend (§6.3).

6.1 RACEMOB'S EVALUATION

In this section, we address the following questions about RaceMob: Can it effectively detect true races in real code (§6.1.2)? Is it efficient (§6.1.3)? How does it compare to state-of-the-art data race detectors (§6.1.4) and interleaving-based concurrency testing tools (§6.1.5)? Finally, how does RaceMob scale with the number of threads (§6.1.6)?

6.1.1 *Experimental Setup*

We evaluated RaceMob using a mix of server, desktop and scientific software: Apache httpd is a Web server that serves around 65% of the Web [6]—we used the `mpm-worker` module of Apache to operate it in multi-threaded server mode and detected data races in this specific module. SQLite [179] is an embedded database used in Firefox, iOS, Chrome, and Android, and has 100% branch coverage with developer's tests. Memcached [56] is a distributed memory-object caching system, used by Internet services like Twitter, Flickr, and YouTube. Knot [15] is a web server. Pbzip2 [66] is a parallel implementation of the popular `bzip2` file compressor. Pfscan [55] is a parallel file scanning tool that provides the combined functionality of `find`, `xargs`, and `fgrep` in a parallel way. Aget is a parallel variant of `wget`. Fmm, Ocean, and Barnes are applications from the SPLASH2 suite [175, 204]. Fmm and Barnes simulate interactions of bodies (n-body simulation), and Ocean simulates eddy currents in oceans.

Our evaluation results are obtained primarily using a test environment simulating a crowdsourced setting, and we also have a small scale, real deployment of RaceMob on our laptops. For the experiments, we use a mix of workloads derived from actual program runs, test suites, and test cases devised by us and other researchers [209]. We configured the hive to assign a single dynamic validation task per user at a time. Altogether, we have execution information from 1,754

Program		**Apache**	**SQLite**	**Memcached**	**Fmm**	**Barnes**	**Ocean**	**Pbzip2**	**Knot**	**Aget**	**Pfscan**
Size (LOC)		138,456	113,326	19,397	9,126	7,580	6,551	3,521	3,586	2,053	2,033
Race candidates		118	88	7	176	166	115	65	65	24	17
True Race	Causes hang	0	3	0	0	0	0	0	0	0	0
	Causes crash	0	0	0	0	0	0	3	0	0	0
	Both orders	0	0	1	5	10	0	2	0	0	0
	Single order	8	0	0	53	6	3	4	2	4	2
Likely FP	Not aliasing	10	31	0	33	65	13	0	18	2	0
	Context	61	10	2	61	28	42	21	28	10	4
	Synchronization	1	37	3	10	49	47	34	13	7	11
Unknown		38	7	1	14	8	10	1	4	1	0

Table 1 – Data race detection with RaceMob. The static phase reports *Data race candidates* (row 2). The dynamic phase reports verdicts (rows 3-10). *Causes hang* and *Causes crash* are data races that caused the program to hang or crash. *Single order* are true data races for which either the primary or the alternate executed (but not both) with no intervening synchronization; *Both orders* are data races for which both executed without intervening synchronization.

simulated user sites. Our test bed consists of a 2.3 GHz 48-core AMD Opteron 6176 machine with 512 GB of RAM running Ubuntu Linux 11.04 and a 2 GHz 8-core Intel Xeon E5405 machine with 20 GB of RAM running Ubuntu Linux 11.10. The hive is deployed on the 8-core machine, and the simulated users on both machines. The real deployment uses ThinkPad laptops with Intel 2620M processors and 8 GB of RAM, running Ubuntu Linux 12.04.

We used C programs in our evaluation because RELAY operates on CIL, which does not support C++ code. Pbzip2 is a C++ program, but we converted it to C by replacing references to STL `vector` with an array-based implementation. We also replaced calls to `new/delete` with `malloc/free`.

6.1.2 *Effectiveness*

To investigate whether RaceMob provides an effective way to detect data races, we look at whether RaceMob can detect true data races, and whether its false positive and false negative rates are sufficiently low.

RaceMob's data race detection results are centralized in Table 1. RaceMob detected a total of 106 data races in ten programs. Four data races in pbzip2 caused the program to crash, three data races in SQLite caused the program to hang, and one data race in Aget caused a data corruption (that we confirmed manually). The other data races did not lead to any observable failure. We manually confirmed that the "True Race" verdicts are correct, and that RaceMob has no false positives in our experiments.

The "Likely FP" row represents the data races that RaceMob identified as likely false positives: (1) *Not aliasing* are reports with accesses that do not alias to the same memory location at runtime; (2) *Context* are reports whose accesses are only made by a single thread at runtime; (3) *Synchronization* are reports for which, the accesses are synchronized, an artifact that the static detector missed. The first two sources of likely false positives (53% of all static reports) are identified using DCI, whereas the last source (24% of all static reports) is identified using on demand race detection. In total, 77% of all statically detected data races are likely false positives.

As we discussed in §3.5, RaceMob's false negative rate is determined by its static data race detector. We rely on prior work's results to partially confirm the absence of false negatives in RaceMob. In particular, Chimera [114], a deterministic record/replay system, relies on RELAY; for deterministic record/replay to work, all data races must be detected; in Chimera's evaluation (which included Apache, Pbzip2, Knot, Ocean, Pfscan, Aget), RELAY did not have any false negatives [114]. We therefore cautiously conclude that RaceMob's static phase had no false negatives in our evaluation. However, this does not exclude the possibility that for other programs there do exist false negatives.

For all the programs, we initially set the timeout for schedule steering to $\tau = 1$ ms. As timeouts fired during validation, the hive increased the timeout 50 ms at a time, up to a maximum of 200 ms. Developers may choose to modify this basic scheme depending on the characteristics of their programs. For instance, the timeout could be increased multiplicatively instead of linearly.

In principle, false negatives may also arise from τ being too low or from there being insufficient executions to prove a true data race. We increased τ in our experiments by $4\times$, to check if this would alter our results, and the final verdicts were the same. After manually examining data races that were not encountered during dynamic validation, we found that they were either in functions that are never called but are nonetheless linked to the programs, or they are not encountered at runtime due to the workloads used in our evaluation.

6.1.3 *Efficiency*

The more efficient a detector is, the less runtime overhead it introduces, i.e., the less it slows down a user's application (as a percentage of uninstrumented execution). The static detection phase is offline, and it took less than 3 minutes for all programs, except Apache and SQLite, for which it took less than 1 hour. Therefore, in this section, we focus on the dynamic phase.

Apache	SQLite	Memcached	Fmm	Barnes	Ocean	Pbzip2	Knot	Aget	Pfscan
1.74	1.60	0.10	4.54	2.98	2.05	2.90	1.27	3.00	3.03

Table 2 – Runtime overhead of data race detection as a percentage of uninstrumented execution. Average overhead is 2.32%, and maximum overhead is 4.54%.

Table 2 shows that runtime overhead of RaceMob is typically less than 3%. The static analysis used to remove instrumentation from empty loop bodies reduced our worst case overhead from 25% to 4.54%. The highest runtime overhead is 4.54%, in the case of Fmm, a memory-intensive application that performs repetitive computations, which gives the instrumentation more opportunity to introduce overhead. Our results suggest that there is no correlation between the number of data race candidates (row 2 in Table 1) and the runtime overhead (Table 2)—overhead is mostly determined by the frequency of execution of the instrumentation code.

Program	Apache	SQLite	Memcached	Fmm	Barnes	Ocean	Pbzip2	Knot	Aget	Pfscan
RaceMob	8	3	1	58	16	3	9	2	4	2
TSAN	8	3	0	58	16	3	9	2	2	1
RELAY	118	88	7	176	166	115	65	157	256	17

Table 3 – Data race detection results with RaceMob, ThreadSanitizer (TSAN), and RELAY. Each cell shows the number of reported data races. The data races reported by RaceMob and TSAN are all true data races. The only true data races among the ones detected by RELAY are the ones in the row "RaceMob". To the best of our knowledge, two of the data races that cause a hang in SQLite were not previously reported.

The overhead introduced by RaceMob is due to the instrumentation plus the overhead introduced by validation (DCI, on-demand detection, and schedule steering). Fig. 24 shows the breakdown of overhead for our ten target programs. We find that the runtime overhead without detection is below 1% for all cases, except the memory-intensive Fmm application, for which it is 2.51%. We conclude that, in the common case when a program is instrumented by RaceMob but no detection is performed, the runtime overhead is negligible; this property is what makes RaceMob suitable for always-on operation.

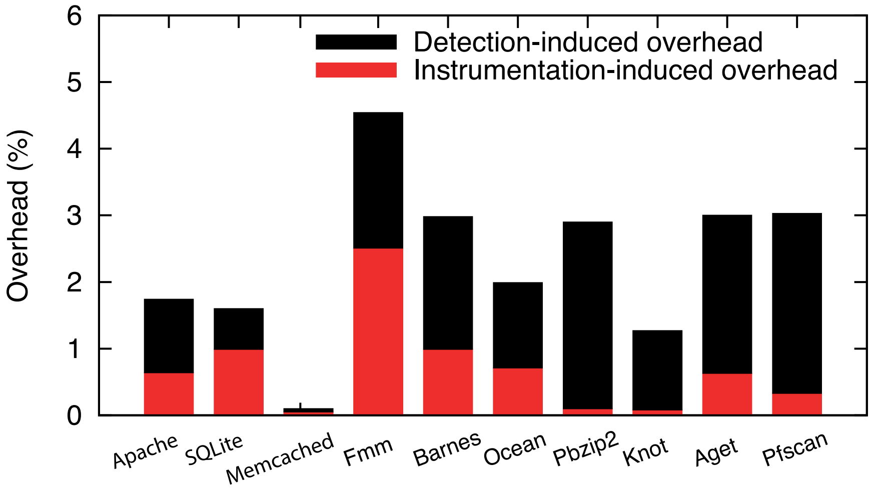


Figure 24 – Breakdown of average overhead into instrumentation-induced overhead and detection-induced overhead.

The dominant component of the overhead of data race detection (the black portion of the bars in Fig. 24) is due to dynamic data race validation. The effect of DCI is negligible: it is below 0.1% for all cases; thus, we don't show it in Fig. 24. Therefore, it is feasible to leave DCI on for all executions. This can help RaceMob to promote a data race from "Likely FP" to "True Race" with low overhead.

If RaceMob assigns more than one validation task at a time per user, the aggregate overhead that a user experiences will increase. In such a scenario, the user site would pick a validation candidate at runtime depending on which potentially racing access is executed. This scheme introduces a lookup overhead to determine at runtime which racing access is executed, however, it would not affect the per-race overhead, because of RaceMob's on-demand data race detection algorithm.

6.1.4 *Comparison to Other Detectors*

In this section, we compare RaceMob to state-of-the art dynamic, static, and sampling-based race detectors.

We compare RaceMob to the RELAY static data race detector [192] and to ThreadSanitizer [174] (TSAN), an open-source dynamic data race detector developed by Google. We also compare RaceMob to PACER [27], a sampling-based data race detector. Our comparison is in terms of detection results and runtime overhead. We do not compare to LiteRace, which is another sampling-based data race detector, because LiteRace has higher overhead and lower data race detection coverage than PACER [130]. The detection results are shown in Table 3.

6.1.4.1 *Comparative Accuracy*

We first compared RaceMob to TSAN by detecting data races for all the test cases that were available to us, except for the program executions from the real deployment of RaceMob, because we do not record real user executions. RaceMob detected 4 extra data races relative to TSAN: For Memcached and Pfscan, RaceMob detected, with the help of schedule steering, 2 data races missed by TSAN. RaceMob also detected 2 input-dependent data races in Aget that were missed by TSAN (of which one causes Aget to corrupt data), because RaceMob had access to executions from the real deployment, which were not accessible to TSAN. These data races required the user to manually abort and restart Aget. For 3 data races in pbzip2, RaceMob triggered a particular interleaving that caused the program to crash as a result of schedule steering, which did not happen in the case of TSAN. Furthermore, we have not observed any crash during detection with TSAN; this shows that, without schedule steering, the consequences of a detected data race may remain unknown.

Note that we give TSAN the benefit of access to all executions that RaceMob has access to (except the executions from the real users). This is probably overly generous, because in reality, dynamic data race detection is not crowdsourced, so one would run TSAN on fewer executions and obtain lower data race detection coverage than shown here. We did not use TSAN's hybrid data race detection algorithm, because it is known to report false positives and therefore lower the accuracy of data race detection.

Program	**Aggregate overhead with RaceMob [# of race candidates × # of users] in %**	**TSAN user-perceived overhead in %**
Apache	339.30	25,207.79
SQLite	281.60	1,428.57
Memcached	2.20	3,102.32
Fmm	1,598.08	47,888.07
Barnes	989.36	30,640.00
Ocean	360.70	3,069.39
Pbzip2	377.00	3,001.00
Knot	165.10	751.47
Aget	144.00	184.22
Pfscan	103.20	13,402.15

Table 4 – RaceMob aggregate overhead vs. TSAN's average overhead, relative to uninstrumented execution. RaceMob's aggregate overhead is across all the executions for all users. For TSAN, we report the average overhead of executing all the available test cases.

RELAY typically reports at least an order of magnitude more data races than the real data races reported by RaceMob, with no indication of whether they are true data races or not. Consequently, the developers would not have information on how to prioritize their bug fixing. This would in turn impact the users, because it might take longer to remove the data races with severe consequences. The benefit of tolerating a 2.32% average detection overhead with RaceMob is that data race detection results are more detailed and helpful. To achieve a similar effect as RaceMob, static data race detectors use unsound heuristics to prune some data race reports, and thus introduce false negatives.

6.1.4.2 *Comparative Overhead*

RELAY's static data race detection is offline, and the longest detection we measured was below 1 hour.

We compared the overheads of dynamic data race detection in RaceMob and TSAN. We chose TSAN because it is freely available, actively maintained, and works for C programs. The results are shown in Table 4. The average overhead of TSAN ranged from almost 49× for Fmm to 1.84× for Aget. The average overhead of RaceMob per user is about three orders of magnitude less than that of TSAN for all three programs.

The aggregate overhead of RaceMob represents the sum of all the overheads of all the executions at all the user sites. It represents RaceMob's overall overhead for detecting the data races in row 2 of Table 3. We compare RaceMob's aggregate overhead to TSAN's overhead because these overheads represent what both tools incur for all the data

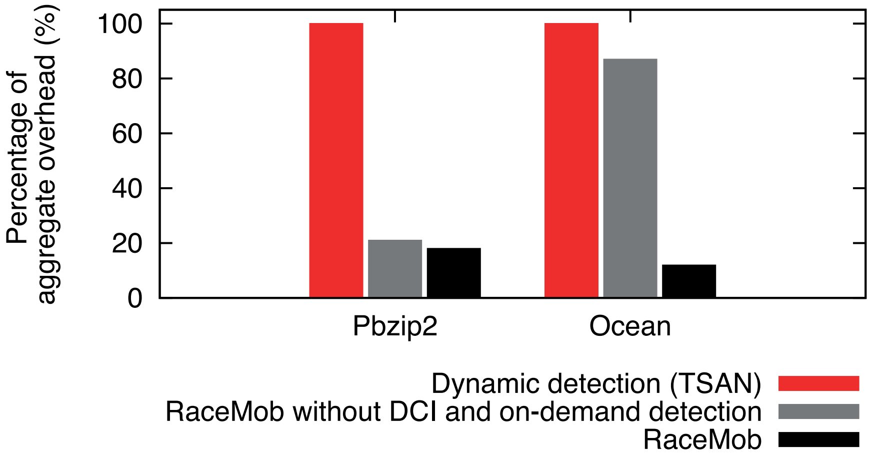


Figure 25 – Contribution of each technique to lowering the aggregate overhead of RaceMob. Dynamic detection represents detection with TSAN. RaceMob without DCI and on-demand detection just uses static data race detection to prune the number of accesses to monitor.

races they detect. The aggregate overhead of RaceMob is an order of magnitude less than the overhead of TSAN. This demonstrates that mere crowdsourcing of TSAN would not be enough to reduce its overhead (it would still be one order of magnitude away from RaceMob), and so the other techniques proposed in RaceMob are necessary too.

