IMPROVING REGRESSION TESTING EFFICIENCY AND RELIABILITY VIA TEST-SUITE TRANSFORMATIONS

BY

AUGUST SHI

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2020

Urbana, Illinois

ABSTRACT

As software becomes more important and ubiquitous, high quality software also becomes crucial. Developers constantly make changes to improve software, and they rely on regression testing—the process of running tests after every change—to ensure that changes do not break existing functionality. Regression testing is widely used both in industry and in open source, but it suffers from two main challenges. (1) Regression testing is costly. Developers run a large number of tests in the test suite after every change, and changes happen very frequently. The cost is both in the time developers spend waiting for the tests to finish running so that developers know whether the changes break existing functionality, and in the monetary cost of running the tests on machines. (2) Regression test suites contain flaky tests, which nondeterministically pass or fail when run on the same version of code, regardless of any changes. Flaky test failures can mislead developers into believing that their changes break existing functionality, even though those tests can fail without any changes. Developers will therefore waste time trying to debug non-existent faults in their changes.

This dissertation proposes three lines of work that address these challenges of regression testing through test-suite transformations that modify test suites to make them more efficient or more reliable. Specifically, two lines of work explore how to reduce the cost of regression testing and one line of work explores how to fix existing flaky tests.

First, this dissertation investigates the effectiveness of test-suite reduction (TSR), a traditional test-suite transformation that removes tests deemed redundant with respect to other tests in the test suite based on heuristics. TSR outputs a smaller, reduced test suite to be run in the future. However, TSR risks removing tests that can potentially detect faults in future changes. While TSR was proposed over two decades ago, it was always evaluated using program versions with seeded faults. Such evaluations do not precisely predict the effectiveness of the reduced test suite on the future changes. This dissertation evaluates TSR in a real-world setting using real software evolution with real test failures. The results show that TSR techniques proposed in the past are not as effective as suggested by traditional TSR metrics, and those same metrics do not predict how effective a reduced test suite is in the future. Researchers need to either propose new TSR techniques that produce more effective reduced test suites or better metrics for predicting the effectiveness of reduced test suites.

Second, this dissertation proposes a new transformation to improve regression testing cost when using a modern build system by optimizing the placement of tests, implemented in a technique called TestOptimizer. Modern build systems treat a software project as a group of inter-dependent modules, including test modules that contain only tests. As such, when developers make a change, the build system can use a developer-specified dependency graph among modules to determine which test modules are affected by any changed modules and to run only tests in the affected test modules. However, wasteful test executions are a problem when using build systems this way. Suboptimal placements of tests, where developers may place some tests in a module that has more dependencies than the test actually needs, lead to running more tests than necessary after a change. TestOptimizer analyzes a project and proposes moving tests to reduce the number of test executions that are triggered over time due to developer changes. Evaluation of TestOptimizer on five large proprietary projects at Microsoft shows that the suggested test movements can reduce 21.7 million test executions (17.1%) across all evaluation projects. Developers accepted and intend to implement 84.4% of the reported suggestions.

Third, to make regression testing more reliable, this dissertation proposes iFixFlakies, a framework for fixing a prominent kind of flaky tests: order-dependent tests. Order-dependent tests pass or fail depending on the order in which the tests are run. Intuitively, order-dependent tests fail either because they need another test to set up the state for them to pass, or because some other test pollutes the state before they are run, and the polluted state makes them fail. The key insight behind iFixFlakies is that test suites often already have tests, which we call helpers, that contain the logic for setting/resetting the state needed for order-dependent tests to pass. iFixFlakies searches a test suite for these helpers and then recommends patches for order-dependent tests using code from the helpers. Evaluation of iFixFlakies on 137 truly order-dependent tests from a public dataset shows that 81 of them have helpers, and iFixFlakies can fix all 81. Furthermore, among our GitHub pull requests for 78 of these order-dependent tests (3 of 81 had been already fixed), developers accepted 38; the remaining ones are still pending, and none are rejected so far.

To my family, for their love and support.

ACKNOWLEDGMENTS

It has been a long seven years of graduate school, culminating in this Ph.D. While these seven years were challenging, they were also very rewarding and fun, in large part due to all the wonderful people I met and interacted with. I am extremely grateful for all the support and help from everyone, and I apologize in advance if I do not mention you here due to my own bad memory. Nonetheless, you have my gratitude.

First, I would like to thank my advisor, Professor Darko Marinov, who took a chance on me and accepted me as his Ph.D. student in 2013. Darko taught me everything I know about research, and he was always there pushing me to do my best. Darko has been extremely supportive of everyone in our group, spending countless hours with us discussing ideas, hacking code, writing papers late into the night until the deadline, and just chatting about life in general. Darko brings an amazing amount of excitement and passion to research and to advising students. I hope I can do the same in the future.

I would also like to thank Professors Vikram Adve, Sarfraz Khurshid, and Sasa Misailovic for serving on my thesis committee. They provided excellent feedback on how to improve the presentation of the work in this dissertation. Beyond serving on the committee, I am grateful for all the help and advice they provided with my job search. Although Professor Tao Xie could not be on my thesis committee, I am very grateful that he served on my prelim committee and gave me invaluable advice, both for how to best present my work and for my job search.

One of the fun parts of research is the opportunity to collaborate with others. I have had the pleasure to work with a large number of wonderful people. I would like to thank all my collaborators: Mohammad Amin Alipour, Jonathan Bell, Adhithya Bhaskar, Nikolaj Bjørner, Rutvik Choudhary, Hayes Converse, Jacek Czerwonka, Saikat Dutta, Michael D. Ernst, Vimuth Fernando, Maria Garzaran, Milos Gligoric, Rahul Gopinath, Ruben Gran Tejero, Alex Groce, Alex Gyori, Milica Hadzi-Tanovic, Farah Hariri, Josie Holmes, Aryaman Jain, Yanjie Jiang, Sarfraz Khurshid, Matthew Krafczyk, Shuvendu Lahiri, Wing Lam, Ben Lambeth, Owolabi Legunsen, Yafeng Lu, Suleman Mahmood, Sasa Misailovic, Reed Oei, Qianyang Peng, Victoria Stodden, Suresh Thummalapenta, Ehsan Totoni, Anjiang Wei, Tao Xie, Tifany Yung, Andrey Zaytsev, Lingming Zhang, Peilun Zhang, Sai Zhang, Zhekun Zhang, Peiyuan Zhao, and Chenguang Zhu. There is no way I could have done the work I did without the help of all these people, and I look forward to future collaborations with them.

Lamyaa Eloussi, Alex Gyori, Farah Hariri, and Owolabi Legunsen were my lab mates from the beginning. The five of us started graduate school and joined Darko's research group in the same semester, in Fall 2013. Maybe it was because we started at the same time that we developed a strong sense of camaraderie with each other throughout our years in graduate school. Lamyaa was the sole Master's student among us, which we constantly ribbed her about, but that did not stop her from making incredible accomplishments as a researcher in the two years before she graduated. One of my biggest regrets is that I never managed to publish with Lamyaa. Alex and I worked together almost constantly from the very beginning. The two of us published at least one paper together every year until he graduated, and we certainly brainstormed many more ideas together. Part of the work presented in this dissertation directly came from our collaborations. Farah was my office mate for most of my time in graduate school, and it was always nice to have her around. The arrangement also made it easier for us to work together, and I still remember how we would work late into the night on a number of projects. Finally, Owolabi braved the academic job market the year before I did, and he graciously provided me with an enormous wealth of knowledge about the process. I am forever grateful for all the help Owolabi provided me during my own job search, from sitting with me late into the night reviewing my job application all the way to how to write a thank you email after an interview.

I could not have asked for a better mentor than Milos Gligoric when I started in graduate school. Milos was deeply involved in my first research project and it was thanks to his guidance that I first experienced what it was like to take a research idea all the way to a submitted paper. Indeed, Milos was often like a second advisor whenever Darko was too busy, and I greatly appreciate his mentorship and support.

The Software Engineering Seminar was a great place for everyone in our area to share ideas and discuss research. I found this seminar to be a reliable place where I could get great feedback on everything from conference talks to paper submissions. Thank you to all the people who have joined this seminar throughout the years, making it as great of an experience as it was for me.

Part of the fun of going to the office every day is to chat with the folks around. Angello Astorga became my office mate after Farah graduated. Although we never collaborated on a research project, I really enjoyed all the discussion we had on our respective topics. Wing Lam sat in the office next door, and we ended up collaborating a lot in the last few years, particularly on flaky tests. I recall many afternoons where Wing, Angello, and I would sit in one of our offices and just bounce ideas back and forth the whole time, and that would feel like a good day of work.

I had two wonderful summer internships, at Google and at Microsoft Research. At Google, I worked with my mentors Teresa Johnson and Easwaran Raman on debugging performance problems in the compiler. I learned a lot during that summer about optimizations and performance profiling as well as what software development looks like in a company. At Microsoft Research, I worked with my mentors Shuvendu Lahiri and Suresh Thummalapenta on optimizing test placements for more efficient builds. The work I did that summer forms a part of this dissertation.

The Computer Science Department support staff are the unsung heroes who make our lives so much easier. I would like to thank Elaine Wilson, Jen Dixson, and Kim Baker for all the help they provided, from handling travel reimbursements to reserving conference rooms for events.

Several professors helped me when I was on the academic job market, from commenting on my job talk to conducting mock interviews with me. I would like to thank Professors Sarita Adve, Vikram Adve, Matthew Caesar, Brighten Godfrey, Sarfraz Khurshid, Andrew Miller, Sasa Misailovic, David Padua, Lawrence Rauchwerger, Victoria Stodden, Hari Sundaram, Matus Telgarsky, Josep Torrellas, Gang Wang, Tao Xie, and Tianyin Xu for all their help. Also, once again, the amazing members of the Software Engineering Seminar were there to help as well. Thank you everyone!

Parts of this dissertation were published at the International Symposium on Software Testing and Analysis (ISSTA) 2018 [160] (Chapter 2), the International Conference on Software Engineering (ICSE) 2017 [163] (Chapter 3), and the European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE) 2019 [162] (Chapter 4). I would like to thank the anonymous reviewers of these papers for their comments and feedback that helped improve the presentation of the work. I am also honored that the ICSE 2017 paper on TestOptimizer won an Association for Computing Machinery Special Interest Group on Software Engineering (ACM SIGSOFT) Distinguished Paper Award.

My research was funded by the National Science Foundation, Facebook, Futurewei, Google, Microsoft, and Qualcomm. I was also supported by the Ray Ozzie Computer Science Fellowship and the Mavis Future Faculty Fellowship

Finally, and certainly most importantly, I would like to express my deepest gratitude to my family. I could certainly have never made it this far without their love and support. I am forever grateful for my wife, Hang Yuan, for all the happiness she brings to my life. My brother, Justin Shi, has been my best friend since childhood, and it has always been fun to swap stories about our respective career paths (he is pursuing a medical degree, which I honestly believe is the harder path between the two of us, so more power to him!). Lastly, I thank my parents, Hongchi Shi and Hong Wang, for all their love, care, and encouragement throughout not just my time in graduate school but my entire life.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1
1.1 Thesis Statement 2
1.2 Evaluating Test-Suite Reduction 4
1.3 Optimizing Test Placement 6
1.4 Automatically Fixing Order-Dependent Flaky Tests 7
1.5 Dissertation Organization 9

CHAPTER 2 EVALUATING TEST-SUITE REDUCTION 11
2.1 Background 11
2.2 Example 13
2.3 Failed-Build Detection Loss (FBDL) 15
2.4 Methodology 18
2.5 Results and Analysis 23
2.6 Threats to Validity 32
2.7 Summary 32

CHAPTER 3 OPTIMIZING TEST PLACEMENT 34
3.1 Background 34
3.2 Problem Statement 37
3.3 TestOptimizer 40
3.4 Implementation 44
3.5 Evaluation 46
3.6 Threats to Validity 55
3.7 Summary 55

CHAPTER 4 AUTOMATICALLY FIXING ORDER-DEPENDENT FLAKY TESTS . . . 57
4.1 Formalization of Order-Dependent Tests 57
4.2 iFixFlakies 61
4.3 Evaluation Setup 68
4.4 Evaluation 69
4.5 Threats to Validity 77
4.6 Summary 78

CHAPTER 5 RELATED WORK 79
5.1 Regression Testing Techniques 79
5.2 Continuous Integration (CI) 82
5.3 Flaky Tests 83
5.4 Mutation Testing 84

CHAPTER 6 CONCLUSIONS AND FUTURE WORK 86
6.1 Future Work 86

REFERENCES 88

CHAPTER 1: INTRODUCTION

Software has become an integral part of our daily lives. For example, we rely on the software in the apps on our smart phones, the navigation systems in our vehicles, the medical devices that save people's lives, and much more. As software becomes more important and ubiquitous, high quality software also becomes crucial. We depend on software developers who write the software to also maintain and improve its quality. When developers make changes to software, they rely on *continuous integration* and *regression testing* to check that changes do not break existing functionality.

