Hybrid Symbolic/Concrete Testing

Matthew Law

A thesis submitted for the degree of Bachelor of
Advanced Computing with Honours.
Supervised by: Zac Hatfield-Dodds
The Australian National University

May 2021

Except where otherwise indicated, this report is my own original work.

Matthew Law
30 May 2021

Acknowledgments

Who do you want to thank? I hope it includes your supervisor(s).

Abstract

Property based testing is a testing approach where tests specify properties for possible test inputs, which are used to generate test cases to show bugs in the system under test. Property based tests find more bugs than traditional unit tests, as traditional unit tests only test a subset of test cases which can be generated by property based tests. However, any testing approach involving concrete execution of test cases is never guaranteed to catch all bugs that possibly exist in a program. Symbolic solver-based testing can find bugs that are missed by concrete property based testing and is a possible solution to this limitation. However, solver-based approaches suffer from path explosion and solvers can fail for complex operations and code. To this end a novel approach for hybrid concrete/symbolic testing is proposed. The approach involves running a test twice with a concrete property based testing tool Hypothesis and a symbolic solver-based tool Crosshair. This was implemented by building support for reading Hypothesis tests into Crosshair. This combined testing approach was validated using various benchmarks and real-world bug hunting with open-source projects. TODO summarise findings.

Contents

Acknowledgments **ii**

Abstract **iii**

1 Introduction **1**
1.1 Report Outline 2

2 Literature Review **3**
2.1 Literature Review 3
2.2 Summary 3

3 Design and Implementation **4**
3.1 Converting Strategies for Crosshair 4
3.2 Summary 4

4 Experimental Methodology **5**
4.1 Software platform 5
4.2 Hardware platform 5

5 Results **7**
5.1 Direct Cost 7
5.2 Summary 7

6 Conclusion **9**
6.1 Future Work 9

Bibliography **10**

List of Figures

3.1 Failing Hypothesis Test 4
3.2 Hello world in Java and C. This short caption is centered. 4

5.1 The cost of zero initialization 8

List of Tables

4.1 Processors used in our evaluation. Note that the caption for a table is at the top. Also note that a really long comment that wraps over the line ends up left-justified. 6

Chapter 1

Introduction

Testing is a fundamental aspect of any software development. Bugs in software are unavoidable and can be detrimental for software quality. The disasters of software bugs have been well documented. A report from CISQ found that in 2020 the total cost of poor software quality was US$2.08 trillion. There are also further overheads from coding bug fixes and reconfiguring software when patching bugs. Recently, there has been increasing adoption of CI/CD and automated testing workflows to alleviate some of these overheads. Despite this and the wide availability of open-source software testing tools, most software today is still not sufficiently tested. Part of the problem lies in the way we test software.

Unit testing is the most common method of software testing and involves testing a single software behaviour. A unit test can only show that a program works for the specific conditions outlined in the unit test. Thus, unit tests don't find bugs beyond the behaviour specified in the test and the test inputs used. Consider an example where we are testing a function that sorts a list. The typical approach to unit testing such a function would be to come up with some inputs of a variety of sizes and then specifying test conditions such as the length of the output should not change etc. The bugs which can be found by our unit testing approach are limited to what inputs and conditions we specify. The problem is that humans can never write enough test cases to effectively test our program.

Property based testing is a testing approach which addresses this problem. In property based testing we assert some logical properties that a test should fulfill and then attempt to generate examples to break those properties. The advantage of property based testing over traditional unit testing is that a single property based test gives us a wide variety of test cases whereas unit tests can only test for a single test case. Property based tests also allow us to feed our specifications for software correctness into tests.

In most software enforcing formal correctness is overkill. The purpose of this research is to exploit the benefits of solvers that test for program correctness to extend concrete testing for improved bug finding.

Hypothesis is a prominent property based testing library for python which utilizes concrete execution MacIver et al. [2019]. In hypothesis tests properties are used to generate concrete test cases for testing code. Such tests are effective at catching bugs. However, Hypothesis fails when testing properties which have a low probability of being invalidated. Crosshair is another prominent property based testing library based on symbolic execution. In Crosshair tests a symbolic value is fed into tests and the test attempts to find a counterexample by executing all feasible program paths. Symbolic execution is useful for catching bugs which are program path dependent and wouldn't otherwise be caught by Hypothesis. However, it does not scale well to large systems because the feasible execution paths increase exponentially with increase in program size. A solution to both the limitations of symbolic and concrete PBT is proposed by combining them. This is achieved by building support for running Hypothesis tests into Crosshair.

