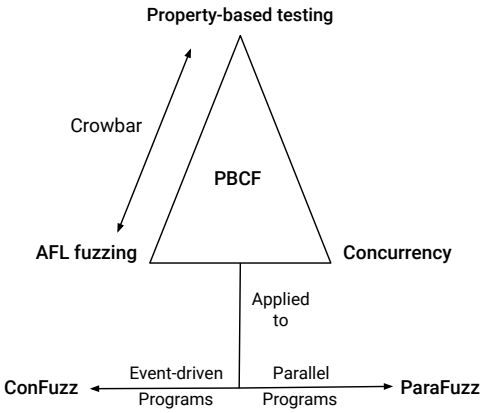
DEPARTMENT OF
COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF
TECHNOLOGY
MADRAS
CHENNAI-600 036

Coverage-guided Property Fuzzing for Event-driven and Parallel programs



A thesis

Submitted by

Sumit Padhiyar

For the award of the degree

Of

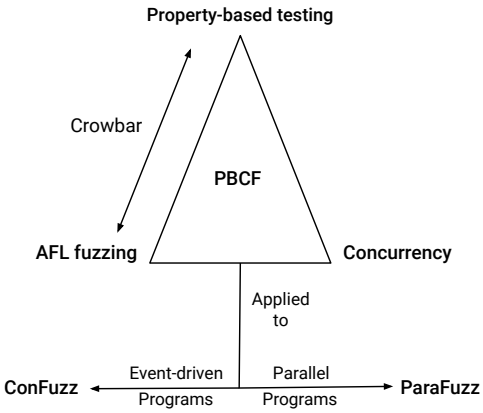
MASTER OF SCIENCE by

Research

June, 2021

DEPARTMENT OF
COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF
TECHNOLOGY
MADRAS
CHENNAI-600 036

Coverage-guided Property Fuzzing for Event-driven and Parallel programs



A thesis

Submitted by

Sumit Padhiyar

For the award of the degree

Of

MASTER OF SCIENCE by

Research

June, 2021

To my parents for their unconditional support.

THESIS CERTIFICATE

This is to undertake that the thesis titled, *Coverage-guided Property Fuzzing for event-driven and parallel programs* submitted by me to the Indian Institute of Technology Madras, for the award of M.S. is a bonafide record of the research work done by me under the supervision of Dr. KC Sivaramakrishnan. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Chennai 600 036 **Research Scholar**

Date:

Research Guide/ Co-ordinator*

**Applicable to External Registration candidates only*

List of Publications

The publications arising out of the work mentioned in this thesis are given as follows:

1. Sumit Padhiyar, KC Sivaramakrishnan, **ConFuzz: Coverage-guided Property Fuzzing for Event-driven Programs**, in The 23rd International Symposium on Practical Aspects of Declarative Languages (PADL) 2021, https://doi.org/10.1007/978-3-030-67438-0_8

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor KC Sivaramakrishnan, who enthusiastically took a novice without any research experience under him and guided him for the next 2.5 years of the MS research journey. He taught me the nuances of research, how to distill ideas, trying out different things, and communicating your research to a wider audience. I am thankful for his guidance and support.

I also take this opportunity to express my sincere thanks to my General Test Committee members Prof. B. Ravindran, Prof. Krishna Nandivada V, and Dr. S P Suresh for their interest, encouragement, valuable suggestions, and thoughtful reviews.

I would take this opportunity to thank my research group colleague Shashank Dubey who was my go-to person for the entire MS program. His guidance helped me immensely during my research journey. A special note of thanks to Adharsh Kamath who helped me get the ParaFuzz tool up and running. Thanks to Anmol Sahoo for helping me with the OCaml ecosystem.

Finally, I would like to thanks my parents for their unconditional support in every decision of my life.

ABSTRACT

KEYWORDS: Concurrency testing; Fuzzing.

With the advancement in processor technologies, more and more software is being developed to utilize the multi-processing capabilities of multicore processors. Multicore processors have become the norm in mobile, desktop, and enterprise computing. Cloud computing which has enabled easy access to fast multicore machines has fueled the growth of such applications. Programmers use concurrent programming to harness the power of multicore processors. Concurrent programming enables simultaneous execution of tasks and I/O operations to reduce the time it takes for an application to process the user request. These simultaneous executions introduce non-determinism in concurrent and parallel programs. Due to non-determinism, a program may exhibit different behaviors when executed multiple times. This non-determinism arising in program execution gives rise to concurrency bugs. Concurrency bugs make bug-free concurrent programs hard to write.

Testing concurrent program is a hard problem as concurrency bugs in programs typically manifest only when the program is executed under a particular buggy thread/event schedule. As a result, conventional testing methodologies for concurrent programs like stress testing and random testing, which explore random schedules, have

a strong chance of missing buggy schedules. Techniques under the umbrella of model checking are time-consuming and not practical enough to be used during application development. This leaves a void for an efficient and practical concurrency testing technique for programmers wanting to test their applications for concurrency bugs in a reasonable amount of time.

In this thesis, we argue that coverage-guided fuzzing can be used effectively for concurrency testing event-driven concurrent and shared-memory parallel programs. The key insight is to state high-level program properties as assertions in the program and then use fuzzing to generate a schedule for the concurrent and the parallel programs that can falsify the assertion, thus finding the concurrency bug. The thesis presents three major contributions each focused on concurrency testing: (1) a novel concurrency testing technique called coverage-guided property-based concurrency fuzzing (PBCF) that combines property-based testing with mutation-based, grey box fuzzing (2) ConFuzz which is an instantiation of PBCF to test event-driven concurrent OCaml programs (3) ParaFuzz which is an instantiation of PBCF to test shared-memory parallel OCaml programs.

TABLE OF CONTENTS

Page

ACKNOWLEDGEMENTS i

ABSTRACT ii

LIST OF TABLES viii

LIST OF FIGURES x

ABBREVIATIONS xi

1 INTRODUCTION 1

1.1 The problem 1

1.2 My Thesis 2

1.3 Contributions 3

1.3.1 PBCF: Coverage-guided Property-based Concurrency Fuzzing 3

1.3.2 ConFuzz: Coverage-guided Property Fuzzing for Event-driven Programs 4

1.3.3 ParaFuzz: Coverage-guided Property Fuzzing for Parallel Programs 5

1.4 Thesis overview 7

2 Preliminaries 8

2.1 Existing concurrency testing techniques 8

2.1.1 Stress and Random Testing 9

2.1.2 Model checking 9

2.2 Property-based Testing 10

2.3 Fuzzing 11

2.3.1 American Fuzzy Lop (AFL) 13

2.4 Crowbar 13

2.5 Concurrent Programming in OCaml 14

2.5.1 Lwt: Event-driven programming 14

2.5.2 Multicore OCaml: Shared-memory Parallel programming . . 15

3 COMBINING CONCURRENCY WITH COVERAGE-GUIDED PROPERTY FUZZING 17

3.1 Problem with property-based fuzzing + concurrency 17

3.2 Coverage-guided property-based concurrency fuzzing 18

4 CONFUZZ 21

4.1 Motivating example 24

4.2 Concurrency bugs in Lwt 27

4.2.1 Lwt: Event-driven model 28

4.2.2 Bug patterns 30

4.3 ConFuzz 32

4.3.1 Architecture 33

4.3.2 ConFuzz on the motivating example 35

4.4 Fuzzing under Non-determinism 36

4.4.1 Non-determinism in Lwt 36

4.4.2 Capturing Non-determinism 38

4.4.3 ConFuzz scheduler 41

4.5 Evaluation 42
4.5.1 Experimental subjects and setup 43
4.5.2 Experimental results 45
4.6 Limitations 52
4.7 Related Work 53
4.8 Conclusion 55

5 PARAFUZZ **56**

5.1 Motivating Example 57
5.2 Multicore OCaml 60
5.2.1 Domains 60
5.2.2 Effect handlers 60
5.3 ParaFuzz 62
5.3.1 ParaFuzz scheduler 63
5.4 Fuzzing under non-determinism 68
5.4.1 Synchronization points in Multicore OCaml 68
5.4.2 Capturing synchronization points in Multicore OCaml 72
5.5 Assumptions 77
5.6 Evaluation 79
5.6.1 Experimental subjects and setup 80
5.6.2 Finding novel bug in domainslib 82
5.6.3 Experimental Results 83
5.7 Related work 89
5.8 Conclusion 92

6 CONCLUDING REMARKS AND FUTURE WORK **93**

6.1 PBCF 93

6.2 ConFuzz and ParaFuzz 94

LIST OF TABLES

Table	Title	Page
4.1	Comparing different testing techniques	25
4.2	Experimental subjects	43
4.3	Bug detection capability of the techniques. Each entry is the fraction of the testing runs that manifested the concurrency bug.	45
4.4	Mean time to find the concurrency bug (seconds)	46
5.1	Comparing different testing techniques	59
5.2	Experimental subjects	80
5.3	Bug detection capability of the techniques. Each entry is the fraction of the testing runs that manifested the concurrency bug.	83
5.4	Mean time to find the concurrency bug (seconds)	84

LIST OF FIGURES

3.1 PBCF technique and its application in ConFuzz and ParaFuzz . . . 19

4.1 A program with a concurrency bug . . . 24

4.2 A program with a concurrency bug . . . 26

4.3 Lwt event model . . . 28

4.4 ocaml-redis#25: Order violation in ocaml-redis. Both Lwt_unix.connect and send_request are asynchronous, and can execute in any order. . . . 30

4.5 ocaml-redis#40: Atomicity violation in ocaml-redis. . . . 32

4.6 ConFuzz architecture . . . 33

4.7 ConFuzz scheduler . . . 41

4.8 Efficiency of ConFuzz as schedule space increases. The total number of schedules is given by $f(n) = (3!)^{(10*n+20)/2)}$. The labels on the x-axis show $(n, f(n))$. . . . 47

4.9 Irmin bug #270 . . . 49

4.10 Mirage-tcpip bug #86 . . . 51

5.1 A program with input+schedule concurrency bug . . . 58

5.2 ParaFuzz architecture . . . 62

5.3 Scheduler API . . . 65

5.4 ParaFuzz Scheduler effect implementation . . . 67

5.5 ParaFuzz Scheduler run function . . . 69

5.6 Domain API . . . 70

5.7 Atomic API . . . 71

5.8 Mutex API . . . 72

5.9 Condition API 73

5.10 Scheduler.fork 74

5.11 Scheduler.context_switch 75

5.12 Scheduler.suspend 76

5.13 Scheduler.resume 76

5.14 A program with schedule-dependent concurrency bug 87

5.15 Efficiency of ParaFuzz as schedule space increases. The total number of schedules is given by $f(n) = \binom{2n}{n}$. The labels on the x-axis shows $(n, f(n))$. 88

ABBREVIATIONS

AFL	American Fuzzy Lop
PBCF	Property-based Concurrency Fuzzing
CV	Condition Variable

CHAPTER 1

INTRODUCTION

With the advancement in processor technologies, modern software applications increasingly utilize the multi-processing capabilities of multicore processors. Multicore processors have become the norm in mobile, desktop, and enterprise computing. Cloud computing which has enabled easy access to fast multicore machines has fueled the growth of such applications. As a result, programmers use concurrent programming to harness the power of multicore processors. On the other hand, concurrent programming is also used for applications that run on uniprocessors. Event-driven systems such as UI applications, web browsers, and web services involve heavy usage of concurrent programming. Indeed, both client and server-side applications use concurrent programming to provide a fast and glitch-free user experience to their users. This wide adoption of concurrent programming makes it an important programming model for application programmers.

1.1 THE PROBLEM

Concurrent programming enables simultaneous execution of tasks and I/O operations to reduce the time it takes for an application to process the user request. These simultaneous executions introduce non-determinism in concurrent and parallel programs. Due to non-determinism, a program may exhibit different behaviors when

executed multiple times. Non-determinism arising in program execution may give rise to concurrency bugs. Concurrency bugs in programs typically manifest only when the program is executed under a particular buggy input, and event or thread schedule. Thus, non-determinism makes it hard to write bug-free concurrent and parallel programs.

Due to the ubiquitous use of concurrent programming in mobile, web, and desktop computing, it becomes imperative to test programs for concurrency bugs. Existing testing methodologies used during application development like unit testing often fails to find concurrency bugs, as it does not alter the schedule of the program. On the other hand, concurrency testing techniques like stress and random testing are not effective in finding concurrency bugs because of the large schedule and input space in concurrent program. Techniques under the umbrella of model checking are time-consuming and not practical enough to be used during application development. This leaves a void for an efficient and practical concurrency testing technique for programmers wanting to test their applications for concurrency bugs in a reasonable time budget.

1.2 MY THESIS

In this thesis, we argue that coverage-guided fuzzing can be used effectively for testing event-driven concurrent and multicore parallel programs. The key insight is to state high-level program properties as *assertions* in the program and then use fuzzing to generate schedules of concurrent and parallel programs that can falsify the assertion, eventually finding the concurrency bug.

The thesis presents three major contributions each focused on concurrency testing:(1)

proposing a novel concurrency testing technique called coverage-guided property-based concurrency fuzzing (PBCF) that combines property-based testing with mutation-based, grey box fuzzing (2) instantiating the technique in ConFuzz to test event-driven concurrent OCaml programs (3) instantiating the technique in ParaFuzz to test parallel multicore OCaml programs.

1.3 CONTRIBUTIONS

In this section, we provide an overview of the contributions made by the thesis.

1.3.1 PBCF: Coverage-guided Property-based Concurrency Fuzzing

The first contribution of the thesis is a novel concurrency testing technique, PBCF that combines property-based testing [Claessen and Hughes (2000)] with mutation-based, grey-box fuzzing [Zalewski (2021)]. We demonstrate the effectiveness and the practicality of PBCF in finding concurrency bugs in real-world programs by implementing PBCF in two *directed* concurrency testing tools, ConFuzz and ParaFuzz to test event-driven OCaml and parallel Multicore OCaml programs respectively.

Bug-free concurrent programs are hard to write due to non-determinism arising out of concurrency and program inputs. Concurrent programs suffer from concurrency bugs such as data race, deadlocks, and race conditions. Moreover, the erroneous condition in a concurrent program may not be the mere presence of a race, but a complex assertion expressed over the current program state. It is unclear how to express this type of

concurrency bugs. Some concurrency bugs typically manifest under specific inputs and event or thread schedules. Conventional testing methodologies for concurrent programs like stress testing and random testing, which explore random schedules, have a strong chance of missing these bugs.

To expose such complex concurrency bugs that may arise in concurrent programs, we present a novel technique that combines property-based testing on the lines of QuickCheck [Claessen and Hughes (2000)] with AFL fuzzer [Zalewski (2021)], the state-of-the-art mutation-based, grey box fuzzer, and apply it to generate not only inputs that may cause the property to fail, but also to drive the various scheduling decisions in the concurrent program. Our key observation is that we can use AFL's grey box fuzzing capability to direct the search towards new schedules, and thus lead to property failure.

1.3.2 ConFuzz: Coverage-guided Property Fuzzing for Event-driven Programs

The second contribution of the thesis is ConFuzz, a concurrency property fuzz testing tool for event-driven concurrent OCaml programs written using the popular Lwt [Lwt (2021)] concurrency library. ConFuzz instantiates PBCF for event-driven concurrent OCaml programs. Using ConFuzz, programmers specify high-level program properties as *assertions* in the concurrent program. ConFuzz uses the popular grey box fuzzer AFL to generate inputs as well as concurrent schedules to maximize the likely hood of finding new schedules and paths in the program to make the assertion fail.

The key contribution in ConFuzz's implementation is that ConFuzz is developed as a

drop-in replacement for the Lwt library and does not require any modification to the test program. ConFuzz controls the non-determinism present in Lwt programs by capturing the various sources of non-determinism. ConFuzz scheduler then generates and enforces event schedule with the help of AFL. The capability of ConFuzz to capture non-determinism enables it to precisely enforce the schedule generated by AFL and *deterministically* reproduce the concurrency bug. We evaluate ConFuzz against Node.Fz [Davis *et al.* (2017)] - a random fuzzer for event-driven JavaScript programs and stress testing on real-world OCaml applications and benchmark programs. Our experimental results show that ConFuzz is easy-to-use, effective, detects concurrency bugs faster than Node.Fz and can reproduce known concurrency bugs in widely used OCaml libraries built using Lwt. These results were published in PADL 2021 [Padhiyar and Sivaramakrishnan (2021)].

1.3.3 ParaFuzz: Coverage-guided Property Fuzzing for Parallel Programs

The final contribution of the thesis is proposed by ParaFuzz, a PBCF instantiation in the form of a concurrency testing tool for multi-threaded Multicore OCaml programs. Similar to ConFuzz, in ParaFuzz programmers specify high-level program properties as assertions in the source code. ParaFuzz aims to identify the input and the schedule that will cause the assertion to fail. ParaFuzz too readily supports record and replay to reproduce the failure.

ParaFuzz make a single assumption to make it an efficient and practical testing tool. ParaFuzz only requires the test program to be free from data races to enable certain optimizations. We argue that the assumption is reasonable enough to make

ParaFuzz efficient. ParaFuzz captures the various source of non-determinism present in Multicore OCaml programs by controlling thread executions and generate thread schedules with the help of AFL. The main challenges faced by ParaFuzz is AFL integration and testing parallel programs without requiring code to change. AFL is known not to work with multi-threaded programs. Also, to make ParaFuzz a practical tool for concurrency testing, it is imperative to handle unmodified test programs. We solve both the challenges by employing a novel approach to control threads using effect handlers [Plotkin and Pretnar (2009)] in ParaFuzz. Effect-handlers enable ParaFuzz to control threads without changing Multicore OCaml thread APIs. This makes ParaFuzz easier to adopt and test a wide variety of parallel programs. ParaFuzz simulates the parallel execution of threads using effect-handlers thereby allowing AFL to retain tight control over the thread scheduling decisions. Thus enabling ParaFuzz to use AFL to test Multicore OCaml parallel programs. We evaluate ParaFuzz against random and stress testing on parallel benchmark programs commonly used in concurrency testing research. We show that ParaFuzz is efficient and effective in finding concurrency bugs that depend on some combination of input and thread schedule compared to random and stress testing. ParaFuzz was able to find one *previously unknown* concurrency bug [1] in parallel programming library: domainslib [domainslib (2021)].

[1] https://github.com/ocaml-multicore/domainslib/issues/25

1.4 THESIS OVERVIEW

The rest of the thesis is organized as follows. Chapter 2 presents an overview of property-based testing and fuzzing which forms the basis of ConFuzz and ParaFuzz. Chapter 3 proposes combining concurrency testing with property-based testing and fuzzing. Chapter 4 presents ConFuzz, a concurrency testing tool for event-driven concurrent OCaml programs. ParaFuzz, a concurrency testing tool for parallel multicore OCaml programs is introduced in Chapter 5. Related work is presented at the end of each chapter. We conclude the thesis with the future direction of this work in Chapter 6.

CHAPTER 2

Preliminaries

To put this work in context, we begin with a discussion on existing concurrency testing techniques. Next we talk about property-based testing and its applicability in testing programs. We then describe Fuzzing and introduce AFL, the most widely used fuzzer. Then we move on to how Crowbar [Dolan and Preston (2017)], a property-based fuzz testing tool for OCaml, which brings the best of both property-based testing and fuzzing. Finally, we discuss concurrent programming in OCaml which forms the basis of the rest of this thesis.