Continuous integration (CI) automates the process of building and testing software after every change [100, 101]. Once a developer pushes a change to a version-control system, that push triggers a CI *build*, which checks out the changed version of the code from the version-control system onto a remote server, e.g., a server in the cloud provided by Travis CI, a popular cloud-based CI platform [23]. The build involves both compiling the changed code and running tests. The process of running tests on the code after every change is known as regression testing [183]. If the tests fail after a change, yet they passed in the previous build, then the tests have ideally detected a fault introduced by the change, and the developers can proceed to debug the change and fix that fault. The process repeats as developers trigger more builds after their changes to fix the fault, until all tests pass and developers can release the changed code. The goal of regression testing is to allow developers to detect and fix faults early on, ideally the moment the faults are introduced. Prior work found that detecting faults early is also critical to addressing them [91, 153, 154]. Regression testing is widely used in both industry and open source, but regression testing suffers from two main challenges: (1) regression testing is costly, and (2) regression test suites often contain flaky tests.

Regression testing is costly because many tests run on every code change, and changes occur very frequently [134]. As such, a lot of human developer time can be spent waiting for test results before they can decide what next steps to take. In addition to time, there is also a monetary cost in utilizing machines to run the tests after every change [98]. To illustrate how much companies spend on testing, Facebook reported that they make around 60K code changes per day [11], and they run on the order of 10K tests per change [130]. Google also reported high costs of testing, running on the order of 150 million tests per day [135].

Regression testing also suffers from the presence of flaky tests in the regression test suites. A *flaky test* is a test that can nondeterministically pass or fail when run on the same code, without any changes [128]. As such, flaky test failures can send misleading signals to developers concerning the effects of their recent changes [91, 128], because failures do not necessarily indicate a fault introduced by the change. Developers will waste time trying to debug a fault unrelated to recent changes [50]. Flaky tests are also quite prevalent. For example, Labuschagne et al. [115] found that 13% of the failed builds they studied in open-source projects using Travis CI are due to flaky tests. The software industry also widely reports major problems with flaky tests [76, 91, 99, 134, 193], e.g., Luo et al. [128] reported that at one point 73K of 1.6M test failures per day at Google were due to flaky tests.

This dissertation tackles these two challenges of high cost and unreliability of regression testing through a common theme, which we call *test-suite transformations*. A test suite is a set of tests in a project, and our test-suite transformations modify the code within the test suite to make it run faster or more reliably. This dissertation presents three kinds of test-suite transformations that can improve efficiency and reliability of regression testing.

Specifically, this dissertation investigates the effectiveness of test-suite reduction (TSR), a traditional transformation that removes likely redundant tests in the test suite to speed up regression testing [92, 151, 176, 183]. While TSR was proposed in the past, we investigate TSR in a more realistic setting than evaluated in prior work, where we use real-world software and real test failures. TSR is inherently unsafe, i.e., it may remove tests that are not redundant with respect to future faults and therefore compromise the fault-detection effectiveness of the test suite on future builds. Instead, we propose a novel, safe test-suite transformation that involves moving around tests in the test suite. The insight is that we can take advantage of a modern, module-based build system, which includes developer-specified information concerning the relationship between tests and code, to run fewer tests after every change. However, we need to optimize the placement of tests within modules to reduce the number of wasteful test executions in future builds. Finally, we propose a technique to automatically fix a prominent kind of flaky tests, namely order-dependent tests, by modifying existing test code using logic that already exists in the test suite.

1.1 THESIS STATEMENT

The thesis statement of this dissertation is as follows:

Test-suite transformations can improve the efficiency and reliability of regression testing.

This dissertation makes the following contributions:

- This dissertation evaluates the effectiveness of test-suite reduction (TSR), which transforms a test suite for faster regression testing in the future by removing redundant tests in the test suite, creating a smaller reduced test suite to be run on future builds. However, TSR risks removing tests that can help detect faults in the future, so a developer needs some way to *predict* how well the reduced test suite will detect faults in the future compared to the original test suite. While TSR was proposed over two decades ago, different TSR techniques were evaluated using program versions with seeded faults, but such evaluations do not precisely predict the effectiveness of the reduced test suite in future builds. This dissertation evaluates TSR using real test failures in (failed) builds that occurred on real code changes. We analyze 1,478 failed builds from 32 GitHub projects that run their tests on Travis CI. Each failed build can have multiple faults, so we propose a family of mappings from test failures to faults. We use these mappings to compute *Failed-Build Detection Loss* (FBDL), the percentage of failed builds where the reduced test suite misses to detect all the faults detected by the original test suite. We find that FBDL can be up to 52.2%, which is higher than suggested by traditional TSR metrics. Moreover, traditional TSR metrics are not good predictors of FBDL, making it difficult for developers to decide whether to use a reduced test suite or not. Researchers need to either propose new TSR techniques that produce more effective reduced test suites or better metrics for predicting the effectiveness of reduced test suites.

- As an alternative to TSR (which can be unsafe due to removing tests that can help detect faults in the future), this dissertation proposes a different test-suite transformation involving the placement of tests by leveraging the underlying build system. Modern build systems help increase developer productivity by performing incremental building and testing. These build systems treat a software project as a group of inter-dependent modules and perform regression test selection at the module level [74, 150, 183]. However, many large software projects have imprecise dependency graphs that lead to wasteful test executions. If tests are placed in a module that has more dependencies than those the tests actually depend on, then the build system can unnecessarily execute those tests even when none of their actual dependencies are affected by a change. In this dissertation, we formulate the problem of wasteful test executions due to suboptimal placement of tests in modules. We propose a greedy algorithm to reduce the number of test executions by suggesting test movements while considering historical build information and the actual dependencies of tests. We implement our technique, called *TestOptimizer*, on top of CloudBuild, a build system developed at Microsoft. We evaluate TestOptimizer on five large proprietary Microsoft projects. Our results show that the suggested test movements can lead to a reduction of 21.7 million test executions (17.1%) in these evaluation projects. We received encouraging feedback from the developers of these projects; they accepted and intend to implement 84.4% of our reported suggestions.

- Concerning flaky tests, this dissertation explores automatically transforming tests within the test suite to fix a specific kind of flaky tests, namely order-dependent tests. Order-dependent tests pass or fail depending on the order in which tests are run. We present *iFixFlakies*, a framework for automatically fixing order-dependent tests. The key insight in iFixFlakies is that test suites often already have tests, which we call *helpers*, whose logic resets or sets the states for order-dependent tests to pass. iFixFlakies automatically searches a test suite for helpers that make the order-dependent tests pass and then recommends patches for the order-dependent tests using code from the helpers. Our evaluation on 137 truly order-dependent tests from a public dataset shows that 81 of them have helpers, and iFixFlakies can fix all 81. We opened GitHub pull requests for 78 order-dependent tests (3 of 81 had been already fixed), and developers accepted our pull requests for 38 of them, with all the remaining ones still pending and none rejected so far.

The rest of this chapter describes these contributions in more detail.

1.2 EVALUATING TEST-SUITE REDUCTION

Researchers have proposed TSR as a way to speed up regression testing [53, 59, 61, 86, 92, 104, 125, 151, 183, 188, 191]. TSR identifies a subset of tests in the test suite that are redundant with respect to the other tests. The redundant tests are then removed from the test suite. The remaining, non-redundant tests form a *reduced test suite*, which is run instead of the original test suite for future builds [151]. Running the smaller, reduced test suite instead of the original test suite leads to faster regression testing in the future. However, there is a risk in using that reduced test suite. TSR techniques determine redundancy based on heuristics, e.g., a test that covers the same statements as another test is considered redundant with that other test and therefore should be removed. These heuristics may not completely capture the true fault-detection capability of a test, and a developer using TSR techniques runs the risk of removing a useful test that can detect faults in future builds. In general, a reduced test suite should ideally detect *all* faults that the original, non-reduced test suite detects in future builds, so developers can debug and fix all those faults before moving on to making more changes. However, because TSR techniques cannot guarantee that the reduced test suite achieves this ideal (the heuristics only operate on the current test suite and do not "know" what will happen in the future), TSR is *unsafe* as it can lead to missing faults in the future. Before using a TSR technique, the developer must decide whether the benefit from using a reduced test suite is worth the cost of missed future faults [151].

Since over two decades ago, when Wong et al. [177] and Rothermel et al. [151] evaluated the loss in fault-detection capability of TSR using program versions with seeded faults, all evaluations of TSR [183] (1) used seeded faults or mutants, (2) used only one fault seeded for each faulty version, and (3) ignored new tests added to the test suite during software evolution. It is unknown how TSR performs for real software evolution. Developers have no practical guidance to make cost-benefit decisions about TSR, and researchers have no real data for improving TSR.

This dissertation presents the first study of TSR on real software evolution. Our study (1) uses real test failures from the Travis CI builds of open-source projects, (2) considers multiple faults per failed build, and (3) accounts for new tests added during software evolution. More precisely, we use the following experimental procedure to evaluate the cost and benefit of TSR. First, we collect historical failed build logs for several open-source projects and determine which tests failed in which build. We define a metric *Failed-Build Detection Loss* (FBDL) as the percentage of failed builds where the reduced test suite misses to detect *all* the faults that the original test suite detects. A *lower* FBDL means that a reduced test suite is *better*. For a reduced tests suite computed at an earlier reduction point, we then measure FBDL using the future failed builds relative to that point. Since we have only failures from historical build logs, we also need to map test failures to faults detected, which can range from each test failure detecting a unique fault to all test failures detecting the same single fault [151]. Prior studies obtained the failure-to-fault mapping precisely because they created the faulty versions themselves. We further propose *Failure-to-Fault Map* (FFMap), a family of mappings from test failures to faults.

Given that the goal of creating a reduced test suite is to use it for *future* builds, a developer needs to *predict* a reduced test suite's FBDL, so we also measure how well metrics collected at the reduction point can predict the FBDL in future builds. We evaluate two traditional TSR metrics [183] as predictors, size reduction and test-requirements loss. Past studies measured test-requirements loss at the reduction point *implicitly* as a proxy for TSR effectiveness in the future. We are the first to *explicitly* evaluate these metrics to predict the quality of a reduced test suite for future failed builds.

In this dissertation, we evaluate (1) the FBDL of TSR in real-world software evolution and (2) the power of various TSR metrics to predict the FBDL of TSR. An increasing number of open-source projects on GitHub [28], the most popular hosting platform for open-source projects [43, 52, 54, 55], utilize Travis CI [23] to run tests. The usage of Travis CI [101] allows us to create a dataset of historical build logs with test failures [51], similar to what some companies might have [65, 168]. We analyze 1,478 failed builds (from a total of 27,461 builds) from Travis CI for 32 GitHub projects that are written in Java and built using Maven [35]. The results show that FBDL is quite high, up to 52.2%, using the most "pessimistic" mapping of failures to faults from FFMap. Moreover, we find that the traditional metrics used to evaluate TSR are *not* good predictors of FBDL; the low correlation between these metrics and FBDL suggests that a developer cannot trust these metrics to predict FBDL. We propose a new predictor, historical FBDL computed on historical failed build logs to predict the FBDL of future failed builds. While the historical FBDL is a better predictor than the other predictors are, it is still not a good predictor.

In summary, our results confirm important concerns about TSR [151, 152, 183]. *Developers* need to exercise great caution when deciding to use TSR, because the FBDL of reduced test suites can be high, and the available predictors of FBDL are not very effective. While our proposed historical FBDL predictor performs better than the others do, it is still not highly reliable. *Researchers* considering TSR need to develop (1) TSR techniques that have lower FBDL, (2) TSR techniques that result in reduced test suites that are more predictable, and/or (3) new predictors that can more reliably predict the FBDL of the reduced test suites in future builds.

1.3 OPTIMIZING TEST PLACEMENT

Given that traditional TSR transforms the test suite (by removing tests) in an unsafe manner that can potentially miss to detect faults in future builds, we explore a means to transform the test suite through better placement of tests. In particular, we take advantage of existing infrastructure developers are already using as part of the development process. Large-scale software development projects use *build systems* to manage the process of building source code, applying static analyzers, and executing tests as part of their CI process. Major companies, such as Facebook, Google, and Microsoft, have made huge investments in developing efficient, incremental, parallel, and distributed build systems, such as Buck [36], Bazel [9], and CloudBuild [67], respectively. These build systems treat a software project as a group of inter-dependent modules and inherently perform (safe) regression test selection [74, 150, 183] at the module level. Given a change, they can use the build dependency graph to identify test modules (modules with only test code) that are affected by the change and run only tests in those modules.