1.1 Report Outline

How many chapters you have? You may have Chapter , Chapter 3, Chapter 4, Chapter 5, and Chapter 6.

Chapter 2

Literature Review

At the beginning of each chapter, please give the motivation and high-level picture of the chapter. You also have to introduce the sections in the chapter, e.g.

Section 2.1 gives background material necessary in order to read this report.

2.1 Literature Review

Hybrid concrete/symbolic testing was first pioneered by the testing tools DART and CUTE. In these tools symbolic execution is used during a concrete execution of a test case to generate logical conditions for paths encountered but not entered during the concrete execution. These path conditions are then used to generate new concrete test cases which explore further code paths. This aims to address the problem of code paths that have a low probability of executing when generating test cases. The result of this work is higher test coverage is achieved than using concrete testing alone. However, DART and CUTE are limited by their constraint solvers.

2.2 Summary

Summarize what you discussed in this chapter, and introduce the story of next chapter. Readers should roughly understand what your report talks about by only reading words at the beginning and the end (Summary) of each chapter.

Chapter 3

Design and Implementation

Same as the last chapter, give the motivation and the high-level picture of this chapter to readers, and introduce the sections in this chapter.

3.1 Converting Strategies for Crosshair

The initial implementation for concolic testing was achieved by converting strategies into crosshair pre-conditions and symbolic inputs for testing with crosshair. Hypothesis strategies specify properties about test inputs to be generated. Thus, it's fairly straightforward to convert these properties into logical statements which can be evaluated by Python's inbuilt eval function. For example, given a hypothesis test:

```
@given(st.integers(0, 100000))
def test_div_zero(x):
    1 / (x - 13242)
```

Figure 3.1: Failing Hypothesis Test

Figure 3.2: Hello world in Java and C. This short caption is centered.

3.2 Summary

Same as the last chapter, summarize what you discussed in this chapter and be a bridge to next chapter.

Chapter 4

Experimental Methodology

4.1 Software platform

We use Jikes RVM, which we defined in a macro in `macros.tex`. We ran the avrora benchmark, which we're typesetting in sans-serif font to make it clear it's a name.

You can also use inline code, like `a && b`. Notice how, unlike when using the `texttt` command, the `icode` macro also scales the x-height of the monospace font correctly.

4.2 Hardware platform

Table 4.1 shows how to include tables and Figure 3.2 shows how to include codes. Notice how we can also use the cleveref package to insert references like Table 4.1, by writing just \cref{tab:machines}.

Table 4.1: Processors used in our evaluation. Note that the caption for a table is at the top. Also note that a really long comment that wraps over the line ends up left-justified.

Architecture	**Pentium 4**	**Atom D510**	**Sandy Bridge**
Model	P4D 820	Atom D510	Core i7-2600
Technology	90nm	45nm	32nm
Clock	2.8GHz	1.66GHz	3.4GHz
Cores $\times$ SMT	2×2	2×2	4×2
L2 Cache	1MB $\times$ 2	512KB $\times$ 2	256KB $\times$ 4
L3 Cache	none	none	8MB
Memory	1GB DDR2-400	2GB DDR2-800	4GB DDR3-1066