In particular, there are two other factors that contribute to lower overhead: the static data race detection phase and the lightweight dynamic validation phase. The contribution of each such phase depends on whether the application for which RaceMob performs data race detection is synchronization-intensive or not. To show the benefit of each phase, we picked Ocean (synchronization-intensive) and pbzip2 (uses less synchronization), and measured the contribution of each phase.

The results are shown in Fig. 32. This graph shows how the overhead of full dynamic detection reduces with each phase. The contribution of static data race detection is more significant for Pbzip2 in comparison to Ocean. This is because, for Pbzip2, narrowing down the set of accesses to be monitored has a good enough contribution. On the other hand, Ocean benefits more from DCI and on-demand data race detection, because static data race detection is inaccurate in this case (and is mitigated by DCI), and Ocean employs heavy synchronization (mitigated by on-demand data race detection). Thus, we conclude that both the static data race detection phase and DCI followed by on-demand data race detection are essential to lowering the overhead of aggregate data race detection in the general case.

We also compared the runtime overhead with PACER, a sampling-based data race detector. We do not have access to a PACER implementation for C/C++ programs; therefore, we modified RaceMob to operate like PACER. We allow PACER to have access to the static data race detection results from RELAY, and we assumed PACER starts sampling whenever a potential racing access is performed (as in RaceMob) rather than at a random time. We refer to our version of PACER as PACER-SA.

PACER-SA's runtime overhead is an order of magnitude larger than that of RaceMob for non-synchronization-intensive programs: 21.56% on average for PACER-SA vs. 2.32% for RaceMob. RaceMob has lower overhead mainly because it performs data race detection selectively: it does not perform on-demand data race detection for every potential data race detected statically, rather it only does so after DCI has proven that the relevant accesses can indeed alias and that they indeed can occur in a multithreaded context. Table 1 shows that DCI excludes on this basis more than half the data race candidates from further analysis.

For synchronization-intensive programs, like Fmm, Ocean and Barnes, PACER-SA's overhead can become up to two orders of magnitude higher than that of RaceMob. This is due to the combined effect of DCI and on-demand data race detection. The latter factor is more prominent for synchronization-intensive applications. To illustrate this, we picked Fmm and used RaceMob and PACER-SA to detect data races. For typical executions of 200 msec, where we ran Fmm with its default workload, Fmm performed around $15,000$ synchronization operations, which incur a 200% runtime overhead with PACER-SA compared to 4.54% with RaceMob.

We conclude that, even if PACER-SA's performance might be considered suitable for production use for non-synchronization-intensive programs, it is prohibitively high in the case of synchronization-intensive programs. This is despite giving the benefit of a static data race detection phase to vanilla PACER. PACER could have lower overhead than RaceMob if it stopped sampling soon after having started and before even detecting a data race, but it would of course also detect fewer data races.

This section showed that RaceMob detects more true data races than state-of-the art detectors while not introducing additional false negatives relative to what the static race detectors already do. It also showed that RaceMob's runtime overhead is lower than state-of-the-art detectors.

6.1.5 *Comparison to Concurrency Testing Tools*

A concurrency testing tool can be viewed as a type of data race detector, and vice versa. In this vein, one could imagine using RaceMob for testing, by using schedule steering (§3.3.3) to explore data races that may otherwise be hard to witness and that could lead to failures. As a simple test, we ran SQLite with the test cases used in our evaluation 10,000 times and never encountered any hang when not instrumented. When running it under RaceMob, we encountered 3 hangs within 176 executions. Similarly, we ran the Pbzip2 test cases 10,000 times and never encountered a crash, but RaceMob caused the occurrence of 4 crashes within 130 executions. This suggests that RaceMob could also be used as a testing tool to quickly identify and prioritize data races.

Some existing concurrency testing tools perform an analysis similar to schedule steering to detect and explore data races. In the rest of this section we compare RaceMob to two such state-of-the-art tools: RaceFuzzer [172] and Portend [98] (whose design we described in detail in Chapter §5). These tools were not intended for use in production, and thus have high overheads (up to 200× for RaceFuzzer and up to 5,000× for Portend), so we do not compare on overhead, but focus instead on comparing their respective data race detection coverage.

RaceFuzzer works in two stages: First, it uses imprecise hybrid data race detection [146] to detect potential data races in a program and instrument them. Second, it uses a randomized analysis to determine whether these potential data races are actual races. Portend uses precise happens-before dynamic data race detection and explores a detected data race's consequences along multiple paths and schedules.

To compare data race detection coverage, we use benchmarks bench$_1$, bench$_2$, bench$_3$ (taken from Google TSAN) and bench$_4$ (taken from [98]). The bench$_4$ benchmark has three data races that only manifest under specific inputs in_1, in_2, and in_3. Simplified versions of the benchmarks are shown in Fig. 26 and Fig. 2.

The RaceFuzzer implementation is not available, so we simulate it: we use TSAN in imprecise hybrid mode, as done in RaceFuzzer, and then implement RaceFuzzer's random scheduler. The results appear in Table 5. For bench$_1$, bench$_2$, and bench$_3$, RaceFuzzer performs as well as RaceMob in terms of data race detection coverage. For bench$_4$, RaceFuzzer's data race detection coverage varies between $0/3 - 3/3$.

Tool	**bench$_1$**	**bench$_2$**	**bench$_3$**	**bench$_4$**
RaceMob	1 / 1	1 / 1	1 / 1	3 / 3
RaceFuzzer	1 / 1	1 / 1	1 / 1	0 – 3 / 3
Portend	0 / 1	0 / 1	0 / 1	3 / 3

Table 5 – RaceMob vs. concurrency testing tools: Ratio of data races detected in each benchmark to the total number of data races in that benchmark.

To understand this variation, we run the following experiment: we assume that initially neither tool has access to any test case with input in_1, in_2, or in_3. Thus, RaceFuzzer cannot detect any data race, so it cannot instrument the racing accesses, and generates an instrumented

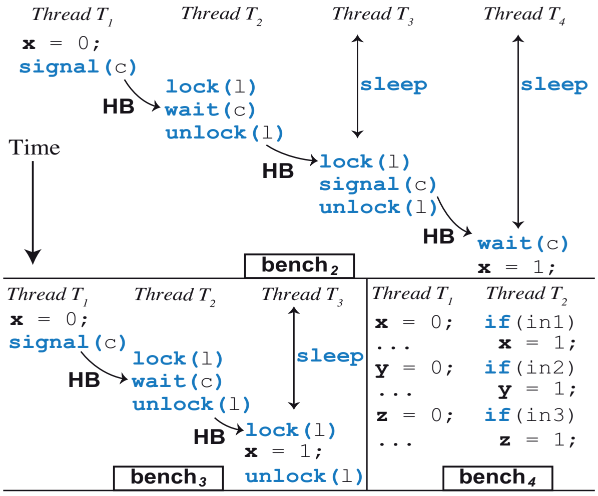

Thread T1
Thread T2
Thread T3
Thread T4
x = 0;
signal(c)
HB
lock(l)
wait(c)
unlock(l)
sleep
sleep
Time
HB
lock(l)
signal(c)
unlock(l)
HB
wait(c)
x = 1;
bench2
Thread T1
Thread T2
Thread T3
x = 0;
signal(c)
HB
lock(l)
wait(c)
unlock(l)
sleep
HB
lock(l)
x = 1;
unlock(l)
bench3
Thread T1
Thread T2
x = 0;
...
y = 0;
...
z = 0;
...
if(in1)
x = 1;
if(in2)
y = 1;
if(in3)
z = 1;
bench4

Figure 26 – Concurrency testing benchmarks: bench$_1$ is shown in Fig. 2, thus not repeated here. In bench$_2$, the accesses to x in thread T_1 and T_3 can race, but the long sleep in T_3 and T_4 causes the signal-wait and lock-unlock pairs to induce a happens-before edge between T_1 and T_4. bench$_3$ has a similar situation to bench$_2$. In bench$_4$, the accesses to variables x, y, z from T_1 and T_2 are racing if the input is either in_1, in_2, or in_3.

version of bench$_4$ we call RaceFuzzer$_0$. RaceMob, however, detects all three potential data races in bench$_4$, thanks to static data race detection, and instruments bench$_4$ at the potentially racing accesses. If we allow RaceFuzzer to see a test with input in_1, then it generates a version of bench$_4$ we call RaceFuzzer$_1$; if we allow it to see both a test with input in_1 and in_2, then it generates RaceFuzzer$_2$. RaceFuzzer$_3$ corresponds to having seen all three inputs.

We run both RaceFuzzer's and RaceMob's versions of the instrumented benchmark and plot data race detection coverage in Fig. 27. When run on random inputs different from in_1, in_2, and in_3, neither tool finds any data race (0/3), as expected. When given input in_1, RaceMob finds the data race, RaceFuzzer$_0$ doesn't, but RaceFuzzer$_1$, RaceFuzzer$_2$, and RaceFuzzer$_3$ do. And so on.

Of course, giving RaceFuzzer the benefit of access in advance to all test cases is overly generous, but this experiment serves to illustrate how the tool works. In contrast, RaceMob achieves data race detection coverage proportional to the number of runs with different inputs in_1, in_2, in_3, irrespective of which test cases were available initially, since it performs static data race detection to identify potential data races. RaceFuzzer could potentially miss all input-dependent data races even when the program under test is run with the inputs that expose such data races, because it may have missed those data races in its initial instrumentation stage. However, this is not a fundamental shortcoming: it is possible to mitigate it by replacing Race-

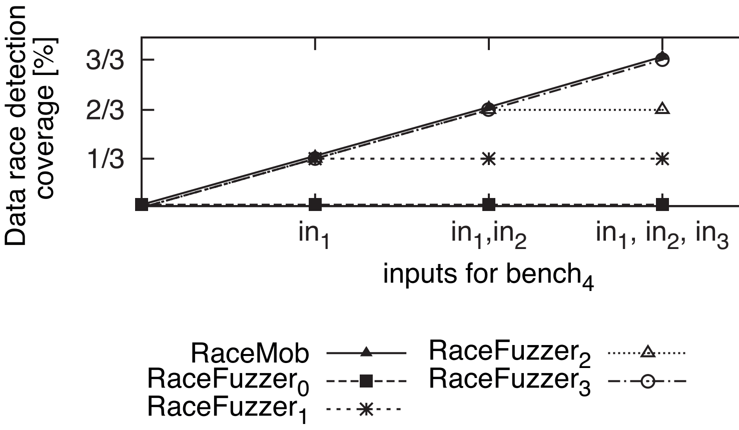


Figure 27 – Data race detection coverage for RaceMob vs. RaceFuzzer. To do as well as RaceMob, RaceFuzzer must have a priori access to all test cases (the RaceFuzzer3 curve).

Fuzzer's dynamic data race detection phase with a static data race detector.

The results of the comparison with Portend appear in Table 5. RaceMob detects all three test cases for bench4, as well as all the data races in all the other benchmarks. On the other hand, Portend discovered all the input-dependent data races in bench4, but failed to detect the data races in the other benchmarks, because it employs a precise dynamic data race detector that does not do schedule steering. However, Portend is able to explore the consequences of a data race more thoroughly than RaceMob, and in that regard RaceMob and Portend are complementary.

6.1.6 *Scalability with Application Threads*

RaceMob uses atomic operations to update internal shared structures related to dynamic data race validation and signal-wait synchronization to perform schedule steering; in this section, we analyze the effect these operations have on RaceMob's scalability as the number of application threads increases.

We configured multiple clients to concurrently request a 10 MB file from Apache and Knot using the Apache benchmarking tool `ab`. For SQLite and Memcached, we inserted, modified, and removed $5,000$ items from the database and the object cache, respectively. We used Pbzip2 to decompress a 100 MB file. For Ocean, we simulated currents in a 256×256 ocean grid. For Barnes, we simulated interactions of $16,384$ bodies (default number for Barnes). We varied the number of threads from 2 – 32. For all programs, we ran the instrumented versions of the programs while performing data race detection and measured the overhead relative to uninstrumented versions on the 8-core machine.

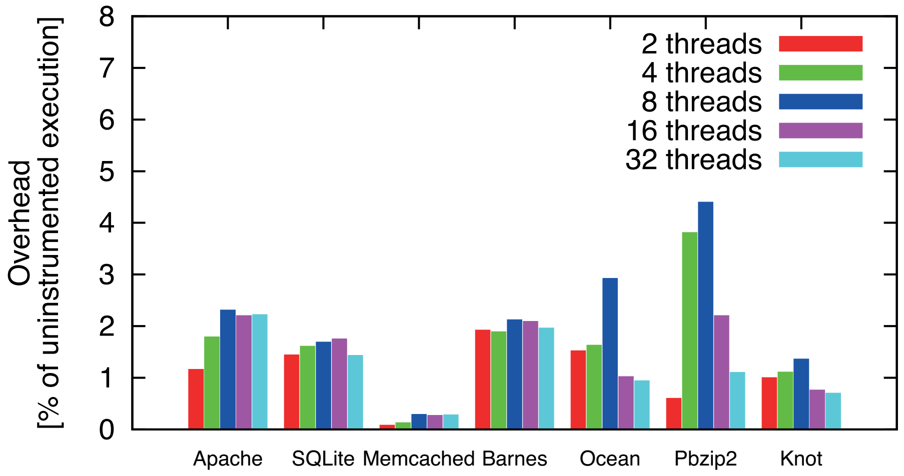


Figure 28 – RaceMob scalability: Induced overhead as a function of the number of application threads.

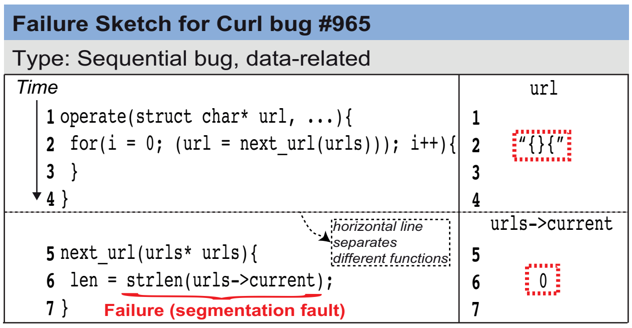
Fig. 28 shows the results. We expected RaceMob's overhead to become less visible after the thread count reached the core count. We wanted to verify this, and that is why we used the 8-core machine. For instance, for Apache the overhead is 1.16% for 2 threads, it slightly rises to its largest value of 2.31% for 8 threads, and then it decreases as the number of threads exceeds the number of cores. We observe a similar trend for all other applications. We conclude that RaceMob's runtime overhead remains low as the number of threads in the test programs increases.