2.1 EXISTING CONCURRENCY TESTING TECHNIQUES

Multi-threaded parallel programs are difficult to get right as it suffers from various concurrency bugs such as data races, race conditions and deadlocks. Such concurrency bugs are often difficult to find because the bugs are exposed only on specific interleavings. More moving parts in the application in the form of thread non-determinism warrants sufficient testing of multi-threaded programs. Testing a shared-memory multi-threaded program is a hard problem. The specific interleavings that cause the bugs to manifest are hard to find as thread interleaving is non-deterministic and there can be large number of thread interleavings to test. Even for a small program, the number of thread interleavings can be huge. We now discuss the existing concurrency testing techniques.

2.1.1 Stress and Random Testing

Most common approach to test parallel program is *stress testing*, where the program is made to execute under heavy system load for hours or even days in the hope to find any notorious concurrency bug. Stress testing suffers from a few problems. Stress testing depends heavily on the test environment as some of the interleavings may be exposed only on heavily-loaded systems. The thread interleavings are not explicitly controlled by stress testing. The interleavings are subject to OS scheduling which can cause stress testing to execute same interleaving again and again which makes it less likely to find that rare buggy interleaving.

Random testing techniques [Edelstein *et al.* (2003)] improves stress testing by randomizing the thread interleavings in order to exercise different interleavings in different test runs. These random testing techniques differ in the way they randomize thread interleavings. Both random and stress testing does not recognize the tested interleavings and naively executes the same interleavings multiple times. Also, the probability of finding the buggy interleaving out of huge interleaving state space is very low.

2.1.2 Model checking

Another approach is model checking or systematic testing [Godefroid (1997); Musuvathi *et al.* (2019)] which controls thread scheduler and systematically generate all legal thread schedules. But it's hard to model check programs due to well known problem of state space explosion. State-space explosion problem refers to combinatorial explosion with increase in number of threads and instructions in each thread. There have

been prior works to deal with state- space explosion problem like partial-order reduction [Flanagan and Godefroid (2005); Godefroid (1995)], context bounding [Musuvathi and Qadeer (2007*b*)] to reduce state space, but cannot substantially reduce it to test long-running program. Even if the state space of a program is not huge, it will take a long time to test multi-threaded programs, which makes it unlikely to use as a practical technique for programmers wanting to test their code in a reasonable amount of time.

Most of the existing testing technique focuses on finding only buggy interleaving, which is inadequate for bugs that depend on some combination of program input and thread schedule. Techniques focusing on testing programs under different schedule requires test input to be fixed. Even after finding no buggy schedule during testing for a given test input, programmers cannot determine whether to continue testing with different input or stop.

2.2 PROPERTY-BASED TESTING

Property-based testing is a testing technique introduced by QuickCheck (Claessen and Hughes (2000)). In property-based testing, the properties under test are invariants that must hold on all possible inputs. The properties are specified as test methods with formal parameters. The program is then tested with randomly generated inputs in an attempt to violate the property. Inputs that violate the invariant are reported by the property-based testing tool.

Property-based testing allows a wide range of input to the property ranging from simple primitive types (Integers, floating-point numbers, etc) to Algebraic data type by the use

of generators. Generators can be composed with simple combinators provided by the QuickCheck tool to write generators for user-defined types, which enables programmers to test arbitrary interfaces. The values in QuickCheck as defined by generators are generated automatically by random testing. The values that do not conform to the generator's specification are simply discarded. Thus, property-based testing enables programmers to validate the program properties on a wide variety of inputs.

2.3 FUZZING

Fuzzing (Fuzz testing) is an effective technique for testing software by feeding random input to induce the program to crash. Fuzz testing was introduced first by [Miller *et al.* (1990)] to test the reliability of Unix utilities twenty-five years before and has been used widely since then to detect bugs and vulnerabilities in software. Fuzz testing has gained popularity as a testing technique to test programs due to its effectiveness, efficiency, and ease of use in finding software bugs and vulnerabilities. Fuzz testing is used especially in the security domain to detect software security bugs. Fuzz testing can be divided into three types: black-box fuzzing, white-box fuzzing, and grey-box fuzzing.

Black-box fuzzing does not require knowing the internal structure of the program under test nor its source code. The program is treated as a black box for testing purposes. As a result, many generated test cases are not interesting or cannot sufficiently test the program. There have been works [Sparks *et al.* (2007); Wang *et al.* (2010)] in black-box fuzzing to generate effective test inputs and explore deeper paths in the program with the help of domain knowledge of the program.

As opposed to black-box fuzzing, white-box fuzzing requires source code as well knowledge of the internal logic of the program to be tested. White-box fuzzing uses program analysis techniques such as dynamic symbolic execution, model-based testing to generate test inputs to systematically increase code coverage or to reach certain critical program locations. White-box fuzzers are very effective in finding bugs deep in the program. But white-box fuzzer may not work well with large programs as program analysis is often quite time-consuming. To overcome this, there have been attempts [Stephens *et al.* (2016)] to combine the efficiency of black-box fuzzers and the effectiveness of white-box fuzzers.

Finally grey box, fuzzing leverages program instrumentation rather than program analysis to learn more about the program. It uses the collected information to generate test cases that can reach new paths in the program. Prominent grey-box fuzzers like AFL [Zalewski (2021)], libFuzzer [libFuzzer (2021)], honggfuzz [honggfuzz (2021)] have been able to detect many bugs and vulnerabilities in open source projects. Compared to black-box fuzzing, grey-box fuzzing is aware of the internal program structure via automatic instrumentation introduced during compilation, makes it effective in guiding the fuzzing procedure to more interesting paths. On the other hand, grey-box fuzzing is extremely lightweight and scalable to large real-world programs than white-box fuzzing. We now discuss American Fuzzy Lop (AFL), one of the popular grey-box fuzzer in detail.

2.3.1 American Fuzzy Lop (AFL)

American Fuzzy Lop [Zalewski (2021)] is a coverage-guided fuzzer [Padhye *et al.* (2019)], which inserts lightweight instrumentation in the program under test to collect code coverage information such as program execution paths. AFL instruments a targeted program at every conditional jump instruction at compile time. Being a coverage-guided fuzzer, AFL tries to explore new branches in the program in the hope to find more bugs. AFL starts with a seed test case, which it mutates with a combination of random and deterministic techniques that aims to find new execution paths in the program. On detecting a new execution path increasing code coverage, the corresponding input test case is saved for further mutation. During fuzzing, the input test cases that result in a crash are saved, thus finding the exact test case that results in a crash. AFL has successfully discovered thousands of bugs and security vulnerabilities[1] in various applications such as image libraries, web browsers, and networking tools [Zalewski (2021)].

2.4 CROWBAR

Fuzzing uses sophisticated techniques to generate input data, but the condition being checked is generally simple: *"does the program crash"*? This is in contrast to property-based testing where input data is generated in a simple way (randomly), but the testing conditions (properties) are complex. In addition, while property-based testing works well in many cases, random generation of inputs may not cover large parts of the programs where the bugs may lie. Crowbar [Dolan and Preston (2017)] is a testing tool

[1] https://lcamtuf.coredump.cx/afl/

for OCaml combining property-based testing with coverage-guided fuzzing by using AFL to generate test data for QuickCheck-style tests. Rather than generating input data randomly, the test inputs are generated by AFL to maximise the discovery of execution paths in the function under test. Crowbar is similar to QuickCheck, with the user specifying generators and properties to be tested. Crowbar does not offer the options for controlling the frequency and distribution of test cases as done by QuickCheck, but instead the distribution of test data is controlled automatically by AFL, to maximize the number of code paths tested.

2.5 CONCURRENT PROGRAMMING IN OCAML

Concurrent programming is used to develop applications for both single and multicore systems. We next describe precisely the notion of concurrent and parallel programs in OCaml language. OCaml is a general-purpose multi-paradigm programming language. OCaml offers a powerful static type system and type inference capability to help programmers get rid of type-related runtime bugs typically associated with dynamically typed languages.

2.5.1 Lwt: Event-driven programming

Event-driven programming is a programming paradigm in which programs react to events such as mouse clicks, keypresses, or any user interactions with the application. Event-driven programs are typically single-threaded with the main idea to process I/O in a non-blocking way. An event loop sits at the heart of the programming model that concurrently performs the I/O operations, and schedules the callback functions

to be resumed when the corresponding I/O is completed. Event-driven concurrent programming is used in I/O-heavy applications such as web browsers, network servers, web applications. For the rest of the thesis, by concurrent programs, we mean both single-threaded event-driven and multi-threaded parallel programs. While referring to individual concurrent programs (event-driven/parallel) we provide sufficient information to distinguish between two types of concurrency.

OCaml language lacks native support for single-thread and multi-thread concurrency. So threading libraries like Lwt [Lwt (2021)] and Aysnc [Async (2021)] are used to write single-threaded event-driven concurrent programs in OCaml. In this thesis, we focus on Lwt based event-driven programs. Lwt (Lightweight threads) [Lwt (2021)] is a cooperative threading library for writing concurrent programs in OCaml. Under cooperative threading, each task voluntarily yields control to other tasks when it is no longer able to make progress. Lwt is the most widely used asynchronous I/O library in the OCaml ecosystem. Lwt's event handling model is similar to that of Node.js [Node.jsEventLoop (2021)]. Internal working of Lwt is discussed in detail in Chapter 4.

2.5.2 Multicore OCaml: Shared-memory Parallel programming

Shared-memory parallel programming is a type of parallel programming used to develop applications for multicore systems. The aim of parallel programming is to harness the processing capability of multiple processor cores to increase application performance. Parallel programming creates multiple threads that are executed in parallel (simultaneously) on different processor cores. Threads in multicore programs

share data by sharing common memory (shared memory).

OCaml is one of the few modern managed system programming languages to lack support for shared memory parallel programming. Although OCaml offers threading capabilities, threads are executed concurrently on a single-core instead of parallel execution. Multicore OCaml is an extension of the OCaml language to take advantage of multicore processors. Multicore OCaml adds native support for Concurrent and shared-memory Parallel programming to OCaml. Relevant details of Multicore OCaml are discussed in Chapter 5.

CHAPTER 3

COMBINING CONCURRENCY WITH COVERAGE-GUIDED PROPERTY FUZZING

In this chapter, we start with how existing property-based fuzzing techniques fail to test concurrent programs. Then we introduce the novel technique that combines concurrency testing and coverage-guided property fuzzing to test concurrent and parallel OCaml programs. Finally, we introduce two practical concurrency testing tools: ConFuzz and ParaFuzz based on the novel technique to test concurrent and parallel OCaml programs respectively.

3.1 PROBLEM WITH PROPERTY-BASED FUZZING + CONCURRENCY

Property-based fuzzing tools like Crowbar [Dolan and Preston (2017)] work quite well in finding bugs in programs without concurrency. Crowbar has found bugs in many widely used OCaml libraries. But property-based fuzzing is ineffective in finding concurrency bugs. Due to concurrency-induced non-determinism, concurrency bugs typically manifest under a specific schedule of events or threads out of many possible schedules. As property-based fuzzing cannot control the scheduling decisions of a concurrent program, it fails to find concurrency bugs. Scheduling decisions of a concurrent program are left to the OS scheduler, which makes property-based fuzzing on par with stress testing in terms of concurrency bug finding capability, which runs the

program normally without controlling the scheduling decisions. Property-based fuzzing and stress testing explore random schedules of a concurrent program which makes it least likely to find a buggy schedule. Despite being an effective testing technique to find bugs, property-based fuzzing fails to test programs with concurrency.

3.2 COVERAGE-GUIDED PROPERTY-BASED CONCURRENCY FUZZING

As discussed in the previous section property-based testing combined with coverage-guided fuzzing is an effective testing techniques, but cannot be used to test concurrent programs. Based on this observation, we introduce a novel concurrency testing technique called property-based concurrency fuzzing (PBCF) that combines property-based testing with coverage-guided fuzzing applied to concurrent programs. We use AFL fuzzer [Zalewski (2021)], the state-of-the-art mutation-based, coverage-guided grey box fuzzer to provide fuzzing capability in PBCF. We apply PBCF to generate not only inputs that may cause the property to fail, but also to drive various scheduling decisions in the concurrent program. Our key observation is that we can use AFL's grey box fuzzing capability to direct the search towards new schedules, and thus lead to property failure and detect concurrency bugs. PBCF not only finds bugs that manifest under a particular schedule but also bugs that depend upon a particular combination of the input to the program and the schedule.

PBCF uses AFL to generate inputs as well as concurrent schedules to maximize the likelihood of finding new schedules and paths in the program to make the assertion fail. To have absolute control over scheduling decisions, PBCF requires direct control of the concurrent program scheduler. PBCF then uses AFL to randomize the scheduling

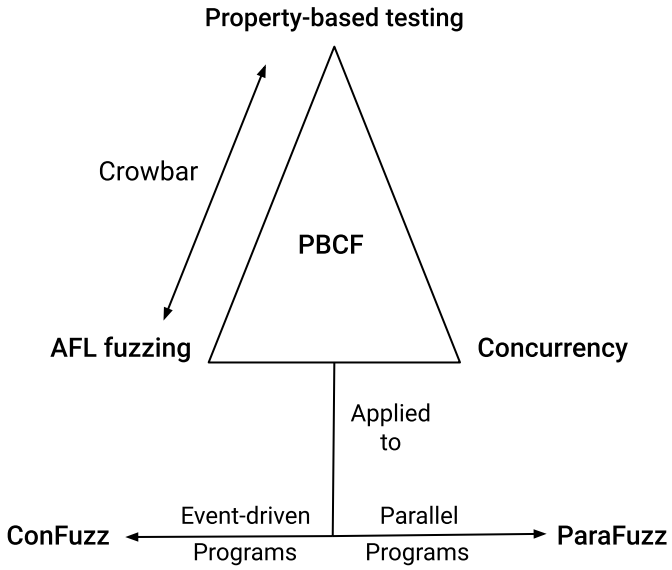


Figure 3.1: PBCF technique and its application in ConFuzz and ParaFuzz

order and enforces the scheduling order through the concurrent program scheduler. This results in the concurrent program being executed under the enforced particular schedule. AFL works by instrumenting the program under test to observe the control-flow edges, mutates the input such that new paths are uncovered. The scheduling order affects the program's execution path. Due to the program instrumentation, AFL recognizes the program execution path in every program run. AFL being a coverage-guided fuzzer, tries to increase coverage in terms of execution paths, thus generating schedule order that increases schedule space coverage.

PBCF completes the trinity of property-based testing, AFL fuzzing, and concurrency. Property-based fuzzing combines property-based testing and AFL fuzzing as implemented by Crowbar. PBCF extends property-based fuzzing with concurrency.

We apply PBCF to event-driven and parallel OCaml programs. We implement PBCF in *directed* concurrency bug-finding tool ConFuzz and ParaFuzz for event-driven OCaml and Multicore OCaml programs respectively as shown in Figure 3.1. While we apply PBCF to OCaml programs, PBCF can be replicated for other AFL-compatible languages too.

We instantiate PBCF in two new *practical* concurrency testing tools: ConFuzz and ParaFuzz for testing event-driven concurrent and multi-threaded parallel OCaml programs respectively. We discuss ConFuzz and ParaFuzz in detail in Chapter 4 and Chapter 5 respectively.

CHAPTER 4

CONFUZZ

Event-driven concurrent programming is used in I/O heavy applications such as web browsers, network servers, web applications and file synchronizers. On the client-side, JavaScript natively supports event-driven programming through promises and async/await [AsynchronousJavaScript (2021)] to be able to retrieve multiple resources concurrently from the Web, without blocking the user-interface rendering. On the server-side, several popular and widely used frameworks such as Node.js (JavsScript) [Node.js (2021)], Lwt (OCaml) [Lwt (2021); Vouillon (2008)], Async (OCaml) [Async (2021)], Twisted (Python) [Twisted (2021)], use event-driven concurrent programming model for building scalable network services.

Event-driven programs are typically single-threaded, with the idea that rather than performing I/O actions synchronously, which may block the execution of the program, all the I/O is performed asynchronously, by attaching a *callback function* that gets invoked when the I/O operation is completed. An *event loop* sits at the heart of the programming model that concurrently performs the I/O operations, and schedules the callback functions to be resumed when the corresponding I/O is completed. The concurrent I/O is typically offloaded to a library such as libuv [libuv (2021)] and libev [Libev (2021)], which in turn discharge concurrent I/O through efficient operating system dependent mechanisms such as epoll [epoll (2021)] on Linux, kqueue [kqueue

(2021)] on FreeBSD, OpenBSD and macOS, and IOCP [IOCP (2021)] on Windows.

Single-threaded event-driven programs avoid concurrency bugs arising from multi-threaded execution such as data races and race conditions. Despite this, event-driven programs suffer from concurrency bugs due to the non-deterministic order in which the events may be resolved. For example, callbacks attached to a timer event and DNS resolution request may execute in different orders based on the order in which the events arrive. As event-driven programs are single-threaded, they do not contain data races related bugs which makes it unsuitable to apply data race detectors developed for detecting multi-threading bugs [Flanagan and Freund (2009*a*); Yu *et al.* (2005)].

Moreover, the erroneous condition in a concurrent program may not be the mere presence of a race, but a complex assertion expressed over the current program state. For example, in the case of a timer event and DNS resolution request, the timer may be intended for timing out the DNS resolution request. On successful resolution, the timer event is canceled. Then, the safety property is that if the timer callback is running, then the DNS resolution request is still pending. It is unclear how to express this complex property as races.

To help uncover such complex concurrency bugs that may arise in event-driven concurrent programs, we present a novel technique that combines property-based testing on the lines of QuickCheck [Claessen and Hughes (2000)] with AFL fuzzer [Zalewski (2021)], the state-of-the-art mutation-based, grey box fuzzer, and apply it to generate not only inputs that may cause the property to fail, but also to drive the various scheduling decisions in the event-driven program. AFL works by instrumenting the

program under test to observe the control-flow edges, mutates the input such that new paths are uncovered. In addition to different paths, a concurrent program also has to contend with the exponential number of schedules available, many of which may lead to the same behavior. Our key observation is that we can use AFL's grey box fuzzing capability to direct the search towards new schedules, and thus lead to property failure.

We have implemented this technique in ConFuzz, a concurrent property fuzz testing tool for concurrent OCaml programs using the popular Lwt [Lwt (2021); Vouillon (2008)] library (asynchronous I/O library). Properties are expressed as assertions in the source code, and ConFuzz aims to identify the input and the schedule that will cause the assertion to fail. ConFuzz supports record and replay to reproduce the failure. Once a bug is identified, ConFuzz can *deterministically reproduce* the concurrency bug. ConFuzz is developed as a drop-in replacement for the Lwt library and does not require any change to the code other than writing the assertion and the wrapper code to drive the tool.

The main contributions of ConFuzz work are as follows:

- We present a novel technique that combines property-based testing with mutation-based, grey box fuzzer applied to test the schedules of event-driven OCaml programs.
- We implement the technique in ConFuzz, a drop-in replacement for testing event-driven OCaml programs written using the Lwt library.
- We show by experimental evaluation that ConFuzz is more effective and efficient than the state-of-the-art random fuzzing tool Node.Fz and stress testing in finding concurrency bugs. We reproduce known concurrency bugs by testing ConFuzz on 8 real-world concurrent OCaml programs and 3 benchmark programs.

