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

June 2021

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

Matthew Law
1 June 2021

Acknowledgments

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

Abstract

Hybrid testing is an approach to testing where we combine different testing approaches by running a single set of tests with multiple testing backends leading to greater test coverage. Property based testing is a testing approach where tests specify properties for possible test inputs which are used to generate concrete test cases to find bugs in the system under test. Symbolic testing is another approach which involves executing a program symbolically to test all viable execution paths in a program. Despite the apparent advantages of combining concrete property based testing and symbolic testing, hybrid concrete/symbolic testing has seen limited adoption and hasn't yet been applied in python applications.To this end a novel approach for hybrid concrete/symbolic testing in python is proposed. This involves running a test twice with a concrete property based testing tool Hypothesis and a symbolic solver-based tool Crosshair. This is provided by building support for reading Hypothesis tests into Crosshair. The implementation was then validated using various benchmarks and real-world bug hunting with open-source projects. TODO summarise findings.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Testing is a fundamental technique for ensuring software quality and correctness during 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 and specifying these tests manually is tedious and time-consuming. 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 allows us to test a range of test cases whereas unit tests can only test for a single test case. Property based tests find more bugs than traditional unit tests. However, property based tests are still limited by use of concrete execution. Property based tests can never definitively prove that a program is running correctly and therefore is never guaranteed to catch all bugs that possibly exist in a program. In most software development of non-safety critical software strictly enforcing formal correctness is overkill. However, there is a lot to be gained for test coverage and bugs from using symbolic execution based methods for bug finding in software. Symbolic execution can find further bugs by executing code paths that are missed by concrete property based testing. However, symbolic execution suffers from path explosion and solvers can fail for complex operations and code. Symbolic execution is also dependent on the formal specification provided for the program, and bugs in there can be bugs in specifications. Combining concrete execution and symbolic execution aims to address these problems and exploit the benefits of both approaches.

Hypothesis is the leading property based testing library for python MacIver et al. [2019]. In Hypothesis test 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 to provide hybrid symbolic/concrete testing in python. Thus, we can exploit the benefits of the two state-of-the-art testing tools each with years of development without building a new tool.

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

Combining concrete execution and symbolic execution for testing was first pioneered by the concolic 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. DART and CUTE employ a singular testing backend whereas this research aims to explore hybrid testing via combining testing backends.

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

This chapter presents how Crosshair is extended to execute Hypothesis tests for implementing hybrid concrete/symbolic testing in Python.

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:

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

Figure 3.1: Failing Hypothesis Test

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 **??** 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

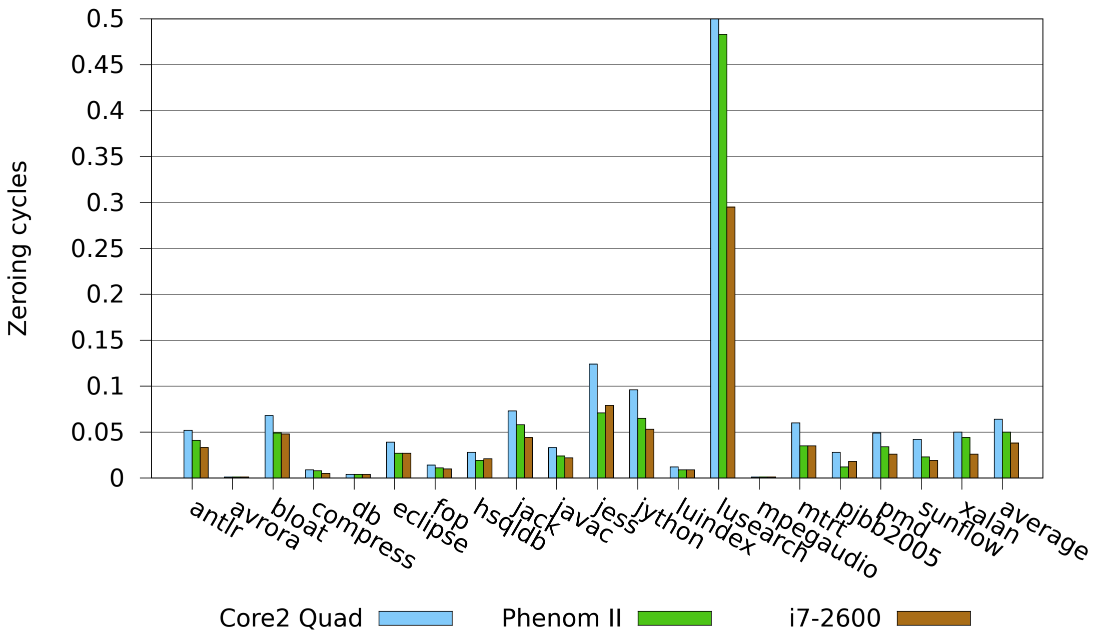
Chapter 5

Results

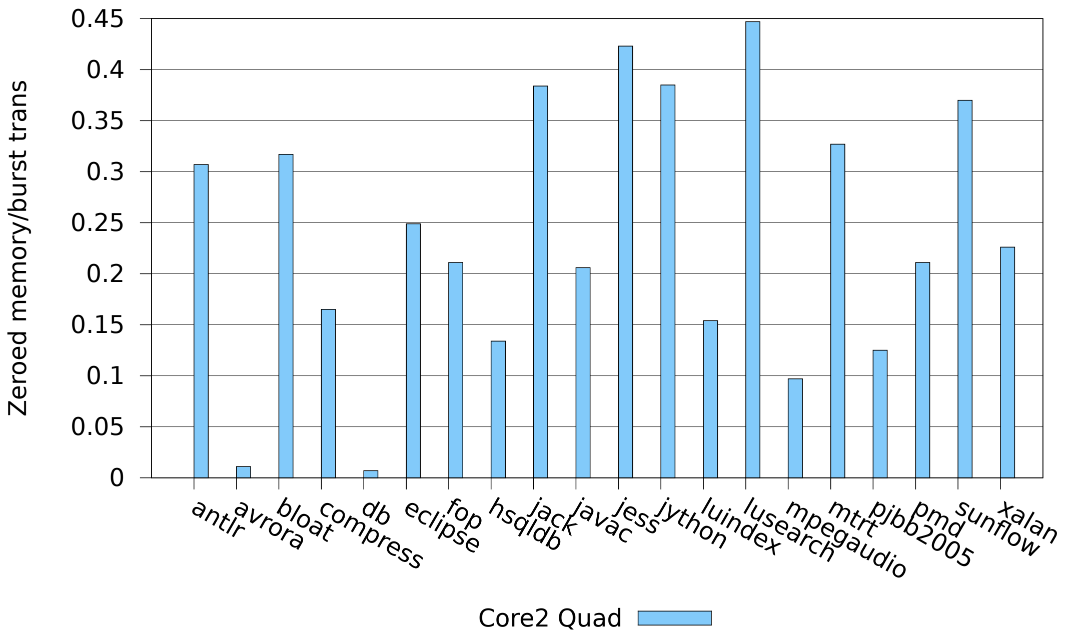
5.1 Direct Cost

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)