Despite the increasing sophistication of build systems, large software projects with many modules often have dependency graphs that make module-level test selection less efficient by executing tests that are not affected by a change. This dissertation highlights a source of inefficiency in the form of wasteful test executions due to test placement in test modules that have more dependencies than the tests actually need. Therefore, tests that are not affected by a change often are executed due to changes in the developer-specified dependencies of the module that contains the tests, even though such changes cannot alter the behavior of these tests. From our experience with CloudBuild, the build system in use at Microsoft, the common reasons for such suboptimal placement of tests include a lack of comprehensive knowledge of all test modules in the project and developers making large refactorings to the code base. The goal of our work is to provide a practical solution to reduce the number of wasteful test executions under the constraints imposed by the underlying build systems.

In this dissertation, we focus on reducing wasteful test executions due to suboptimal placement of tests within test modules. An ideal placement of tests to avoid wasteful test executions is to place each test in a test module that shares exactly the same set of dependencies that are exercised by the test during its execution. This placement ensures that only relevant tests are executed for a given change. However, in large projects, moving thousands of tests to achieve the ideal placement requires huge effort. Given that developers are always under pressure to meet customer requirements, they often have limited time to perform these activities. Therefore, it is not practical to invest huge one-time effort to achieve the ideal placement. Furthermore, introducing a large number of modules can significantly increase the time to build all modules in the project. Our technique addresses these issues by suggesting movements that reduce the number of test executions while minimizing the number of suggestions to give to developers.

Our technique, called *TestOptimizer*, accepts the following inputs: dependency graph, actual dependencies for tests, and number of times modules have been built over a given range of time. TestOptimizer formulates the problem as a decision problem, asking if there is a way to split test modules to reduce the overall number of wasteful test executions. TestOptimizer also uses a heuristic, called *affinity*, which is the set of dependencies the tests in a test module should be testing. Affinity helps determine the tests that need not be moved, i.e., tests that execute the affinity of its test module are not amenable to movement. TestOptimizer uses a greedy algorithm to iteratively suggest test movements that lead to reductions in the number of test executions. These suggestions involve moving tests into a newly created test module or to an existing test module that shares the exact same dependencies as the tests. Furthermore, TestOptimizer suggests the test movements ranked in the order of highest reduction in the number of test executions first. We envision that developers can use TestOptimizer once to get the correct placement of all tests. After developers implement the suggestions, they can run TestOptimizer on only added tests.

We implement TestOptimizer in a tool on top of CloudBuild. To evaluate TestOptimizer, we apply it on five large proprietary projects in Microsoft. Our results show that the suggested movements can result in a reduction of 21.7 million test executions (17.1%) across all our evaluation projects. We received positive feedback from the developers of these projects at Microsoft, who also accepted and intend to implement 84.4% of our suggestions.

1.4 AUTOMATICALLY FIXING ORDER-DEPENDENT FLAKY TESTS

An important kind of flaky tests are *order-dependent* tests, which pass or fail based solely on the order of the sequence in which the tests run [118, 190]. Each order-dependent test has at least one *test order* (a sequence of tests in the test suite) where the order-dependent test passes, and at least one other different test order where the order-dependent test fails; if the two test orders do not differ, the test is not flaky solely due to the ordering. Prior work [128] showed that order-dependent tests are among the top three most common kinds of flaky tests. As an example, a widely reported case happened when Java projects updated from Java 6 to Java 7. Java 7 changed the implementation of reflection, which JUnit uses to determine the test order in which to run tests. Many tests failed due to the tests being run in a different test order from before, requiring developers to manually fix their test suites [1, 3, 114]. Prior work developed automated techniques for detecting order-dependent tests in test suites [69, 118, 190]. We recently released a dataset of flaky tests, where about half are order-dependent [29, 118].

In this dissertation, we describe a framework, *iFixFlakies*, which can automatically fix many order-dependent tests. The key insight is that test suites often (but not always) already have tests, which we call *helpers*, whose logic (re)sets the state required for order-dependent tests to pass. We first identify that an order-dependent test can be classified into one of two types based on the result of running the test in isolation from the other tests. One type is a *victim*, an order-dependent test that passes when run in isolation but fails when run with some other tests. The other type is a *brittle*[1], an order-dependent test that fails when run in isolation but passes when run with some other test(s).

Given these types of order-dependent tests, our insight for iFixFlakies is that running some helper(s) directly before victims and brittles makes these order-dependent tests pass. Therefore, we can use the code from these helpers to fix order-dependent tests so that they pass even if helpers are not run (directly) before the order-dependent tests. iFixFlakies searches for helpers and, when it can find them, uses them to automatically recommend patches for order-dependent tests. As inputs, iFixFlakies takes an order-dependent test, a test order where the test passes, and a test order where the test fails. iFixFlakies outputs a patch that can be applied to the order-dependent test to make it pass even when run in the test order where it was failing before. The code in the patch comes from a helper. Although simply using all the code from the helper can create a patch to make the order-dependent test pass, such a patch would be complex and undesirable, because helpers typically contain many statements irrelevant to why the tests are order-dependent. iFixFlakies produces effective patches by applying delta debugging [184] on the helpers to produce the minimal patch for order-dependent tests.

We evaluate iFixFlakies on all 137 truly order-dependent tests from a public dataset [118] that includes the order-dependent tests and their corresponding passing and failing test orders[2]. We find that 120 tests are victims and 17 are brittles. We also find that 81 of these 137 order-dependent tests have helpers, allowing iFixFlakies to propose patches for all 81 of these tests (64 victims and 17 brittles). These patches have, on average, only 30.2% of the statements of the original helper, and

[1]The word "brittle" is commonly used as an adjective but can also be used as a noun.

[2]This dissertation describes results for additional order-dependent tests not described in our original paper on iFixFlakies [162].

65.1% of these patches consist of only *one statement*. The overall time that iFixFlakies takes to find the first helper and to produce a patch using that helper is only 207 seconds on average. When an order-dependent test has no helper, iFixFlakies takes 325 seconds on average to determine that it cannot produce a patch. These timing results show that iFixFlakies is efficient.

We opened GitHub pull requests for 78 order-dependent tests with helpers (3 of 81 had been already fixed in the latest version of the code). While all patches generated by iFixFlakies semantically fixed the flaky test, not all patches were syntactically the most appropriate, e.g., matched the formatting style for the project. For 33 tests, we created the pull requests using exactly the patch recommended by iFixFlakies, while the remaining 45 involved some manual changes, mostly refactorings to make the code more similar to the style of the project. Developers accepted our pull requests fixing 38 order-dependent tests; the pull requests for the remaining 40 order-dependent tests are still under consideration, but none have been rejected so far. We describe the pull requests in more detail on our website [30].

1.5 DISSERTATION ORGANIZATION

The rest of this dissertation is organized as follows:

Chapter 2: Evaluating Test-Suite Reduction

This chapter presents our work on evaluating traditional test-suite reduction techniques on real-world, open-source projects with real software evolution and test failures [160].

Chapter 3: Optimizing Test Placement

This chapter presents our technique TestOptimizer, which suggests test placements for reducing the number of test executions in module-level regression testing [163].

Chapter 4: Automatically Fixing Order-Dependent Flaky Tests

This chapter presents our technique iFixFlakies, which automatically proposes fixes for order-dependent flaky tests, such that these tests do not fail regardless of the order in which the tests are run [162].

Chapter 5: Related Work

This chapter presents an overview of other research related to the topics presented in this dissertation.

Chapter 6: Conclusions and Future Work

This chapter concludes the dissertation and describes future work that can extend the work presented already in this dissertation.

CHAPTER 2: EVALUATING TEST-SUITE REDUCTION

This chapter presents our evaluation of test-suite reduction (TSR) during real software evolution using real test failures. Section 2.1 provides some background on TSR. Section 2.2 shows an illustrative example of how we evaluate TSR given real software evolution. Section 2.3 presents the new metric that we introduce for evaluating the effectiveness of TSR. Section 2.4 describes the methodology for our evaluation. Section 2.5 presents the results. Section 2.6 presents threats to validity. Finally, Section 2.7 concludes and summarizes the chapter.

2.1 BACKGROUND

We first describe traditional TSR and commonly used TSR algorithms. The goal of TSR is to take an existing original test suite (a set of tests that developers run on their code under test) and create an ideally smaller, reduced test suite from that original test suite.

Definition 2.1. Traditionally [183], a TSR algorithm Algo takes two inputs: (1) a function ρ that returns the set of satisfied test-requirements for a given test suite and (2) the original test suite $\mathcal{O}$ to be reduced. It returns a reduced test suite $\mathcal{R} \subseteq \mathcal{O}$ that satisfies the same test-requirements as the original test suite:

$$\mathsf{Algo}(\rho, \mathcal{O}) = \mathcal{R}, \text{such that } \mathcal{R} \subseteq \mathcal{O} \wedge \rho(\mathcal{O}) = \rho(\mathcal{R}) \tag{2.1}$$

2.1.1 Test-requirements

Code coverage is widely used for measuring the quality of test suites. As such, tests may be considered redundant w.r.t. the coverage each test achieves, e.g., two tests that cover the same statements can be considered redundant w.r.t. statement coverage. We use *TSR criterion* to refer to the type of test-requirements used to guide TSR in creating the reduced test suite, i.e., what the function ρ is. In the remaining text, we use the term *TSR technique* to refer to an algorithm instantiated with a specific TSR criterion.

Previous research on TSR has often used statement coverage as the TSR criterion [53, 60, 61, 86, 188], Other TSR criteria were also used, e.g., block coverage [176, 177], branch coverage [104, 151], and def-use coverage [92, 104]. For such techniques that use code coverage as the TSR criterion, we say they use the function cov in place of ρ, where cov returns the set of covered elements by the test suite. Later in this chapter for our evaluation, we use statement/line coverage for guiding TSR (i.e., the set of elements returned by cov are lines covered).

2.1.2 TSR Algorithms

The most popular traditional TSR algorithms create reduced test suites that satisfy the same test-requirements as the original test suite, i.e., $\rho(\mathcal{O}) = \rho(\mathcal{R})$. We thus call them *adequate* TSR algorithms. Some of our prior work has also evaluated inadequate TSR algorithms [158], but in this dissertation we focus on just the adequate TSR algorithms.

The most widely used algorithms for TSR include Greedy [61, 108], GRE [61], and HGS [92]; their details are available elsewhere, e.g., in the regression-testing survey [183]. Conceptually, they all work by starting with an empty test suite and iteratively adding tests from the original test suite until all test-requirements are satisfied, resulting in the reduced test suite.

2.1.3 Evaluating TSR Algorithms

Studies [53, 86, 104, 109, 125, 151, 152, 176, 177, 188] that evaluated TSR algorithms used mainly two metrics—*size of the reduced test-suite relative to original test suite* and *loss in fault-detection capability*—to measure the quality of the reduced test suites on the *one software version* on which the reduction is performed, which we call the *reduction point*. For both these metrics, the smaller the better. The size reduction is measured as the ratio of tests kept in the reduced test suite from the original test suite over the number of tests in the original test suite:

$$\mathsf{SizeKept} = |\mathcal{R}|/|\mathcal{O}| \times 100\% \tag{2.2}$$

The loss in fault-detection capability is measured as the ratio of the number of faults missed by the reduced test suite over the number of faults detected by the original test suite. However, due to the challenges in collecting a large number of known real faults per project, researchers most commonly evaluate through a proxy of faults, typically by some other test-requirements that approximate fault-detection capability (but not the TSR criterion used to create the reduced test suite, as that would by definition lead to no loss).

$$\mathsf{ReqLoss} = (|\rho'(\mathcal{O})| - |\rho'(\mathcal{R})|)/|\rho'(\mathcal{O})| \times 100\% \tag{2.3}$$

where ρ' is a function that returns the set of satisfied test-requirements different from the TSR criterion used to create the reduced test suite $\mathcal{R}$.

As prior studies on TSR would commonly use code coverage as the TSR criterion, they would in turn evaluate $\mathsf{ReqLoss}$ based on mutants from mutation testing. Mutation testing [46, 62, 85, 86, 89, 105, 187, 188, 189] systematically inserts syntactic changes, called *mutants*, in code and measures how many of these mutants are killed by a given test suite; a mutant is considered *killed* if at least one of the tests fails (and the same test passes in the non-mutated run). The quality of a test suite is measured as the ratio of the number of killed mutants over the total number of systematically inserted mutants; this ratio is called the *mutation score*. Previous research on TSR measured the effectiveness of TSR algorithms by comparing the mutation score of the reduced test suite to the mutation score of the original test suite [86, 188], i.e., use function mut that returns set of mutants killed by the test suite in place of ρ' in the equation for $\mathsf{ReqLoss}$. When $\mathsf{ReqLoss}$ uses mut, we call it $\mathsf{MutLoss}$.