The remainder of this chapter is organized as follows. We present a motivating example

```
let linear_eq i =
  let x = ref i in
  let p1 = pause () >>= fun () ->
    x := !x - 2; return_unit in
  let p2 = pause () >>= fun () ->
    x := !x * 4; return_unit in
  let p3 = pause () >>= fun () ->
    x := !x + 70; return_unit in
  Lwt_main.run (join[p1;p2;p3]);
  assert (!x <> 0)
```

Figure 4.1: A program with a concurrency bug

in Section 4.1 that illustrates the effectiveness of ConFuzz on an adversarial example. Section 4.2.1 provides an overview of the event-driven model in Lwt. ConFuzz is introduced in Section 4.3 along with a discussion of non-determinism arising in event-driven programs. Section 4.4 discusses the implementation details of ConFuzz. Experimental evaluation is described in Section 4.5. The limitations of our approach are discussed in Section 4.6. Related work is discussed in Section 4.7, and we conclude the chapter with Section 4.8.

4.1 MOTIVATING EXAMPLE

We describe a simple, adversarial example to illustrate the effectiveness of ConFuzz over Node.Fz and stress testing. Figure 4.1 shows an OCaml concurrent program written using the Lwt library [Vouillon (2008)]. The program contains a single function `linear_eq` that takes an integer argument `i`. `linear_eq` creates three concurrent tasks `p1`, `p2`, and `p3`, each modifying the shared mutable reference `x`. The `pause` operation pauses the concurrent task, registering the function `fun ()-> ...`

Table 4.1: Comparing different testing techniques

Testing Technique	Executions (millions)	Time (minutes)	Bug Found
ConFuzz	3.26	18	**Yes**
Node.Fz [Davis *et al.* (2017)]	110	60	**No**
Stress	131	60	**No**

following the >>= operator as a callback to be executed in the future. Importantly, the tasks `p1`, `p2`, and `p3` may be executed in any order.

This program has a concurrency bug; there exists a particular combination of input value `i` and interleaving between the tasks that will cause the value of `x` to become 0, causing the assertion to fail. There are 2^{63} [1] possibilities for the value of `i` and 6 (3!) possible schedules for the 3 tasks. Out of these, there are only 3 possible combinations of input and schedule for which the assertion fails.

- `i` = -17 and schedule = `[p2; p1; p3]` : $((-17 * 4) - 2) + 70 = 0$.
- `i` = -68 and schedule = `[p1; p3; p2]` : $((-68 - 2) + 70) * 4 = 0$.
- `i` = -68 and schedule = `[p3; p1; p2]` : $((-68 + 70) - 2) * 4 = 0$.

As the bug in the example program depends on input and interleaving, concurrency testing techniques focusing only on generating different interleavings will fail to find this bug. This is evident when the program is executed under different testing techniques. Table 4.1 shows a comparison of ConFuzz with the random concurrency fuzzing tool Node.Fz [Davis *et al.* (2017)] and stress testing for the example program.

[1] OCaml uses tagged integer representation [Leroy (1990)] where 1 bit is used to distinguish immediate values from pointers.

```
let pipe_chars a b c =
  let res = ref [] in
  let ic, oc = pipe () in
  let sender =
    write_char oc a >>= fun () ->
    write_char oc b >>= fun () ->
    write_char oc c >>= fun () ->
    return_unit
  in
  let recvr () =
    read_char ic >>= fun c ->
    res := Char.uppercase_ascii c::!res;
    return_unit
  in
  Lwt_main.run (join [recvr(); recvr();
                      recvr(); sender]);
  assert (!res <> ['B';'U';'G'])
```

Figure 4.2: A program with a concurrency bug

Node.Fz is a concurrency fuzzing tool similar to ConFuzz, which generates random interleavings rather than being guided by AFL. Node.Fz focuses only on finding buggy interleavings. As Node.Fz is implemented in JavaScript, we port the underlying technique in OCaml. We refer to the OCaml port of Node.Fz technique when referring to Node.Fz. Stress testing runs a program repeatedly with random input values. We test the example program with each technique until a bug is found or a timeout of 1 hour is reached. We report the number of executions and time taken if the bug was found. Only ConFuzz was able to find the bug. Although this example is synthetic, we observe similar patterns in real-world programs where the bug depends on the combination of the input value and the schedule, and cannot be discovered with a tool that only focuses on one of the sources of non-determinism.

Real-world event-driven programs also involve file and network I/O, timer completions, etc. ConFuzz can test *unmodified* programs that involve complex I/O behavior. Figure 4.2 shows a function `pipe_chars` that takes three character arguments. The function creates a shared `pipe` as a pair of input (`ic`) and output (`oc`) file descriptors. The `sender` task sends the characters over `oc`. The three `recvr` tasks each receive a single character, convert that to the corresponding upper case character, and append it to a global list reference `res`. The assertion checks that the final result in `res` is not `['B';'U';'G']`. Due to input and scheduling non-determinism, there are plenty of schedules. However, the assertion failure is triggered with only 6 distinct inputs, each of which is a permutation of `'b'`, `'u'`, `'g'` for the input arguments to the function, and a corresponding permutation of the `recvr` tasks. ConFuzz was able to find a buggy input and schedule in under a minute. This illustrates that ConFuzz is applicable to real-world event-driven concurrent programs.

4.2 CONCURRENCY BUGS IN LWT

In this section, we discuss the event-driven model in Lwt and then give an overview of concurrency bug patterns in Lwt programs. Lwt is the most widely used asynchronous I/O library in the OCaml ecosystem. Lwt lies at the heart of the stack in the MirageOS [Madhavapeddy *et al.* (2013)], a library operating system for constructing Unikernels. MirageOS is embedded in Docker for Mac and Windows apps [Docker (2021)] and hence, runs on millions of developer machines across the world. Hence, our choice of Lwt is timely and practical. That said, the ideas presented in this chapter can be applied to other event-driven libraries such as Node.js [Node.js (2021)].

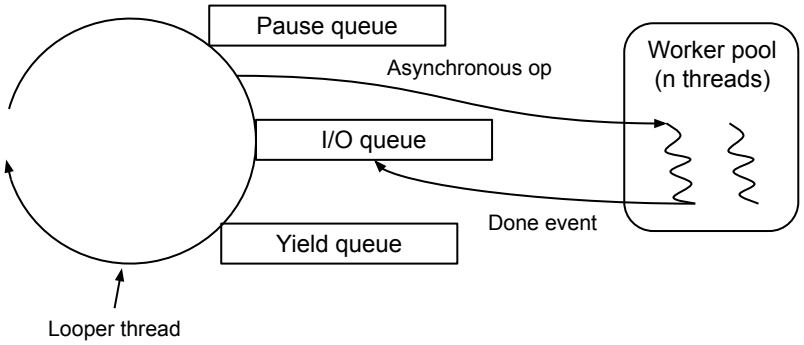


Figure 4.3: Lwt event model

4.2.1 Lwt: Event-driven model

Lwt (Lightweight threads) [Vouillon (2008)] is a cooperative threading library for writing concurrent programs in OCaml. Under cooperative threading, each task voluntarily yields control to other tasks when it is no longer able to make progress. Lwt event model is shown in Figure 4.3. The event handling model is similar to that of Node.js [Node.jsEventLoop (2021)]. Lwt event model consists of an event loop engine and a worker pool. The event loop engine manages timers, read and write I/O events on registered file descriptors and executes the callbacks registered with the events. Lwt event loop engine can be configured to use various engines such as `libev` [Libev (2021)], Unix's select [Select (2021)] and poll [Poll (2021)]. As `libev` is the default event loop engine, in the sequel, we refer to `libev` when referring to Lwt's event loop engine.

The *looper thread* executing the event loop engine waits for the events to occur and then executes user callbacks registered with respective events. The looper thread executes every callback atomically, without any interruption, until the callback itself gives up

control. The computationally intensive tasks and blocking system calls are offloaded to the *worker pool* of threads that execute the tasks so that they do not block the event loop.

Lwt event loop consists of three event queues, each holding a different class of events with their attached callbacks. The three queues are `yield`, `pause` and `I/O` queue. All yielded and paused callbacks are inserted in the yield and pause queue respectively. The primitives `yield` and `pause` both yield control to the looper thread allowing it to execute other event callbacks in the queue. The difference is that yield allows I/O to be handled in between, but pause does not. The `I/O` queue comprises the timer and I/O events and is handled by `libev` engine.

The event loop examines each of the queues for pending callbacks. The looper thread executes all the callbacks in an event queue before moving on to the next queue. Lwt does not guarantee the relative execution order between two events in the same or different queues. For example, two yield events, as well as yield and an I/O event, can be processed in any order. The non-determinism in the execution order of the events gives rise to concurrency bugs. The asynchronous I/O operations like network I/O and file read-write invoked in the callbacks are offloaded to the worker pool. The worker thread on completion of the offloaded task signals `libev` to indicate task completion (*Done event*). The callbacks associated with the asynchronous tasks are then executed by the `libev` engine.

```
let connect host port =
 let open Lwt_unix in
 let conn =
  socket (Unix.PF_INET) Unix.SOCK_STREAM 0
 in
 let port = string_of_int port in
 getaddrinfo host port [] >>= function
 | [] -> failwith "Could not resolve redis host!"
 | addrinfo::_ -> return addrinfo.Lwt_unix.ai_addr
 >>= fun sock_addr ->
 - ignore (Lwt_unix.connect conn sock_addr);
 + Lwt_unix.connect conn sock_addr >>= fun () ->
 return conn

let ping connection =
  let command = [ "PING" ] in
  IO.try_bind
  (fun () -> send_request connection command)
  (function
  |`Status "PONG" -> IO.return true
  | _ -> IO.return false)
  (fun e -> IO.fail e)
```

Figure 4.4: ocaml-redis#25: Order violation in ocaml-redis. Both Lwt_unix.connect and send_request are asynchronous, and can execute in any order.

4.2.2 Bug patterns

Bugs in Lwt programs are classified into two types: *order violations* and *atomicity violations*.

Order violation

An order violation occurs when the intended order between events or asynchronous operations is not enforced by the execution. A developer may assume that some sequence of operations will execute in a certain order. It may be the case that the operations are actually executed in a different order due to non-determinism.

Figure 4.4 shows an order violation in the `ocaml-redis` library [ocaml redis (2021)] issue 25 [2]. The `connect` method creates a connection to a Redis server asynchronously. `ping` sends a *ping* message using `send_request` asynchronous operation. The developer's intention to execute the `connect` and the `ping` operations sequentially by issuing them one after the other is violated due to asynchronous operations `Lwt_unix .connect` and `send_request`. As the operations are asynchronous, they can be executed in any order resulting in a buggy execution where *ping* message is sent before the connection is established. To fix this order violation, `connect` should wait for `Lwt_unix.connect` to establish a connection before returning the connection.

Atomicity violation

An atomicity violation occurs when two or more operations are assumed to be executed atomically, but another operation happens to be interleaved between them, such that the effect is no longer atomic. In Lwt programs, atomicity violation is the violation of the intended atomic execution of a sequence of callbacks or asynchronous operations. Several concurrency studies [Wang *et al.* (2017); Davis *et al.* (2017)] have shown that a vast majority of concurrency bugs in the wild are caused by atomicity violations.

One such example of atomicity violation, ocaml-redis issue #40 [3], is illustrated in Figure 4.5. The function `read_reply` reads the reply from the input channel `ch` by invoking one of the asynchronous operations, `read_line` and `read_integer`, based

[2] https://github.com/0xffea/ocaml-redis/issues/25
[3] https://github.com/0xffea/ocaml-redis/issues/40

```
let read_reply ch =
+ IO.atomic (fun ch ->
  IO.input_char ch >>= fun c ->
  match c with
  | '+' ->
    read_line ch
  | ':' ->
    read_integer ch
+ ) ch
```

Figure 4.5: ocaml-redis#40: Atomicity violation in ocaml-redis.

on the reply type `c`. The reply type is read asynchronously by `IO.input_char` before the rest of the reply is read. Thus the order between these two asynchronous operations cannot be violated. But another asynchronous operation can be interleaved between `IO.input_char` and `read_*`, making `read_reply` non-atomic.

The execution of two concurrent `read_reply` operations may conflict with each other and read corrupted data. Before the first `read_reply` completes reading the reply, a second `read_reply` may start reading the reply from the input channel. If they are of different types, then they will read incorrect replies. The fix for this bug is to make `read_reply` atomic by wrapping the body of the `read_reply` function in `IO.atomic`.

4.3 CONFUZZ

In this section, we present the architecture of the ConFuzz tool and show how to test event-driven concurrent programs using ConFuzz.

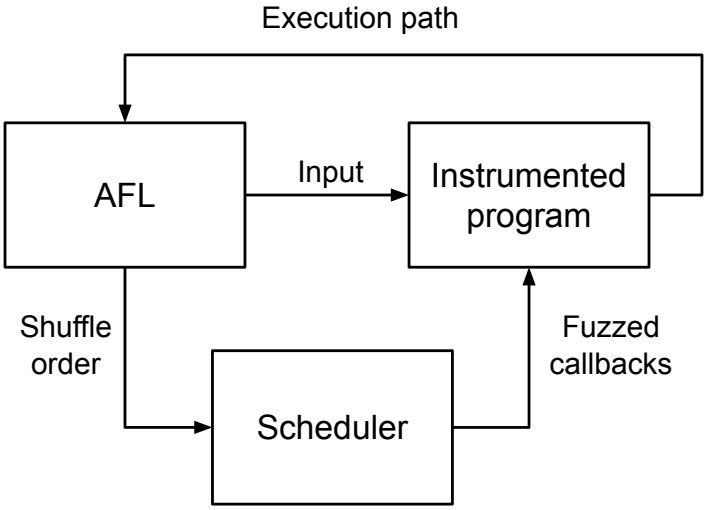


Figure 4.6: ConFuzz architecture

4.3.1 Architecture

We have implemented the novel technique of concurrent property-based testing using the AFL fuzzer in ConFuzz tool. ConFuzz tool can handle OCaml programs written using the Lwt library. Figure 4.6 shows ConFuzz's architecture. ConFuzz combines property-based testing and AFL (through crowbar library) with concurrency testing. ConFuzz uses the crowbar library to enable programmers to write concurrency tests as properties. Concurrency tests are then compiled with AFL compatible instrumentation. The resulting instrumented binary is then executed using AFL for fuzzing.

ConFuzz controls Lwt's scheduler by capturing the non-determinism present in the Lwt programs. To explore properties on a wide range of different schedules, ConFuzz generates various legal event schedules by alternating the order of event callback execution with the help of AFL. AFL generates execution order (*shuffle order*) for the

captured concurrent events, which is then enforced by a controlled scheduler (*fuzzed callbacks*). The properties are tested repeatedly with different test inputs and event schedules. The test input and the event schedules that result in property failures are detected as a crash by AFL, resulting in the detection of a concurrency bug. ConFuzz also offers a random testing mode where *shuffle order* is generated by OCaml pseudo-random number generator (PRNG) instead of AFL. We use this mode to evaluate AFL's performance to generate *shuffle order* effectively in Section 4.5.

To capture the non-determinism in Lwt concurrent programs, ConFuzz captures the concurrent event callbacks in the Lwt event loop in a list. To change the order of callbacks, AFL fuzzes the callback list while recording the fuzzer input. As the test program is instrumented, AFL can classify crashes as duplicates by looking at the execution path taken, which corresponds to the same bug that can be exposed in multiple schedules. AFL, being a coverage-guided fuzzer aims to maximize the code coverage by generating unique callback orders, thus leading to generating newer event schedules. ConFuzz aims to provide a quick testing solution to enable programmers to test their concurrent code on a wide variety of schedules, giving them the confidence to run programs in production.

Many of the concurrency testing tools [Davis *et al.* (2017); Musuvathi and Qadeer (2007*a*)] focus on generating and enforcing different schedules to find concurrency bugs that occur only in some specific schedule. This works well when test input to the program is fixed, leaving the concurrency tool to just find a buggy schedule. But in the case where a concurrency bug occurs on some specific test input and schedule,

these testing tools are unable to catch the bug. ConFuzz is also designed to catch bugs that depend on some combination of inputs and buggy schedules. Test input and buggy schedule to programs are sourced from AFL.

Similar to other concurrency testing tools, ConFuzz also supports the record and replay feature, which records the event schedule that leads to a concurrency bug. The input and the buggy schedule are saved in a separate file, which when executed with test binary, deterministically reproduces the bug. Thus, ConFuzz helps to debug concurrency bugs by reliably reproducing it.

4.3.2 ConFuzz on the motivating example

In this section, we describe how to test the example shown in Figure 4.1 with ConFuzz.

```
let () =
  Confuzz.(add_test ~name:"test_linear_eq"
    [Confuzz.int] (fun i -> linear_eq i ))
```

To test `linear_eq` function, a test should be registered with `Confuzz.add_test`. As `linear_eq` takes an integer argument, the `Confuzz.int` generator is specified, which will generate random numbers. Last argument to `Confuzz.add_test` takes a callback function that takes an integer and calls `linear_eq` with generated integer `i`. ConFuzz tests `linear_eq` repeatedly by calling callback function with different integers and executing `linear_eq` with different event schedules. Observe that integrating and testing the concurrent program with ConFuzz does not require any change to the function under test. Hence, the technique is readily applicable to the production code.

4.4 FUZZING UNDER NON-DETERMINISM

In this section, we discuss the non-determinism present in Lwt concurrent programs. We then show how ConFuzz captures and explores the non-determinism.

4.4.1 Non-determinism in Lwt

I/O and timer non-determinism

The I/O events registered with the event loop such as file and network operations (read and write) are highly non-deterministic. The I/O events can be triggered in any order as there is uncertainty regarding the completion time of the operations. There is also the possibility of races between the callbacks of different I/O events to shared resources such as sockets and file descriptors.

The timer API is used to defer action until a certain time duration has passed. The timers are also often used for ad-hoc synchronization and timeout. Lwt does not guarantee the precise expiration of timers. For example, a timer registered for `t` seconds will expire *at least* `t` seconds in the future. Thus, the assumption regarding the precise expiration of timers in a concurrent program could lead to bugs.

As described in Section 4.2.1, the timer and the I/O events are processed in the I/O event queue. The non-deterministic nature of the execution of I/O and timer events in the I/O queue leads to a large number of callback schedules. For example, a user code expecting a network request to complete within a certain time period can go wrong. This bug can be caught by fuzzing the I/O event queue.

Worker pool non-determinism

Blocking system calls and long-running user tasks handled by the worker pool are other sources of non-determinism in Lwt programs. The worker pool comprises a pre-configured number of kernel threads for task execution. Due to the fixed number and dynamic scheduling of threads, the order of execution of tasks by the worker pool is non-deterministic. As the processing time of each task varies, the completion order of tasks relative to each other also varies. To indicate task completion, the worker thread writes to a common file descriptor, which is processed as an I/O event. This results in the relative order of execution between task callbacks and other callbacks to be non-deterministic.