6.2 GIST'S EVALUATION

In this section we aim to answer the following questions about Gist and failure sketching: Is Gist capable of automatically computing failure sketches (§6.2.2)? Are these sketches accurate (§6.2.3)? How efficient is the computation of failure sketches in Gist (§6.2.4)?

6.2.1 *Experimental Setup*

To answer these questions we benchmark Gist with several real world programs: Apache hhtpd, SQLite, Memcached, and Pbzip2 were previously described in §6.1.1. Cppcheck [131] is a C/C++ static analysis tool integrated with popular development tools such as Visual Studio, Eclipse, and Jenkins. Curl [180] is a data transfer tool for network protocols such as FTP and HTTP, and it is part of most Linux distributions and many programs, like LibreOffice and CMake. Transmission [189] is the default BitTorrent client in Ubuntu and Fedora Linux, as well as Solaris.



```
operate(struct char* url, ...){
  for(i = 0; (url = next_url(urls))); i++){
  }
}
next_url(urls* urls){
  len = strlen(urls->current);
}
```

Failure Sketch for Curl bug #965
Type: Sequential bug, data-related
Time
url
"{}{"
urls->current
0
horizontal line separates different functions
Failure (segmentation fault)

Figure 29 – The failure sketch of Curl bug #965.

We developed an extensible framework called Bugbase [14] in order to reproduce the known bugs in the aforementioned software. Bugbase can also be used to do performance benchmarking of various bug finding tools. We used Bugbase to obtain our experimental results.

We benchmark Gist on bugs (from the corresponding bug repositories) that were used by other researchers to evaluate their bug finding and failure diagnosis tools [7, 151, 98]. Apart from bugs in Cppcheck and Curl, all bugs are concurrency bugs (e.g., data races and atomicity). We use a mixture of workloads from actual program runs, test suites, test cases devised by us and other researchers [210], Apache's benchmarking tool `ab`, and SQLite's test harness. We gathered execution information from a total of 11,360 executions.

The distributed cooperative setting of our test environment is simulated, as opposed to employing real users, because CPUs with Intel PT support are still scarce, having become available only recently. In the future we plan to use a real-world deployment. Altogether we gathered execution information from 1,136 simulated user endpoints. Client-side experiments were run on a 2.4 GHz 4 core Intel i7-5500U (Broadwell) machine running a Linux kernel with an Intel PT driver [119]. The server side of Gist ran on a 2.9 GHz 32-core Intel Xeon E5-2690 machine with 256 GB of RAM running Linux kernel 3.13.0-44.

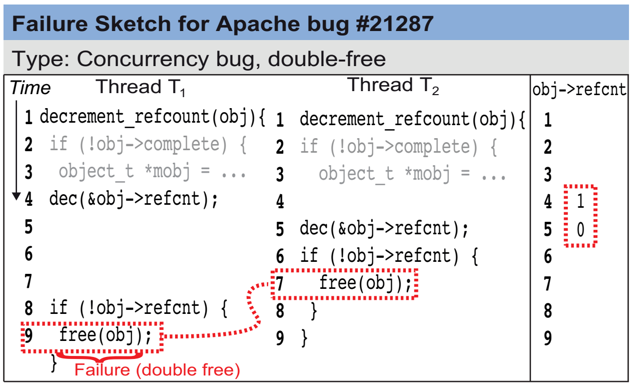
6.2.2 *Automated Generation of Sketches*

For all the failures shown in Table 6, Gist successfully computed the corresponding failure sketches after gathering execution information from 11,360 runs in roughly 35 minutes. The results are shown in the rightmost two columns. We verified that, for all sketches computed by Gist, the failure predictors with the highest F-measure indeed correspond to the root causes that developers chose to fix.

In the rest of this section, we present two failure sketches computed by Gist, to illustrate how developers can use them for root cause diagnosis and for fixing bugs. These two complement the failure sketch

Bug name / software	**Software version**	**Software size [LOC]**	**Bug ID from bug DB**	**Static slice size, in source [LOC] (LLVM instructions)**	**Ideal failure sketch size, in source [LOC] (LLVM instrs)**	**Gist-computed sketch size, in source [LOC] (LLVM instrs)**	**Duration of failure sketch computation by Gist: # failure recurrences <time> (offline analysis time)**
Apache-1	2.2.9	224,533	45605	7 (23)	8 (23)	8 (23)	5 <4m:22s> (1m:28s)
Apache-2	2.0.48	169,747	25520	35 (137)	4 (16)	4 (16)	4 <3m:53s> (0m:55s)
Apache-3	2.0.48	169,747	21287	354 (968)	6 (6)	8 (8)	3 <4m:17s> (1m:19s)
Apache-4	2.0.46	168,574	21285	335 (805)	9 (12)	13 (16)	4 <5m:34s> (1m:23s)
Cppcheck-1	1.52	86,215	3238	3,662 (10,640)	11 (16)	11 (16)	4 <5m:14s> (2m:32s)
Cppcheck-2	1.48	76,009	2782	3,028 (8,831)	3 (8)	3 (8)	3 <3m:21s> (1m:40s)
Curl	7.21	81,658	965	15 (46)	6 (17)	6 (17)	5 <1m:31s> (0m:40s)
Transmission	1.42	59,977	1818	680 (1,681)	2 (7)	3 (8)	3 <0m:23s> (0m:17s)
SQLite	3.3.3	47,150	1672	389 (1,011)	3 (4)	3 (4)	2 <2m:47s> (1m:43s)
Memcached	1.4.4	8,182	127	237 (1,003)	6 (13)	8 (16)	4 <0m:56s> (0m:02s)
Pbzip2	0.9.4	1,492	N/A	8 (14)	6 (13)	9 (14)	4 <1m:12s> (0m:03s)

Table 6 – Bugs used to evaluate Gist. Bug IDs come from the corresponding official bug database. Source lines of code are measured using sloccount [200]. We report slice and sketch sizes in both source code lines and LLVM instructions. Time is reported in minutes:seconds.


Failure Sketch for Apache bug #21287
Type: Concurrency bug, double-free
Time
Thread T_1
Thread T_2
obj->refcnt
Failure (double free)

```
1 decrement_refcount(obj){  1 decrement_refcount(obj){  1
2  if (!obj->complete) {    2  if (!obj->complete) {     2
3   object_t *mobj = ...    3   object_t *mobj = ...     3
4  dec(&obj->refcnt);       4                            4  1
5                           5  dec(&obj->refcnt);        5  0
6                           6  if (!obj->refcnt) {       6
7                           7   free(obj);               7
8  if (!obj->refcnt) {      8  }                         8
9   free(obj);              9 }                          9
  }
```

Figure 30 – The failure sketch of Apache bug #21287. The grayed-out components are not part of the ideal failure sketch, but they appear in the sketch that Gist automatically computes.

for the Pbzip2 bug already described in Fig. 8. Aside from some formatting, the sketches shown in this section are exactly the output of Gist. We renamed some variables and functions to save space in the figures. The statements or variable values in dotted rectangles denote failure predicting events with the highest F-measure values. We integrated Gist with KCachegrind [197], a call graph viewer that allows easy navigation of the statements in the failure sketch.

Fig. 29 shows the failure sketch for Curl bug #965, a sequential bug caused by a specific program input: passing the string “{}{“ (or any other string with unbalanced curly braces) to Curl causes the variable `urls->current` in function `next_url` to be `NULL` in step 6. The value of `url` in step 2 (“{}{“) and the value of `urls->current` in step 6 (0) are the best failure predictors. This failure sketch suggests that fixing the bug consists of either disallowing unbalanced parentheses in the input url, or not calling `strlen` when `urls->current` is `NULL`. Developers chose the former solution to fix this bug [181].

Fig. 30 shows the failure sketch for Apache bug 21287, a concurrency bug causing a double free. The failure sketch shows two threads executing the `decrement_refcount` function with the same `obj` value. The `dec` function decrements `obj->refcount`. The call to `dec`, the `if` condition checking, namely `!obj->refcount`, and the call to `free` are not atomic, and this can cause a double free if `obj->refcount` is 0 in step 6 in T_3 and step 8 in T_2. The values of `obj->refcount` in steps 4 and 5 (1 and 0 respectively), and the double call to `free(obj)` are the best failure predictors. Developers fixed this bug by ensuring that the decrement-check-free triplet is executed atomically [182].

The grayed-out statements in the failure sketch in Fig. 30 are not part of the ideal failure sketch. The adaptive slice tracking in Gist tracks them during slice refinement, because Gist does not know the statements in the ideal failure sketch a priori. For the Curl bug in Fig. 29, we do not show any grayed-out statements, because, adaptive slice tracking happens to track only the statements that are in the ideal failure sketch.

6.2.3 *Accuracy of Failure Sketches*

In this section, we measure the accuracy (A) of failure sketches computed by Gist (Φ_G), as compared to ideal failure sketches that we computed by hand (Φ_I), according to our ideal failure sketch definition (§4.3). We define two components of failure sketch accuracy:

1) ***Relevance*** measures the extent to which a failure sketch contains all the statements from the ideal sketch and no other statements. We define relevance as the ratio of the number of LLVM instructions in $\Phi_G \cap \Phi_I$ to the number of statements in $\Phi_G \cup \Phi_I$. We compute relevance accuracy as a percentage, and define it as $A_R = 100 \cdot \frac{|\Phi_G \cap \Phi_I|}{|\Phi_G \cup \Phi_I|}$

2) ***Ordering*** measures the extent to which a failure sketch correctly represents the partial order of LLVM memory access instructions in the ideal sketch. To measure the similarity in ordering between the Gist-computed failure sketches and their ideal counterparts, we use the normalized Kendall tau distance [103] τ, which measures the number of pairwise disagreements between two ordered lists. For example, for ordered lists <A, B, C> and <A, C, B>, the pairs (A, B) and (A, C) have the same ordering, whereas the pair (B, C) has different orderings in the two lists, hence $\tau = 1$. We compute the ordering accuracy as a percentage defined by $A_O = 100 \cdot (1 - \frac{\tau(\Phi_G, \Phi_I)}{\#\text{ of pairs in }\Phi_G \cap \Phi_I})$. Note that # of pairs in $\Phi_G \cap \Phi_I$ can't be zero, because both failure sketches will at least contain the failing instruction as a common instruction.

We define overall accuracy as $A = \frac{A_R + A_O}{2}$, which equally favors A_O and A_R. Of course, different developers may have different subjective opinions on which one matters most.

We show Gist's accuracy results in Fig. 31. Average relevance accuracy is 92%, average ordering accuracy is 100%, and average overall accuracy is 96%, which leads us to conclude that Gist can compute failure sketches with high accuracy. The accuracy results are deterministic from one run to the next.

Note that, for all cases when relevance accuracy is below 100%, it is because Gist's failure sketches have (relative to the ideal sketches) some excess statements in the form of a prefix to the ideal failure sketch, as shown in gray in Fig. 30. We believe that developers find it significantly easier to visually discard excess statements clustered as a prefix than excess statements that are sprinkled throughout the failure sketch, so this inaccuracy is actually not of great consequence.

We show in Fig. 32 the contribution of Gist's three analysis and tracking techniques to overall sketch accuracy. To obtain these measurements, we first measured accuracy when using just static slicing, then enabled control flow tracking and re-measured, and finally en-

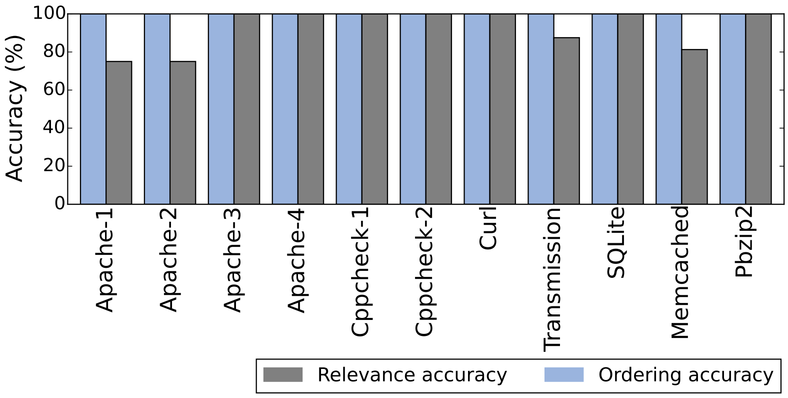


Figure 31 – Accuracy of Gist, broken down into relevance accuracy and ordering accuracy.

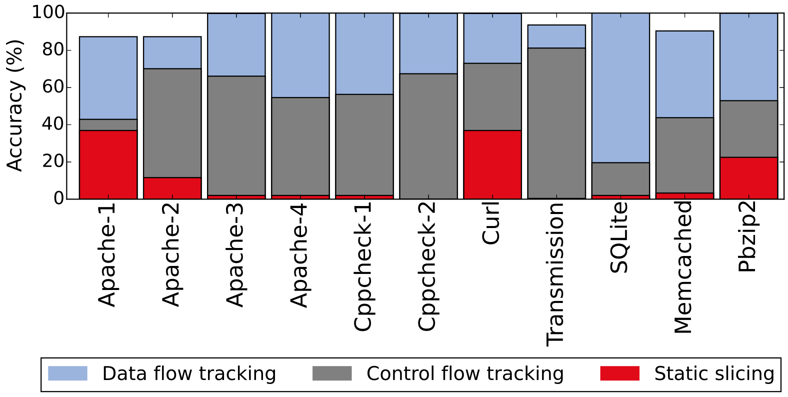


Figure 32 – Contribution of various techniques to Gist's accuracy.

abled also data flow tracking and re-measured. While the accuracy results are consistent across runs, the individual contributions may vary if, for example, workload non-determinism causes different paths to be exercised through the program.

A small contribution of a particular technique does not necessarily mean that it does not perform well for a given program, but it means that the other techniques that Gist had enabled prior to this technique "stole its thunder" by being sufficient to provide high accuracy. For example, in the case of Apache-1, static analysis performs well enough that control flow tracking does not need to further refine the slice. However, in some cases (e.g., for SQLite), tracking the inter-thread execution order of statements that access shared variables using hardware watchpoints is crucial for achieving high accuracy.

We observe that the amount of individual contribution varies substantially from one program to the next, which means that neither of these techniques would achieve high accuracy for all programs on its own, and so they are all necessary if we want high accuracy across a broad spectrum of software.

6.2.4 *Efficiency*

Now we turn our attention to the efficiency of Gist: how long does it take to compute a failure sketch, how much runtime performance overhead does it impose on clients, and how long does it take to perform its offline static analysis. We also look at how these measures vary with different parameters.

The last column of Table 6 shows Gist's failure sketch computation latency broken down into three components. We show the number of failure recurrences required to reach the best sketch that Gist can compute, and this number varies from 2 to 5 recurrences. We then show the total time it took in our simulated environment to find this sketch; this time is always less than 6 minutes, varying from <0m:23s> to <5m:34s>. Not surprisingly, this time is dominated by how long it takes the target failure to recur, and in practice this depends on the number of deployed clients and the variability of execution circumstances. Nevertheless, we present the values for our simulated setup to give an idea as to how long it took to build a failure sketch for each bug in our evaluation. Finally, in parentheses we show Gist's offline analysis time, which consists of computing the static slice plus generating instrumentation patches. This time is always less than 3 minutes, varying between <0m:2s> and <2m:32s>. We therefore conclude that, compared to the debugging latencies experienced by developers today, Gist's automated approach to root cause diagnosis presents a significant advantage.