Interestingly, previous studies on TSR reported conflicting findings in terms of the loss of fault-detection capability measured through loss in mutants killed: reduced test suites sometimes had low loss [158, 176, 177, 188] and sometimes had high loss [86, 104, 125, 151, 152].

Note that mutants can also be used as the TSR criterion to create the reduced test suite [158]. If so, then the reduced test suite cannot be evaluated using mutants again, but another test-requirement must be used. For mutant-based TSR, we can then use code coverage to measure any loss in fault-detection capability, i.e., use function cov as ρ' for computing $\mathsf{ReqLoss}$. When $\mathsf{ReqLoss}$ uses cov, we call it $\mathsf{CovLoss}$.

2.2 EXAMPLE

We present one example that illustrates how developers would apply TSR in their project and how to evaluate the effectiveness of the reduced test suite. Consider the `caelum/vraptor4` GitHub project, "A web MVC action-based framework [...] for fast and maintainable Java development" [25]. This project uses Travis CI, a cloud-based continuous-integration platform, to build and run tests for every push [25]. Through our large-scale study, we identify 1,939 build logs for this project from Travis CI, and 124 of those are failed builds. Intuitively, when developers of `caelum/vraptor4` consider whether to apply TSR, they would need to consider the trade-off between the benefit of removing some number of tests and the cost of missing future faults due to removing the tests.

Assume the developer chooses the commit `b2437ab1` [6] as the reduction point for performing TSR. At this reduction point, the original test suite had 753 test units[1]. Coverage-based TSR (we also evaluate mutant-based TSR) using the Greedy algorithm [61, 108] (we also evaluate three more algorithms) finds that 419 of those test units are redundant and can be *removed*, so only 334 are *kept* in the reduced test suite, giving a $\mathsf{SizeKept}$ of 44.4%.

Given a reduced test suite and its corresponding original test suite, we categorize each failed test unit in a future failed build based on its presence or not in the reduced test suite. The failed

[1]The term *test unit* refers to either a test method or a test class; we discuss later how the PIT tool that we use to collect test coverage and mutation results produces information at the level of test methods or classes.

```
    request.setParameter(name, URLDecoder.decode(m.group(i),
-       "UTF-8"));
+       encodingHandler.getEncoding()));
```

Figure 2.1: Relevant part of a commit that failed a build

test can be removed from the reduced test suite, kept in the reduced test suite, or a new test that was added between the reduction point and the failed build (not in the corresponding original test suite). Failed tests that were removed can lead to missed faults. For example, the failed build for the commit `f810dd0d` has only one failed test unit, but this failed test unit would have been removed had TSR been applied on the earlier commit `b2437ab1`, and thus the build would have not failed; in this case, using TSR would have definitely lead to a *miss-build*, because the developer would not see *any* failure that indicates a fault. As another example, the failed build for the commit `10668287` has five failed test units, but *all* five failed test units would have been kept had TSR been applied, and thus the build not only would have still failed but also would have reported the same failed test units as when the original test suite had been run; in this case, using TSR would have definitely revealed the fault(s).

In general, however, checking whether the failures from a reduced test suite miss a fault in a failed build is challenging because failed builds can have a mix of failed test units that are removed or kept. For example, the build number `303` [7] for the commit `021d10b7` [4] has nine failed test units, one kept and eight removed by TSR. Because one of the failed test units would have been kept, this build would have failed even if TSR had been applied. While it is positive that the build would have failed, it can be negative that the build would have had only one failed test unit rather than nine. In general, two or more dynamic test failures may map to either one, same static fault in the code, or they may map to several different faults. Ideally, we would like to determine whether the developers applying TSR and seeing only a subset of failures would still be able to find *all the faults*. However, it is rather challenging to determine whether multiple test failures map to the same fault: given only a subset of failures, the developers may fix the code such that the other failures get fixed anyhow, or the developers may fix only the given failures while the others would still remain as failures (with the developers applying TSR unaware of that).

In this particular example, our manual inspection shows that all nine failures are actually due to the same fault. Figure 2.1 shows the relevant part of commit `021d10b7` [4] that failed the build [7]. The code had some encoding hard-coded to `UTF-8`, and the commit `021d10b7` [4] changed the code to get the encoding from the `web.xml` configuration file. This change broke all nine test units with a `NullPointerException`; had the developers seen any of the nine failures, they would have likely fixed their code to correct all failures. In fact, all nine test units stopped failing after just a one-line change [5].

Manual inspection of failures is extremely costly and error-prone, so we use automated heuristics to categorize failed builds based on how likely developers would have fixed the code even without seeing the failures from the removed test units. We propose FFMap, a family of mappings from test failures to faults, based on the proximity of removed and kept failed tests. We describe these mappings in more detail in Section 2.3. For example, one of the mappings maps test failures in the same test class to the same fault. In our example, all nine test methods are from the same test class, `DefaultParametersControlTest`, and indeed fail due to the same fault. With such a mapping, we say that using TSR would not miss any fault for that failed build.

2.3 FAILED-BUILD DETECTION LOSS (FBDL)

We describe how we measure *Failed-Build Detection Loss* (FBDL) for a reduced test suite. Given a reduced test suite from some reduction point, the goal is to find the percentage of future failed builds where the reduced test suite does not detect *all* the faults that the original test suite would detect; the reduced test suite cannot detect more faults than the original test suite detects, and we do *not* assume that the original test suite detects all faults in the code. We call a failed build a *miss-build* if the reduced test suite does not detect all the faults the original test suite detects. If F is the set of all future failed builds after a reduction point, and F_r is the subset of F where the reduced test suite detects all the faults that the original test suite detects, then we define FBDL as:

$$\mathsf{FBDL} = (|F| - |F_r|)/|F| * 100. \tag{2.4}$$

Similar to the other TSR metrics (Section 2.1.3), the smaller the FBDL, the better.

Given a reduced test suite, we want to find which of the future failed builds are in F_r, using test failures from historical build logs. Wong et al. [176] and Rothermel et al. [151] defined a similar metric, but their experiments constructed faulty versions of the program, each seeded with one fault. The set of all such faulty versions is the set F, and having one fault per version makes it easy to map test failures to the faults (all test failures map to the same fault). However, in real-world evolution, a program version can have multiple faults, and mapping test failures to faults is much harder, which in turn makes defining F_r harder as well.

2.3.1 Failure-to-Fault Map

We develop a family of mappings from test failures to faults, called *Failure-to-Fault Map* (FFMap). The different mappings are based on heuristics for how likely certain groupings of failed tests are due to the same fault. Our heuristics are based on the groupings of tests defined by developers. The first mapping, FFMap$_S$, is the most "optimistic" and maps every test failure to the same fault, so any test failure detects all the faults[2]. This mapping is the same one used in seminal experiments of TSR [151, 176]. The second mapping, FFMap$_P$, maps failed tests from the same (Java) package to the same fault(s). The third mapping, FFMap$_C$, maps failed tests from the same class to the same fault(s). The final mapping, FFMap$_U$, is the most "pessimistic" mapping and maps each test failure to its own unique fault, so all test failures are needed to detect all faults. Rothermel et al. [151] also mention potentially using this latter mapping for TSR evaluations, but they ultimately did not use it as their experiments are such that there was one seeded fault per program version, so using FFMap$_S$ made more sense.

2.3.2 Classifying Failed Builds

Using the different FFMap mappings, we define which failed builds are considered a miss-build. Given a reduced test suite from some reduction point, we find every failed build after that point, classify its failed tests, and finally classify the entire failed build based on its failed tests. We assume that whenever the original test suite passed, then the reduced test suite would have also passed. This assumption can break due to test-order dependencies [49, 179] or other causes of flakiness [128, 134], but it does hold in a majority of cases [179].

With respect to a reduced test suite (and its corresponding original test suite) from an early, passed commit, we give a classification for each failed test from a future failed build had the developer used with the reduced test suite. Each failed test is classified as: (1) REMOVED: the test existed in the original test suite on which TSR was performed but the test was *removed* and was not in the reduced test suite—this test would have not failed if using the reduced test suite; (2) KEPT: the test existed in the original test suite on which TSR was performed and the test was kept in the reduced test suite—this test would have still failed if using the reduced test suite; (3) NEW: the test did not exist in the original test suite and is newly added between the reduction point and the failed build—this test could have influenced the reduced test suite (e.g., a re-reduction could have been performed if there are many new tests), but the simplest assumption is to consider that the new tests would just be added to the reduced test suite. We assume the test suite evolves by adding all new tests into the reduced test suite, and we consider such new test failures.

Based on the classification of failed tests in a failed build, we classify the failed build into one of the six classifications: DEFMISS, LIKELYMISS, SAMEPACKAGE, SAMECLASS, HIT, and NEWONLY (listed in order of "badness"). Table 2.1 shows how we classify entire builds based on

[2]When a failed build is due to only one fault, as evaluated in prior TSR studies that used one seeded fault per version, FFMap$_S$ is the correct mapping, but in real evolution a failed build may be due to multiple regression faults.

Table 2.1: Build classification based on failed tests

Failed Tests \ Build	DefMiss	In-between			Hit		NewOnly
#Removed	>0	>0			0		0
#Kept	0	>0	0	>0	>0		0
#New	0	0	>0	>0	0	>0	>0

the number of Removed, Kept, and New test units. We present the build classifications in the order that is the easiest to understand.

DefMiss *Builds.* A DefMiss build is one in which the reduced test suite definitely cannot detect all the fault(s) that the original test suite would detect, no matter what FFMap mapping is used. We classify a build as DefMiss if *all* of the failed tests are Removed, i.e., the only tests that fail are tests that would have been removed from the reduced test suite, so the build would have not even failed if using the reduced test suite.

Hit *Builds.* In contrast to a DefMiss build where the reduced test suite fails to detect any fault, a Hit build is one where the reduced test suite detects *all* the faults that the original test suite would detect, no matter what FFMap mapping is used. We classify a build as Hit if none of the failed tests are classified as Removed and at least one failed test is classified as Kept. The number of New tests does not matter.

NewOnly *Builds.* We classify a build as NewOnly if all the failed tests are classified as New. In these builds, neither the reduced test suite nor the original test suite detect any fault. If the reduced test suite is modified such that all new tests are added into the reduced test suite, a NewOnly build would not be a miss-build.

"In Between" Builds. When a build has a mix of Removed and Kept/New tests, it is less clear if the failed build is a miss-build or not. While the reduced test suite would have failed on the build (because at least one failed test is included in the reduced test suite), at least one failed test would have been removed, so we cannot easily establish whether the reduced test suite would have detected *all* the faults. These builds can be miss-builds based on which FFMap mapping is used. We classify such builds into three separate (sub)classifications. A build is SameClass if each Removed test is from the same class as some Kept/New test. The intuition is that failed tests from the same class likely detect the same fault (as illustrated in Section 2.2). A build is SamePackage if it is not SameClass but each Removed test is from the same package as some Kept/New test. The reasoning is similar as for SameClass. All remaining builds are LikelyMiss, i.e., at least one failed Removed test does not share the same package (thus not the same class) as any Kept/New test. Because the Removed tests are rather different from all

KEPT/NEW tests, it is unlikely (though not impossible) that they failed due to the same fault(s).

2.3.3 Computing FBDL

With the failed builds classified, we compute which of those failed builds are miss-builds based on the FFMap mapping used. FBDL values for mappings FFMap_S, FFMap_P, FFMap_C, and FFMap_U are called FBDL_S, FBDL_P, FBDL_C, and FBDL_U, respectively.

For FBDL_S, a miss-build is only a failed build classified as DEFMISS, because the reduced test suite does not contain any of the failed tests in that build. For FBDL_P, a miss-build is a failed build classified as DEFMISS or LIKELYMISS, because the reduced test suite removed failed tests from different packages. For FBDL_C, a miss-build is a failed build classified as DEFMISS, LIKELYMISS, or SAMEPACKAGE, because the reduced test suite removed failed tests from different classes. Finally, for FBDL_U, a miss-build is a failed build classified as DEFMISS, LIKELYMISS, SAMEPACKAGE, or SAMECLASS (i.e., only failed builds classified as HIT or NEWONLY are not miss-builds), because at least one failed test is in the reduced test suite.

2.4 METHODOLOGY

For our evaluation of TSR, we need to obtain a dataset of projects, passed builds where we can apply TSR, and failed builds that we can use to compute the FBDL of the reduced test suites. We then evaluate if there are good predictors of FBDL. We model our experimental setup for evaluating TSR in a manner that simulates the approach by which developers could use TSR (Section 2.2).