Callback non-determinism

As callbacks are executed atomically, `yield` and `pause` primitives enable long-running computation to give up execution to another callback voluntarily. The yielded callbacks are guaranteed to be called sometime in the future. A sequence of yield and paused callbacks can be executed in any order. Also, there can be multiple callbacks attached to a single event. In that case, Lwt does not guarantee the order of execution of callbacks. The alternative ordering of yielded and paused callbacks and the event callbacks leads to an increase in the number of schedules. The assumptions regarding the callback execution order between the yield paused and I/O event callbacks can lead to inconsistent program states or bugs. These types of bugs can be caught by fuzzing yield, pause and I/O event callback queues.

4.4.2 Capturing Non-determinism

In this section, we discuss how ConFuzz controls the non-determinism described in Section 4.4.1. The captured non-deterministic events are controlled by ConFuzz scheduler (Section 4.4.3) to generate alternative callback execution schedules.

Event loop queues

Lwt event loop queues (Section 4.2.1) such as yield, pause and I/O are the primary sources of non-determinism in an Lwt program. To capture and control the non-determinism arising out of these queues, we have made architectural changes to the event loop. We have inserted calls to ConFuzz scheduler in the event loop before executing the callbacks of yield and pause queues. ConFuzz scheduler then changes the order of callbacks to generate alternative schedules. To capture I/O queue non-determinism, expired timers and ready file descriptors (I/O events) are inserted in a list by the event loop. The list is then passed to ConFuzz scheduler to change the order of I/O events. By fuzzing the I/O queue, we are able to generate schedules with alternative timer and I/O event order.

Worker pools

Non-determinism in the worker pool is influenced by multiple factors such as the number of threads, the thread scheduling by the operating system and the order in which the tasks are offloaded. We take a systematic approach to capture the worker pool non-determinism. First, for deterministic processing and completion order of tasks, we

reduce the worker pool size to one. This change serializes the tasks handled by the worker pool. The worker pool tasks are executed one after another. By reducing the worker pool to one thread, ConFuzz can deterministically replay the order of worker pool task execution.

To signal task completion, the worker pool thread writes to a single common file descriptor registered with `libev`. The write to a common file descriptor is intercepted by the event loop and processed as an I/O event. The single file descriptor is shared by all the tasks for indicating task completion. Thus, Lwt multiplexes a single file descriptor for many worker pool tasks. Multiplexing prevents changing the order of task completion relative to I/O and timer events. ConFuzz can still change the task completion order relative to other tasks. But this prevents ConFuzz from generating broader schedule space and as a result, misses some of the bugs.

To overcome this, ConFuzz eliminates multiplexing by assigning a file descriptor per task. The assigned file descriptors are registered with `libev`. On task completion, the respective file descriptor is written with a byte which is then processed as an I/O event. During the event loop I/O phase, task completion I/O events are fuzzed along with other I/O events and timers. De-multiplexing enables ConFuzz to shuffle the order of task completion relative to other tasks as well as timer and I/O events.

To change the processing order of worker pool tasks, we delay the execution of tasks. During each iteration of the event loop, the tasks are collected in a list. At the start of the next iteration of the event loop, ConFuzz scheduler shuffles the processing order of the tasks. The tasks are then executed synchronously. By delaying the task execution by

one iteration, ConFuzz collects enough tasks to shuffle. We believe that delaying tasks by one iteration would suffice to generate the task processing orders that would occur in production environments. It is highly unlikely that a task from the second iteration is started and completed before tasks from first the iteration, given that Lwt tasks are started off in a FIFO manner.

Executing tasks synchronously enables greater control over the completion order of tasks. As tasks are already completed, ConFuzz can change the order of task completion to generate alternate completion orders. In addition, the task completion order is shuffled relative to the I/O events by the I/O queue fuzzing. Synchronous task execution also helps in deterministically generating a buggy schedule. As the number of completed tasks remains the same in every schedule, ConFuzz has to just reorder tasks to reproduce a bug. This design choice let ConFuzz generate task processing and completion order independently.

However, delaying and synchronous task execution can prevent ConFuzz from missing schedule containing bugs arising from the worker pool. In ConFuzz, we trade-off schedule space generation to reliably reproduce concurrency bugs by deterministic schedule generation. ConFuzz does not guarantee the absence of bugs but reliably reproduces discovered concurrency bugs.

Promise callbacks

As promise callbacks are executed non-deterministically, promise callback ordering is also fuzzed by ConFuzz. Before execution, the order of callbacks attached to a promise

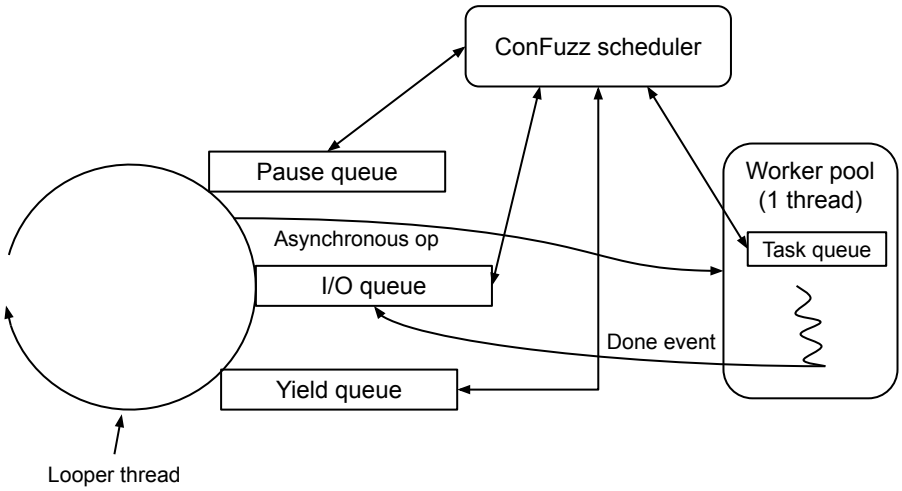


Figure 4.7: ConFuzz scheduler

is changed by ConFuzz scheduler. By fuzzing promise callbacks, ConFuzz generates alternative ordering of callback execution.

4.4.3 ConFuzz scheduler

To generate varied event schedules, ConFuzz scheduler controls the Lwt event loop and the worker pool as shown in Figure 4.7. To change the order of events, ConFuzz scheduler exposes `fuzz_list : 'a list -> 'a list` function, which takes a list and returns a shuffled list. The changes to the Lwt scheduler that require changing the order of events (Section 4.4.2) call this function to shuffle the callback list. On executing the shuffled list, the program is executed under a particular schedule.

To reorder callbacks, ConFuzz scheduler asks AFL to generate random numbers. The random numbers then determine the ordering of the callbacks. On detecting a concurrency bug, the generated random numbers are saved in a separate file as a

schedule trace. With the schedule trace, the scheduler can reproduce a schedule. Using this capability of the scheduler, ConFuzz can replay a schedule to reliably expose the detected concurrency bugs. Deterministic replay helps programmers find the exact cause of concurrency bugs.

The order of callback execution affects the program's execution path. Due to the program instrumentation, AFL recognizes the program execution path in every program run. AFL being a coverage-guided fuzzer, tries to increase coverage (execution paths). AFL thus generates random numbers that produce alternative callback orderings. Alternative callback orderings result in new schedules that exercise new program execution paths. ConFuzz scheduler keeps on generating new schedules until AFL is able to find new execution paths. ConFuzz thus uses AFL fuzzing to execute a program under different execution schedules.

4.5 EVALUATION

In this section, we evaluate the effectiveness of ConFuzz in finding concurrency bugs in real-world OCaml applications and benchmark programs. Additionally, we check the efficiency of ConFuzz in terms of time required to detect concurrency bugs in comparison to Node.Fz and stress testing. Node.Fz [Davis *et al.* (2017)] is a concurrency bug-finding fuzzing tool for event-driven JavaScript programs. As Node.Fz randomly perturbs the execution of a JavaScript program, we use ConFuzz's random testing mode (Section 4.3.1) to simulate Node.Fz technique. Stress testing runs a program repeatedly with random input values. Stress testing does not generate program interleavings as done by ConFuzz and executes programs directly under OS scheduler.

Table 4.2: Experimental subjects

Type	Name (abbreviation)	Description	GitHub Stars	Size (LoC)	Issue #
Real world applications	irmin (IR)	Distributed database	1,284	18.6K	270
	lwt (LWT)	Concurrent programming library	448	12.2k	583
	mirage-tcpip (TCP)	Networking stack for Mirage OS	253	4.9K	86
	ghost (GHO)	Blogging engine	35,000	50K	1834
	porybox (PB)	Pokémon platform	29	7.9K	157
	node-mkdirp (NKD)	Recursive mkdir	2,200	0.5K	2
	node-logger-file (CLF)	Logging module	2	0.9K	1
	fiware-pep-steelskin (FPS)	Policy enforcement point proxy	11	8.2K	269
Benchmark programs	Motivating example (MX)	Linear equation with concurrency	-	-	-
	Benchmark 1 (B1)	Bank transactions	-	-	-
	Benchmark 2 (B2)	Schedule coverage	-	-	-

We design and conduct experiments to answer the following questions:

1. **RQ1: Effectiveness** – How frequently is ConFuzz able to find bugs?
2. **RQ2: Efficiency** – How many executions are required to detect bugs by ConFuzz as compared to Node.Fz and stress testing?
3. **RQ3: Practicality** – Can ConFuzz detect and reproduce known concurrency bugs in real-world OCaml applications?

4.5.1 Experimental subjects and setup

We evaluated ConFuzz on both real-world OCaml applications and benchmark programs. Table 4.2 summarises the applications and benchmark programs used for the evaluation. We have used eight real-world applications and three benchmark programs as experimental subjects for evaluating ConFuzz. All of the programs contain at least one known concurrency bug.

To identify known concurrency bugs, we searched across GitHub bug reports for closed

bugs in Lwt based OCaml projects. We select a bug only if the bug report contains a clear description or has an automated test case to reproduce the bug. We found three Lwt based OCaml bugs - IR, LWT and TCP as shown in Table 4.2. Apart from OCaml bugs, we have build a dataset of 15 known concurrency real-world JavaScript bugs mentioned in the related work [Chang *et al.* (2019); Davis *et al.* (2017); Wang *et al.* (2017)]. We abstracted the buggy concurrent code of JavaScript bugs and ported it to standalone OCaml programs. We excluded those bugs from the JavaScript dataset which could not be ported to OCaml or have an incomplete bug report. We were able to port 5 JavaScript bugs from the dataset. The five JavaScript bugs used in the evaluation are GHO, PB, NKD, CLF and FPS. MX is the motivating example from section 4.1. Benchmark B1 simulates concurrent bank transactions, adapted from the VeriFIT repository of concurrency bugs [Repository (2021)]. Concurrent bank transactions in B1 cause the bank account log to get corrupted. Benchmark B2 simulates a bug depending on a particular concurrent interleaving and gets exposed only when B2 is executed under that buggy interleaving. B2 is explained in detail in section RQ2.

We design our experiments to compare ConFuzz's bug detection capability with Node.Fz and stress testing (hereby referred to as the testing techniques). We perform 30 testing runs for each experimental subject (Table 4.2) and testing technique. A *testing run* is a single invocation of the testing technique. The performance metric we focus on is mean time to failure (MTTF), which measures how quickly a concurrency bug is found in terms of time. A single *test execution* indicates one execution of the respective application's test case. For each subject and testing technique, we execute the respective

Table 4.3: Bug detection capability of the techniques. Each entry is the fraction of the testing runs that manifested the concurrency bug.

	Stress	**Node.Fz**	**ConFuzz**
IR	1.00	0.00	1.00
LWT	0.00	1.00	1.00
TCP	0.00	1.00	1.00
GHO	0.00	0.00	1.00
PB	0.00	0.00	1.00
NKD	0.4	0.53	1.00
CLF	0.43	0.56	1.00
FPS	0.00	0.96	1.00
MX	0.00	0.00	1.00
B1	0.87	0.6	1.00
B2	0.00	0.00	1.00
Avg	0.24	0.42	1.00

subject application until the first concurrency bug is found or a timeout of 1 hour occurs. For each such run, we note the time taken to find the first concurrency bug and whether a bug was found or not. We ran all of our experiments on a machine with a 6-Core Intel i5-8500 processor, 16GB RAM, running Linux 4.15.0-1034.

4.5.2 Experimental results

RQ1: Effectiveness

Table 4.3 shows the bug detection capabilities of the three testing techniques. The first column shows the abbreviation of the experimental subjects. The second to fourth column shows the bug detection results of Stress, Node.Fz and ConFuzz testing, respectively. Each cell in the table shows the fraction of the testing runs that detected a concurrency bug out of the total 30 testing runs per experimental subject and testing

Table 4.4: Mean time to find the concurrency bug (seconds)

	Stress	**Node.Fz**	**ConFuzz**
IR	37.7	-	1.03
LWT	-	295.73	243.3
TCP	-	315.03	94.16
GHO	-	-	0.33
PB	-	-	0.3
NKD	1738.83	1104.62	42.23
CLF	685.1	1086.2	231.96
FPS	-	696.55	103.13
MX	-	-	981.17
B1	918.8	1333.89	384.6
B2	-	-	59.26

technique.

As shown in Table 4.3, ConFuzz detected concurrency bugs in every testing run for all experimental subjects (all cells are 1.00). In the case of GHO, PB, MX and B2, only ConFuzz was able to detect a bug. Despite capturing the non-determinism, Node.Fz could not detect a bug in IR, GHO, PB, MX and B2. This confirms that ConFuzz was able to generate concurrent schedules along with inputs more effectively. Stress testing was more effective in the case of IR and B1 than Node.Fz with a ratio of 1.00 and 0.87 respectively. Both IR and B1 comprise a lot of files I/O. We suspect that due to OS-level non-determinism, stress testing is more effective than Node.Fz, as Node.Fz finds it difficult to generate the exact buggy schedule for file I/O. This provides a helpful insight that ConFuzz is good at generating a prefix or exact schedule that can cause concurrency errors. In addition, ConFuzz does not produce *false positives*, as schedules explored by ConFuzz are all legal schedules in Lwt. Thus, the results confirm that ConFuzz is effective at detecting concurrency bugs.

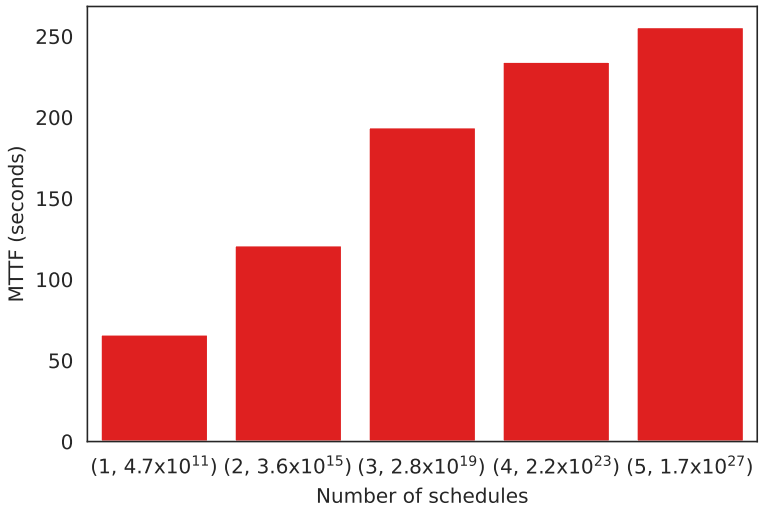


Figure 4.8: Efficiency of ConFuzz as schedule space increases. The total number of schedules is given by $f(n) = (3!)^{(10*n+20)/2)}$. The labels on the x-axis show $(n, f(n))$.

RQ2: Efficiency

Table 4.4 shows the efficiency results of the three testing techniques. The second to fourth column shows the efficiency results of stress, Node.Fz and ConFuzz testing respectively. Each cell represents the average time (in seconds) taken to detect the first concurrency bug per experimental subject and testing technique over 30 testing runs. '-' in the cell indicates that none of the 30 testing runs detected a concurrency bug within the timeout of 1 hour.

As shown in Table 4.4, for every experimental subject, ConFuzz took significantly less time (column 4) to find bugs than other techniques. ConFuzz is 26×, 6× and 4.7× faster than Node.Fz for NKD, FPS and CLF bugs respectively. For NKD and IR bugs, ConFuzz is 41× and 36× faster than stress testing respectively. Except for LWT, ConFuzz is at least 2× faster than second-fastest technique.

Note that for NKD, CLF, FPS and B1 bugs, the average time of Node.Fz and stress testing does not include testing runs which failed to detect concurrency bug. Due to its efficiency, ConFuzz enables a developer to explore a broader schedule space of the concurrent program than Node.Fz and other techniques with the same test time budget. Thereby increasing the chances of finding bugs in the same limited test time budget. Thus, these results illustrate that ConFuzz is efficient in detecting concurrency bugs.

To evaluate the efficiency of ConFuzz on a program containing a large schedule space, we modify the motivating example in Figure 4.1 to have a large number of concurrent schedules. We define a concurrent schedule as the order in which the callbacks attached to the events are executed. The total number of concurrent schedules of the modified program is given by the following formula parameterized over *n*:

$$\textit{Total number of schedules} = (3!)^{(10*n+20)/2} \tag{4.1}$$

where *n* controls the degree of concurrency in the program. Only one concurrent schedule out of the many schedules results in a concurrency bug. Figure 4.8 shows the efficiency of ConFuzz over large schedule spaces. We increase *n* from 1 to 5 to generate a large schedule space. Note that benchmark B2 used as an experimental subject in evaluation is a modified program with *n* equals to 1. Figure 4.8 graph shows mean time to failure (MMTF) as the schedule space is increased. As evident from the graph, even for the program with a large schedule space, ConFuzz was able to detect the bug within minutes. Note that Node.Fz and stress testing fail to detect the bug for the

```
let start_watchdog ~delay dir =
  match watchdog dir with
  | Some _ ->
    assert (nb_listeners dir <> 0);
    Lwt.return_unit
  | None   ->
-    Log.debug "Start watchdog for %s" dir;
+    (* Note:  multiple threads can wait here *)
     listen dir ~delay ~callback >|= fun u ->
-    Hashtbl.add watchdogs dir u
+    match watchdog dir with
+    | Some _ -> u ()
+    | None ->
+    Log.debug "Start watchdog for %s" dir;
+    Hashtbl.add watchdogs dir u
```

Figure 4.9: Irmin bug #270

modified program. Despite the number of schedules increasing exponentially, MTTF increased linearly. This shows the efficiency of ConFuzz to find bugs even in programs with large schedule spaces.

RQ3: Practicality

As shown in results regarding RQ1 and RQ2, ConFuzz can effectively and efficiently detect concurrency bug in real-world applications. Moreover, tested real-world applications are widely used and have large codebase. We were able to reproduce detected bug reliably in real-world application as shown in Table 4.3. We shall look at each of these three bugs in detail:

[4]https://github.com/mirage/irmin/issues/270

Irmin #270 [4] Irmin is a distributed database built on the principles of Git. Similar to Git, the objects in Irmin are organized into directories. Irmin allows users to install *watchers* on directories which are callback functions that get triggered once when for every change in the directory. The bug had to do with the callbacks begin invoked multiple times if multiple watchers were registered to the same directory in quick succession. The patch is shown in Figure 4.9. When there are concurrent calls to `start_watchdog` in succession, it might turn out that all of them are blocked at `listen`. When the callback is triggered, each of these callbacks now adds an entry to the `watchdogs` hash table. The fix is to only add one entry to the hash table and for the rest, directly call the callback function. The property that we tested was that the callback function is invoked only once. Observe that the bug is input dependent; the bug is triggered only if concurrent calls to `start_watchdog` work on the same directory `dir` and the `delay` is such that they are all released in the same round.