In the context of adaptive slice tracking, the overhead incurred on the client side increases monotonically with the size of the tracked slice, which is not surprising. Fig. 33 confirms this experimentally. The portion of the overhead curve between the slice sizes 16 and 22 is relatively flat compared to the rest of the curve. This is because, within that interval, Gist only tracks a few control flow events for Apache-1 and Curl (these programs have no additional data flow elements in that interval), which introduces negligible overhead.

The majority of the overhead incurred on the client side stems from control flow tracking. In particular, the overhead of control flow tracking varies from a low of 2.01% to a high of 3.43%, whereas the overhead of data flow tracking varies from a low of 0.87% to a high of 1.04%.

What is perhaps not immediately obvious is the trade-off between initial slice size σ and the resulting accuracy and latency. In Fig. 34, we show the average failure sketch accuracy across all programs we measured (right y-axis) and Gist's latency in # of recurrences (left y-axis) as a function of σ that Gist starts with (x-axis). As long as the initial slice size is less than the one for the best sketch that Gist can find, Gist's adaptive approach is capable of guiding the developer to the highest accuracy sketch. Of course, the time it takes to find

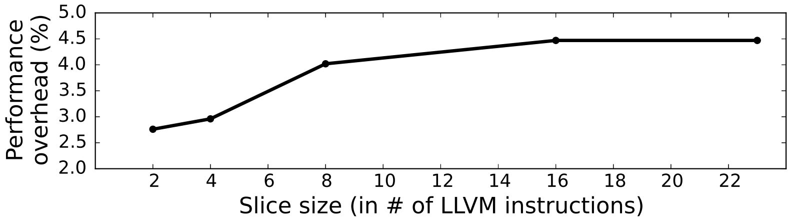


Figure 33 – Gist's average runtime performance overhead across all runs as a function of tracked slice size.

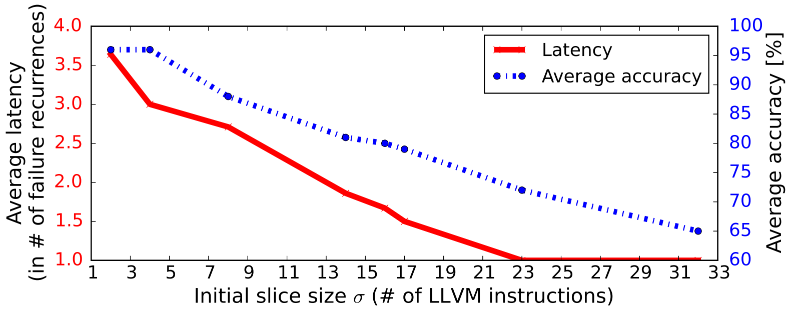


Figure 34 – Tradeoff between slice size and the resulting accuracy and latency. Accuracy is in percentage, latency is in the number of failure recurrences.

the sketch is longer the smaller the starting slice size is, because the necessary # of recurrences is higher. There is thus an incentive to start with a larger slice size. Unfortunately, if this size overshoots the size of the highest accuracy sketch, then the accuracy of the outcome suffers, because the larger slice includes extraneous elements.

As we mentioned in §6.2.3, the extraneous statements that can lower Gist's accuracy are clustered as a prefix to the ideal failure sketch, allowing developers to easily ignore them. Therefore, if lower root cause diagnosis latency is paramount to the developers, they are comfortable ignoring the prefix of extraneous statements, and they can tolerate the slight increase in Gist's overhead, it is reasonable to configure Gist to start with a large σ (e.g., $\sigma = 23$ achieves a latency of *one failure recurrence* for all our benchmarks).

For the benchmarks in our evaluation, starting AsT at $\sigma = 4$ would achieve the highest average accuracy at the lowest average latency of 3, with an average overhead of 3.98%.

Finally, Fig. 35 compares Intel PT, the hardware-based control flow tracking mechanism we use in Gist, to Mozilla rr, a software-based state-of-the-art record & replay system. In particular, we compare the performance overhead imposed by the two tracking mechanisms on the client application. The two extremes are Cppcheck, where Mozilla rr is on par with Intel PT, and Transmission and SQLite, where Mozilla rr's overhead is over many orders of magnitude higher

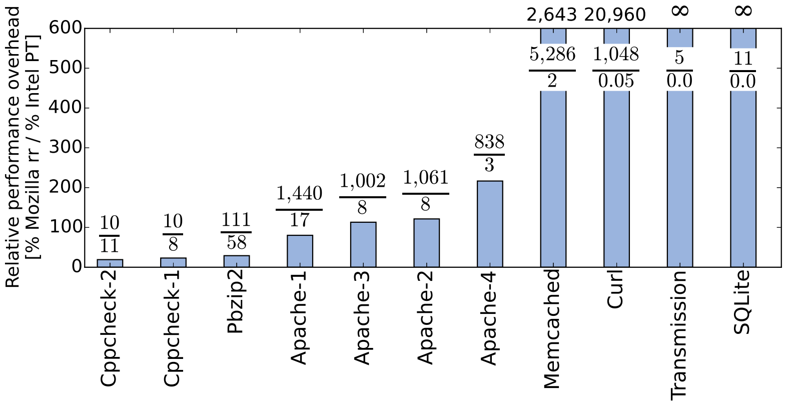


Figure 35 – Comparison of the full tracing overheads of Mozilla rr and Intel PT.

than Intel PT's [1]. For the benchmarks in our evaluation, full tracing using Intel PT incurs an average overhead of 11%, whereas full program record & replay incurs an average runtime overhead of 984%. Unlike Intel PT, Mozilla rr also gathers data flow information, but with Gist we have shown that full program tracing is not necessary for automating root cause diagnosis.

In conclusion, our empirical evaluation shows that Gist is capable of automatically computing failure sketches for failures caused by real bugs in real systems (§6.2.2), these sketches have a high accuracy of 96% on average (§6.2.3), and the average performance overhead of failure sketching is low at 3.74% with $\sigma = 2$ (§6.2.4). We therefore believe failure sketching to be a promising approach for helping developers debug elusive bugs that occur only in production.

6.3 PORTEND'S EVALUATION

In this section, we answer the following questions: Is Portend effective in telling developers which data races are true bugs and in helping them fix buggy data races (§6.3.2)? How accurately does it classify data race reports into the four categories of data races (§6.3.3)? How long does classification take, and how does it scale (§6.3.4)? How does Portend compare to the state of the art in data race classification (§6.3.5)? How effectively and efficiently does Portend implement symbolic memory consistency modeling and what is its memory overhead (§6.3.6, §6.3.7)?. Throughout this section, we highlight the synergy of the techniques used in Portend: in particular §6.3.2 shows how symbolic output comparison allows more accurate data race classification compared to post-data race state comparison, and §6.3.3

---

[1] Full tracing overheads of Transmission and SQLite for Intel PT are too low to be reliably measured, thus they are shown as 0%, and the corresponding Mozilla rr/Intel PT overheads for these systems are shown as ∞.

shows how the combination of multi-path multi-schedule analysis improves upon traditional single-path analysis.

6.3.1 *Experimental Setup*

We apply Portend to 7 applications: SQLite, Pbzip2, Memcached, Ocean, and Fmm, which we previously described in §6.1.1; Ctrace [132], a multi-threaded debug library; Bbuf [208], a shared buffer implementation with a configurable number of producers and consumers.

We additionally evaluate Portend on homegrown micro-benchmarks that capture most classes of data races considered as harmless in the literature [174, 142]: "redundant writes" (RW), where racing threads write the same value to a shared variable, "disjoint bit manipulation" (DBM), where disjoint bits of a bit-field are modified by racing threads, "all values valid" (AVV), where the racing threads write different values that are nevertheless all valid, and "double checked locking" (DCL), a method used to reduce the locking overhead by first testing the locking criterion without actually acquiring a lock. Additionally, we have 4 other micro-benchmarks that we used to evaluate the SMCM. We detail those micro-benchmarks in §6.3.6. Table 7 summarizes the properties of our 15 experimental targets.

Note that these "harmless" data races are anti-patterns for some languages and platforms, because their behavior is highly dependent on the compiler and the hardware [134].

Program	**Size (LOC)**	**Language**	**# Forked threads**
SQLite 3.3.0	113,326	C	2
ocean 2.0	11,665	C	2
fmm 2.0	11,545	C	3
memcached 1.4.5	8,300	C	8
pbzip2 2.1.1	6,686	C++	4
ctrace 1.2	886	C	3
bbuf 1.0	261	C	8
AVV	49	C++	3
DCL	45	C++	5
DBM	45	C++	3
RW	42	C++	3
no-sync	45	C++	3
no-sync-bug	46	C++	3
sync	47	C++	3
sync-bug	48	C++	3

Table 7 – Programs analyzed with Portend. Source lines of code are measured with the `cloc` utility.

We ran Portend on several other systems (e.g., HawkNL, swarm), but no races were found in those programs with the test cases we ran, so we do not include them here. For all experiments, the Portend parameters were set to $M_p = 5$, $M_a = 2$, and the number of symbolic inputs to 2. We found these numbers to be sufficient to achieve high accuracy in a reasonable amount of time. To validate Portend's results, we used manual investigation, analyzed developer change logs, and consulted with the applications' developers when possible. All experiments were run on a 2.4 GHz Intel Core 2 Duo E6600 CPU with 4 GB of RAM running Ubuntu Linux 10.04 with kernel version 2.6.33. The reported numbers are averages over 10 experiments.

6.3.2 *Effectiveness*

Of the 93 distinct data races detected in 7 real-world applications, Portend classified 5 as definitely harmful by watching for "basic" properties (Table 8): one hangs the program and four crash it.

Program	**Total # of data races**	**# of "Spec violated" races**		
		Deadlock	**Crash**	**Semantic**
SQLite	1	1	0	0
pbzip2	31	0	3	0
ctrace	15	0	1	0
Manually inserted errors				
fmm	13	0	0	1
memcached	18	0	1	0

Table 8 – "Spec violated" data races and their consequences.

To illustrate the checking for "high level" semantic properties, we instructed Portend to verify that all timestamps used in fmm are positive. This caused it to identify the 6th "harmful" data race in Table 8; without this semantic check, this data race turns out to be harmless, as the negative timestamp is eventually overwritten.

To illustrate a "what-if analysis" scenario, we turned an arbitrary synchronization operation in the memcached binary into a no-op, and then used Portend to explore the question of whether it is safe to remove that particular synchronization point (e.g., we may be interested in reducing lock contention). Removing this synchronization induces a data race in memcached; Portend determined that the data race could lead to a crash of the server for a particular interleaving, so it classified it as "spec violated".

Portend's main contribution is the classification of data races. If one wanted to eliminate all harmful data races from their code, they could use a static data race detector (one that is complete, and, by necessity, prone to false positives) and then use Portend to classify these reports.

Program	**Number of data races**						
	Distinct data races	Data race instances	Spec violated	Output differs	K-witness harmless		Single ordering
					states same	*states differ*	
SQLite	1	1	1	0	0	0	0
ocean	5	14	0	0	0	1	4
fmm	13	517	0	0	0	1	12
memcached	18	104	0	2	0	0	16
pbzip2	31	97	3	3	0	0	25
ctrace	15	19	1	10	0	4	0
bbuf	6	6	0	6	0	0	0
AVV	1	1	0	0	1	0	0
DCL	1	1	0	0	1	0	0
DBM	1	1	0	0	1	0	0
RW	1	1	0	0	1	0	0

Table 9 – Summary of Portend's classification results. We consider two data races to be distinct if they involve different accesses to shared variables; the same data race may be encountered multiple times during an execution—these two different aspects are captured by the *Distinct data races* and *Data race instances* columns, respectively. Portend uses the stack traces and the program counters of the threads making the racing accesses to identify distinct data races. The last 5 columns classify the distinct data races. The *states same/differ* columns show for how many data races the primary and alternate states were different after the data race, as computed by the Record/Replay Analyzer [142].

For every harmful data race, Portend's comprehensive report and replayable traces (i.e., inputs and thread schedule) allowed us to confirm the harmfulness of the data races within minutes. Portend's report includes the stack traces of the racing threads along with the address and size of the accessed memory field; in the case of a segmentation fault, the stack trace of the faulting instruction is provided as well—this information can help in automated bug clustering. According to developers' change logs and our own manual analysis, the data races in Table 8 are the only known harmful data races in these applications.

6.3.3 *Accuracy and Precision*

To evaluate Portend's accuracy and precision, we had it classify all 93 data races in our target applications and micro-benchmarks. Table 9 summarizes the results. The first two columns show the number of distinct data races and the number of respective instances, i.e., the number of times those data races manifested during data race detection. The "spec violated" column includes all data races from Table 8 minus the semantic data race in fmm and the data race we introduced in memcached. In the "k-witness harmless" column, we show for which data races the post-data race states differed vs. not.

By accuracy, we refer to the correctness of classification: the higher the accuracy, the higher the ratio of correct classification. Precision on the other hand, refers to the reproducibility of experimental results: the higher the precision, the higher the ratio with which experiments are repeated with the same results.

To determine accuracy, we manually classified each data race and found that Portend had correctly classified 92 of the 93 data races (99%) in our target applications: all except one of the data races classified "k-witness harmless" by Portend are indeed harmless in an absolute sense, and all "single ordering" data races indeed involve ad-hoc synchronization.

To measure precision, we ran 10 times the classification for each data race. Portend consistently reported the same data set shown in Table 9, which indicates that, for these data races and applications, it achieves full precision.

As can be seen in the "k-witness harmless" column, for each and every one of the 7 real-world applications, a state difference (as used in [142]) does not correctly predict harmfulness, while our "k-witness harmless" analysis correctly predicts that the data races are harmless with one exception.

This suggests that differencing of concrete state is a poor classification criterion for data races in real-world applications with large memory states, but may be acceptable for simple benchmarks. This also supports our choice of using symbolic output comparison.