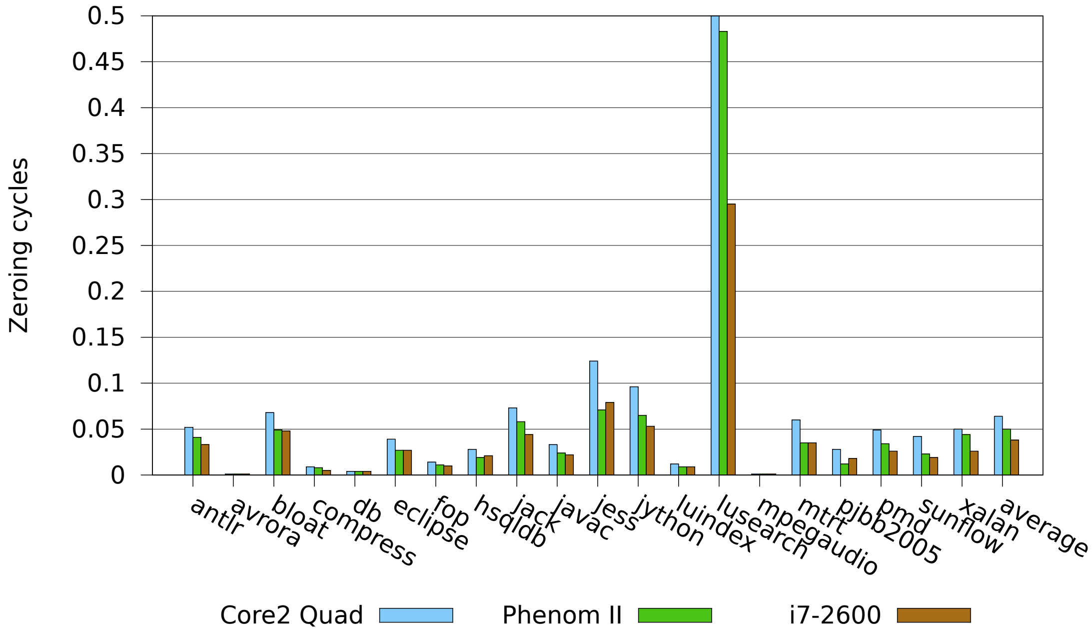
Chapter 5

Results

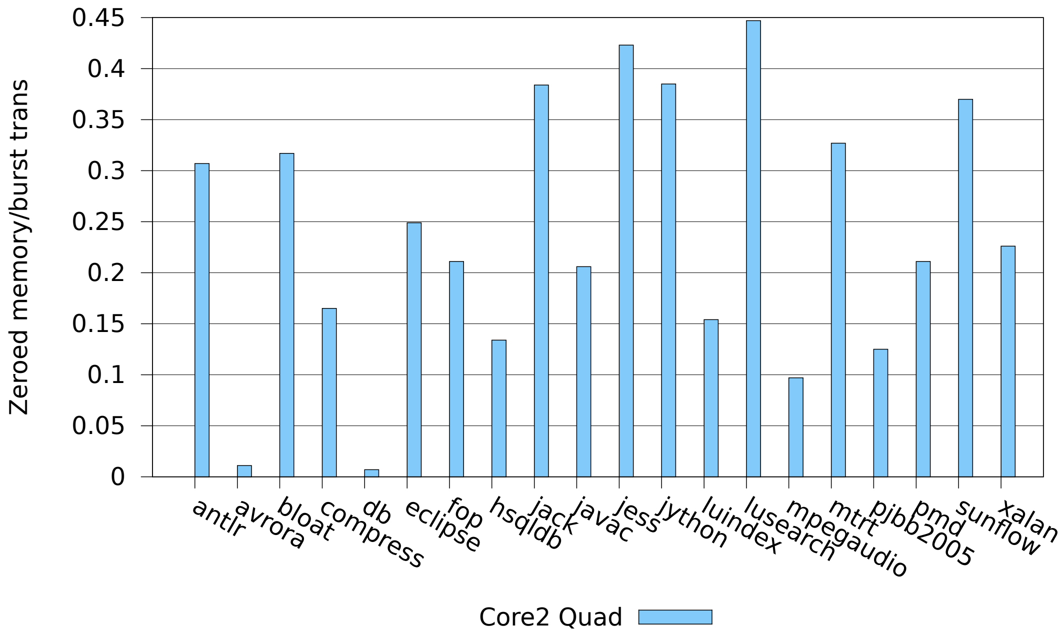
5.1 Direct Cost

Here is the example to show how to include a figure. Figure 5.1 includes two subfigures (Figure 5.1(a), and Figure 5.1(b));

5.2 Summary



(a) Fraction of cycles spent on zeroing



(b) BytesZeroed / BytesBurstTransactionsTransferred

Figure 5.1: The cost of zero initialization

Chapter 6

Conclusion

Summarize your work, state your main findings and discuss what you are going to do in the future in Section 6.1.

6.1 Future Work

Good luck.

Bibliography

MacIver, D. R.; Hatfield-Dodds, Z.; and Contributors, M. O., 2019. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software*, 4, 43 (2019), 1891. doi:10.21105/joss.01891. https://doi.org/10.21105/joss.01891. (cited on page 2)