Lwt #583 [5] Lwt has support for concurrent processing of an event stream through the `Lwt_react` module. An event stream can be thought of as a broadcast channel. When values are pushed into the channel, all of the receivers are notified with the value. `Lwt_react` has a function `limit` to rate an event stream `event_stream`.

```
let limited_stream =
  limit (fun _ -> sleep 1.0) event_stream
```

`limit` takes a callback function and an `event_stream` and returns a new event stream `limited_stream`, which only emits an event when the `event_stream` has a new

[5] https://github.com/ocsigen/lwt/issues/583

```
let query t ip =
  if Hashtbl.mem t.cache ip then (
    Hashtbl.find t.cache
  ) else (
    let cond = Lwt_condition.create () in
    Hashtbl.add t.cache ip (Incomplete cond);
+    let result = Lwt_condition.wait cond in
    output_probe t ip >>= fun () ->
    (* Note: cond can be signalled here *)
-    Lwt_condition.wait cond
+    result
  )
```

Figure 4.10: Mirage-tcpip bug #86

event and the callback function is resolved. Here, `limited_stream` emits an event once per second if the event stream emits an event more frequently. The problem occurs when the callback function itself emits an event on `event_stream`. In this case, `limited_stream` does not rate-limit this event and is emitted immediately. The property that we tested with ConFuzz is that no more than one event is emitted in the one-second time period from the `limited_stream`.

Mirage-tcpip #86 [6] Mirage-tcpip provides a networking stack for MirageOS [Madhavapeddy *et al.* (2013)]. It provides implementations for networking protocols such as IP, ICMP, UDP and TCP. The bug in mirage-tcpip occurs in the implementation of the ARP query. The `query` as shown in Figure 4.10 takes an IP address and returns the corresponding MAC address. The bug occurs when an IP address is not found in the internal ARP cache `cache` which results in initiating an external query `output_probe` to fetch the IP-MAC entry. To get notified about the cache update for the corresponding

[6] https://github.com/mirage/mirage-tcpip/pull/86

IP, `query` registers a condition variable `cond` with `cache` and starts waiting on it. The response handling of `output_probe` and notification of `cond` happens asynchronously outside `query`. The bug manifests in a particular situation when, before `query` starts waiting on `cond`, the `output_probe` response arrives and updates the `cache` notifying all the waited condition variables attached to it except `cond`. As `cond` was not waited upon, `query` waits indefinitely on it. The patch is shown in Figure 4.10 involving waiting on `cond` before initiating `probe_request`. The property that we tested was query should not be waiting when `cache` has corresponding IP entry.

These results show that ConFuzz can detect and reproduce concurrency bugs in real-world applications.

4.6 LIMITATIONS

While our experimental evaluation shows that ConFuzz is highly effective in finding bugs, we discuss some of the limitations of our approach. While ConFuzz captures most of the non-determinism present in event-driven concurrent programs, it cannot capture and control external non-determinism such as file read/write or network response. External non-determinism arises when interacting with external resources like file system, database, etc. which are outside the scope of ConFuzz.

To be completely certain about the order in which the asynchronous tasks are executed, ConFuzz serializes the worker pool tasks which might result in missing some of the concurrency bugs arising out of the worker pool-related races (although the concurrency bug study by Davis et al. [Davis *et al.* (2017)] did not identify any such races).

Serializing worker pool tasks help ConFuzz to *deterministically reproduce* detected bugs. We trade-off missing some of the worker pool-related concurrency bugs with the deterministic reproducibility of the detected bugs. Being a property-based testing framework, ConFuzz aims to generate failing tests cases that falsify the property. Hence, ConFuzz does not aim to detect traditional concurrency bugs such as data races and race conditions.

4.7 RELATED WORK

To the best of our knowledge, ConFuzz is the first tool to apply coverage-guided fuzzing, not just to maximize the coverage of the source code of program, but also to maximize the schedule space coverage introduced by a non-deterministic event-driven program. In this section, we compare ConFuzz to related work.

Concurrency fuzzing: AFL has been used previously to detect concurrency vulnerabilities in a Heuristic Framework [Liu *et al.* (2018)] for multi-threaded programs. Unlike ConFuzz, Heuristic Framework generates interleavings by changing thread priorities instead of controlling the scheduler directly, thereby losing the bug replay capability. Due to its approach, the Heuristic Framework can only find a specific type of concurrency bugs and has false positives. Heuristic Framework is applied to multi-threaded programs whereas ConFuzz is applied to event-driven programs. The most similar work to ConFuzz is the concurrency fuzzing tool Node.Fz [Davis *et al.* (2017)]. Node.Fz fuzzes the order of events and callbacks randomly to explore different schedules. Node.Fz can only find bugs that manifest purely as a result of particular scheduling, not as a property of program inputs. As Section 4.5 illustrates, the coverage-

guided fuzzing of ConFuzz is much more effective than Node.Fz at finding the same concurrency bugs.

Multithreaded programs: Many approaches and tools have been developed to identify concurrency bugs in multi-threaded programs. FastTrack Flanagan and Freund (2009*b*), Eraser [Savage *et al.* (1997)], CalFuzzer [Joshi *et al.* (2009)] aims to detect multi-threaded concurrency bugs like data races, deadlock. ConTest [Edelstein *et al.* (2003)], RaceFuzzer [Sen (2008)] uses random fuzzing to generate varied thread schedules. These approaches apply to multi-threaded programs for detecting concurrency bugs such as atomicity violations and race conditions on shared memory and are not directly applicable to event-driven programs interacting with the external world by performing I/O. Systematic exploration techniques such as model checking attempt to explore the schedule space of a given program exhaustively to find concurrency bugs. CHESS [Musuvathi and Qadeer (2007*a*)] is a stateless model checker exploring the schedule space in a systematic manner. While exhaustive state-space exploration is expensive and given a limited test time budget, ConFuzz explores broader input and schedule space, which is more likely to detect bugs.

Application domains: There are bug detection techniques to identify concurrency errors in client-side JavaScript web applications. WAVE [Hong *et al.* (2014)], WebRacer [Petrov *et al.* (2012)] and EventRacer [Raychev *et al.* (2013)] propose to find concurrency bugs in client-side elements like browser's DOM and webpage loading through static analysis or dynamic analysis. Though client-side web apps are event-driven, these techniques are tuned for client-side key elements like DOM and web page

loading which are not present in server-side like OCaml concurrent programs. Thus, the above approaches cannot be directly applied to event-driven OCaml applications. Android is another event-driven programming environment in combination with a multi-threaded programming model. Several dynamic data race detectors [Hsiao *et al.* (2014); Maiya *et al.* (2014); Bielik *et al.* (2015)] have been proposed for Android apps. These tools are tightly coupled with the Android system and target mainly shared memory races rather than violations due to I/O events.

4.8 CONCLUSION

In this chapter, we have presented, ConFuzz, a *directed* concurrency fuzzing tool that employs novel concurrency fuzzing technique PBCF for finding concurrency bugs in event-driven OCaml programs written using the Lwt library. Our performance evaluation shows that coverage-guided fuzzing of ConFuzz is more effective and efficient than the random fuzzing tool Node.Fz in finding the bugs. We also show that ConFuzz can detect bugs in large and widely used real-world OCaml applications without having to modify the code under test.

CHAPTER 5

PARAFUZZ

With the advancement in processor technology, multicore processors have become the norm in mobile, desktop and enterprise computing. Fast multicore machines are easily accessible at the click of a button thanks to cloud computing. Programmers must write multi-threaded softwares to realize the processing potential of multicore processors. Multi-threaded programming allows programmers to use threads to parallelize computation and reduce application processing time.

In this chapter, we apply novel technique of coverage-guided property-based concurrency fuzzing (PBCF) described in Chapter 3 to test multi-threaded programs. We instantiate PBCF technique in ParaFuzz, a concurrent property fuzz testing tool for Multicore OCaml parallel programs as an extension to ConFuzz. ParaFuzz also supports record and replay to *deterministically reproduce* the concurrency bug. ParaFuzz is developed as a drop-in replacement for the Multicore OCaml compiler and employs a novel approach to control threads of multi-threaded parallel programs without requiring the *any changes* to the test programs.

Unlike model checking, ParaFuzz does not guarantee testing on all possible thread schedules, but generate and test programs on interesting inputs and thread schedules that maximize the likely hood of finding a property failure. Due to its efficiency, ParaFuzz enables a programmer to explore a broader schedule space of the multi-

threaded program with a limited test time budget, thereby increasing the chances of finding bugs in the same limited time. This makes ParaFuzz an efficient and practical concurrency testing tool for programmers wanting to test their multi-threaded code in a reasonable amount of time.

The main contributions of ParaFuzz work are as follows:

- We implement the PBCF technique in ParaFuzz, a drop-in replacement for testing Multicore OCaml parallel programs.
- We present a novel approach to implement thread scheduler by *effect handlers* to control threads without changing thread API.
- We show by experimental evaluation that ParaFuzz is more effective and efficient than random and stress testing in finding concurrency bugs. ParaFuzz was able to find 1 previously *unknown concurrency bug* in widely used Multicore OCaml library.

The remainder of this chapter is organized as follows. We present a motivating example in Section 5.1 that illustrates the effectiveness of ParaFuzz on an adversarial example. Section 5.2 provides an overview of the parallel programming in Multicore OCaml. ParaFuzz is introduced in Section 5.3. Section 5.4 discusses the implementation details of ParaFuzz. Assumptions made by ParaFuzz along with reasonable justification are described in Section 5.5. Experimental evaluation is described in Section 5.6. Related work is discussed in Section 5.7, and we conclude the chapter with Section 5.8.

5.1 MOTIVATING EXAMPLE

We describe a simple, adversarial example to illustrate the effectiveness of ParaFuzz in finding concurrency bugs in parallel programs that depend on both input and thread

```
1 let test i =
2     let x = Atomic.make i in
3     let y = Atomic.make 0 in
4     let dom = Domain.spawn(fun () ->
5         if (Atomic.get x = 10) then Atomic.set y 2)
6     in
7     Atomic.set y 1;
8     Domain.join dom;
9     assert (Atomic.get y <> 2)
```

Figure 5.1: A program with input+schedule concurrency bug

schedule. Figure 5.1 shows a Multicore OCaml parallel program. The program contains a single function `test` that takes an integer parameter `i`. `test` function initializes two atomic variables `x` and `y` with value `i` and `0` respectively. `Domain.spawn` creates a different thread the runs in parallel with the main thread. `dom` thread set the `y` to `2` if the value of `x` is `10`. `test` function on line 7 sets `y` to `1`. `Domain.join` waits for `dom` to terminate. Line 5 and Line 7 execute in parallel by different threads. At the end `test` function asserts the value of `y` to be not equal to `2`.

This program has a concurrency bug; there exists a particular combination of input value `i` and schedule between the two threads that will cause the value of `y` to become equal to `2`, causing the assertion to fail. There are 2^{63} [1] possibilities for the value of `i` and therefore the value of `x` and 3 possible thread schedules for the 2 threads. Out of these, there is only one possible combination of input and thread schedule for which the assertion fails.

- `i` = 10 and schedule = **[Atomic.get x = 10; Atomic.set y 1; Atomic.set y 2]**.

[1] OCaml uses tagged integer representation [Leroy (1990)] where 1 bit is used to distinguish immediate values from pointers.

Table 5.1: Comparing different testing techniques

Testing Technique	Executions (millions)	Time (minutes)	Bug Found
ParaFuzz	0.55	10.5	**Yes**
Random	108.6	60	**No**
Stress	25.2	60	**No**

ParaFuzz aims to find concurrency bugs that depend on a combination of input and thread schedule alongside bugs that only depend on a particular thread schedule. As the bug in example program depends on input and interleaving, concurrency testing techniques focusing only on generating different interleavings will fail to find this bug. This is evident when the program is executed under different testing techniques. Table 5.1 shows a comparison of ParaFuzz with the random and stress testing for the example program.

Random testing controls the thread scheduling decisions and randomizes the order of thread executions. Stress testing runs the program normally without controlling the scheduling decisions. Inputs to the programs are generated randomly in stress and random testing, whereas in ParaFuzz, inputs are generated by AFL. We test the example program with each technique until a bug is found or a timeout of 1 hour is reached. We report the number of executions and time taken if the bug was found. Only ParaFuzz was able to find the bug. Random and Stress testing were not able to find the bug despite running significantly longer with far more executions. Although this example is synthetic, we observe similar patterns in real-world programs where the bug depends on the combination of the input value and the schedule, and cannot be

discovered with a tool that only focuses on one of the sources of non-determinism.

5.2 MULTICORE OCAML

Recall from Section 2.5.2 that, Multicore OCaml adds native support for concurrency and parallelism to the OCaml language. Multicore OCaml implements parallelism and concurrency abstraction through domains and fibers respectively. We shall discuss them in detail now.

5.2.1 Domains

Multicore OCaml implements shared-memory parallelism through domains. A domain is a unit of parallelism in Multicore OCaml. Domains map one-to-one with OS threads and thus execute in parallel with other domains. Domains are fairly heavyweight as they include their own heap and the synchronization required to manage a multicore garbage collection (GC). The domainslib library [domainslib (2021)] - a high-level parallel programming library makes it easy to write parallel programs over domains. From here onwards, we refer to the parallel thread of execution as a domain to avoid.

5.2.2 Effect handlers

Effect handlers [Plotkin and Pretnar (2009)] provide a modular foundation for user-defined effects. The key idea is to separate the definition of the effectful operations from their interpretations, which are given by *handlers* of the effects. Effect handlers are a generalization of exception handlers, where, in addition to the effect being handled, the handler is provided with the delimited continuation [Danvy and Filinski (1990)]

of the perform site. Effect handlers allow for non-local control flow mechanisms such as generators, async/await, lightweight threads and coroutines to be composably expressed. Effect handlers in Multicore OCaml are implemented using efficient runtime managed stack segments called *fibers* [Sivaramakrishnan *et al.* (2021)].

Multicore OCaml extends OCaml with the ability to declare user-defined effects with the help of the *effect* keyword. For example,

```
effect E: int -> int
```

declares an effect `E`, which is parameterized with an integer, which when performed returns an integer. A computation can perform the effect `E` without knowing how the effect `E` is implemented. This computation may be enclosed by different handlers that handle `E` differently. Effects are performed with the `perform` primitive, which performs the effect and returns the result.

```
let v = try perform (E 42) with
| effect (E i) k -> continue k (i+1)
```

In the code above, effect `E` is performed with 42 as the argument. To handle this effect `E`, effect handler are used. The pattern `effect (E i)k` is the effect handler which handles the effect `E`, `i` is the parameter passed to `E` while performing it, and `k` is the delimited continuation. Intuitively, `k` represents suspended computation between the site where the effect was performed and the effect handler. You can continue the continuation using `continue` keyword and program control will resume from the effect perform site. Here in this case effect `E` is continued with `i+1`. This results in `perform`

`E 42` returning with value 43 and variable `v` is assigned 43. This capability of effect handlers to resume suspended computations later enables Multicore OCaml to have asynchronous I/O in direct-style instead of callback-oriented style found in languages like JavaScript.

5.3 PARAFUZZ

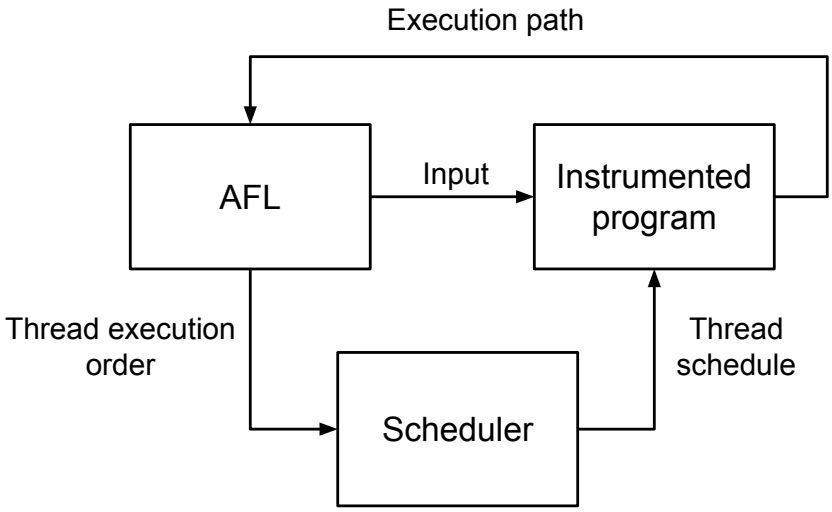


Figure 5.2: ParaFuzz architecture

We have implemented the PBCF technique for Multicore OCaml parallel programs in ParaFuzz tool. ParaFuzz combines property-based testing and AFL with concurrency testing for parallel programs. Figure 5.2 shows ParaFuzz 's architecture. Unlike ConFuzz, where the scheduler is implemented in OCaml code, the scheduler of Multicore OCaml parallel programs written using domains, is controlled by OS. This poses a challenge in the form of how to perform ConFuzz style concurrency fuzzing for parallel programs? The key observation is to simulate the Domain programming

interface using a user-defined scheduler written using effect handlers (Section 5.2.2). Since effect-handlers allow you to write concurrent programs in direct style, we can implement the Domain programming interface as a drop-in replacement for the Multicore OCaml standard library. As a result, ParaFuzz can test unmodified programs developed using standard library's Domain programming interface or any other library built on top of it (domainslib). ParaFuzz being an extension of ConFuzz for Multicore OCaml programs supports all the features of ConFuzz such as record and replay, *no false positives* etc.

5.3.1 ParaFuzz scheduler

ParaFuzz scheduler is the core part of the ParaFuzz. ParaFuzz scheduler's main job is to generate domain schedule with the help of AFL.The changes made in Multicore OCaml compiler to capture non-determinism (Section 5.4.2) call ParaFuzz scheduler's API (Figure 5.3). Scheduler tracks domain ready to execute next in a *ready list*. ParaFuzz scheduler then chooses the next domain to run with the number generated by AFL. By choosing a domain to run at each step, ParaFuzz scheduler generates domain schedules. On detecting a concurrency bug, the AFL-generated random numbers are saved in a separate file as a schedule trace. With the schedule trace, the ParaFuzz scheduler can reproduce and enforce a domain schedule. Using this capability of the scheduler, ParaFuzz can replay a domain schedule to reliably expose the detected concurrency bugs. Deterministic replay helps programmers find the exact cause of concurrency bugs in parallel programs.

The order in which each domain is executed to run next affects the parallel program's

execution path. We make every context switch between domains seem as a branching condition to AFL as AFL can recognize branching conditions in programs and interprets the sequence of branching conditions as program's execution path. ParaFuzz essentially converts domain schedule to sequential program's execution path which can be recognized by AFL. Due to the program instrumentation inserted in programs, AFL recognizes the program execution path in every program run. AFL being a coverage-guided fuzzer, tries to increase coverage (execution paths). AFL thus generates random numbers that produce alternative domain execution orderings. Alternative domain execution orderings result in new domain schedules that exercise new program execution paths. ParaFuzz scheduler keeps on generating new domain schedules until AFL is able to find new execution paths. ParaFuzz thus uses AFL fuzzing to generate and execute programs under varied domain schedules.