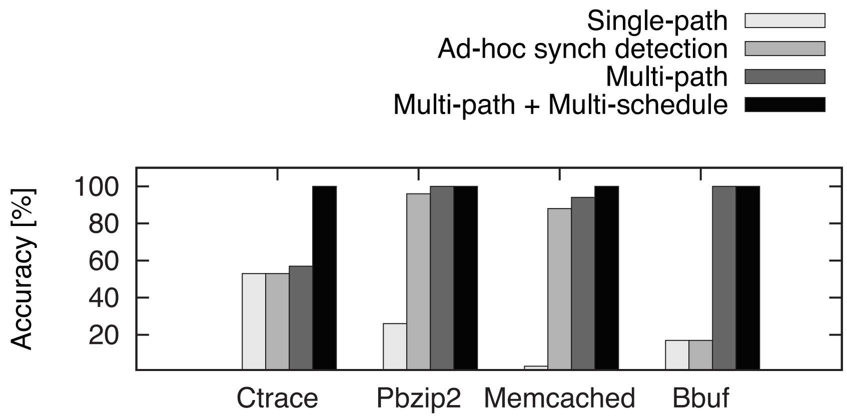
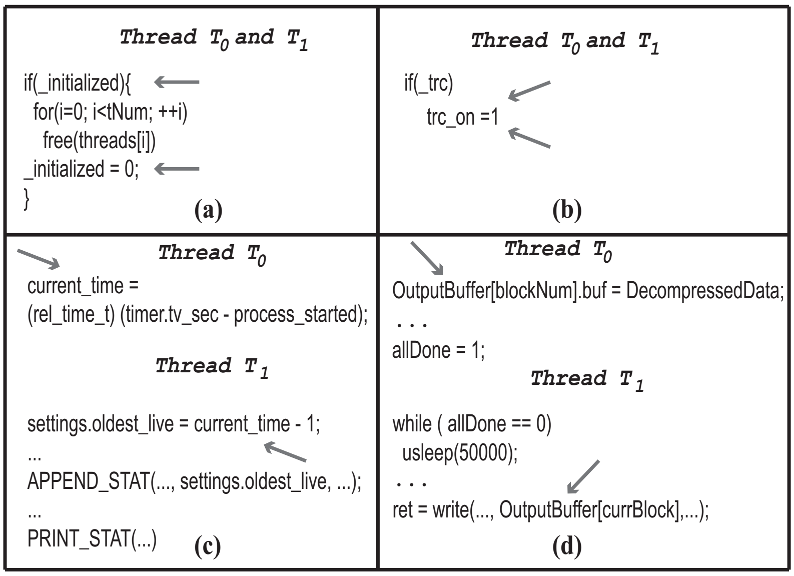


Figure 36 – Breakdown of the contribution of each technique toward Portend's accuracy. We start from single-path analysis and enable one by one the other techniques: ad-hoc synchronization detection, multi-path analysis, and finally multi-schedule analysis.

Multi-path multi-schedule exploration proved to be crucial for Portend's accuracy. Fig. 36 shows the breakdown of the contribution of each technique used in Portend: ad-hoc synchronization detection, multi-path analysis, and multi-schedule analysis. In particular, for 16 out of 21 "output differs" data races (6 in Bbuf, 9 in Ctrace, 1 in pbzip2) and for 1 "spec violated" data race (in ctrace), single-path analysis revealed no difference in output; it was only multi-path multi-schedule exploration that revealed an output difference (9 data races required multi-path analysis for classification, and 8 data races required also multi-schedule analysis). Without multi-path multi-schedule analysis, it would have been impossible for Portend to accurately classify those data races by just using the available test cases. Moreover, there is a high variance in the contribution of each technique for different programs, which means that none of these techniques alone would have achieved high accuracy for a broad range of programs.

We also wanted to evaluate Portend's ability to deal with false positives, i.e., false data race reports. Data race detectors, especially static ones, may report false positives for a variety of reasons, depending on which technique they employ. To simulate an imperfect detector for our applications, we deliberately removed from Portend's data race detector its awareness of mutex synchronizations. We then eliminated the data races in our micro-benchmarks by introducing mutex synchronizations. When we re-ran Portend with the erroneous data race detector on the micro-benchmarks, all four were falsely reported as data races by the detector, but Portend ultimately classified all of them as "single ordering". This suggests Portend is capable of properly handling false positives.

Fig. 37 shows examples of real data races for each category: (a) a "spec violated" data race in which resources are freed twice, (b) a "k-



```
if(_initialized){
  for(i=0; i<tNum; ++i)
    free(threads[i])
_initialized = 0;
}
```

```
if(_trc)
   trc_on =1
```

```
current_time =
(rel_time_t) (timer.tv_sec - process_started);
```

```
settings.oldest_live = current_time - 1;
...
APPEND_STAT(..., settings.oldest_live, ...);
...
PRINT_STAT(...)
```

```
OutputBuffer[blockNum].buf = DecompressedData;
. . .
allDone = 1;
```

```
while ( allDone == 0)
  usleep(50000);
. . .
ret = write(..., OutputBuffer[currBlock],...);
```

Figure 37 – Simplified examples for each data race class from real systems. (a) and (b) are from ctrace, (c) is from memcached and (d) is from pbzip2. The arrows indicate the pair of racing accesses.

witness harmless" data race due to redundant writes, (c) an "output differs" data race in which the schedule-sensitive value of the shared variable influences the output, and (d) a "single ordering" data race showing ad-hoc synchronization implemented via busy wait.

6.3.4 *Efficiency*

We evaluate the performance of Portend in terms of efficiency and scalability. Portend's performance is mostly relevant if it is to be used interactively, as a developer tool, and also if used for a large scale bug triage tool, such as in Microsoft's Windows Error Reporting system [67].

We measure the time it takes Portend to classify the 93 data races; Table 10 summarizes the results. We find that Portend classifies all detected data races in a reasonable amount of time, the longest taking less than 11 minutes. For Bbuf, Ctrace, Ocean and Fmm, the slowest classification time is due to a data race from the "k-witness harmless" category, since classification into this category requires multi-path multi-schedule analysis.

The second column reports the time it took Cloud9 to interpret the programs with concrete inputs. This provides a sense of the overhead incurred by Portend compared to regular LLVM interpretation in Cloud9. Both data race detection and classification are disabled when measuring baseline interpretation time. In summary, the overhead introduced by classification ranges from 1.1× to 49.9× over Cloud9's interpreter's overhead.

In order to get a sense of how classification time scales with program characteristics, we measured it as a function of program size, number of preemption points, number of branches that depend (directly or indirectly) on symbolic inputs, and number of threads. We found that program size plays almost no role in classification time. Instead, the other three characteristics play an important role. We show in Fig. 38 how classification time varies with the number of dependent branches and the number of preemptions in the schedule (which is roughly proportional to the number of preemption points and the number of threads). Each vertical bar corresponds to the classification time for the indicated data race. We see that, as the number of preemptions and branches increase, so does classification time.

Program	**Cloud9 running time (sec)**	**Portend classification time (sec)**		
		Avg	*Min*	*Max*
SQLite	3.10	4.20	4.09	4.25
ocean	19.64	60.02	19.90	207.14
fmm	24.87	64.45	65.29	72.83
memcached	73.87	645.99	619.32	730.37
pbzip2	15.30	360.72	61.36	763.43
ctrace	3.67	24.29	5.54	41.08
bbuf	1.81	4.47	4.77	5.82
AVV	0.72	0.83	0.78	1.02
DCL	0.74	0.85	0.83	0.89
DBM	0.72	0.81	0.79	0.83
RW	0.74	0.81	0.81	0.82

Table 10 – Portend's classification time for the 93 data races in Table 9.

We analyzed Portend's accuracy with increasing values of k and found that $k = 5$ is sufficient to achieve overall 99% accuracy for all the programs in our evaluation. Fig. 39 shows the results for Ctrace, Pbzip2, Memcached, and Bbuf. We therefore conclude that it is possible to achieve high classification accuracy with relatively small values of k.

6.3.5 *Comparison to Existing Data Race Detectors*

We compare Portend to the Record/Replay-Analyzer technique [142], Helgrind$^+$'s technique [86], and Ad-Hoc-Detector [186] in terms of the accuracy with which data races are classified. We implemented the Record/Replay-Analyzer technique in Portend and compared accuracy empirically. For the ad-hoc synchronization detection tech-

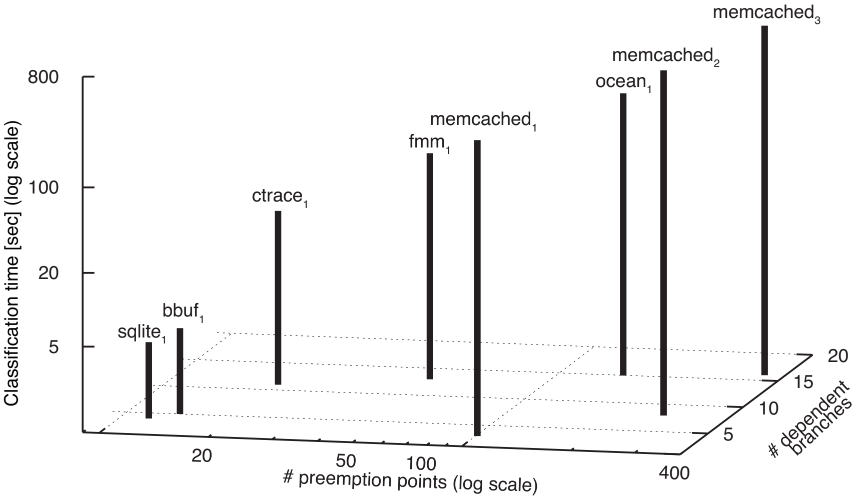


Figure 38 – Change in classification time with respect to number of preemptions and number of dependent branches for some of the data races in Table 9. Each sample point is labeled with data race id.

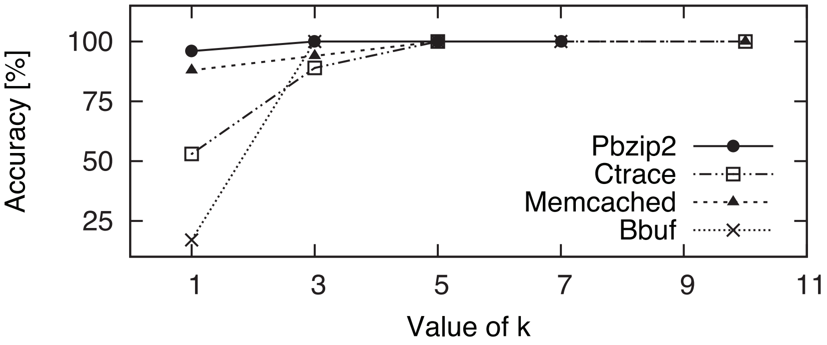


Figure 39 – Portend's accuracy with increasing values of k.

niques, since we do not have access to the implementations, we analytically derive the expected classification based on the published algorithms. We do not compare to RaceFuzzer [172], because it is primarily a bug finding tool looking for harmful data races that occur due to exceptions and memory errors; it therefore does not provide a fine-grained classification of data races. Similarly, no comparison is provided to DataCollider [91], since data race classification in this tool is based on heuristics that pertain to data races that we rarely encountered in our evaluation.

In Table 11 we show the accuracy, relying on manual inspection as "ground truth". Record/Replay-Analyzer does not tolerate replay failures and classifies data races that exhibit a post-data race state mismatch as harmful (shown as specViol), causing it to have low accuracy (10%) for that class. When comparing to Helgrind$^+$ and Ad-Hoc-Detector, we conservatively assume that these tools incur no false positives when ad-hoc synchronization is present, even though this is unlikely, given that both tools rely on heuristics. This notwithstanding, both tools are focused on weeding out data races due to ad-hoc synchronization, so they cannot properly classify the other data races (36 out of 93). In contrast, Portend classifies a wider range of data races with high accuracy.

<table>
<tr><th></th><th>specViol</th><th>k-witness</th><th>outDiff</th><th>singleOrd</th></tr>
<tr><td>Ground Truth</td><td>100%</td><td>100%</td><td>100%</td><td>100%</td></tr>
<tr><td>Record/Replay Analyzer</td><td>10%</td><td>95%</td><td colspan="2">-
(not-classified)</td></tr>
<tr><td>Ad-Hoc-Detector, Helgrind$^+$</td><td colspan="3">-
(not-classified)</td><td>100%</td></tr>
<tr><td>Portend</td><td>100%</td><td>99%</td><td>99%</td><td>100%</td></tr>
</table>

Table 11 – Accuracy for each approach and each classification category, applied to the 93 data races in Table 9. "Not-classified" means that an approach cannot perform classification for a particular class.

The main advantage of Portend over Record/Replay-Analyzer is that it is immune to replay failures. In particular, for all the data races classified by Portend as "single ordering", there was a replay divergence (that caused replay failures in Record/Replay-Analyzer), which would cause Record/Replay-Analyzer to classify the corresponding data races as harmful despite them exhibiting no apparent harmfulness; this accounts for 57 of the 84 misclassifications. Note that even if Record/Replay-Analyzer were augmented with a phase that pruned "single ordering" data races (57/93), it would still diverge on 32 of the remaining 36 data races and classify them as "spec violated", whereas only 5 are actually "spec violated". Portend, on the other hand, correctly classifies 35/36 of those remaining data races. Another advantage is that Portend classifies based on symbolic output comparison, not concrete state comparison, and therefore, its classification results can apply to a range of inputs rather than a single input.

We manually verified and, when possible, checked with developers that the data races in the "k-witness harmless" category are indeed harmless. Except for one data race, we concluded that developers intentionally left these data races in their programs because they considered them harmless. These data races match known patterns [142, 91], such as redundant writes to shared variables (e.g., we found such patterns in Ctrace). However, for one data race in Ocean, we confirmed that Portend did not figure out that the data race belongs in the "output differs" category (the data race can produce different output if a certain path in the code is followed, which depends indirectly on program input). Portend was not able to find this path even with $k = 10$ after one hour. Manual investigation revealed that this path is hard to find because it requires a very specific and complex combination of inputs.

6.3.6 *Efficiency and Effectiveness of Symbolic Memory Consistency Modeling*

The previous evaluation results were using the SMCM plugin in the sequential consistency mode. The sequential memory consistency mode is the default in Cloud9 as well as in Portend. In this section, we answer the following questions while operating the SMCM plugin in Portend's weak consistency mode: (1) Is Portend effective in discovering bugs that may surface under the weak consistency model?, (2) What is Portend's efficiency and (3) memory usage while operating the SMCM plugin in Portend's weak memory consistency mode?

We use simple micro-benchmarks that we have constructed to test the basic functionality of SMCM. The simplified source code for these micro-benchmarks can be seen in Fig. 40. These benchmarks are:

— *no-sync*: The source code for this micro-benchmark can be seen in Fig. 40-a: A program with opportunities for write reordering. Reorderings cause the `printf` statement on line 17 to produce different program outputs.

— *no-sync-bug*: The source code for this benchmark can be seen in Fig. 40-b: A program with opportunities for write reordering. A particular write reordering causes the program to crash; however the program does not crash under sequential consistency.

— *sync*: The source code for this micro-benchmark can be seen in Fig. 40-c: A program with no opportunities for write reordering. There is a data race on both `globalx` and `globaly`. Since both threads 1 and 2 write the same value 2 to `globalx`, the output of the program is the same for any execution, assuming writes are atomic[2].

— *sync-bug*: The source code for this micro-benchmark can be seen in Fig. 40-d: A program with opportunities for write reordering. The barrier synchronization does not prevent the write to `globalx` and `globaly` from reordering. A particular write reordering causes the program to crash; however the program does not crash under sequential consistency.