2.4.1 Projects and Failures

To determine what projects to use for our experimental evaluation, we start with the dataset provided by TravisTorrent [51]. This dataset includes a large number of build logs harvested from Travis CI for a variety of projects from GitHub. We filter the TravisTorrent dataset to obtain a set of Java projects that use Maven. We focus on projects that use Maven because we rely on the PIT mutation testing tool [19] (Section 2.4.3) to obtain code coverage and mutation results. As in previous research for coverage-based TSR, we use mutants to measure the loss of reduced test suites as a part of our evaluation. We also use mutants as a different test-requirement for mutant-based TSR. We choose to use PIT because it is robust and increasingly used in research [57, 77, 79, 126, 157, 158, 164].

We aim to analyze projects with a non-trivial number of failed builds, because we want a representative sample of failed builds per project. A small number of failed builds would lead to only a few possible values for FBDL (e.g., if a project has only one failed build, then the FBDL is either 0% or 100%). We find projects that have over 20 failed builds that Travis CI marks as either "failed" or "errored" (because both kinds can have failed tests) and at least one passed build. We consider only the builds that are either a pull request for the `master` branch or a direct push into that branch. We focused on the master branch because (1) missing failures on it is the most problematic for the project, (2) the master branch has a linear history, so we can precisely determine whether a reduced test suite from some reduction point could have been propagated to a build at another point, and (3) the master branch is more likely to have commits available for reproducing runs. We obtain 144 projects that satisfy these requirements.

2.4.2 Reduction Points

We attempt to rerun old builds, going back to commits from 2013. While we need not rerun the failed builds (because TSR can be evaluated from failed tests, which can be determined from logs as described in Section 2.4.4), we do need to rerun some passed builds to collect code coverage and killed mutants, which are not available from the Travis CI logs, to perform TSR and evaluate test-requirements loss. However, reproducing old builds (even just compiling the code) is *challenging* for many reasons (e.g., changing Java version, missing dependencies, different environment). We create a Docker [14] image similar to the one used by Travis CI [24]. (Travis CI does not make public their actual Docker image.) We aim for multiple reduction points for each project so we can study (1) the effects of newly added tests on the results and (2) whether the distance from the reduction point to the failed build affects the FBDL.

We first try to reproduce a very early passed build for each project. Specifically, for each of the initially selected 144 projects, we find from the TravisTorrent dataset the earliest 10 passed builds whose commits could still be checked out from the GitHub repository. We check out these commits and try to reproduce the passed build by building the project in our Docker image using the commands specified in the project's `.travis.yml` file (e.g., `mvn install -DskipTests` followed by `mvn test`). We use the earliest passed build of those 10 builds as a candidate point for TSR. If none of those 10 builds pass, we exclude that project. We are left with 51 projects.

For each of these 51 projects, we further search for more commits on which we can reproduce passed builds to use as candidate points for TSR. Travis CI has many passed builds for each project. Ideally, we would evaluate TSR using all the passed builds, but given limited time and resources, we do not try them all. We search for the builds that ran *right before* a failed build. Selecting these builds as reduction points and then evaluating FBDL on *all* the failed builds after each point gives us (1) a diverse range of commit distances from the reduction point to the failed build and (2) a diverse number of newly added tests. Specifically, for each failed build, we take up to 3 passed builds (whose commits can be checked out) right before that failed build. Each reproducible passed build provides a candidate point.

We aim to analyze TSR for test suites with a non-trivial number of tests. We use 10 test methods (as reported by Surefire) as the threshold. We exclude the reduction points with fewer tests, obtaining 875 candidate reduction points for 51 projects.

2.4.3 Test-Suite Reduction

For each commit that is a candidate reduction point, we attempt to perform TSR. We use PIT [19] to obtain coverage matrices that map individual tests to lines they cover. PIT most often reports this mapping from test *methods* to lines covered, but in some cases (e.g., when a test class has a `@BeforeClass` annotation), PIT can only map the test *classes* to lines covered (e.g., for the test class `GraphHopperServletIT` in `graphhopper/graphhopper`). We can detect when PIT reports coverage for a test class because it repeats the fully-qualified name for the test class in place of the test method. In the rest of this chapter, when we refer to a *test unit* in a test suite, we mean either the test method or the test class that PIT reports as test units. For each project, we count the number of these test units reported by PIT.

For some commits, PIT fails to collect coverage, either crashing altogether or having some test fail while collecting coverage even though it passed without collecting coverage (e.g., the test may be flaky [128, 134]). We filter such commits from further analysis. Moreover, if PIT reports fewer than 10 test units in the original test suite for some commit, we filter out those commits, again because we need commits with sufficient number of test units such that performing TSR on that commit makes sense.

For the remaining commits, we first perform coverage-based TSR. We implement four different TSR algorithms: Greedy, GE, GRE, and HGS (Section 2.1.2). We apply each algorithm with the line-coverage information on each reduction point to create coverage-based reduced test suites. We exclude any reduced test suite that is the same as the original test suite, as they would behave the same. If we end up excluding all reduction points for a project, then we remove the project. We obtain 32 projects with 321 reduction points.

We also use PIT to collect what tests kill which mutants. We used all 16 mutation operators available in PIT, including the experimental ones [18]. Because mutation testing can be expensive, we limit PIT to run mutation testing up to 12 hours per reduction point. PIT times out or crashes on 95 reduction points, and we exclude them from any further analysis that involves mutants, which can result in excluding even entire projects from some analyses. We compute $\mathsf{MutLoss}$ at the reduction point (Section 2.1.3). Moreover, we apply mutant-based TSR [140, 158] to construct reduced test suites that kill all mutants as the original test suites. We perform mutant-based TSR also using the same four TSR algorithms. For test suites reduced using mutant-based TSR, we measure CovLoss at the reduction point.

2.4.4 Extracting Failed Tests

We do not collect test failures by rerunning failed builds because rerunning old builds is challenging, as mentioned for passed builds. From the logs of failed builds, we need to extract the names of the failed tests. For each build, the TravisTorrent dataset already provides the names of *some* failed tests extracted from the build logs. However, our sampling finds that the test names in TravisTorrent were often not properly extracted; the names only included the test method names but not the (fully qualified) test class names, and some failed tests were omitted altogether. We patch the TravisTorrent analyzer for finding failed tests to extract the fully qualified test names more correctly. This extraction requires the complete textual logs for each build, and we harvest these build logs ourselves from Travis CI. We harvest all the logs since the first build of each project on Travis CI up until April 1, 2017. Sometimes the build failed such that no failed test is reported in the log, e.g., the build failed in the compilation phase. For our further analysis, we only consider the failed builds where the failed test names are in the log. As a result, the analysis for some projects includes fewer than 20 failed builds.

A seemingly trivial but actually tricky aspect when extracting failed test names for use in classifying failed builds is to *match the names* of test units from the reduction point and the failed build. PIT provides test units that are sometimes test methods and sometimes test classes. Likewise, the reported failed tests from the Travis CI build logs are sometimes test methods and sometimes test classes. Our matching is as follows. If the failed test unit is a test method, then it matches either the exact same name or the test class of the test method in the original test suite. If the failed test unit is a test class, then it matches either the exact same name or *any* test method from the same class in the original test suite. If the matching finds the name, the test is REMOVED or KEPT; otherwise, it is NEW. We do not consider test renames as they are not frequent in test evolution [144], and method renames are hard to track in general [145, 167]. Note that when a failed test in a future build has the same name as a test at the reduction point, the developer may have actually modified the test body between the reduction point and the failed build.

2.4.5 Predicting FBDL

We want to evaluate the predictive power of traditional TSR metrics that can be generated at the reduction point (Section 2.1.3). We utilize a linear regression model to see if there is a linear correlation between the predictor and FBDL. The linear regression model outputs an R^2 value,

Table 2.2: Statistics about our evaluation projects; $\mu \pm \sigma$ values are across all algorithms (Greedy, GE, GRE, HGS)

ID	Project slug	#Builds		Avg #Failed	#Red.	Orig.	Coverage-Based		Mutant-Based	
		Total	Failed	Test Units	Points	Size	SizeKept	MutLoss	SizeKept	CovLoss
P1	addthis/stream-lib	184	11	1.5	8	106.6	48.1% ± 1.6%	3.1% ± 0.3%	61.9% ± 1.0%	0.3% ± 0.0%
P2	azagniotov/stubby4j	886	13	3.0	9	50.1	45.2% ± 3.2%	2.8% ± 1.3%	49.2% ± 2.7%	2.5% ± 2.4%
P3	caelum/vraptor4	1,939	124	4.4	7	770.6	44.1% ± 0.4%	6.5% ± 0.8%	51.1% ± 1.0%	7.1% ± 2.1%
P4	dynjs/dynjs	383	18	264.0	10	798.3	34.6% ± 0.8%	t/o	t/o	t/o
P5	FasterXML/jackson-core	580	23	2.3	20	274.4	67.5% ± 3.1%	1.8% ± 0.3%	76.9% ± 2.3%	0.3% ± 0.1%
P6	google/auto	595	33	2.8	3	27.3	65.3% ± 2.1%	0.6% ± 0.4%	63.2% ± 5.1%	8.2% ± 5.5%
P7	google/truth	419	34	10.4	3	222.7	53.3% ± 3.1%	3.3% ± 0.6%	57.0% ± 3.3%	1.6% ± 1.8%
P8	graphhopper/graphhopper	2,269	117	18.3	33	686.0	23.7% ± 1.7%	t/o	t/o	t/o
P9	HubSpot/jinjava	398	6	1.7	16	307.1	41.2% ± 0.5%	4.9% ± 1.1%	53.4% ± 0.7%	1.0% ± 1.0%
P10	iluwatar/java-desig... [13]	1,537	16	2.2	1	52.0	96.2% ± 0.0%	0.8% ± 0.0%	96.2% ± 0.0%	0.9% ± 0.0%
P11	jOOQ/jOOQ	1,993	88	5.8	34	29.2	69.0% ± 2.4%	0.0% ± 0.0%	69.1% ± 2.5%	0.0% ± 0.0%
P12	jsonld-java/jsonld-... [15]	290	17	2.1	3	45.7	21.4% ± 7.9%	t/o	t/o	t/o
P13	kongchen/swagger-ma... [22]	511	87	3.0	2	22.0	31.8% ± 0.0%	3.0% ± 0.3%	47.7% ± 2.4%	0.0% ± 0.0%
P14	ktoso/maven-git-com... [16]	349	34	7.5	8	39.0	42.7% ± 4.3%	4.2% ± 1.3%	49.9% ± 5.3%	6.6% ± 1.7%
P15	larsga/Duke	146	15	2.6	8	640.4	34.1% ± 1.2%	6.9% ± 0.5%	48.3% ± 0.6%	0.4% ± 0.1%
P16	lviggiano/owner	582	19	4.8	25	216.7	34.5% ± 1.0%	4.5% ± 1.0%	38.0% ± 1.4%	7.0% ± 5.4%
P17	mgodave/barge	184	40	3.1	4	27.5	42.6% ± 3.4%	1.7% ± 0.2%	55.4% ± 2.5%	0.3% ± 0.6%
P18	myui/hivemall	671	50	6.9	9	85.2	57.7% ± 2.5%	t/o	t/o	t/o
P19	notnoop/java-apns	229	75	8.5	5	99.8	33.5% ± 1.0%	6.2% ± 2.1%	39.3% ± 0.6%	9.8% ± 1.8%
P20	nurkiewicz/spring-d... [21]	101	13	53.5	1	34.0	30.9% ± 1.7%	1.4% ± 1.2%	32.4% ± 0.0%	0.0% ± 0.0%
P21	perwendel/spark	862	55	11.9	13	16.5	82.1% ± 2.5%	0.0% ± 0.0%	75.5% ± 2.0%	0.3% ± 0.1%
P22	rackerlabs/blueflood	2,296	300	4.7	3	121.0	54.8% ± 1.0%	4.7% ± 0.5%	62.6% ± 0.9%	1.0% ± 0.4%
P23	redline-smalltalk/r... [20]	228	19	2.4	3	112.3	74.5% ± 1.5%	2.9% ± 1.7%	76.8% ± 4.2%	2.2% ± 1.1%
P24	relayrides/pushy	738	30	5.6	4	47.2	66.7% ± 2.8%	0.8% ± 0.6%	68.6% ± 3.9%	0.3% ± 0.1%
P25	sanity/quickml	643	52	1.5	14	37.1	63.0% ± 6.4%	4.5% ± 2.4%	83.3% ± 2.2%	2.4% ± 1.8%
P26	scobal/seyren	453	21	13.4	3	23.3	41.0% ± 5.6%	0.9% ± 0.5%	37.5% ± 5.5%	3.4% ± 1.3%
P27	spotify/cassandra-reaper	382	21	3.7	5	20.8	68.9% ± 3.0%	0.7% ± 0.5%	77.4% ± 3.3%	0.1% ± 0.2%
P28	square/dagger	758	13	31.7	7	116.7	38.5% ± 2.3%	3.5% ± 0.7%	49.3% ± 1.9%	1.6% ± 0.6%
P29	square/wire	1404	32	11.5	2	73.0	52.5% ± 5.4%	1.1% ± 0.3%	62.0% ± 11.3%	1.4% ± 1.6%
P30	tananaev/traccar	2,960	44	2.8	36	131.5	93.7% ± 2.5%	0.7% ± 0.6%	96.0% ± 1.1%	0.2% ± 0.2%
P31	twilio/twilio-java	431	13	1.5	6	88.5	73.6% ± 4.3%	0.7% ± 0.5%	72.7% ± 4.3%	1.7% ± 1.1%
P32	weld/core	2,060	45	4.8	16	280.6	35.6% ± 2.2%	t/o	t/o	t/o
	Overall (Sum or Average)	27,461	**1,478**	504.2	321	175.1	**51.9%**	**2.7%**	**61.1%**	**2.2%**

ranging from 0 to 1, and we are looking for an R^2 value larger than 0.7, showing strong linear correlation [81]. The p-value then shows statistical significance (the lower the better).