We now describe the novel approach used in ParaFuzz scheduler to control domains without requiring the test programs to change. ParaFuzz scheduler uses effect handlers (Section 5.2.2) to mock domains. ParaFuzz scheduler internally implements domain as fibers (Section 5.2.2) instead of OS threads which can be executed concurrently (time-sharing fashion) with other domains. This enables ParaFuzz to have tight control over domain executions which is necessary for precisely enforcing domain schedules. There are mainly two major advantages of using effect handlers in ParaFuzz scheduler. First, the Domain API is not modified which helps ParaFuzz to test programs without any modification. This makes ParaFuzz easier to adopt and test a wide variety of parallel programs. Second, ParaFuzz essentially converts a Multicore OCaml parallel program

```
type 'a cont
(** Represents a blocked computation that waits for a value of
    type 'a. *)

val context_switch : unit -> unit
(** [context_switch] switches to the next runnable as
   determined
    by scheduler. *)

val fork : (unit -> unit) -> int
(** [fork f] spawns a new runnable in the scheduler and returns
    [id] of the forked runnable. *)

val suspend : (('a cont * int) -> unit) -> 'a
(** [suspend f] applies [f] to the current continuation, and
    suspends the execution of the current runnable, and
    switches to the next runnable in the scheduler's queue. *)

val resume : ('a cont * 'a * int) -> unit
(** [resume (k,v,id)] prepares the suspended continuation [k]
    with value [v], domain identifier [id] and enqueues it as
    well current runnable to the scheduler queue. *)

val resume_without_context_switch : ('a cont * 'a * int) ->
   unit
(** [resume_without_context_switch (k,v,id)] prepares the
    suspended continuation [k] with value [v], domain
    identifier [id] and enqueues it to the scheduler queue and
    continues with the current runnable. *)

val run : (module AFLQueue) -> (unit -> unit) -> unit
(** [run m f] runs [f] with the AFL controlled queue [m]. *)

val get_current_domain_id : unit -> int
(* Returns the ID of the current thread running *)
```

Figure 5.3: Scheduler API

to a single domain sequential OCaml program by mocking parallelly executed domains as concurrently executed fibers running on a single OS thread. AFL doesn't work with parallel programs and requires programs to be single-threaded. The effect handlers approach makes it possible for ParaFuzz to use AFL to test Multicore OCaml parallel programs.

To mock domains using effect handlers, ParaFuzz scheduler API functions (Figure 5.3) called for capturing non-determinism are internally implemented as simply performing effects using `perform` primitive. The effect implementation is shown in Figure 5.4. For instance ParaFuzz scheduler's `context_switch` function performs effect `Context_switch`. The `Context_switch` effect yields control to ParaFuzz scheduler. The `Fork` effect takes a thunk which is spawned as a concurrent domain. Effect `Suspend` and `Resume` suspends and resumes a domain respectively. The `Id` effect returns the id of the calling domain. We also implement our version of synchronization primitives: *Mutex* and *Condition variables* using effect handlers to enable ParaFuzz scheduler to control the order in which a Mutex is acquired/released or a Condition is waited upon/signalled.

The scheduler effects are handled by effect handlers defined in `run` function. `run` is the core function of the ParaFuzz scheduler. Figure 5.5 describes the `run` function. It takes the program to test as a thunk (`main` argument) and handles the effect performed in the test program. `afl_module` is the AFL controlled queue required by ParaFuzz scheduler. `current_id` tracks currently executing domain

```
type 'a cont = ('a,unit) continuation
effect Context_switch : unit
effect Fork           : (unit -> unit) -> int
effect Suspend : (('a cont * int) -> unit) -> 'a
effect Resume  : ('a cont * 'a * int) -> unit
effect Id : int

let fork f = perform (Fork f)
let suspend f = perform (Suspend f)
let resume (k,v, id) = perform (Resume (k,v,id))
let context_switch () = perform Context_switch
let get_current_domain_id () = perform Id
```

Figure 5.4: ParaFuzz Scheduler effect implementation

id, `next_id` provides for id for next spawned domain and `domains_finished` denotes number of domains terminated. Apart from generating and enforcing domain schedules ParaFuzz scheduler, scheduler checks for deadlock in parallel programs through `check_for_deadlock` function. `check_for_deadlock` checks whether all spawned threads are terminated or not. The function spawn (line 17) evaluates the computation `f` in an effect handler. The computation `f` may return normally with a value, or perform scheduler effects. The pattern effect `Context_switch k` handles the effect `Context_switch` and binds `k` to the continuation of the corresponding `perform` delimited by this handler. The scheduler queue `M` backed by AFL-controlled queue `afl_module` maintains a queue of these continuations. `enqueue` pushes functions into the queue which when called resumes the continuations using the `continue` primitive and updates the current domain id. `dequeue` pops functions from the queue and calls it. If the queue is empty, we check for deadlock using `check_for_deadlock`. In the case of the `Context_switch` effect, we enqueue the current continuation k and resume the next available continuation. In the case of the `Fork f` effect, we enqueue the current continuation, update the `current_id` with the

id of the newly spawned domain `next_id` and recursively call spawn on `f` in order to run `f` concurrently. For `Suspend f` effect, we first call the function `f` with current continuation `k` and currently executing domain id `current_id`. Then we resume the next available continuation using `deqeue`. In the case of `Resume k',v,id` effect, we enqueue the current continuation and the continuation to be resumed `k` with value `v` and domain id `id`. Finally, resuming the next available continuation. Effect `Id` simply continues current continuation with the calling domain id `current_id`.

5.4 FUZZING UNDER NON-DETERMINISM

In this section, we discuss the non-determinism present in Multicore OCaml programs. We then show how ParaFuzz captures and fuzzes this non-determinism.

5.4.1 Synchronization points in Multicore OCaml

Non-determinism in Multicore OCaml mainly comes from non-determinism in domains execution. To generate domain schedule ParaFuzz context switches only before every synchronization point due to the optimization mentioned in Section 5.5. In this section, we discuss the synchronization points present in Multicore OCaml parallel program.

Domain synchronization points Domains in Multicore OCaml execute in parallel and are scheduled by operating system which makes the order of execution of domains at runtime non-deterministic. Concurrency bugs that depend on a particular schedule of domain execution can be caught by fuzzing the order of domain executions. Multicore

```
let run afl_module main =
    let current_id = ref 0 in
    let next_id = ref 1 in
    let domains_finished = ref 0 in

    let check_for_deadlock () =
        if (!next_id - !domains_finished) = 0 then ()
        else failwith "Deadlock!"  in

    let module M = (val afl_module : AFLQueue) in
    let enqueue id k v =
        M.enqueue (fun () -> current_id := id; continue k v) in
    let dequeue () =
        if M.is_empty () then check_for_deadlock ()
        else M.dequeue () () in

    let rec spawn f =
        match f () with
        | () -> incr domains_finished; dequeue ()
        | effect Context_switch k ->
            enqueue !current_id k ();
            dequeue ()
        | effect (Fork f) k ->
            enqueue !current_id k (!next_id);
            current_id := !next_id;
            incr next_id;
            spawn f
        | effect (Suspend f) k ->
            f (k,!current_id);
            dequeue ()
        | effect (Resume (k',v, id)) k ->
            enqueue id k' v;
            enqueue !current_id k ();
            dequeue ()
        | effect (Id) k ->
            continue k (!current_id)
    in
    spawn main
```

Figure 5.5: ParaFuzz Scheduler run function

OCaml exposes threading abstraction via Domain API. Domain API provides methods for programs to create domain for parallel execution and wait for it to terminate. Domain API is shown in Figure 5.6.

```
val spawn : (unit -> 'a) -> 'a t
(** [spawn f] creates a new domain that runs in parallel with
    the current domain. *)

val join : 'a t -> 'a
(** [join d] blocks until domain [d] runs to completion.
    If [d] results in a value, then that is returned by
    [join d]. If [d] raises an uncaught exception, then
    that is thrown by [join d]. Domains may only be joined
    once: subsequent uses of [join d] raise Invalid_argument.
    *)
```

Figure 5.6: Domain API

Atomic synchronization points Multicore OCaml provides two types of mutable variables: atomic and non-atomic mutable variables. Accesses to atomic mutable variables are executed atomically and are therefore synchronized with respect to accesses from different domains. Atomic API in Multicore OCaml provides support for mutable atomic variables and supporting methods that operate on them. Figure 5.7 shows Multicore OCaml atomics API. During parallel program execution, order of each of the method in Atomic API affects the shared variable state. Any assumption on the order in which Atomic methods are executed may lead to concurrency bugs.

Synchronization primitives synchronization points Synchronization primitives namely locks and condition variables (CV) are used to provide synchronized access

```
(** Create an atomic reference. *)
val make : 'a -> 'a t

(** Get the current value of the atomic reference. *)
val get : 'a t -> 'a

(** Set a new value for the atomic reference. *)
val set : 'a t -> 'a -> unit

(** Set a new value for the atomic reference, and return the
   current value. *)
val exchange : 'a t -> 'a -> 'a

(** [compare_and_set r seen v] sets the new value of [r] to [v]
    only if its current value is physically equal to [seen] --
    the comparison and the set occur atomically. Returns [true]
    if the comparison succeeded (so the set happened) and
    [false] otherwise. *)
val compare_and_set : 'a t -> 'a -> 'a -> bool

(** [fetch_and_add r n] atomically increments the value of [r]
    by [n], and returns the current value (before the
    increment). *)
val fetch_and_add : int t -> int -> int

(** [incr r] atomically increments the value of [r] by [1]. *)
val incr : int t -> unit

(** [decr r] atomically decrements the value of [r] by [1]. *)
val decr : int t -> unit
```

Figure 5.7: Atomic API

to mutable non-atomic variables in Multicore OCaml. Non-synchronization access of non-atomic variables can lead to data races. Still, usage of synchronization primitives cannot eliminate the risk of race conditions in parallel programs. The order of acquiring/releasing a lock or waiting/signalling a CV by domains can affect the state of non-atomic variables. Race conditions arising out of synchronization primitives can be caught by fuzzing the order of execution of synchronization primitives. Multicore OCaml provides two synchronization primitives to use in parallel programs: *Mutex*

```
val create : unit -> t
(** Return a new mutex. *)

val lock : t -> unit
(** Lock the given mutex. Only one thread can have the mutex
    locked at any time. A thread that attempts to lock a mutex
    already locked by another thread will suspend until the
    other thread unlocks the mutex. *)

val try_lock : t -> bool
(** Same as {!Mutex.lock}, but does not suspend the calling
    thread if the mutex is already locked: just return [false]
    immediately in that case. If the mutex is unlocked, lock it
    and return [true]. *)

val unlock : t -> unit
(** Unlock the given mutex. Other threads suspended trying to
    lock the mutex will restart.  The mutex must have been
    previously locked by the thread that calls {!Mutex.unlock}.
    *)
```

Figure 5.8: Mutex API

and *Condition* for locks and CV respectively. Figure 5.8 and 5.9 describe Mutex and Condition API respectively.

5.4.2 Capturing synchronization points in Multicore OCaml

In this section, we discuss how ParaFuzz controls and captures the synchronization points described in Section 5.4.1. ParaFuzz scheduler (Section 5.3.1) uses captured synchronization points to generate varying domain schedules.

Capturing Domain synchronization points To capture domain synchronization points we insert calls to ParaFuzz scheduler in Domain API described in Figure 5.6. The

```
val create : unit -> t
(** Return a new condition variable. *)

val wait : t -> Mutex.t -> unit
(** [wait c m] atomically unlocks the mutex [m] and suspends
    the calling process on the condition variable [c]. The
    process will restart after the condition variable [c]
    has been signalled. The mutex [m] is locked again before
    [wait] returns. *)

val signal : t -> unit
(** [signal c] restarts one of the processes waiting on the
   condition variable [c]. *)

val broadcast : t -> unit
(** [broadcast c] restarts all processes waiting on the
   condition variable [c]. *)
```

Figure 5.9: Condition API

inserted calls in Domain API inform ParaFuzz scheduler about domains creation and termination. This enables ParaFuzz scheduler control precisely when a particular domain starts execution and when it is allowed to terminate. By delaying domain execution and termination, ParaFuzz scheduler can catch concurrency bugs that depend upon fixed order/time of domain creation and termination. The inserted calls to ParaFuzz scheduler do not modify Domain API in any way, which makes it possible to test unmodified Multicore OCaml programs to be tested by ParaFuzz. `Domain.spawn` creates a new domain to execute function passed in parallel. We insert calls to ParaFuzz scheduler's fork method `Scheduler.fork` in `Domain.spawn`. `Scheduler.fork` as shown in Figure 5.10 takes a function that forms the code to be executed parallelly in a separate domain and returns an integer `id` of the newly created domain. `Scheduler.fork` enables ParaFuzz scheduler start tracking and controlling

```
val fork : (unit -> unit) -> int
(** [fork f] spawns a new runnable in the scheduler and returns
    [id] of the forked runnable. *)
```

Figure 5.10: Scheduler.fork

the newly created domain's execution.

Capturing Atomics synchronization points ParaFuzz treats Atomic methods described in Figure 5.7 as synchronized regions and makes domains context switch before executing each Atomic method to generate domain schedule that can violate property defined on a shared variable. By making domain context switch before each atomic access, ParaFuzz scheduler can control the order in which domains access a particular atomic variable. ParaFuzz scheduler can generate domain schedule that can manifest concurrency bugs that depend on a particular access order of atomic variables. To enable ParaFuzz scheduler to context switch to different domain before accessing an atomic variable, we insert calls to ParaFuzz scheduler in each of the Atomic API methods. We insert calls to scheduler's `Scheduler.context_switch` method that transfers the control to ParaFuzz scheduler. Once the control is transferred to ParaFuzz scheduler, the next domain to be executed is chosen according to the generated domain schedule. Figure 5.11 describes `Scheduler.context_switch` method.

```
val context_switch : unit -> unit
(** [context_switch] switches to the next runnable as
    determined by scheduler. *)
```

Figure 5.11: Scheduler.context_switch

Capturing synchronization primitives synchronization points Multicore OCaml synchronization primitives *Mutex* and *Condition* provides synchronized access to non-atomic variables. Still, race conditions can arise on the state of non-atomic variables. This type of race condition can be caught by controlling the order in which a Mutex is acquired/released or when a Condition is waited upon/signaled. We insert calls to ParaFuzz scheduler before executing Mutex and Condition API. The inserted calls enable ParaFuzz scheduler to fuzz the order of acquiring/releasing a Mutex or waiting/signaling a Condition.

To capture the synchronization points in Mutex API shown in Figure 5.8, we insert calls to `Scheduler.context_switch` (Figure 5.11) to let ParaFuzz scheduler context switch before acquiring/releasing a Mutex. Further, to handle domains blocked waiting on a Mutex, we insert calls to `Scheduler.suspend` while acquiring a Mutex. `Scheduler.suspend` shown in Figure 5.12 takes a function `f` and blocks waiting to be unblocked by Mutex release. The function `f` pushes domain wanting to acquire Mutex in the list of domains waiting for the Mutex to be released. `Scheduler.suspend` enables ParaFuzz scheduler to stop tracking blocked domain for schedule generation by removing it from the list of ready domains.

```
val suspend : (('a cont * int) -> unit) -> 'a
(** [suspend f] applies [f] to the current computation, and
    suspends the execution of the current runnable, and
    switches to the next runnable in the scheduler's queue. *)
```

Figure 5.12: Scheduler.suspend

Similarly, to unblock domains waiting on a Mutex, we insert calls to `Scheduler.resume` in `Mutex.unlock`. `Scheduler.resume` calls into ParaFuzz scheduler to unblock and resumes a waiting domain on a Mutex. Figure 5.13 describes `Scheduler.resume` that takes blocked domain enqueued while executing `Scheduler.suspend`, a value to resume with and a domain identifier. `Scheduler.resume` enables ParaFuzz scheduler to resume tracking of the blocked domain and include blocked domain in the list of domains ready to be run next.

```
val resume : ('a cont * 'a * int) -> unit
(** [resume (k,v,id)] prepares the suspended computation [k]
    with value [v], domain identifier [id] and enqueues it
    as well current runnable to the scheduler queue. *)
```

Figure 5.13: Scheduler.resume

For capturing the synchronization points in Condition API described in Figure 5.9, we insert calls to `Scheduler.context_switch` (Figure 5.11) in similar way like done in Mutex to let ParaFuzz scheduler context switch before waiting/signaling a Condition. As Condition internally uses Mutex, we do not insert calls to either `Scheduler.suspend` or `Scheduler.resume` to simulate domains waiting on condition or released from waiting on condition respectively as those calls are called by Mutex API internally.

5.5 ASSUMPTIONS

To make ParaFuzz an efficient and practical tool to find concurrency bugs in Multicore OCaml programs, we make certain assumptions which we discuss next. To generate domain schedules, ParaFuzz controls domains execution by context switching at pre-defined program location in each domain execution. Theoretically, as each domain can context switch before each instruction, ParaFuzz should context switch before every instruction instead of pre-defined program locations to generate domain schedule. But this would make ParaFuzz too slow to be a practical tool to find concurrency bugs. As shown by D.Bruening [Bruening (1999)], it is sufficient just to enumerate possible orders of the synchronized regions of the program to covers all possible behaviors of the program. Synchronized regions are atomic blocks of the program that accesses shared variables and are protected by synchronized primitives. By enumerating only synchronized regions, the number of schedules to consider are reduced considerably. Rather than considering all possible schedules at the instruction level, it is sufficient to only consider schedules of the program's atomic blocks. To apply this optimization, a program should follow the *mutual-exclusion locking discipline*. A *mutual-exclusion locking discipline* dictates that each shared variable is associated with at least one mutual-exclusion lock and that the lock or locks are always held whenever any thread accesses that variable. To see why enumerating the orders of the atomic blocks is sufficient, consider the following. The order of two instructions in different threads can only affect the behavior of the program if the instructions access some common variable - one instruction must write to a variable that the other reads or writes. In a program that correctly follows a mutual-exclusion locking discipline, there must be at

least one lock that is always held when either instruction is executed. Thus they cannot execute simultaneously, and their order is equivalent to the order of their synchronized regions.

ParaFuzz uses D.Bruening's optimization to reduce the number of domain schedule to be explored to find concurrency bugs. For this optimization, ParaFuzz requires each shared variable in a Multicore OCaml program to be either *atomic variable* or every read/write access be properly synchronized using synchronization primitives so as to follow *mutual-exclusion locking discipline* as required by the optimization.