To evaluate Portend's effectiveness in finding bugs that may only arise under Portend's weak ordering, we ran the micro-benchmarks with Portend's SMCM plugin configured in two modes: sequential consistency (Portend-seq) and Portend's weak consistency (Portend-weak). We provide the number of bugs found by each configuration of Portend and also the percentage of possible execution states

---

[2] If writes are non-atomic, even a seemingly benign data race, where two threads write the same value to a shared variable, may end up producing unexpected results. Details of how this can happen can be found in [23]

```
int volatile globalx = 0;
int volatile globaly = 0;
```

Thread T_1

```
void * work0 (void *arg) {
  globalx = 2;
  globaly = 1;
  return 0;
}
```

Thread T_2

```
void * work1 (void *arg) {
  globalx = 2;
  return 0;
}
```

Thread Main

```
int main (int argc, char *argv[]){
  pthread_t t0, t1;
  int rc;
  rc = pthread_create (&t0, 0, work0, 0);
  rc = pthread_create (&t1, 0, work1, 0);
  printf("%d,%d", globalx, globaly);
  pthread_join(t0, 0);
  pthread_join(t1, 0);
  return 0;
}
```

(a)

```
int volatile globalx = 0;
int volatile globaly = 0;
```

Thread T_1

```
void * work0 (void *arg) {
  globalx = 2;
  globaly = 1;
  return 0;
}
```

Thread T_2

```
void * work1 (void *arg) {
  globalx = 2;
  return 0;
}
```

Thread Main

```
int main (int argc, char *argv[]){
  pthread_t t0, t1;
  int rc;
  rc = pthread_create (&t0, 0, work0, 0);
  rc = pthread_create (&t1, 0, work1, 0);
  if(globalx == 0 && globaly == 2)
    ; //crash!
  pthread_join(t0, 0);
  pthread_join(t1, 0);
  return 0;
}
```

(b)

```
int volatile globalx = 0;
int volatile globaly = 0;
```

Thread T_1

```
void * work0 (void *arg) {
  globalx = 2;
  pthread_barrier_wait(&barr);
  globaly = 1;
  return 0;
}
```

Thread T_2

```
void * work1 (void *arg) {
   globalx = 2;
   pthread_barrier_wait(&barr);
   return 0;
}
```

Thread Main

```
int main (int argc, char *argv[]){
  pthread_t t0, t1;
  int rc;
  pthread_barrier_init(&barr, NULL, 2);
  rc = pthread_create (&t0, 0, work0, 0);
  rc = pthread_create (&t1, 0, work1, 0);
  printf("%d,%d", globalx, globaly);
  pthread_join(t0, 0);
  pthread_join(t1, 0);
  return 0;
}
```

(c)

```
int volatile globalx = 0;
int volatile globaly = 0;
```

Thread T_1

```
void * work0 (void *arg) {
  globalx = 2;
  globaly = 1;
  pthread_barrier_wait(&barr);
  return 0;
}
```

Thread T_2

```
void * work1 (void *arg) {
   globalx = 2;
   pthread_barrier_wait(&barr);
   return 0;
}
```

Thread Main

```
int main (int argc, char *argv[]){
  pthread_t t0, t1;
  int rc;
  pthread_barrier_init(&barr, NULL, 2);
  rc = pthread_create (&t0, 0, work0, 0);
  rc = pthread_create (&t1, 0, work1, 0);
  if(globalx == 0 && globaly == 2)
    ; //crash!
  pthread_join(t0, 0);
  pthread_join(t1, 0);
  return 0;
}
```

(d)

Figure 40 – Micro-benchmarks used for evaluating SMCM.

that each configuration covers if Portend's weak consistency were assumed. Note that ground truth (that is the total number of states that can be covered under Portend's weak consistency) in this case is manually identified, because the number of possible states is small. Effectively identifying this percentage for arbitrarily large programs is undecidable.

We present the results in Table 12. As it can be seen, Portend-weak discovers the bugs that can only be discovered under Portend's weak consistency whereas Portend-seq cannot find those bugs because of sequential consistency assumptions. A similar reasoning applies to state exploration. Portend covers all the possible states that may arise from returning multiple values at "read"s whereas Cloud9 simply returns the last value that was written to a memory location and hence has lower coverage.

System	**Number of bugs**		**State coverage (%)**	
	Portend-seq	Portend-weak	Portend-seq	Portend-weak
no-sync	0/0	0/0	50	100
no-sync-bug	0/1	1/0	50	100
sync	0/0	0/0	100	100
sync-bug	0/1	1/1	50	100

Table 12 – Portend's effectiveness in bug finding and state coverage for two memory model configurations: sequential memory consistency mode and Portend's weak memory consistency mode.

We also evaluate the performance of Portend-weak for our microbenchmarks and compare its running time to that of Portend-seq. The results of this comparison can be seen in Fig. 41. The running times of the benchmarks under Portend-seq essentially represent the "native" LLVM interpretation time. For the *no-sync* benchmark we can see that the running time of Portend is about 2 seconds more than that of Portend-seq. This is expected as Portend-weak covers more states compared to Portend-seq.

However, it should be noted that in the case of the *no-sync-bug* benchmark, the running times are almost the same for Portend-weak and Portend-seq (although not visible on the graph, the running time of Portend-weak is slightly larger than that of Portend-seq, on the order of a few milliseconds). This is simply due to the fact that the bug in *no-sync-bug* is immediately discovered after the exploration state has been forked in Portend. The bug is printed at the program output, and the exploration ends for that particular state. Similar reasoning applies to the other benchmark pair, namely *sync* and *sync-bug*.

6.3.7 *Memory Consumption of Symbolic Memory Consistency Modeling*

In this final section of the evaluation, we measure the peak memory consumption of Portend-weak and Portend-seq for the microbenchmarks we have tested. The results can be seen in Fig. 42. The memory consumption increases for all the benchmarks. This is be-

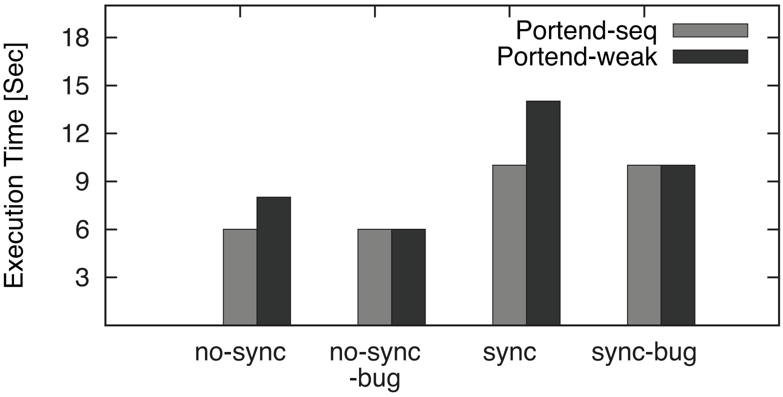


Figure 41 – Running time of Portend-weak and Portend-seq

cause for all the benchmarks, Portend-weak always forks off more states and/or performs more bookkeeping than Portend-seq, even though it does not always explore those states.

Although the memory consumption consistently increases for Portend-weak, it does not increase proportionally with the state forking. This is possible due to the copy-on-write mechanism employed for exploring states and keeping track of the happens-before graph. However, when we ran Portend-weak on real world programs, the memory consumption quickly exceeded the memory capacity of the workstation we used for our experiments. We plan on incorporating techniques like partial order reduction [62] from model checking in order to reduce the number of states that SMCM needs to explore and improve its scalability as part of future work.

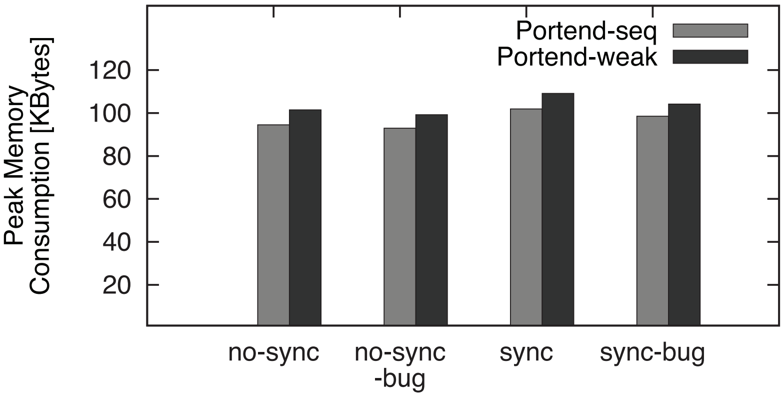


Figure 42 – Memory usage of Portend-weak and Portend-seq

In summary, Portend is able to classify with 99% accuracy and full precision all the 93 data races into four data race classes defined in §5.1 in under 5 minutes per data race on average. Furthermore, Portend correctly identifies 6 serious harmful data races. Compared to previously published data race classification methods, Portend performs more accurate classification (92 out of 93, 99%) and is able to correctly classify up to 89% more data races than existing replay-based tools (9 out of 93, 10%). Portend also correctly handles false positive data race reports. Furthermore, SMCM allows Portend to accurately perform data race classification under relaxed memory consistency models, with low overhead.

Part III

WRAPPING UP

In this final part, we discuss ongoing and future work and we present concluding remarks.

7

ONGOING AND FUTURE WORK

Our effort to better understand concurrent programs is ongoing, and in this section, we briefly mention some ongoing and future work in that direction. In particular, we talk about how various techniques we developed for root cause diagnosis can be extended to improve the security of software systems (7.1); how can we tackle privacy challenges of collaborative approaches that work in production (7.2)?; how mixed static-dynamic analysis can be used to expose deadlocks and other concurrency bugs (7.3)?; and how our detection, root cause diagnosis and classification techniques can be applied to large-scale distributed systems (7.4)?

7.1 ENHANCING SECURITY THROUGH PATH PROFILING

Root cause diagnosis using Gist allows gathering information about the control flow of a program, which can be used to enhance the security of a software system. Prior work on root cause diagnosis showed that path profiles [9] embody richer execution information than mere branches, data values or constraints on values. We speculate that we can use path profiles to also enhance the security of real-world software.

The first way in which we can improve security through path profiles is by speeding up the security auditing cycles for critical security exploits such as control flow hijacks. In order to do this, we can first build a profile of *secure* control flows by monitoring multiple user executions. We can then identify stray executions that deviate from secure control flow (à la [54] or using machine learning) and examine whether such executions exhibit any violations of security properties. We are currently exploring this avenue.

The second way in which we can use path profiles presents a generalization of the idea we presented in the previous paragraph. In particular, we are interested in finding answers to the following questions as part of future work: Can we identify *good* paths versus *bad* paths? Can we automatically infer properties about paths (e.g., performance behavior) and help developers better structure their code based on such properties? What are the meaningful boundaries of programs to monitor when gathering path information? Can we have intelligent strategies to sample path behaviors of programs (e.g., strategies better than random sampling)? We believe that answering those questions requires deep insight into software systems and constitutes an exciting future research agenda.

7.2 PRIVACY IMPLICATIONS OF COLLABORATIVE APPROACHES

In this dissertation we introduce collaborative approaches for detecting data races and for finding the root causes of failures. Both of these approaches rely on gathering execution information from end users, and therefore they both have privacy implications. In particular, the information gathered from user endpoints may compromise users' privacy by leaking sensitive user information.

We believe that privacy implications for data race detection (i.e., RaceMob) are minimal, because data race detection does not have access to the actual data values of the variables involved in a data race.

Root cause diagnosis (i.e., Gist) on the other hand, has access to actual data values that it monitors as part of the static slice that it tracks at user endpoints, and therefore it can potentially leak more private information.

We believe that both Portend and Gist can benefit from ways of quantifying [183] and limiting the amount of execution information extracted from user endpoints. It is also possible to forego monitoring data values as part of root cause diagnosis: while this will improve the level of privacy preservation, privacy can still leak through control flow events. One possible way to anonymize control flow could be through computing hashes describing the control flow of a program. However, it remains to be seen what are the right boundaries for computing hashes of control flow events. Effective computation of hashes in the presence of nondeterminism is also an open question.

7.3 EXPOSING CONCURRENCY BUGS

An effective strategy in detecting concurrency bugs is exposing them by increasing the probability of their occurrence. In this dissertation we showed how RaceMob uses schedule steering to increase the probability of manifestation of concurrency bugs (§3). Prior work used various approaches for systematically exposing concurrency bugs in the user space [150] and in the kernel space [64]. As a starting point, we have looked at program transformations to expose deadlock bugs [4].

In the future, we would like to explore whether we can use legitimate compiler transformations to increase the likelihood of a program to violate a specification (e.g., cause a crash or a hang). Prior work [193, 194] used static analysis to identify places in the code where compilers leveraged undefined behavior in their advantage, while introducing bugs. We would like to explore whether we can infer potential compiler transformations (i.e., not being performed today) that might introduce bugs in the future.

7.4 CONCURRENCY IN LARGE-SCALE DISTRIBUTED SYSTEMS

Finally, many of the problems we attacked in this dissertation have incarnations in large-scale distributed systems (e.g., Internet-scale systems). For instance, data races and atomicity violations manifest themselves as process-level races [109] that cause correctness and performance problems as well as resource leaks. We would like to adapt our techniques for the detection and root cause diagnosis of concurrency bugs in the context of large-scale distributed systems.

8

CONCLUSIONS

Concurrency bugs are some of the nastiest bugs that affect modern software. As hardware becomes increasingly parallel, concurrency bugs become more relevant. Concurrency bugs are hard to detect efficiently, because existing concurrency bug detection techniques rely on heavyweight analyses. Even when concurrency bugs are detected, it is challenging to understand which ones are truly harmful, and how can they manifest during a program's execution. The most obscure concurrency bugs are however ones that rarely recur only in production. Detection, root cause diagnosis, and classification is even more challenging in production. This is because of the strict efficiency requirements of in-production software.

In this dissertation, we develop techniques for the detection, root cause diagnosis and classification of in-production concurrency bugs. In particular this dissertation introduces:

- **The first highly-accurate data race detection technique that can be used always-on in production.** . The key idea behind this data race detection technique is a combination of in-house static analysis and a novel in-production adaptive and crowd-sourced dynamic analysis.
- **Failure sketching, the first in-production root cause diagnosis technique that does not rely on custom hardware or system checkpointing infrastructure.** The key idea behind failure sketching is performing heavyweight in-house static analysis followed by lightweight in-production dynamic analysis to gather execution events from user endpoints and build failure sketches that point developers to the root causes of failures.
- **The first highly accurate data race classification technique.** The key idea behind highly accurate data race classification is to explore multiple program paths and schedules and to observe the effects of data races on programs' externally visible outputs (rather than programs' internal state) in order to perform classification.

We built prototypes for all the techniques we introduce in this dissertation and show that our prototypes are effective, efficient, accurate and precise.

The techniques we develop collectively help reason clearly about subtle behaviors of concurrent programs and avoid concurrent programming pitfalls. We believe that in the future, our techniques can be extended to better reason about large-scale distributed systems and security properties of software systems.

BIBLIOGRAPHY

[1] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. “An Evaluation of Similarity Coefficients for Software Fault Localization.” In: *PRDC*. 2006.

[2] Sarita Adve. “Data Races Are Evil with No Exceptions: Technical Perspective.” In: *Commun. ACM* (2010).

[3] Rahul Agarwal et al. “Optimized Run-time Race Detection and Atomicity Checking Using Partial Discovered Types.” In: *ASE*. 2005.

[4] Baris Kasikci Ali Kheradmand and George Candea. “Lockout: Efficient Testing for Deadlock Bugs.” In: *Workshop on Determinism and Correctness in Parallel Programming*. 2014.

[5] Glenn Ammons and James R. Larus. “Improving Data-flow Analysis with Path Profiles.” In: *Intl. Conf. on Programming Language Design and Implem.* 1994.

[6] *Apache httpd*. `http://httpd.apache.org`. 2013.

[7] Joy Arulraj, Guoliang Jin, and Shan Lu. “Leveraging the Short-term Memory of Hardware to Diagnose Production-run Software Failures.” In: (2014).

[8] Joy Arulraj et al. “Production-run Software Failure Diagnosis via Hardware Performance Counters.” In: *ASPLOS*. 2013.

[9] Piramanayagam Arumuga Nainar and Ben Liblit. “Adaptive Bug Isolation.” In: *Intl. Conf. on Software Engineering*. 2010.