In case the predictor is not linearly correlated with FBDL, we also evaluate using Kendall-τ_b rank correlation. Kendall-τ_b rank correlation does not assume the two are linearly correlated, and instead measures if the predictor and FBDL both go up or both go down. Kendall-τ_b rank correlation outputs a τ_b value ranging from -1 to 1, where a negative value indicates negative correlation and a positive value indicates positive correlation. The higher the absolute value, the better the correlation, and we are again looking for an absolute value greater than 0.7, showing strong correlation [81]. There is also a p-value for statistical significance.

2.4.6 Summary of Analyzed Projects

Table 2.2 shows a summary of the final 32 evaluation projects: the short ID to be used later, the GitHub user/repo slug for the evaluation project, the total number of builds, the number of failed builds, the average number of failed test units per failed build, the number of reduction points, the average size (i.e., the number of test units) of the test suite at the reduction points, the SizeKept for coverage-based TSR using four TSR algorithms, the MutLoss of the coverage-based reduced test suites, the SizeKept for mutant-based TSR using four TSR algorithms, and the CovLoss of the mutant-based reduced test suites. The tabulated mean±std.dev. values are across all reduction points and all four algorithms. The cells with "t/o" mark the cases where PIT does not complete mutation testing. The overall summary numbers are: (1) SizeKept, 51.9% and 61.1% for coverage- and mutant-based TSR, respectively, indicate that almost *half of the tests are redundant* with respect to those TSR criteria; and (2) the average MutLoss, 2.7%, shows that, at the reduction points, coverage-based TSR results in only slightly fewer mutants killed than the original test suite; likewise, the average CovLoss, 2.2%, shows that mutant-based TSR results in very little coverage loss compared to the original test suite.

2.5 RESULTS AND ANALYSIS

Our evaluation aims to answer the following research questions:

RQ1 What is the FBDL of TSR in real software evolution?

RQ2 How well can the FBDL of TSR be predicted?

RQ3 How does distance from TSR reduction point affect FBDL?

The goal of RQ1 is to measure the FBDL of TSR, which has not been done before for real-world software evolution. The goal of RQ2 is to see whether FBDL can be predicted well: if the FBDL is high but can be predicted, the developer can still make a cost-benefit analysis about using the reduced test suite. The goal of RQ3 is also related to prediction: if a longer distance from a reduction point leads to worse FBDL, the developer can make an informed decision to stop using the reduced test suite.

2.5.1 RQ1: FBDL

Figures 2.2 and 2.3 show FBDL$_S$, FBDL$_P$, FBDL$_C$, and FBDL$_U$ for each evaluation project using coverage-based Greedy TSR and mutant-based HGS TSR, respectively. For each evaluation project, we compute each metric for each reduction point and then average (using arithmetic mean) across all reduction points; the Avg bar shows the averages across all evaluation projects. The bars overlap the metric values for each evaluation project, as FBDL cannot decrease going from FBDL$_S$ to FBDL$_U$. The top, respectively bottom, half of each figure shows FBDL computed including, respectively excluding, NEWONLY builds.

For example, for P3 (`caelum/vraptor4`) in Figure 2.2, the FBDL percentages of FBDL$_S$, FBDL$_P$, FBDL$_C$, and FBDL$_U$ are 1.4%, 1.4%, 1.4%, and 50.5%, respectively. Excluding the NEWONLY

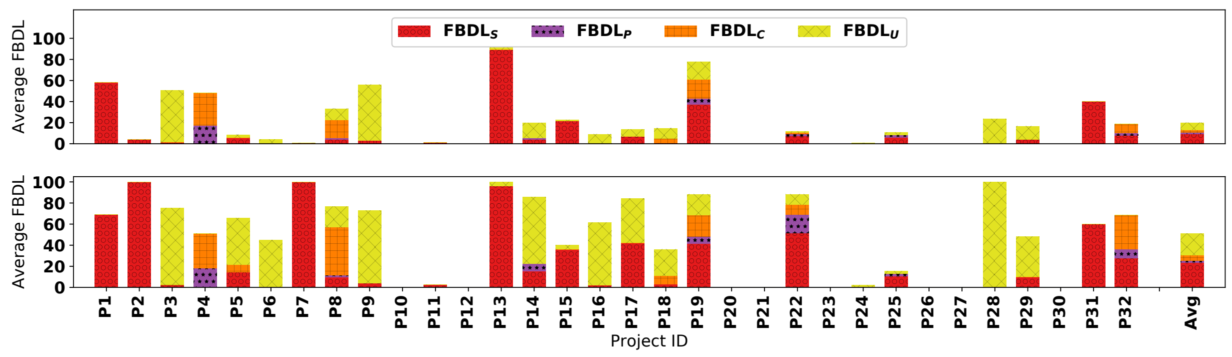

Figure 2.2: Average FBDL when including (top) and excluding (bottom) NEWONLY (Coverage-based Greedy)

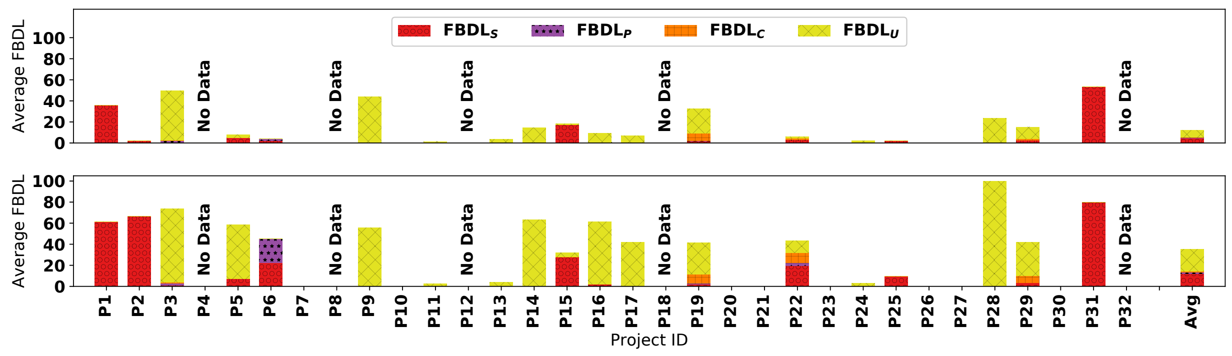

Figure 2.3: Average FBDL when including (top) and excluding (bottom) NEWONLY (Mutant-based HGS)

builds, the percentages are 2.2%, 2.2%, 2.2%, and 75.2%, respectively. For this case, SizeKept (average of 44.2% kept; Table 2.2 shows 44.1% average over all four algorithms) may be worth it as FBDL$_S$, FBDL$_P$, and FBDL$_C$ are rather low. However, if the developer believes that each test failure detects a unique fault, then based on FBDL$_U$, the reduction may not be worth it.

Figure 2.3 shows the breakdown per evaluation project for mutant-based TSR and has no data for five evaluation projects because PIT could not collect mutation testing results for those evaluation projects (Section 2.4.3). For the same P3, the percentages of FBDL$_S$, FBDL$_P$, FBDL$_C$, and FBDL$_U$ are 0.5%, 2.0%, 2.6%, and 49.7%, respectively. Excluding NEWONLY, the percentages are 0.7%, 2.9%, 3.8%, and 73.9%, respectively. The developer may reach similar conclusions on whether or not to use the mutant-based reduced test suite for P3 as with the coverage-based one.

The two figures show the results for only one of four TSR algorithms for each TSR criterion, coverage- and mutant-based. The distributions are visually similar for the same TSR criterion for the other three TSR algorithms. Table 2.3 shows the overall averages for each FBDL for all TSR algorithms, computed excluding NEWONLY builds. This table allows comparing percentages

Table 2.3: Averages of FBDL for different TSR techniques

Technique		FBDL_S	FBDL_P	FBDL_C	FBDL_U
Cov	Greedy	26.1%	27.4%	28.9%	52.2%
	GE	21.3%	22.8%	29.1%	45.2%
	GRE	23.6%	24.8%	30.6%	47.2%
	HGS	22.6%	23.6%	29.1%	48.5%
Mut	Greedy	13.1%	14.7%	15.5%	36.2%
	GE	10.5%	12.0%	12.8%	34.3%
	GRE	12.2%	13.7%	14.8%	36.0%
	HGS	12.1%	13.2%	14.3%	35.5%

across the two TSR criteria, as the percentages are averaged across the builds that are *in common* between coverage- and mutant-based TSR techniques. (Recall from Section 2.4.3 that PIT fails to produce mutation testing results for 95 reduction points.)

We use a statistical analysis to compare each pair of algorithms for the same TSR criterion, e.g., comparing coverage-based Greedy and coverage-based HGS. Because different FBDL are computed based on the classification of individual failed builds, we focus on comparing those classifications. Specifically, we use the Student's paired t-test to compare the ratio of builds classified in the same category, e.g., DEFMISS. We find that the p-value ranges from 0.12 to 0.99 for coverage-based TSR, and from 0.25 to 0.94 for mutant-based TSR. Such high p-values fail to reject the null hypothesis that the reduced test suites from any pair of algorithms for the same TSR criterion are from the same distribution for any failed build classification. While failing to reject the null hypothesis does not imply accepting it, all four algorithms likely behave the same *for the same TSR criterion*.

We further compare the four TSR algorithms by viewing the classification of failed builds as a multiclass classification [17] and computing the accuracy/overlap of classifications for each pair of algorithms for each evaluation project. The average overlap across all evaluation projects and all pairs of algorithms ranges from 0.87 to 0.94 for coverage-based TSR and from 0.95 to 0.98 for mutant-based TSR. Because the overlap of failed builds that are classified the same between algorithms using the same TSR criterion is so high, we show detailed classification results for only one representative TSR algorithm, Greedy for coverage (Figure 2.2) and HGS for mutants (Figure 2.3). The analyses we show later in this chapter are only for those two TSR techniques.

We finally evaluate the correlation between pairs of FBDL variants (for the same TSR technique) using Kendall-τ_b rank correlation for each reduced test suite. The correlation among FBDL_S, FBDL_P, and FBDL_C is rather high, with τ_b ranging from 0.69 to 0.98, all with $p < 0.0001$. However, FBDL_U is not as correlated with the other FBDL values, with τ_b ranging from 0.29

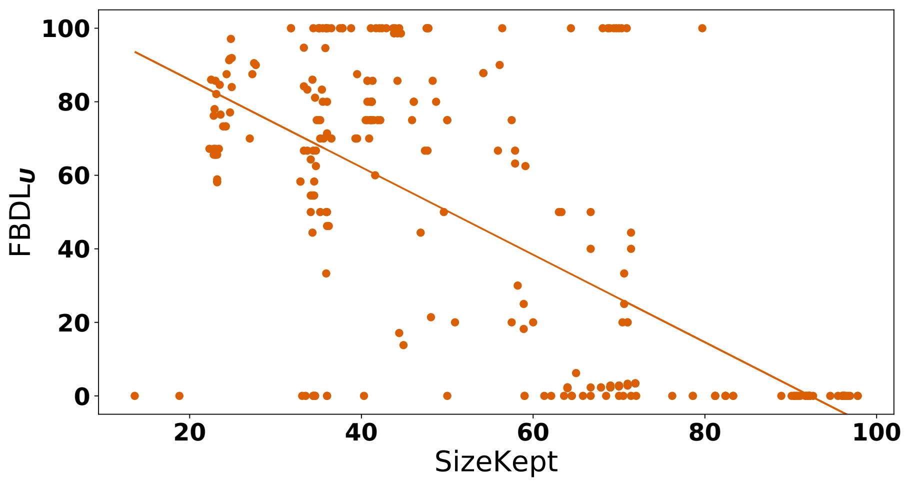

Figure 2.4: SizeKept vs. FBDL$_U$ (Coverage-based Greedy)

to 0.80, with most less than 0.6, all with $p < 0.0001$.