But this limits the application of ParaFuzz to Multicore OCaml programs not containing data races. Data race occurs when more than one thread tries to access shared data without using any synchronization like locks and one of the access being write. Due to data races, programs containing data races do not follow the *mutual-exclusion locking discipline* required by ParaFuzz. Programs with data races also exhibit *relaxed memory ordering* due to compiler optimizations and modern multicore hardware that exhibits relaxed behaviours. So it is not sufficient to just simulate the interleaving of threads, which would only simulate sequential consistency behaviours. Also, data races in Multicore OCaml have well-defined semantics [Dolan *et al.* (2018)] unlike undefined behavior in the C++ memory model. In Multicore OCaml, reads involved in data races can return a set of values. To simulate the data race behavior, we need to keep track of every read and write of every memory location, which can affect ParaFuzz's performance. Due to this, ParaFuzz makes a only single assumption that the program to be tested should be *data race-free*. To make ParaFuzz a practical technique, we

make this trade-off of missing some data race bugs. Still, we believe that this is a reasonable assumption to make. *Why?*

Multicore OCaml programs are mainly written using a high level parallel programming library - domainslib [domainslib (2021)]. The domainslib is being developed and maintained by the core Multicore OCaml team, which make it less likely for a program written using domainslib to contain data races. Even if the program to be tested contains a data race, it is possible that ParaFuzz may actually find the bug. The bug may not be due to the data race and hence would be identified by interleaving at synchronization points. As in some cases it is sufficient to context switch just before every synchronized shared variable access to test the data race behavior. Finally, it is very easy to extend ParaFuzz in the future to consider data race programs due to Multicore OCaml memory model having a simple operational instantiation. However, the scalability of such a technique is to be determined.

5.6 EVALUATION

In this section, we evaluate the effectiveness of ParaFuzz in finding concurrency bugs in real-world Multicore OCaml applications and benchmark programs. Additionally, we check the efficiency of ParaFuzz in terms of time required to detect concurrency bugs in comparison to random and stress testing. In random testing, the thread scheduler is controlled like ParaFuzz, but the thread schedule is generated randomly. We use ParaFuzz 's random testing mode (Section 5.3) to simulate random testing. Stress testing runs the program normally without controlling the scheduling decisions. ParaFuzz generates program input using AFL, while in random and stress testing

Table 5.2: Experimental subjects

Name (abbreviation)	Bug type
mysql-bug (SQL)	race-condition
circular-list (CL)	race-condition
deadlock3 (D3)	deadlock
buffer-if (BI)	deadlock
buffer-notify (BN)	deadlock
RAX-jpf (RAX)	deadlock
domainslib (DL)	deadlock
motivating-example (MX)	race-condition
effective-random-testing-example (ERT)	race-condition

inputs are generated randomly. We design and conduct experiments to answer the following questions:

- **RQ1: Effectiveness** – How frequently is ParaFuzz able to find input+schedule dependent bugs?
- **RQ2: Efficiency** – How many executions are required to detect bugs by ParaFuzz as compared to random and stress testing?
- **RQ3: ParaFuzz's Efficiency to explore schedules** – Is ParaFuzz effective than random testing in finding schedule-only dependent bug?

5.6.1 Experimental subjects and setup

We evaluated ParaFuzz on both real-world Multicore OCaml applications and benchmark programs. Table 5.2 summarises the applications and benchmark programs used for the evaluation. We have used nine experimental subjects comprising of real-world applications and benchmark programs as experimental subjects for evaluating ParaFuzz. All of the programs contain at least one known concurrency bug.

For experimental evaluation, we have build a dataset of 20 known parallel program

bugs mentioned in related work [Bruening (1999); Yu *et al.* (2012); Visser *et al.* (2003); Sen (2007)]. The dataset comprises bugs of real-world and benchmark programs written mainly in C, C++ and Java. Each of the program in the dataset contains at least one known concurrency bug. We select a bug only if the bug report contains a clear description or has an automated test case to reproduce the bug. We exclude data race bugs, as ParaFuzz requires test programs to be data race free. We port the remaining testable bugs to standalone Multicore OCaml programs. We excluded those bugs from the dataset which either could not be ported to Multicore OCaml or were data race bugs. We were able to port 7 multi-threaded bugs from the dataset. Dataset programs contain schedule-only dependent bugs. As ParaFuzz targets input dependent concurrency bugs (exposed only when a program executes with a specific input and thread schedule), we add additional input non-determinism in dataset programs and modify it accordingly to make concurrency bugs in those programs input-dependent. All the experimental subjects are taken from related work, except MX and DL. MX is the motivating example from Section 5.1. DL is the *previously unknown* deadlock bug in channel implementation found by ParaFuzz in Multicore OCaml parallel programming library: domainslib [domainslib (2021)].

We design our experiments to compare ParaFuzz 's bug detection capability with random and stress testing (hereby referred to as the testing techniques). We perform 30 testing runs for each experimental subject (Table 5.2) and testing technique. A testing run is a single invocation of the testing technique. The performance metric we focus on is mean time to failure (MTTF), which measures how quickly a concurrency bug is found in terms of time. A single test execution indicates one execution of the respective

application's test case. For each subject and testing technique, we execute the respective subject application until the first concurrency bug is found or a timeout of 1 hour occurs. For each such run, we note the time taken to find the first concurrency bug and whether a bug was found or not. We ran all of our experiments on a machine with a 6-Core Intel i5-8500 processor, 16GB RAM, running Linux 4.15.0-1034.

5.6.2 Finding novel bug in domainslib

We found one *previously unknown* concurrency bug in Multicore OCaml parallel programming library: domainslib. While running the test suite of domainslib using ParaFuzz, we observed that one of the test cases hangs in some of the runs. On investigating further, we found a deadlock in domainslib's channel implementation. The deadlock had to with two receivers waiting for a message on the same channel. When a message is sent by a sender to the same channel, it causes the message to be lost resulting in a deadlock. The root cause is sender sends the message to the first receiver, but it wakes up the second receiver. As a result the first receiver is not woken up and the second receiver despite being woken up does not receive the message and goes back to waiting. This results in the message being lost and the two receivers still waiting leading to a deadlock. ParaFuzz was able to detect deadlock within 1 second every time as shown discussed in Section 5.6.3. We reported [2] the deadlock to the domainslib team, which was acknowledged as a real bug and a fix was introduce in the domainslib library. This demonstrates the ParaFuzz's capability in detecting concurrency bugs in real-world programs and libraries.

[2] https://github.com/ocaml-multicore/domainslib/issues/25

Table 5.3: Bug detection capability of the techniques. Each entry is the fraction of the testing runs that manifested the concurrency bug.

	Stress	**Random**	**ParaFuzz**
SQL	0.00	0.00	1.00
CL	0.00	0.00	0.96
D3	0.00	0.00	1.00
BI	0.00	0.03	1.00
BN	0.00	0.00	1.00
RAX	0.00	0.00	1.00
DL	0.00	1.00	1.00
MX	0.00	0.00	1.00
ERT	0.00	0.00	1.00
Avg	0.00	0.003	0.99

5.6.3 Experimental Results

RQ1: Effectiveness

Table 5.3 shows the bug detection capabilities of the three testing techniques. The first column shows the abbreviation of the experimental subjects. The second to fourth column shows the bug detection results of Stress, Random and ParaFuzz testing, respectively. Each cell in the table shows the fraction of the testing runs that detected a concurrency bug out of the total 30 testing runs per experimental subject and testing technique.

As shown in Table 5.3, ParaFuzz detected concurrency bugs in every testing run for all experimental subjects (all cells are 1.00) except CL. Even in the case of CL, ParaFuzz was able to find concurrency bugs with the ratio of 0.96 (29 out of 30 testing runs), as compared to stress and random testing which were not able to detect bugs even in a single run. In-fact except BI and DL, ParaFuzz is the only technique able to detect

Table 5.4: Mean time to find the concurrency bug (seconds)

	Stress	**Random**	**ParaFuzz**
SQL	-	-	734.26
CL	-	-	971.16
D3	-	-	469.63
BI	-	100.76	20.36
BN	-	-	875.4
RAX	-	-	111
DL	-	0	0
MX	-	-	625.36
ERT	-	-	88.83

concurrency bugs, that too in every testing runs. This confirms that ParaFuzz was able to generate concurrent schedules along with inputs more effectively. Except for BI and DL, random testing could not detect a bug in the remaining programs. For BI, random testing managed to detect the bug only in single testing run (0.3). ParaFuzz and random testing both were able to detect the novel deadlock bug in DL. DL is not input-dependent and concurrency heavy which makes it easy for random testing to find it. Stress testing failed to detect input dependent bugs in any program. In addition, ParaFuzz does not produce *false positives*, as schedules explored by ParaFuzz are all legal schedules in Multicore OCaml. Thus, the results confirm that ParaFuzz is effective at detecting input-dependent concurrency bugs.

RQ2: Efficiency

Table 5.4 shows the efficiency results of the three testing techniques. The second to fourth column shows the efficiency results of stress, Random and ParaFuzz testing respectively. Each cell represents the average time (in seconds) taken to detect the first

concurrency bug per experimental subject and testing technique over 30 testing runs. '-' in the cell indicates that none of the 30 testing runs detected a concurrency bug within the timeout of 1 hour.

As shown in Table 5.4, for every experimental subject, ParaFuzz took significantly less time (column 4) to find bugs than other techniques. AS discussed in RQ1, ParaFuzz is the only technique to find bugs in all programs except BI and DL. Even in the case of BI, random testing managed to detect bugs only in single testing run out of 30 testing runs. Still, ParaFuzz is 5$\times$ faster than random testing. As DL is not input-dependent and concurrency-heavy, both ParaFuzz and random testing detected the concurrency bug in almost no time.

Note that for BI bug, the average time of random testing does not include testing runs that failed to detect concurrency bug. In addition, due to implementation issue with Multicore OCaml, ParaFuzz could not enable fork-server optimisation of AFL. AFL's fork-server optimization decreases testing time by a factor of at least 5. We have reported the issue to Multicore OCaml team and the issue is being looked at actively. Enabling AFL's fork-server optimization, will further decrease the time ParaFuzz takes to find concurrency bugs. Due to its efficiency, ParaFuzz enables a developer to explore a broader schedule space of the concurrent program than other techniques with the same test time budget. Thereby increasing the chances of finding bug in the same limited test time budget. Thus, these results illustrate that ParaFuzz is efficient in detecting concurrency bugs.

RQ3: ParaFuzz's Efficiency to explore schedules

ParaFuzz is more effective and efficient than random and stress testing in find input-dependent bugs as demonstrated by RQ1 and RQ2. But is ParaFuzz still effective in finding concurrency bugs that are schedule-dependent only compared to other testing techniques? So we conducted an experiment on a parallel program (Figure 5.14) which exposes a *schedule-dependent* concurrency bug only when the program is executed under some thread schedules out of many possible. We gradually increase the state space of the program while recording the bug detection time for each technique. We exclude stress testing, as it was not effective in finding concurrency bugs in RQ1, RQ2 and directly compare ParaFuzz and random testing.

Figure 5.14 shows the test parallel program. The program spawns two threads each executing `write` function `n` number of times. `write` function calls function `f` on each value from 1 to *n*. First and second thread writes each value to atomic variable `x` and `y` respectively. At the end, the program checks whether the value of either of the atomic variables is not equal to half of `n`. Here `n` controls the degree of concurrency in the program. Note that the test program contains more than one buggy thread schedules for which the assertion fails. We increase `n` from 10 to 100 to generate large thread schedule space. As seen, the schedule space is huge and increases exponentially as `n` increases. The total number of thread schedules of the program is given by the following formula

```
let n = 100

let rec write f i max =
    if i <= max then (f i; write f (i+1) max)
    else ()

let test () =
    let x = Atomic.make 0 in
    let y = Atomic.make 0 in
    let d1 = Domain.spawn
    (fun () -> write (Atomic.set x) 1 n) in
    let d2 = Domain.spawn
    (fun () -> write (Atomic.set y) 1 n) in
    let xval = Atomic.get x in
    let yval = Atomic.get y in
    Crowbar.check @@ (xval <> (n/2) || yval <> (n/2));
    Domain.join d1;
    Domain.join d2

let () =
    Crowbar.(add_test ~name:"Large schedule space test"
    [Crowbar.const 1] (fun _ ->
        Parafuzz_lib.run test
  ))
```

Figure 5.14: A program with schedule-dependent concurrency bug

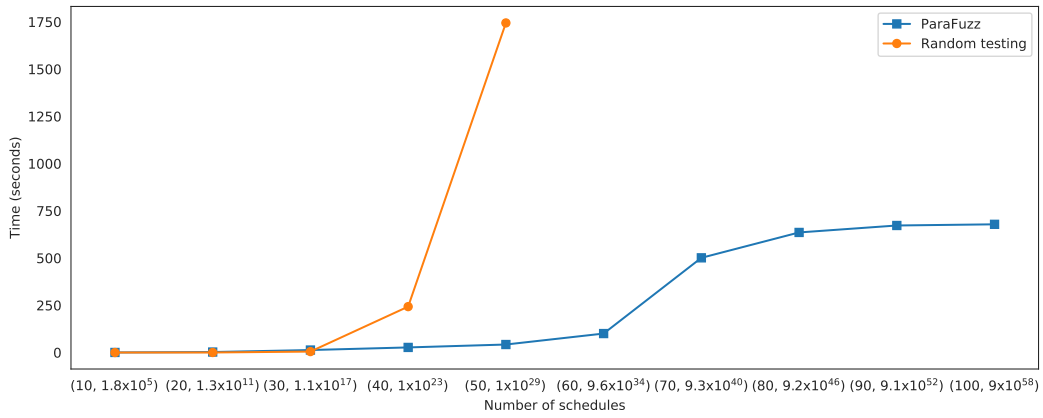


Figure 5.15: Efficiency of ParaFuzz as schedule space increases. The total number of schedules is given by $f(n) = \binom{2n}{n}$. The labels on the x-axis shows $(n, f(n))$.

parameterized over *n*:

$$\textit{Total number of schedules } f(n) = \binom{2n}{n} \tag{5.1}$$

Figure 5.15 graph shows mean time to failure (MMTF) to find the concurrency bug for ParaFuzz and random testing as the schedule space is increased. As seen in the graph, initially random testing is on par with ParaFuzz, but with the degree of concurrency exceeding 30, testing time increases drastically. Meanwhile ParaFuzz's testing time increases linearly as schedule space increases. With the degree of concurrency exceeding 40, random testing is not able to find the bug, in contrast with ParaFuzz which is able to find bugs in a reasonable amount of time. This demonstrates that ParaFuzz is efficient in finding schedule-dependent concurrency bugs even in programs with large thread schedule space. ParaFuzz usage of AFL shines really well when the concurrency bug (like Figure 5.14 bug) requires a sequence of scheduling

decisions to be made in order to expose it. In the case when thread schedule space is small, random testing can be used for concurrency testing. But, for a program with large schedule space or containing input-dependent bug or the concurrency bug does require many sequence of scheduling decisions to be made, ParaFuzz's AFL fuzzing is more effective.

5.7 RELATED WORK

To the best of our knowledge, ParaFuzz is the first tool to apply coverage-guided fuzzing, not just to maximize the coverage of the source code of program, but also to maximize the schedule space coverage introduced by a non-deterministic execution of multi-threaded programs by directly controlling the thread scheduler. In this section, we compare ParaFuzz to related work.

Concurrency fuzzing: The closest work to ParaFuzz is the Heuristic Framework [Liu *et al.* (2018)] to detect concurrency vulnerabilities in multi-threaded programs by AFL. Heuristics Framework does not directly control thread scheduler, instead generates thread schedule by influencing scheduling decisions at runtime by changing thread priorities. Heuristics Framework focuses on finding concurrency bugs involving two concurrent operations. Due to this approach, the Heuristics Framework may fail to find concurrency bugs that require generating a series of scheduling decisions or that involve more than two concurrent operations. Apart from that it cannot reproduce found concurrency bugs deterministically. ParaFuzz does not produce false positives, whereas the Heuristics Framework can generate false positives too. ParaFuzz can expose bugs that depend on some combination of input and thread schedule, while

Heuristics Framework requires input to be fixed for finding input+thread schedule bugs. Another testing tool, Maple [Yu *et al.* (2012)] employs a coverage-driven approach for testing multi-threaded programs. But unlike ParaFuzz, it aims to maximise untested interleavings for *a fixed test input* to increase interleaving coverage. Node.Fz [Davis *et al.* (2017)] is a concurrency fuzzing tool for event-driven JavaScript programs that fuzzes the order of events and callbacks randomly to explore different schedules. Node.Fz can only find bugs that manifest purely as a result of particular scheduling, not as a property of program inputs and cannot be applied to multi-threaded programs. MUZZ [Chen *et al.* (2020)] proposes a new grey-box fuzzing technique that improves over AFL in both multithreading-relevant seed generation and concurrency-vulnerability detection. ParaFuzz uses AFL as a source of randomness and can take advantage of the improvements proposed by MUZZ.

Concurrency testing techniques: Stress and random testing are the common and widely used approaches in software development to test multi-threaded programs. Stress testing simply executes a parallel program for various days/months in the hope to find concurrency bugs. Stress testing is not effective in testing parallel programs as it does not control the thread scheduler and instead executes the program directly under OS load which makes it highly dependent on OS environment. Random testing techniques [Edelstein *et al.* (2003)] improves stress testing by randomizing the thread interleavings in order to exercise different interleavings in different test runs. These random testing techniques differ in the way they randomize thread interleavings. Both random and stress testing does not recognize the tested interleavings and naively executes the same interleavings multiple times. Also, the probability of finding the

buggy interleaving out of huge interleaving state space is very low. ParaFuzz on other hand directly controls the thread scheduler and recognizes the tested interleavings making it more effective than stress and random testing techniques in finding buggy interleavings that can cause concurrency bugs. Systematic exploration techniques such as model checking [Godefroid (1997)] attempt to explore the schedule space of a given program exhaustively to find concurrency bugs. Even with partial order reduction techniques [Flanagan and Godefroid (2005); Godefroid (1995)], the number of thread interleavings to test for a *given input* is still huge. Few heuristics like context and depth bounding [Musuvathi and Qadeer (2007*b*)] have been proposed to further reduce the testing time at the cost of missing potential concurrency errors. CHESS [Musuvathi *et al.* (2019)] is a stateless model checker exploring the schedule space in a systematic manner. Exhaustive state-space exploration is expensive and given a limited test time budget, ParaFuzz explores broader input and schedule space, which is more likely to detect concurrency bugs.

Bug detection tools: Many approaches and tools have been developed to identify concurrency bugs in multi-threaded programs. Bug detection tools are categorised into two types: static and dynamic concurrency bug detection tools. Static concurrency bug detection tools aims to analyze programs statically without executing the concurrent programs. Most of the static bug detection tools produce a lot of false positives, preventing them from being widely used by programmers. Dynamic bug detection tools executes the concurrent programs in various ways to expose concurrency bugs. Dynamic bug detection tools such as FastTrack [Flanagan and Freund (2009*b*)], Eraser [Savage *et al.* (1997)], CalFuzzer [Joshi *et al.* (2009)] aims to detect data races

concurrency bugs in multi-threaded. These tools can complement ParaFuzz by finding data races in concurrent programs, as ParaFuzz requires programs to be data-race free. Another category of dynamic analysis tools such as ConTest [Edelstein *et al.* (2003)], RaceFuzzer [Sen (2008)] and Sherlock [Eslamimehr and Palsberg (2014)] uses random testing to generate varied thread schedules and trigger concurrency bugs. As shown in Section 5.6, ParaFuzz is much more effective and efficient than random testing in finding concurrency bugs in multi-threaded programs.

5.8 CONCLUSION

In this chapter, we have presented, ParaFuzz, a *directed* concurrency fuzzing tool that employs novel concurrency fuzzing technique PBCF for finding concurrency bugs in multi-threaded Multicore OCaml programs. ParaFuzz detected one *previously unknown novel* concurrency bug in parallel programming library: domainslib. Our performance evaluation shows that coverage-guided fuzzing of ParaFuzz is more effective and efficient than the random and stress testing in finding the input-dependent concurrency bugs. ParaFuzz also demonstrates a novel approach using *effect-handlers* in testing multicore programs and libraries without requiring to modify the code under test.