[10] Mohamed Faouzi Atig et al. “On the Verification Problem for Weak Memory Models.” In: *Symp. on Principles of Programming Languages*. 2010.

[11] Amittai Aviram et al. “Efficient system-enforced deterministic parallelism.” In: *Symp. on Operating Sys. Design and Implem.* 2010.

[12] Gogul Balakrishnan and Thomas Reps. “Analyzing Memory Accesses in x86 Executables.” In: *Intl. Conf. on Compiler Construction*. 2004.

[13] Baris Kasikci. *Are "data races" and "race condition" actually the same thing in context of concurrent programming*. `http://stackoverflow.com/questions/11276259/are-data-races-and-race-condition-actually-the-same-thing-in-context-of-conc/`. 2013.

[14] George Candea Baris Kasikci Benjamin Schubert. *Gist*. http://dslab.epfl.ch/proj/gist/. 2015.

[15] Rob von Behren et al. “Capriccio: Scalable threads for Internet services.” In: *Symp. on Operating Systems Principles*. 2003.

[16] Tom Bergan et al. “CoreDet: a compiler and runtime system for deterministic multithreaded execution.” In: *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*. 2010.

[17] Al Bessey et al. “A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World.” In: *Commun. ACM* (2010).

[18] Swarnendu Biswas, Minjia Zhang, and Michael D. Bond. *Lightweight Data Race Detection for Production Runs*. Tech. rep. OSU-CICRC-1/15-TR01. Ohio State University, 2015.

[19] Swarnendu Biswas et al. “DoubleChecker: Efficient Sound and Precise Atomicity Checking.” In: PLDI. 2014.

[20] Burton H. Bloom. “Space/Time Trade-offs in Hash Coding with Allowable Errors.” In: *Commun. ACM* (1970).

[21] Robert L. Bocchino Jr. et al. “A type and effect system for deterministic parallel Java.” In: *OOPSLA*. 2009.

[22] Hans Boehm. *Programming with Threads: Questions Frequently Asked by C and C++ Programmers*. `http://www.hboehm.info/c++mm/user-faq.html`.

[23] Hans-J. Boehm. “How to miscompile programs with "benign" data races.” In: *USENIX Workshop on Hot Topics in Parallelism*. 2011.

[24] Hans-J. Boehm. “Position paper: nondeterminism is unavoidable, but data races are pure evil.” In: *RACES*. 2012.

[25] Hans-J. Boehm and Sarita V. Adve. “Foundations of the C++ concurrency memory model.” In: *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*. Intl. Conf. on Programming Language Design and Implem. 2008.

[26] Hans-J. Boehm and Sarita V. Adve. “You Don’t Know Jack About Shared Variables or Memory Models.” In: *Commun. ACM* (2012).

[27] Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. “PACER: Proportional detection of data races.” In: *Intl. Conf. on Programming Language Design and Implem.* 2010.

[28] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. “An Infrastructure for Adaptive Dynamic Optimization.” In: *CGO*. 2003.

[29] Stefan Bucur et al. “Parallel Symbolic Execution for Automated Real-World Software Testing.” In: *ACM EuroSys European Conf. on Computer Systems*. 2011.

[30] Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. "Bounded Model Checking of Concurrent Data Types on Relaxed Memory Models: A Case Study." In: *Intl. Conf. on Computer Aided Verification*. 2006.

[31] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." In: *Symp. on Operating Sys. Design and Implem.* 2008.

[32] Cristian Cadar et al. "EXE: Automatically Generating Inputs of Death." In: *Conf. on Computer and Communication Security*. 2006.

[33] Subhachandra Chandra and Peter M. Chen. "Whither Generic Recovery from Application Faults? A Fault Study Using Open-Source Software." In: *DSN*. 2000.

[34] Haibo Chen et al. "POLUS: A POwerful Live Updating System." In: *ICSE*. 2007.

[35] Trishul M. Chilimbi et al. "HOLMES: Effective Statistical Debugging via Efficient Path Profiling." In: *Intl. Conf. on Software Engineering*. 2009.

[36] V. Chipounov and G. Candea. "Enabling sophisticated analyses of x86 binaries with RevGen." In: *Intl. Conf. on Dependable Systems and Networks*. 2011.

[37] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. "S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems." In: *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*. 2011.

[38] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. "The S2E Platform: Design, Implementation, and Applications." In: *ACM Transactions on Computer Systems* 30.1 (2012). Special issue: Best papers of ASPLOS.

[39] Vitaly Chipounov et al. "Selective Symbolic Execution." In: *Workshop on Hot Topics in Dependable Systems*. 2009.

[40] Jong-Deok Choi and Andreas Zeller. "Isolating Failure-inducing Thread Schedules." In: *ISSTA*. 2002.

[41] Jong-Deok Choi et al. "Efficient and precise datarace detection for multithreaded object-oriented programs." In: *SIGPLAN Notices* 37.5 (2002), pp. 258–269.

[42] Chris Lattner. *libc++*. http://libcxx.llvm.org/. 2012.

[43] Intel Corporation. *Intel(R) Processor Trace Decoder Library*. `https://github.com/01org/processor-trace`. 2015.

[44] Heming Cui et al. "Stable Deterministic Multithreading through Schedule Memoization." In: *Symp. on Operating Sys. Design and Implem.* 2010.

[45] *CVE's related to races*. `http://www.cvedetails.com/vulnerability-list/cweid-362/vulnerabilities.html`.

[46] Joseph Devietti et al. "DMP: deterministic shared memory multiprocessing." In: *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*. 2009.

[47] Anne Dinning and Edith Schonberg. "Detecting Access Anomalies in Programs with Critical Sections." In: *SIGPLAN Not.* (1991).

[48] *DRD*. `http://valgrind.org/docs/manual/drd-manual.html`. 2015.

[49] Michel Dubois, Christoph Scheurich, and Faye Briggs. "Memory Access Buffering in Multiprocessors." In: *Proc. 13th Ann. Intl. Symp. on Computer Architecture* (1986), pp. 374–442.

[50] Laura Effinger-Dean et al. "IFRit: interference-free regions for dynamic data-race detection." In: Conf. on Object-Oriented Programming, Systems, Languages, and Applications. 2012.

[51] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. "Goldilocks: A race and transaction-aware Java runtime." In: *Intl. Conf. on Programming Language Design and Implem.* San Diego, California, USA, 2007.

[52] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. "Goldilocks: Efficiently Computing the Happens-Before Relation Using Locksets." In: *Intl. Conf. on Runtime Verification*. 2006.

[53] Dawson Engler and Ken Ashcraft. "RacerX: Effective, Static Detection of Race Conditions and Deadlocks." In: *Symp. on Operating Systems Principles*. 2003.

[54] Dawson Engler et al. "Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code." In: *Symp. on Operating Systems Principles*. 2001.

[55] Peter Eriksson. *Parallel File Scanner*. http://ostatic.com/pfscan. 2013.

[56] Brad Fitzpatrick. *Memcached*. http://memcached.org. 2013.

[57] Cormac Flanagan and Stephen N. Freund. "Adversarial memory for detecting destructive races." In: *Intl. Conf. on Programming Language Design and Implem.* 2010.

[58] Cormac Flanagan and Stephen N Freund. "Atomizer: A dynamic atomicity checker for multithreaded programs." In: *SIGPLAN Notices* 39.1 (2004), pp. 256–267.

[59] Cormac Flanagan and Stephen N. Freund. "FastTrack: Efficient and precise dynamic race detection." In: *Intl. Conf. on Programming Language Design and Implem.* 2009.

[60] Cormac Flanagan and Stephen N. Freund. "Type-based Race Detection for Java." In: PLDI '00. 2000.

[61] Cormac Flanagan, Stephen N. Freund, and Jaeheon Yi. "Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs." In: *Intl. Conf. on Programming Language Design and Implem.* 2008.

[62] Cormac Flanagan and Patrice Godefroid. "Dynamic partial-order reduction for model checking software." In: 2005.

[63] Pedro Fonseca, Cheng Li, and Rodrigo Rodrigues. "Finding complex concurrency bugs in large multi-threaded applications." In: *ACM EuroSys European Conf. on Computer Systems*. 2011.

[64] Pedro Fonseca, Rodrigo Rodrigues, and Björn B. Brandenburg. "SKI: exposing kernel concurrency bugs through systematic schedule exploration." In: OSDI. 2014.

[65] Vijay Ganesh and David L. Dill. "A decision procedure for bit-vectors and arrays." In: *Intl. Conf. on Computer Aided Verification*. 2007.

[66] Jeff Gilchrist. *Parallel BZIP2*. http://compression.ca/pbzip2. 2013.

[67] Kirk Glerum et al. "Debugging in the (very) large: ten years of implementation and experience." In: *Symp. on Operating Systems Principles*. 2009.

[68] Patrice Godefroid, Michael Y. Levin, and David Molnar. "Automated Whitebox Fuzz Testing." In: *Network and Distributed System Security Symp.* 2008.

[69] Patrice Godefroid and Nachiappan Nagappan. "Concurrency at Microsoft – An Exploratory Survey." In: *Intl. Conf. on Computer Aided Verification*. 2008.

[70] *Google Courgette*. https://chromium.googlesource.com/chromium/src/courgette/+/master.

[71] Jim Gray. *Why Do Computers Stop And What Can Be Done About It?* 1985.

[72] *Hacking Starbucks for unlimited coffee*. `http://sakurity.com/blog/2015/05/21/starbucks.html`.

[73] Steven Hand. "An experiment in determinism." In: *CACM* (2012).

[74] Matthias Hauswirth and Trishul M. Chilimbi. "Low-overhead Memory Leak Detection Using Adaptive Statistical Profiling." In: *ASPLOS*. 2004.

[75] *Helgrind*. `http://valgrind.org/docs/manual/hg-manual.html`. 2012.

[76] Maurice Herlihy and J. Eliot B. Moss. "Transactional memory: Architectural support for lock-free data structures." In: *Intl. Symp. on Computer Architecture*. 1993.

[77] Maurice P. Herlihy and Jeannette M. Wing. "Linearizability: A Correctness Condition for Concurrent Objects." In: *TOPLAS* (1990).

[78] C. A. R. Hoare. "Monitors: An Operating System Structuring Concept." In: *Communications of the ACM* 17.10 (1974).

[79] Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. "Maximal Sound Predictive Race Detection with Control Flow Abstraction." In: *SIGPLAN Not.* (2014).

[80] IEEE. "1003.1 Standard for Information Technology Portable Operating System Interface (POSIX) Rationale (Informative)." In: *IEEE Std 1003.1-2001. Rationale (Informative)* (2001).

[81] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Vol. 2. 325383-038US. 2015.

[82] Intel Corp. *Parallel Inspector*. http://software.intel.com/en-us/articles/intel-parallel-inspector. 2012.

[83] Intel Corporation. *Intel Processor Trace*. `https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing`. 2013.

[84] *ISO/IEC 14882:2011: Information technology – Programming languages – C++*. International Organization for Standardization. 2011.

[85] *ISO/IEC 9899:2011: Information technology – Programming languages – C*. International Organization for Standardization. 2011.

[86] Ali Jannesari and Walter F. Tichy. "Identifying Ad-hoc Synchronization for Enhanced Race Detection." In: *Intl. Parallel and Distributed Processing Symp.* 2010.

[87] *Java Synchronized Methods*. https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html.

[88] Yang Liu Jiaqi Zhang Weiwei Xiong et al. "ATDetector: Improving the Accuracy of a Commercial Data Race Detector by Identifying Address Transfer." In: *IEEE/ACM International Symposium on Microarchitecture*. 2011.

[89] Guoliang Jin et al. "Instrumentation and sampling strategies for cooperative concurrency bug isolation." In: *SIGPLAN Not.* (2010).

[90] John Criswell. *The Information Flow Compiler*. `https://llvm.org/svn/llvm-project/giri/`. 2011.

[91] Sebastian Burckhardt John Erickson Madanlal Musuvathi and Kirk Olynyk. "Effective Data-Race Detection for the Kernel." In: *Symp. on Operating Sys. Design and Implem.* 2010.

[92] John Regehr. *Race Condition vs. Data Race*. `http://blog.regehr.org/archives/490`. 2011.

[93] James A. Jones and Mary Jean Harrold. "Empirical Evaluation of the Tarantula Automatic Fault-localization Technique." In: *ASE*. 2005.

[94] Vineet Kahlon, Franjo Ivančić, and Aarti Gupta. "Reasoning About Threads Communicating via Locks." In: *Intl. Conf. on Computer Aided Verification*. 2005.

[95] Vineet Kahlon et al. "Fast and Accurate Static Data-race Detection for Concurrent Programs." In: *CAV*. 2007.

[96] Vineet Kahlon et al. "Static Data Race Detection for Concurrent Programs with Asynchronous Calls." In: FSE. 2009.

[97] Baris Kasikci, Cristian Zamfir, and George Candea. "Automated Classification of Data Races Under Both Strong and Weak Memory Models." In: *ACM Transactions on Programming Languages and Systems* 37.3 (2015).

[98] Baris Kasikci, Cristian Zamfir, and George Candea. "Data Races vs. Data Race Bugs: Telling the Difference with Portend." In: *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*. 2012.

[99] Baris Kasikci, Cristian Zamfir, and George Candea. "RaceMob: Crowdsourced Data Race Detection." In: *Symp. on Operating Systems Principles*. 2013.

[100] Baris Kasikci et al. "Efficient Tracing of Cold Code Via Bias-Free Sampling." In: *USENIX Annual Technical Conf.* 2014.

[101] Baris Kasikci et al. "Failure Sketches: A Better Way to Debug." In: *Workshop on Hot Topics in Operating Systems*. 2015.

[102] Baris Kasikci et al. "Failure Sketching: A Technique for Automated Root Cause Diagnosis of In-Production Failures." In: *Symp. on Operating Systems Principles*. 2015.

[103] M. G. Kendall. "A New Measure of Rank Correlation." In: *Biometrika* (1938).

[104] James C. King. "A new approach to program testing." In: *Intl. Conf. on Reliable Software*. 1975.

[105] James C. King. "Symbolic execution and program testing." In: *Communications of the ACM* (1976).

[106] Andi Kleen. *Announcing simple-pt - A simple Processor Trace implementation*. `http://halobates.de/blog/p/344`. 2015.

[107] Andi Kleen. *simple-pt Linux driver*. `https://github.com/andikleen/simple-pt`. 2015.

[108] Volodymyr Kuznetsov et al. “Efficient state merging in symbolic execution.” In: *Intl. Conf. on Programming Language Design and Implem.* 2012.

[109] Oren Laadan et al. “Pervasive detection of process races in deployed systems.” In: *Symp. on Operating Systems Principles*. 2011.

[110] Leslie Lamport. “Time, clocks, and the ordering of events in a distributed system.” In: *Communications of the ACM* 21.7 (1978).

[111] Butler W. Lampson and David D. Redell. “Experience with Processes and Monitors in Mesa.” In: *Communications of the ACM* 23.2 (1980).

[112] Chris Lattner. “Macroscopic Data Structure Analysis and Optimization.” PhD thesis. University of Illinois at Urbana-Champaign, May 2005.

[113] Chris Lattner and Vikram Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation.” In: *Intl. Symp. on Code Generation and Optimization*. 2004.