A1: In sum, the FBDL cost of TSR is rather high, with a lower bound of 9.5% (based on FBDL$_S$) and an upper bound of 52.2% (based on FBDL$_U$ and excluding NEWONLY builds).

2.5.2 RQ2: Predicting FBDL

We evaluate the predictive power of three metrics that can be measured on the reduced test suite when it is created: the two traditional metrics as described in Section 2.1.3 and one new metric we propose based on historical FBDL. To focus evaluation on predicting FBDL due to tests *removed* from the original test suite, we exclude NEWONLY builds; their failed tests did not exist at the reduction point and could not have been removed.

SizeKept Intuitively, the more tests are kept in the reduced test suite, the less likely the reduced test suite results in a miss-build. We expect a negative correlation between SizeKept and FBDL.

Figure 2.4 shows a scatter plot relating SizeKept and the FBDL$_U$ for each coverage-based Greedy reduced test suite. We show the prediction for FBDL$_U$ because SizeKept predicts FBDL$_U$ the best among all four FBDL variants. The plot includes the linear regression line. $R^2 = 0.45$, with $p < 0.0001$, suggests a weak linear fit; $\tau_b = -0.41$, with $p < 0.0001$, suggests a weak negative correlation. For the other FBDL variants, R^2 ranges from 0.04 to 0.23, all with $p < 0.001$, and τ_b ranges from -0.23 to -0.34, all with $p < 0.0001$.

Figure 2.5 shows the same kind of plot as Figure 2.4, but for mutant-based HGS reduced test suites. There are similar trends for mutant-based TSR. For mutant-based HGS reduced test suites, we see SizeKept predicts FBDL$_U$ the best; $R^2 = 0.26$, with $p < 0.0001$, suggests a weak linear fit;

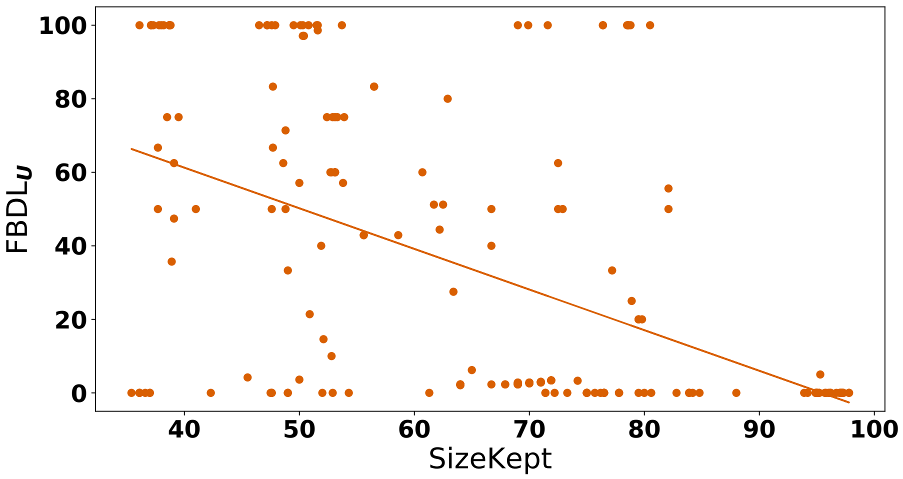

Figure 2.5: SizeKept vs. FBDL$_U$ (Mutant-based HGS)

$\tau_b = -0.40$, with $p < 0.0001$, suggests a weak negative correlation. For the other FBDL variants, the R^2 values are the same, 0.00, but with very high p-values. The τ_b values range from -0.10 to -0.14, all with rather high p-values. Overall, these results show that SizeKept at the reduction point does not correlate well with future FBDL and cannot well predict FBDL.

We also compare SizeKept and FBDL *within individual evaluation projects* that have more than 5 reduced test suites (to ensure enough data points to draw any conclusions on correlations). We check if any evaluation project results in strong correlations, i.e., R^2 value or τ_b value greater than 0.7 [81]. For all coverage-based TSR, only one evaluation project, P18, results in $R^2 > 0.7$, with $p = 0.0030$, for predicting both FBDL$_S$ and FBDL$_P$. For mutant-based TSR, only one evaluation project, P5, results in $R^2 > 0.7$, with $p = 0.0002$, for predicting FBDL$_S$, FBDL$_P$, and FBDL$_C$. For τ_b values, *no* evaluation project results in an absolute value greater than 0.7, for either coverage- or mutant-based TSR. Even per evaluation project, SizeKept and FBDL do not strongly correlate.

ReqLoss We evaluate ReqLoss as a predictor of FBDL. We use MutLoss for coverage-based TSR and CovLoss for mutant-based TSR. We expect a positive correlation between ReqLoss and FBDL.

Our experiments show rather low ReqLoss, with average 2.7% MutLoss across all projects for coverage-based TSR and 2.2% CovLoss for mutant-based TSR; similar low percentages were reported in recent TSR studies [57, 158, 188]. Such low percentages suggest that the reduced test suites may not miss many faults. However, we find the average FBDL$_S$, the most "optimistic" FBDL metric (Section 2.3.3) to be much higher than the average ReqLoss. While the ReqLoss is not equal to FBDL, it may still be a good predictor. Intuitively, the higher the ReqLoss, the more likely the reduced test suite results in a miss-build.

Figure 2.6 shows a scatter plot relating MutLoss and FBDL$_U$ for each coverage-based Greedy

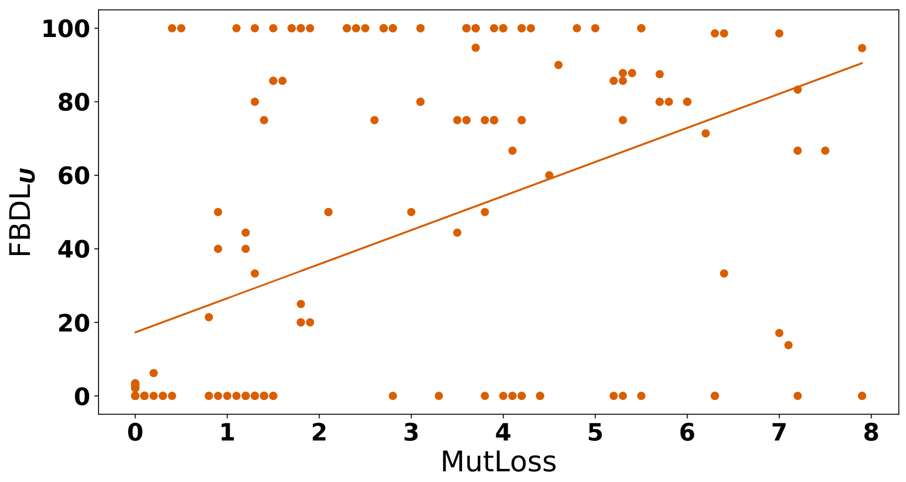

Figure 2.6: MutLoss vs. FBDL$_U$ (Coverage-based Greedy)

reduced test suite where we could measure mutation score. We show the prediction for FBDL$_U$ because MutLoss predicts FBDL$_U$ the best. The plot includes the linear regression line. $R^2 = 0.25$, with $p < 0.0001$, suggests a weak linear fit; $\tau_b = 0.28$, with $p < 0.0001$, suggests a weak positive correlation. For the other FBDL variants, R^2 ranges from 0.02 to 0.03. For τ_b, the values are all the same, -0.04. For other variants, the p-values are all rather high.

Figure 2.7 shows a similar scatter plot, but relating CovLoss of mutant-based HGS reduced test suites. There are similar trends for mutant-based TSR. For mutant-based HGS reduced test suites, we see CovLoss predicts FBDL$_U$ the best. $R^2 = 0.14$, with $p < 0.0001$, suggests a weak linear correlation; $\tau_b = 0.18$, with $p < 0.001$, suggests a weak positive correlation. For the other FBDL variants, the R^2 values are the same, 0.00, but with very high p-values. The τ_b values range from 0.11 to 0.19 (where correlating with FBDL$_P$ has a higher τ_b value than with FBDL$_U$, but has lower R^2 value); the highest p-value is 0.0687.

We also analyze if ReqLoss is a good predictor on a per-project basis, considering only evaluation projects with more than 5 reduced test suites. When correlating MutLoss with FBDL, only one project, P14, has $R^2 > 0.7$, but only for predicting FBDL$_P$ and FBDL$_C$, with $p = 0.0207$. Two projects have $\tau_b > 0.7$, with the highest p-value being 0.0207. When correlating CovLoss with FBDL, two projects result in $R^2 > 0.7$, with the highest p-value being 0.0570. Unfortunately, no project has a good τ_b.

Overall, ReqLoss is not a good predictor of FBDL for either coverage- or mutant-based TSR.

Combining Both Traditional Metrics ReqLoss controlled for SizeKept might result in a good predictor of FBDL. We create a linear model to correlate FBDL with a linear combination of ReqLoss and SizeKept. For coverage-based Greedy reduced test suites, MutLoss and SizeKept

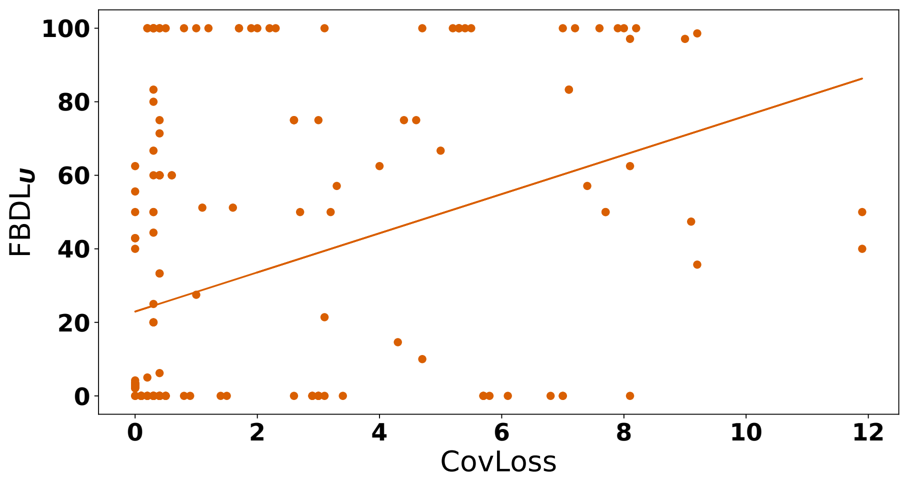

Figure 2.7: CovLoss vs. FBDL$_U$ (Mutant-based HGS)

together predict FBDL$_U$ the best; $R^2 = 0.42$ (higher than before), with $p < 0.0001$. For mutant-based HGS reduced test suites, CovLoss and SizeKept together predict FBDL$_U$ the best; $R^2 = 0.27$ (higher than before), with $p < 0.0001$. The correlations are still not strong. Even a more complex model that accounts for interaction of ReqLoss and SizeKept does not predict FBDL well.

Historical FBDL We propose to use historical FBDL to predict the future FBDL. The scenario is that a developer applies TSR at a past point and measures the FBDL that would have been obtained from that point until the current version; the developer then uses this historical FBDL to predict the FBDL of the *same* reduced test suite for future builds. (Note that this scenario does *not* re-reduce the test suite at the current point but reuses the exact same test suite from the past.) We simulate this scenario in our evaluation for each reduced test suite. We first find all the failed builds after the reduction point, then find the middle build among those failed builds, and finally compute the historical FBDL before the middle build and future FBDL after the middle build. We then compute correlation between historical and future FBDL.

Figure 2.8 shows a scatter plot relating the historical FBDL$_C$ and the future FBDL$_C$ for each coverage-based Greedy reduced test suite. We show the prediction for FBDL$_C$ because it is predicted the best. The plot includes the linear regression line. $R^2 = 0.57$, with $p < 0.0001$, suggests a weak linear fit; $\tau_b = 0.64$, with $p < 0.0001$, suggests a weak positive correlation. For the other FBDL variants, R^2 ranges from 0.42 to 0.56, all with $p < 0.0001$, and τ_b ranges from 0.40 to 0.62, all with $p < 0.0001$. For coverage-based TSR, historical FBDL is a stronger predictor than other metrics. However these values still do not indicate a very strong correlation.