CHAPTER 6

CONCLUDING REMARKS AND FUTURE WORK

Concurrent programming enables simultaneous execution of tasks and I/O operations to reduce the time it takes for an application to process the user request. Non-determinism arising in event-driven concurrent and multi-threaded parallel programs gives rise to concurrency bugs. Non-determinism makes it hard to test concurrent programs as concurrency bugs typically manifest only when the program is executed under a particular buggy thread/event schedule. In this thesis, we present a novel concurrency testing technique (PBCF) and its instantiation in two practical concurrency testing tools ConFuzz and ParaFuzz to test event-driven and multi-threaded concurrent programs respectively. In this chapter, we discuss the future possible extensions of our contributions. The discussion is split based on the contributions : novel concurrency testing technique (PBCF) and two new concurrency testing tools.

6.1 PBCF

Property-based concurrency fuzzing (PBCF) proposes a novel concurrency testing technique called that combines property-based testing with coverage-guided fuzzing applied to concurrent programs. PBCF uses AFL [Zalewski (2021)], the start-of-the-art mutation-based, coverage-guided grey box fuzzer to provide fuzzing capability. We apply PBCF to generate not only inputs that may cause the property to fail, but also

to drive various scheduling decisions in the concurrent program. Our key observation is that we can use AFL's grey box fuzzing capability to direct the search towards new schedules, and thus lead to property failure and detect concurrency bugs. MUZZ is a grey-box fuzzing technique [Chen *et al.* (2020)], that uses thread-aware instrumentation to generate more effective seeds that can exercise buggy schedules. PBCF is compatible with any instrumentation-capable fuzzer. So as a extension of our work, PBCF can be implemented with MUZZ instead of AFL as a backend fuzzing technique to find concurrency bugs.

6.2 CONFUZZ AND PARAFUZZ

We instantiate PBCF technique in two *directed* concurrency bug-finding tool ConFuzz and ParaFuzz for event-driven OCaml and Multicore OCaml programs respectively. Using these tools, programmers specify high-level program properties as assertions in the source code. These tools then identify the input and the schedule that will cause the assertion to fail. Both ConFuzz and ParaFuzz support record and replay feature to *deterministically* reproduce the concurrency bug. ConFuzz and ParaFuzz employ novel approach to let programmers test their concurrent code without any modifications. This combination of features makes ConFuzz and ParaFuzz, a *practical* concurrency tool for programmers wanting to test their concurrent code in reasonable amount of time.

ConFuzz aims to test event-driven OCaml programs written using popular asynchronous I/O library: Lwt [Lwt (2021)]. Our performance evaluation shows that coverage-guided fuzzing of ConFuzz is more effective and efficient than the random

fuzzing tool Node.Fz [Davis *et al.* (2017)] in finding concurrency bugs. ConFuzz can be extended to work with other event-driven languages and frameworks such as JavaScript.

ParaFuzz aims to find concurrency bugs in multi-threaded Multicore OCaml programs. Multicore OCaml is an extension of OCaml language to provide shared-memory parallelism. We present a novel approach to implement a thread scheduler by *effect handlers* [Plotkin and Pretnar (2009)] to control threads without changing the thread API. Experimental evaluation shows that ParaFuzz is more effective and efficient than random and stress testing in finding multi-threaded concurrency bugs. ParaFuzz was able to find 1 previously *unknown concurrency bug* in widely used Multicore OCaml library: domainslib [domainslib (2021)]. Currently, in ParaFuzz, we manually insert context switch points in each thread API. Even though the test program calls the thread API multiple times, AFL recognizing only a single context switch point as it relies on the program location of the context switch point to distinguish branching points in program execution paths. Due to this implementation, AFL may fail to generate some thread schedule. To improve upon this, ParaFuzz can be extended to parse the test program to automatically insert context switch points *before* each thread API call location in the program, instead of within the thread API as done currently. Another extension, is to enable ParaFuzz to work with other *effect-handlers supported* multi-threaded languages.

REFERENCES

1. **Async** (2021). Async: Typeful concurrent programming. URL https://opensource.janestreet.com/async/.

2. **AsynchronousJavaScript** (2021). Asynchronous javascript. URL https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous.

3. **Bielik, P.**, **V. Raychev**, and **M. Vechev** (2015). Scalable race detection for android applications. *SIGPLAN Not.*, **50**(10), 332–348. ISSN 0362-1340. URL https://doi.org/10.1145/2858965.2814303.

4. **Bruening, D.**, Systematic testing of multithreaded java programs. 1999.

5. **Chang, X.**, **W. Dou**, **Y. Gao**, **J. Wang**, **J. Wei**, and **T. Huang**, Detecting atomicity violations for event-driven node.js applications. *In Proceedings of the 41st International Conference on Software Engineering*, ICSE '19. IEEE Press, 2019. URL https://doi.org/10.1109/ICSE.2019.00073.

6. **Chen, H.**, **S. Guo**, **Y. Xue**, **Y. Sui**, **C. Zhang**, **Y. Li**, **H. Wang**, and **Y. Liu**, MUZZ: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs. *In 29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2020. ISBN 978-1-939133-17-5. URL https://www.usenix.org/conference/usenixsecurity20/presentation/chen-hongxu.

7. **Claessen, K.** and **J. Hughes**, Quickcheck: A lightweight tool for random testing of haskell programs. *In Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00. Association for Computing Machinery, New York, NY, USA, 2000. ISBN 1581132026. URL https://doi.org/10.1145/351240.351266.

8. **Danvy, O.** and **A. Filinski**, Abstracting control. *In Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90. Association for Computing Machinery, New York, NY, USA, 1990. ISBN 089791368X. URL https://doi.org/10.1145/91556.91622.

9. **Davis, J.**, **A. Thekumparampil**, and **D. Lee**, Node.fz: Fuzzing the server-side event-driven architecture. *In Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17. Association for Computing Machinery, New York, NY, USA, 2017. ISBN 9781450349383. URL https://doi.org/10.1145/3064176.3064188.

10. **Docker** (2021). Improving docker with unikernels: Introducing hyperkit, vpnkit and datakit. URL https://www.docker.com/blog/docker-unikernels-open-source/.

11. **Dolan, S.** and **M. Preston** (2017). Testing with Crowbar. OCaml Workshop.

12. **Dolan, S.**, **K. Sivaramakrishnan**, and **A. Madhavapeddy** (2018). Bounding data races in space and time. *SIGPLAN Not.*, **53**(4), 242–255. ISSN 0362-1340. URL https://doi.org/10.1145/3296979.3192421.

13. **domainslib** (2021). domainslib - domain-level parallel programming library for multicore ocaml. URL https://github.com/ocaml-multicore/domainslib.

14. **Edelstein, O.**, **E. Farchi**, **E. Goldin**, **Y. Nir-Buchbinder**, **G. Ratsaby**, and **S. Ur** (2003). Framework for testing multi-threaded java programs. *Concurrency and Computation: Practice and Experience*, **15**.

15. **epoll** (2021). epoll: I/o event notification facility. URL http://man7.org/linux/man-pages/man7/epoll.7.html.

16. **Eslamimehr, M.** and **J. Palsberg**, Sherlock: Scalable deadlock detection for concurrent programs. *In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014. Association for Computing Machinery, New York, NY, USA, 2014. ISBN 9781450330565. URL https://doi.org/10.1145/2635868.2635918.

17. **Flanagan, C.** and **S. N. Freund**, Fasttrack: Efficient and precise dynamic race detection. *In Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09. Association for Computing Machinery, New York, NY, USA, 2009*a*. ISBN 9781605583921. URL https://doi.org/10.1145/1542476.1542490.

18. **Flanagan, C.** and **S. N. Freund** (2009*b*). Fasttrack: Efficient and precise dynamic race detection. *SIGPLAN Not.*, **44**(6), 121–133. ISSN 0362-1340. URL https://doi.org/10.1145/1543135.1542490.

19. **Flanagan, C.** and **P. Godefroid** (2005). Dynamic partial-order reduction for model checking software. *SIGPLAN Not.*, **40**(1), 110–121. ISSN 0362-1340. URL https://doi.org/10.1145/1047659.1040315.

20. **Godefroid, P.** (1995). Partial-order methods for the verification of concurrent systems - an approach to the state-explosion problem.

21. **Godefroid, P.**, Model checking for programming languages using verisoft. *In Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97. Association for Computing Machinery, New York, NY, USA, 1997. ISBN 0897918533. URL https://doi.org/10.1145/263699.263717.

22. **Hong, S.**, **Y. Park**, and **M. Kim**, Detecting concurrency errors in client-side java script web applications. *In Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*, ICST '14. IEEE Computer Society, USA, 2014. ISBN 9781479922550. URL https://doi.org/10.1109/ICST.2014.17.

23. **honggfuzz** (2021). honggfuzz - security oriented software fuzzer. URL https://honggfuzz.dev/.

24. **Hsiao, C.-H.**, **J. Yu**, **S. Narayanasamy**, **Z. Kong**, **C. L. Pereira**, **G. A. Pokam**, **P. M. Chen**, and **J. Flinn**, Race detection for event-driven mobile applications. *In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design*

and Implementation, PLDI '14. Association for Computing Machinery, New York, NY, USA, 2014. ISBN 9781450327848. URL https://doi.org/10.1145/2594291.2594330.

25. **IOCP** (2021). I/o completion ports. URL https://docs.microsoft.com/en-us/windows/win32/fileio/i-o-completion-ports.

26. **Joshi, P.**, **M. Naik**, **C.-S. Park**, and **K. Sen**, Calfuzzer: An extensible active testing framework for concurrent programs. *In Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09. Springer-Verlag, Berlin, Heidelberg, 2009. ISBN 9783642026577. URL https://doi.org/10.1007/978-3-642-02658-4_54.

27. **kqueue** (2021). kqueue, kevent – kernel event notification mechanism. URL https://www.freebsd.org/cgi/man.cgi?kqueue.

28. **Leroy, X.** (1990). The ZINC experiment : An Economical Implementation of the ML Language. Technical Report RT-0117, INRIA. URL https://hal.inria.fr/inria-00070049.

29. **Libev** (2021). Libev event loop. URL http://software.schmorp.de/pkg/libev.html.

30. **libFuzzer** (2021). libfuzzer – a library for coverage-guided fuzz testing. URL https://llvm.org/docs/LibFuzzer.html.

31. **libuv** (2021). libuv: Cross-platform asynchronous i/o. URL https://libuv.org/.

32. **Liu, C.**, **D. Zou**, **P. Luo**, **B. B. Zhu**, and **H. Jin**, A heuristic framework to detect concurrency vulnerabilities. *In Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18. Association for Computing Machinery, New York, NY, USA, 2018. ISBN 9781450365697. URL https://doi.org/10.1145/3274694.3274718.

33. **Lwt** (2021). Lwt: Ocaml promises and concurrent i/o. URL https://ocsigen.org/lwt/5.2.0/manual/manual.

34. **Madhavapeddy, A.**, **R. Mortier**, **C. Rotsos**, **D. Scott**, **B. Singh**, **T. Gazagnaire**, **S. Smith**, **S. Hand**, and **J. Crowcroft**, Unikernels: Library operating systems for the cloud. *In Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13. Association for Computing Machinery, New York, NY, USA, 2013. ISBN 9781450318709. URL https://doi.org/10.1145/2451116.2451167.

35. **Maiya, P.**, **A. Kanade**, and **R. Majumdar** (2014). Race detection for android applications. *SIGPLAN Not.*, **49**(6), 316–325. ISSN 0362-1340. URL https://doi.org/10.1145/2666356.2594311.

36. **Miller, B. P.**, **L. Fredriksen**, and **B. So** (1990). An empirical study of the reliability of unix utilities. *Commun. ACM*, **33**(12), 32–44. ISSN 0001-0782. URL https://doi.org/10.1145/96267.96279.

37. **Musuvathi, M.** and **S. Qadeer**, Chess: Systematic stress testing of concurrent software. *In* **G. Puebla** (ed.), *Logic-Based Program Synthesis and Transformation*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007*a*. ISBN 978-3-540-71410-1.

38. **Musuvathi, M.** and **S. Qadeer** (2007*b*). Iterative context bounding for systematic testing of multithreaded programs. *SIGPLAN Not.*, **42**(6), 446–455. ISSN 0362-1340. URL https://doi.org/10.1145/1273442.1250785.

39. **Musuvathi, M.**, **S. Qadeer**, **P. Nainar**, **T. Ball**, **G. Basler**, and **I. Neamtiu**, Finding and reproducing heisenbugs in concurrent programs. *In Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008*, Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008. USENIX Association, 2019. 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008 ; Conference date: 08-12-2008 Through 10-12-2008.

40. **Node.js** (2021). Node.js is a javascript runtime built on chrome's v8 javascript engine. URL https://nodejs.org/en/.

41. **Node.jsEventLoop** (2021). Node.js event loop. URL https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick.

42. **ocaml redis** (2021). Ocaml-redis: Ocaml binding to redis data store. URL https://github.com/0xffea/ocaml-redis.

43. **Padhiyar, S.** and **K. C. Sivaramakrishnan**, Confuzz: Coverage-guided property fuzzing for event-driven programs. *In* **J. F. Morales** and **D. Orchard** (eds.), *Practical Aspects of Declarative Languages*. Springer International Publishing, Cham, 2021. ISBN 978-3-030-67438-0.

44. **Padhye, R.**, **C. Lemieux**, and **K. Sen**, Jqf: Coverage-guided property-based testing in java. *In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019. Association for Computing Machinery, New York, NY, USA, 2019. ISBN 9781450362245. URL https://doi.org/10.1145/3293882.3339002.

45. **Petrov, B.**, **M. Vechev**, **M. Sridharan**, and **J. Dolby**, Race detection for web applications. *In Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12. Association for Computing Machinery, New York, NY, USA, 2012. ISBN 9781450312059. URL https://doi.org/10.1145/2254064.2254095.

46. **Plotkin, G.** and **M. Pretnar**, Handlers of algebraic effects. *In Proceedings of the 18th European Symposium on Programming Languages and Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, ESOP '09. Springer-Verlag, Berlin, Heidelberg, 2009. ISBN 9783642005893. URL https://doi.org/10.1007/978-3-642-00590-9_7.

47. **Poll** (2021). Poll system call. URL http://man7.org/linux/man-pages/man2/poll.2.html.

48. **Raychev, V.**, **M. Vechev**, and **M. Sridharan**, Effective race detection for event-driven programs. *In Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13. Association for Computing Machinery, New York, NY, USA, 2013. ISBN 9781450323741. URL https://doi.org/10.1145/2509136.2509538.

49. **Repository, V.** (2021). Verifit repository of test cases for concurrency testing. URL http://www.fit.vutbr.cz/research/groups/verifit/benchmarks/.

50. **Savage, S.**, **M. Burrows**, **G. Nelson**, **P. Sobalvarro**, and **T. Anderson** (1997). Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, **15**(4), 391–411. ISSN 0734-2071. URL https://doi.org/10.1145/265924.265927.

51. **Select** (2021). Select system call. URL http://man7.org/linux/man-pages/man2/select.2.html.

52. **Sen, K.**, Effective random testing of concurrent programs. *In Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07. Association for Computing Machinery, New York, NY, USA, 2007. ISBN 9781595938824. URL https://doi.org/10.1145/1321631.1321679.

53. **Sen, K.**, Race directed random testing of concurrent programs. *In Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08. Association for Computing Machinery, New York, NY, USA, 2008. ISBN 9781595938602. URL https://doi.org/10.1145/1375581.1375584.

54. **Sivaramakrishnan, K.**, **S. Dolan**, **L. White**, **T. Kelly**, **S. Jaffer**, and **A. Madhavapeddy**, Retrofitting effect handlers onto ocaml. *In Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021. Association for Computing Machinery, New York, NY, USA, 2021. ISBN 9781450383912. URL https://doi.org/10.1145/3453483.3454039.

55. **Sparks, S.**, **S. Embleton**, **R. Cunningham**, and **C. Zou**, Automated vulnerability analysis: Leveraging control flow for evolutionary input crafting. *In Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. 2007.

56. **Stephens, N.**, **J. Grosen**, **C. Salls**, **A. Dutcher**, **R. Wang**, **J. Corbetta**, **Y. Shoshitaishvili**, **C. Kruegel**, and **G. Vigna**, Driller: Augmenting fuzzing through selective symbolic execution. *In NDSS*, volume 16. 2016.

57. **Twisted** (2021). Twisted: an event-driven networking engine written in python. URL https://twistedmatrix.com/trac/.

58. **Visser, W.**, **K. Havelund**, **G. Brat**, **S. Park**, and **F. Lerda** (2003). Model checking programs. *Automated Software Engg.*, **10**(2), 203–232. ISSN 0928-8910. URL https://doi.org/10.1023/A:1022920129859.

59. **Vouillon, J.**, Lwt: A cooperative thread library. *In Proceedings of the 2008 ACM SIGPLAN Workshop on ML*, ML '08. Association for Computing Machinery, New York, NY, USA, 2008. ISBN 9781605580623. URL https://doi.org/10.1145/1411304.1411307.

60. **Wang, J.**, **W. Dou**, **Y. Gao**, **C. Gao**, **F. Qin**, **K. Yin**, and **J. Wei**, A comprehensive study on real world concurrency bugs in node.js. *In Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017. IEEE Press, 2017. ISBN 9781538626849.

61. **Wang, T.**, **T. Wei**, **G. Gu**, and **W. Zou**, Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. *In 2010 IEEE Symposium on Security and Privacy*. 2010.

62. **Yu, J.**, **S. Narayanasamy**, **C. Pereira**, and **G. Pokam**, Maple: A coverage-driven testing tool for multithreaded programs. *In Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12. Association for Computing Machinery, New York, NY, USA, 2012. ISBN 9781450315616. URL https://doi.org/10.1145/2384616.2384651.

63. **Yu, Y.**, **T. Rodeheffer**, and **W. Chen**, Racetrack: Efficient detection of data race conditions via adaptive tracking. *In Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05. Association for Computing Machinery, New York, NY, USA, 2005. ISBN 1595930795. URL https://doi.org/10.1145/1095810.1095832.

64. **Zalewski, M.** (2021). American fuzzy lop. URL https://lcamtuf.coredump.cx/afl.

CURRICULUM VITAE

NAME: Sumit Padhiyar

Educational Qualifications:

2015 Bachelor Of Engineering (B.E.)

Institute : Gujarat Technological University, Ahmedabad
Specialization : Computer Engineering

2021 Master Of Science by Research (M.S)

Institute : Indian Institute Of Technology Madras (IITM), Chennai
Specialization : Computer Science & Engineering

GENERAL TEST COMITTEE

CHAIRPERSON:	Prof. Balaraman Ravindran Professor Dept of Computer Science and Engineering, IIT Madras
GUIDE:	Dr. K. C. Sivaramakrishnan Assistant Professor Dept of Computer Science and Engineering, IIT Madras
MEMBERS:	Prof. V. Krishna Nandivada Professor Dept of Computer Science and Engineering, IIT Madras
	Dr. S P Suresh Associate Professor Chennai Mathematical Institute