[114] Dongyoon Lee et al. “Chimera: Hybrid program analysis for determinism.” In: *Intl. Conf. on Programming Language Design and Implem.* 2012.

[115] Nancy G. Leveson and Clark S. Turner. “An Investigation of the Therac-25 Accidents.” In: *IEEE Computer* (1993).

[116] Ben Liblit et al. “Bug isolation via remote program sampling.” In: *Intl. Conf. on Programming Language Design and Implem.* 2003.

[117] Ben Liblit et al. “Scalable Statistical Bug Isolation.” In: *Intl. Conf. on Programming Language Design and Implem.* 2005.

[118] Benjamin Robert Liblit. “Cooperative Bug Isolation.” PhD thesis. University of California, Berkeley, Dec. 2004.

[119] *Linux branch with Intel PT support*. https://github.com/virtuoso/linux-perf/tree/intel_pt. 2015.

[120] Richard J. Lipton. “Reduction: A Method of Proving Properties of Parallel Programs.” In: *Commun. ACM* (1975).

[121] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. “Dthreads: efficient deterministic multithreading.” In: *Symp. on Operating Systems Principles*. 2011.

[122] Shan Lu. “Understanding, Detecting and Exposing Concurrency Bugs.” PhD thesis. UIUC, 2008.

[123] Shan Lu et al. “AVIO: Detecting Atomicity Violations via Access Interleaving Invariants.” In: ASPLOS. 2006.

[124] Shan Lu et al. “Learning from Mistakes – A Comprehensive Study on Real World Concurrency Bug Characteristics.” In: *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*. 2008.

[125] Brandon Lucia et al. “Atom-Aid: Detecting and Surviving Atomicity Violations.” In: ISCA. 2008.

[126] Chi-Keung Luk et al. “PIN: building customized program analysis tools with dynamic instrumentation.” In: *Intl. Conf. on Programming Language Design and Implem.* 2005.

[127] Nuno Machado, Brandon Lucia, and Luís Rodrigues. “Concurrency Debugging with Differential Schedule Projections.” In: *PLDI*. 2015.

[128] Jeremy Manson, William Pugh, and Sarita V. Adve. “The Java memory model.” In: *Symp. on Principles of Programming Languages*. 2005.

[129] Jeremy Manson, William Pugh, and Sarita V. Adve. “The Java Memory Model.” In: POPL. 2005.

[130] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. “LiteRace: Effective sampling for lightweight data-race detection.” In: *Intl. Conf. on Programming Language Design and Implem.* 2009.

[131] Daniel MarjamÃ€ki. *Cppcheck*. http://cppcheck.sourceforge.net/. 2015.

[132] Cal McPherson. *Ctrace*. http://ctrace.sourceforge.net. 2012.

[133] *Memcached issue 127*. `http://code.google.com/p/memcached/issues/detail?id=127`.

[134] Scott Meyers and Andrei Alexandrescu. *C++ and The Perils of Double-Checked Locking*. `http://www.drdobbs.com/184405772`.

[135] Barton P. Miller et al. “The Paradyn Parallel Performance Measurement Tool.” In: *Computer* (1995).

[136] Madanlal Musuvathi et al. “A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs.” In: *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*. 2010.

[137] Madanlal Musuvathi et al. “Finding and Reproducing Heisenbugs in Concurrent Programs.” In: *Symp. on Operating Sys. Design and Implem.* 2008.

[138] Abdullah Muzahid et al. “SigRace: Signature-based Data Race Detection.” In: ISCA. 2009.

[139] Mayur Naik and Alex Aiken. “Conditional Must Not Aliasing for Static Race Detection.” In: *SIGPLAN Not.* (2007).

[140] Mayur Naik, Alex Aiken, and John Whaley. "Effective static race detection for Java." In: *Intl. Conf. on Programming Language Design and Implem.* 2006.

[141] Mayur Naik, Alex Aiken, and John Whaley. "Effective static race detection for Java." In: *Intl. Conf. on Programming Language Design and Implem.* 2006.

[142] Satish Narayanasamy et al. "Automatically classifying benign and harmful data races using replay analysis." In: *Intl. Conf. on Programming Language Design and Implem.* (2007).

[143] George C. Necula et al. "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs." In: *Intl. Conf. on Compiler Construction*. 2002.

[144] Hiroyasu Nishiyama. "Detecting Data Races Using Dynamic Escape Analysis Based on Read Barrier." In: VM. 2004.

[145] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. "Exterminator: Automatically correcting memory errors with high probability." In: *Intl. Conf. on Programming Language Design and Implem.* 2007.

[146] Robert O'Callahan and Jong-Deok Choi. "Hybrid dynamic data race detection." In: *Symp. on Principles and Practice of Parallel Computing*. 2003.

[147] Christos H. Papadimitriou. "The Serializability of Concurrent Database Updates." In: *J. ACM* (1979).

[148] Mark S. Papamarcos and Janak H. Patel. "A Low-overhead Coherence Solution for Multiprocessors with Private Cache Memories." In: *ISCA*. 1984.

[149] Chang-Seo Park and Koushik Sen. "Randomized Active Atomicity Violation Detection in Concurrent Programs." In: *FSE*. 2008.

[150] Soyeon Park, Shan Lu, and Yuanyuan Zhou. "CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places." In: ASPLOS. 2009.

[151] Soyeon Park et al. "Do You Have to Reproduce the Bug at the First Replay Attempt? – PRES: Probabilistic Replay with Execution Sketching on Multiprocessors." In: *Symp. on Operating Systems Principles*. 2009.

[152] Jeff H. Perkins et al. "Automatically Patching Errors in Deployed Software." In: *Symp. on Operating Sys. Design and Implem.* 2010.

[153] Eli Pozniansky and Assaf Schuster. "Efficient on-the-fly data race detection in multithreaded C++ programs." In: *Symp. on Principles and Practice of Parallel Computing*. 2003.

[154] Eli Pozniansky and Assaf Schuster. “MultiRace: Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs: Research Articles.” In: *Concurrency and Computation: Practice and Experience* (2007).

[155] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. “LOCKSMITH: context-sensitive correlation analysis for race detection.” In: *Intl. Conf. on Programming Language Design and Implem.* 2006.

[156] Christoph von Praun and Thomas R. Gross. “Object Race Detection.” In: *SIGPLAN Not.* (2001).

[157] Christoph von Praun and Thomas R. Gross. “Static Conflict Analysis for Multi-threaded Object-oriented Programs.” In: PLDI. 2003.

[158] Christoph von Praun and Thomas R. Gross. “Static Detection of Atomicity Violations in Object-Oriented Programs.” In: *Journal of Object Technology* 3.6 (2004), pp. 103–122.

[159] Feng Qin et al. “Rx: Treating bugs as allergies – a safe method to survive software failures.” In: *ACM Transactions on Computer Systems* 25.3 (2007).

[160] Quora. *What is a coder’s worst nightmware?* `http://www.quora.com/What-is-a-coders-worst-nightmare`.

[161] Ravi Rajwar and James R. Goodman. “Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution.” In: MICRO. 2001.

[162] Sadun Anik Rastislav Bodik. “Path-sensitive value-flow analysis.” In: *Symp. on Principles of Programming Languages*. 1998.

[163] David D. Redell et al. “Pilot: An Operating System for a Personal Computer.” In: (1980).

[164] C. J. Van Rijsbergen. *Information Retrieval*. Butterworth-Heinemann, 1979.

[165] Martin C. Rinard and Monica S. Lam. “The Design, Implementation, and Evaluation of Jade.” In: *ACM Trans. Program. Lang. Syst.* (1998).

[166] Caitlin Sadowski and Jaeheon Yi. “How Developers Use Data Race Detection Tools.” In: *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools*. PLATEAU. 2014.

[167] Swarup Kumar Sahoo, John Criswell, and Vikram Adve. “An Empirical Study of Reported Bugs in Server Software with Implications for Automated Bug Diagnosis.” In: ICSE. 2010.

[168] Swarup Kumar Sahoo et al. “Using Likely Invariants for Automated Software Fault Localization.” In: (2013).

[169] Amit Sasturkar et al. "Automated Type-based Analysis of Data Races and Atomicity." In: *PPoPP*. 2005.

[170] Stefan Savage et al. "Eraser: A dynamic data race detector for multithreaded programs." In: *ACM Transactions on Computer Systems* 15.4 (1997).

[171] Edith Schonberg. "On-the-fly detection of access anomalies (with retrospective)." In: *SIGPLAN Notices* 39.4 (2004).

[172] Koushik Sen. "Race directed random testing of concurrent programs." In: *Intl. Conf. on Programming Language Design and Implem.* (2008).

[173] Koushik Sen, Darko Marinov, and Gul Agha. "CUTE: a concolic unit testing engine for C." In: *Symp. on the Foundations of Software Eng.* 2005.

[174] Konstantin Serebryany and Timur Iskhodzhanov. "ThreadSanitizer - Data race detection in practice." In: *Workshop on Binary Instrumentation and Applications*. 2009.

[175] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. *SPLASH: Stanford Parallel Applications for Shared Memory*. Tech. rep. CSL-TR-92-526. Stanford University Computer Systems Laboratory, 1992.

[176] Richard L. Sites, ed. *Alpha architecture reference manual*. 1992.

[177] Jiri Slaby. *LLVM Slicer*. https://github.com/jirislaby/LLVMSlicer/. 2014.

[178] Yannis Smaragdakis et al. "Sound Predictive Race Detection in Polynomial Time." In: (2012).

[179] *SQLite*. http://www.sqlite.org/. 2013.

[180] Daniel Stenberg. *Curl*. `http://curl.haxx.se/`. 2015.

[181] Daniel Stenberg. *Curl bug 965*. http://sourceforge.net/p/curl/bugs/965/. 2013.

[182] Bill Stoddard. *Apache bug 21287*. https://bz.apache.org/bugzilla/show_bug.cgi?id=21287. 2003.

[183] Latanya Sweeney. "K-Anonymity: A Model for Protecting Privacy." In: *Intl. Journal on Uncertainty, Fuzziness and Knowledge-based Systems*. 2002.

[184] The Associated Press. *General Electric Acknowledges Northeastern Blackout Bug*. http://www.securityfocus.com/news/8032. Feb. 12, 2004.

[185] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. "StreamIt: A Language for Streaming Applications." In: *CC*. 2002.

[186] Chen Tian et al. "Dynamic recognition of synchronization operations for improved data race detection." In: *Intl. Symp. on Software Testing and Analysis*. 2008.

[187] *TIOBE Programming Community Index*. `http://www.tiobe.com/tiobe_index/`. Nov. 2004.

[188] Nicholas Hunt Tom Bergan Joseph Devietti and Luis Ceze. "The Deterministic Execution Hammer: How Well Does it Actually Pound Nails?" In: *Workshop on Determinism and Correctness in Parallel Programming*. 2011.

[189] *Transmission*. `http://www.transmissionbt.com/`. 2015.

[190] Joseph Tucek et al. "Triage: diagnosing production run failures at the user's site." In: *Symp. on Operating Systems Principles*. 2007.

[191] Kaushik Veeraraghavan et al. "Detecting and surviving data races using complementary schedules." In: *Symp. on Operating Systems Principles*. 2011.

[192] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. "RELAY: Static race detection on millions of lines of code." In: *Symp. on the Foundations of Software Eng.* 2007.

[193] Xi Wang et al. "A Differential Approach to Undefined Behavior Detection." In: *TOCS* (2015).

[194] Xi Wang et al. "Towards Optimization-safe Systems: Analyzing the Impact of Undefined Behavior." In: SOSP. 2013.

[195] Yan Wang et al. "DrDebug: Deterministic Replay Based Cyclic Debugging with Dynamic Slicing." In: *CGO*. 2014.

[196] David L. Weaver and Tom Germond, eds. *The SPARC Architecture Manual, Version 9*. 1994.

[197] Josef Weidendorfer. *KCachegrind*. http://kcachegrind.sourceforge.net/html/Home.html. 2015.

[198] Weining Gu, Zbigniew Kalbarczyk, Ravishankar K. Iyer, Zhen-Yu Yang. *Characterization of Linux Kernel Behavior under Errors*. 2003.

[199] Mark Weiser. "Program slicing." In: *Intl. Conf. on Software Engineering*. 1981.

[200] David Wheeler. *SLOCCount*. http://www.dwheeler.com/sloccount/. 2010.

[201] P. F. Wilson, L. D. Dell, and G. F. Anderson. *Root Cause Analysis : A Tool for Total Quality Management*. American Society for Quality, 1993.

[202] Robert P. Wilson and Monica S. Lam. "Efficient context-sensitive pointer analysis for C programs." In: *Intl. Conf. on Programming Language Design and Implem.* La Jolla, CA, 1995.

[203] *Windows Process and Thread Functions*. https://msdn.microsoft.com/en-us/library/windows/desktop/ms684847(v=vs.85).aspx.

[204] Steven Cameron Woo et al. “The SPLASH-2 programs: characterization and methodological considerations.” In: *Intl. Symp. on Computer Architecture* (1995).

[205] Jingyue Wu, Heming Cui, and Junfeng Yang. “Bypassing races in live applications with execution filters.” In: *Symp. on Operating Sys. Design and Implem.* 2010.

[206] Weiwei Xiong et al. “Ad-Hoc Synchronization Considered Harmful.” In: *Symp. on Operating Sys. Design and Implem.* 2010.

[207] Min Xu, Rastislav Bodík, and Mark D. Hill. “A Serializability Violation Detector for Shared-memory Server Programs.” In: PLDI. 2005.

[208] Yu Yang et al. “Distributed dynamic partial order reduction based verification of threaded software.” In: *Intl. SPIN Workshop*. 2007.

[209] Jie Yu and Satish Narayanasamy. “A case for an interleaving constrained shared-memory multi-processor.” In: *Intl. Symp. on Computer Architecture*. Austin, TX, USA, 2009.

[210] Jie Yu and Satish Narayanasamy. “A Case for an Interleaving Constrained Shared-Memory Multi-Processor.” In: *Intl. Symp. on Computer Architecture*. 2009.

[211] Yuan Yu, Tom Rodeheffer, and Wei Chen. “RaceTrack: Efficient detection of data race conditions via adaptive tracking.” In: *Symp. on Operating Systems Principles*. 2005.

[212] Ding Yuan et al. “SherLog: error diagnosis by connecting clues from run-time logs.” In: *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*. 2010.

[213] Cristian Zamfir and George Candea. “Execution Synthesis: A Technique for Automated Debugging.” In: *ACM EuroSys European Conf. on Computer Systems*. 2010.

[214] Cristian Zamfir et al. “Automated Debugging for Arbitrarily Long Executions.” In: *Workshop on Hot Topics in Operating Systems*. 2013.

[215] Cristian Zamfir et al. “Debug Determinism: The Sweet Spot for Replay-Based Debugging.” In: *Workshop on Hot Topics in Operating Systems*. 2011.

[216] Andreas Zeller and Ralf Hildebrandt. “Simplifying and Isolating Failure-Inducing Input.” In: *IEEE Transactions on Software Engineering* (2002).

[217] Wei Zhang et al. “ConSeq: Detecting Concurrency Bugs through Sequential Errors.” In: *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*. 2011.

[218] Pin Zhou, Radu Teodorescu, and Yuanyuan Zhou. “HARD: Hardware Assisted Lockset-based Race Detection.” In: *International Symposium on High-Performance Computer Architecture*. 2007.