Figure 2.9 shows a similar scatter plot as Figure 2.8, but for mutant-based, HGS reduced test suites and with FBDL$_S$. There are similar trends for mutant-based TSR. For mutant-based HGS

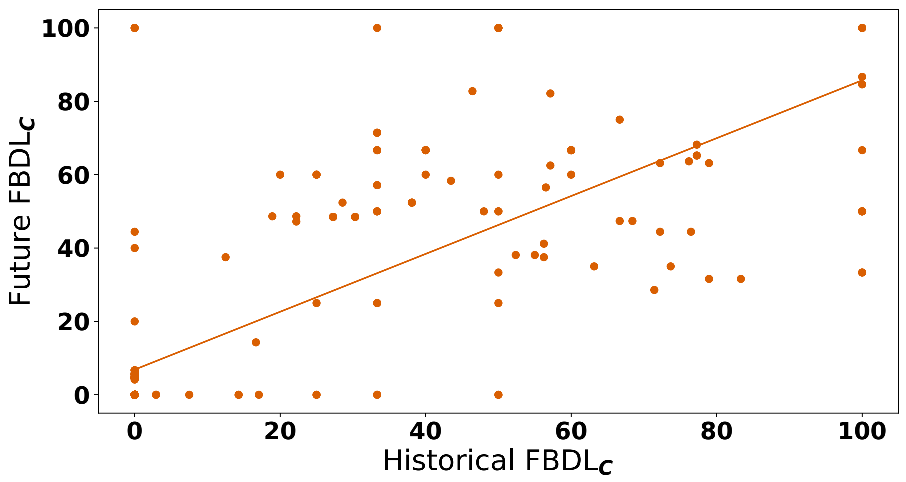

Figure 2.8: Historical vs. future FBDL$_C$ (Coverage-based Greedy)

reduced test suites, $R^2 = 0.55$, with $p < 0.001$, suggests a weak linear fit; $\tau_b = 0.65$, with $p < 0.001$, suggests a weak positive correlation. For the other FBDL variants, the R^2 ranges from 0.48 to 0.50, all with $p < 0.0001$, and τ_b range from 0.59 to 0.67, all with $p < 0.0001$. FBDL$_P$ and FBDL$_C$ are predicted with better τ_b values than FBDL$_S$, but have smaller R^2 values. For mutant-based TSR, historical FBDL is again a stronger predictor than other metrics, but like for coverage-based TSR, it is still not a very strong correlation.

Per evaluation project, coverage-based reduced test suites for six projects achieve $R^2 > 0.7$, with the highest p-value being 0.0073; 5 projects achieve an *absolute* τ_b greater than 0.7, with the highest p-value being 0.0042. However, two of these projects have a *negative* correlation, i.e., $\tau_b < -0.7$. Mutant-based reduced test suites for four projects have $R^2 > 0.7$, with the highest p-value 0.0038; 2 projects achieve $\tau_b > 0.7$, with the highest p-value being 0.0565. Overall, per evaluation project, historical FBDL is a better predictor of future FBDL than other metrics, though still for only a subset of evaluation projects.

A2: In sum, we find that the two traditionally used metrics (size reduction and test-requirements loss) are not good predictors of FBDL. Historical FBDL is a much better predictor of future FBDL but still not strong in most cases.

2.5.3 RQ3: Impact of Evolution

Although we find no good predictor for FBDL at the reduction point, some correlation may still exist between FBDL and the distance, e.g., measured in terms of the number of builds from the reduction point to the failed builds. Intuitively, a reduced test suite may result in more miss-builds at a *larger* distance from the reduction point; if so, that observation could lead to some actionable

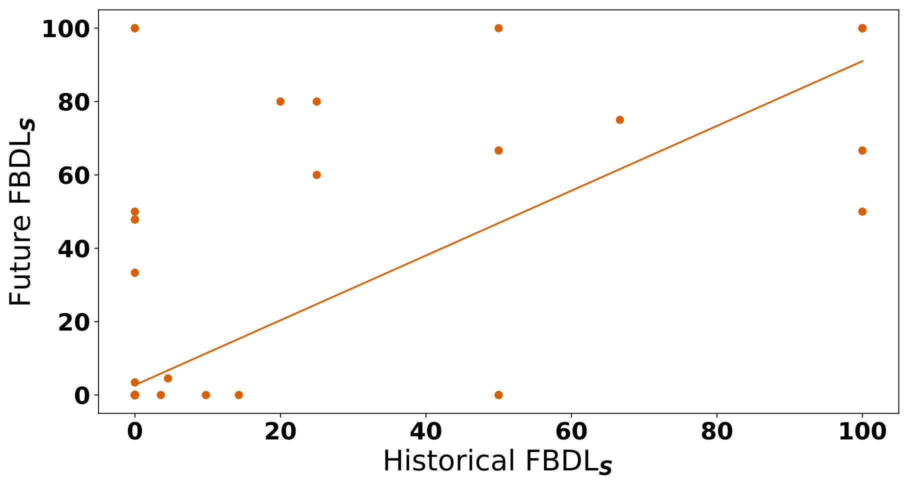

Figure 2.9: Historical vs. future FBDL (Mutant-based HGS)

insight, e.g., the developers can switch back to the original test suite after some time or re-perform TSR to obtain an updated reduced test suite.

We analyze the relationship between the number of builds since reduction and FBDL. We analyze each evaluation project separately. All builds for an evaluation project are split into 10 bins with about the same number of builds per bin. (Evaluation projects that have a different number of builds end up with different bin sizes.) For each bin, we calculate the ratio of miss-builds out of failed builds in that bin, excluding NewOnly builds. We also compute a best fit line through the 10 points given by the ratios. The slope of this line represents a simple measure of trend in FBDL with the number of builds since reduction. A positive slope is increasing FBDL, and a negative slope is decreasing FBDL.

One may expect the slope to be positive for most of the projects, i.e., the farther the builds are from the reduction point, the more likely to have miss-builds. However, the results do not show this to be the case. For example, evaluating FBDL$_U$ (which is the best predicted out of all the FBDL variants) using coverage-based Greedy TSR has 9 projects with negative slopes, 8 with slope 0, and 11 with positive slopes. (4 projects do not have enough failed builds to calculate a slope.) For the projects that have a non-zero slope, we also calculate R^2 and p-values. The R^2 values have a median of 0.28 and p-values have a median of 0.1421. The generally low R^2 values and high p-values suggest that the increasing and decreasing patterns are not strong for individual projects as well. Visual inspection also showed that plots have varying patterns, not only for coverage-based Greedy TSR but for all other techniques.

A3: FBDL does not correlate well with the distances from the reduction point, showing yet again that TSR is quite unpredictable.

2.6 THREATS TO VALIDITY

Our study on TSR has internal, external, and construct threats to the validity of our results, as any other empirical study. Our results may not generalize beyond the evaluation projects we evaluate. To reduce this risk, we use a diverse set of evaluation projects. Our results are based on our choice of reduction points for TSR, so a different choice of reduction points may lead to different FBDL values. We reduce this risk by choosing reduction points diverse in both commit distances to failed builds and numbers of newly added tests. Different TSR algorithms guided by different TSR criteria may result in different reduced test suites. We evaluate four widely-used TSR algorithms with two widely-used TSR criteria.

We introduce a new metric FBDL based on mapping test failures to faults. Some of our evaluation projects have a small number of flaky tests [128], which may affect our study by introducing false failures; we believe that they do not affect our key findings. We have spent substantial engineering effort trying to make the runs more reproducible using Docker, starting from the Travis CI Docker image [24]. Test suites may have test-order dependencies [49, 84, 118, 120, 179, 190]. TSR inherently assumes that test suites do not have test-order dependencies [120], and our experiments assume the same.

A specific question concerning our study is whether the software project history when using TSR would look as it does when project developers likely did not use TSR. (Only 8 of 321 reduction points had original test suites that were smaller than those in the prior reduction point, likely because developers manually removed some tests.) If developers actually kept only the reduced test suite at some point, their behavior in the future could differ from what we see in the code repository and the build logs that used the original test suite. In theory, developers could have a completely different behavior, e.g., making different code changes or testing those changes at different times. More likely, the developers could have modified the test suites differently. For example, the developers may have added more or fewer new tests than we see currently; in the limit, a novice developer may be unaware that TSR was performed and could manually write tests similar to those that were removed. As another example, even if some build has no test failure due to a fault because a test was removed due to reduction, a developer may learn about the fault in another way (e.g., by getting a report from a user), and then the developer could manually add a new test (or reintroduce that originally removed test) to detect that regression fault.

2.7 SUMMARY

Automated TSR is more risky than suggested by prior research. TSR was proposed over two decades ago, but since its inception it was evaluated primarily with mutants or seeded faults. We present the first study that evaluates the cost of TSR using real test failures. Our analysis shows that FBDL can go up to 52.2%, much higher than the mutant-detection loss. Moreover, FBDL is difficult to predict using traditional TSR metrics. Developers who are considering *current* TSR techniques should use FBDL to weigh whether the reduced test-suite size warrants the risk of missing faults. Real builds, however, do have potential for safe(r) TSR, so researchers could develop *novel* TSR techniques that either miss fewer failed builds or at least provide more predictable FBDL. Researchers should use FBDL to evaluate the quality of newly proposed TSR techniques.

CHAPTER 3: OPTIMIZING TEST PLACEMENT

This chapter presents TestOptimizer, a technique for optimizing the placement of tests to reduce the number of test executions over time for module-level regression testing. Section 3.1 provides some background on modern build systems and module-level regression testing. Section 3.2 formalizes the problem that we address with TestOptimizer. Section 3.3 presents the TestOptimizer technique itself. Section 3.4 describes details of implementing TestOptimizer on top of Microsoft's internal build system CloudBuild. Section 3.5 presents the evaluation results. Section 3.6 presents threats to validity. Finally, Section 3.7 concludes and summarizes the chapter.

3.1 BACKGROUND

Large-scale software development projects use *build systems* to manage the process of building source code, applying static analyzers, and executing tests. Any inefficiencies in the underlying build system directly impact developer productivity [64]. Given the significance of build systems, all major companies have made huge investments in developing efficient, incremental, parallel, and distributed cloud-based build systems. Example build systems include Bazel [9], Buck [36], CloudBuild [67, 156], and FASTBuild [10].

3.1.1 CloudBuild

CloudBuild [67] is a distributed, cloud-based, build system widely used internally at Microsoft. CloudBuild handles thousands of builds and executes millions of tests each day. Other companies such as Google also have similar distributed cloud-based build systems to build their large code bases (e.g., Bazel [9] for Google). The high-level principles behind CloudBuild are the same with these other build systems. We focus on CloudBuild because of our own experiences with this build system[1], and we implement our technique specifically on top of CloudBuild (the ideas are applicable to other similar build systems).

CloudBuild views a software project as a group of inter-dependent modules to perform incremental builds after developer changes. Each module is a grouping of related code, and building a module results in a binary, e.g., a DLL file. Developers specify a build dependency graph where a module depends on another module if the former module uses the API from the binary built in the latter module. When a developer makes changes, CloudBuild can determine which modules are *affected* by the changes, i.e., (1) the modules whose code the developers directly changed and

[1]The author of this dissertation did an internship at Microsoft working with the team that develops CloudBuild.

(2) the modules that transitively depend on the changed modules based on the dependency graph. CloudBuild then only builds the affected modules, i.e., it performs incremental builds. After a change, CloudBuild does not build any unaffected modules, but if an unaffected module is needed as a dependency for another affected module, CloudBuild uses a previously cached binary for that unaffected module for building affected ones. When building a module, CloudBuild allocates a clean server from the cloud just to build that module. CloudBuild then copies in all the necessary dependency binaries based on the dependency graph. This setup ensures that the server contains only the necessary dependencies for building the module and nothing else. The output of building a module is a binary. If there are tests in the module, CloudBuild also runs those tests. After building, CloudBuild caches the output binary for future builds.

Based on the nature of its incremental builds, CloudBuild inherently performs (safe) regression test selection (RTS) [74, 150, 183] at the module level. More specifically, whenever any dependency of a test module is affected by the given change, all tests in that module are executed; otherwise, the module is skipped, and no tests in the module are executed. The major advantage of module-level RTS is that it does not require any additional metadata (such as fine-grained dependencies like exercised statements for each test) beyond what is available in the build specification, namely static, compile-time module dependencies. Especially in the case of large software projects that execute millions of tests each day, storage and maintenance of the metadata would add non-trivial overhead. Therefore, module-level RTS is currently the most practical option for performing RTS at this scale.

3.1.2 Running Example

To illustrate how a CloudBuild performs RTS at the module level and the problem we address in this chapter, consider an example dependency graph shown in Figure 3.1 for a hypothetical project. The example project includes four application modules (`A`, `B`, `C`, and `D`), and three test modules (`X`, `Y`, and `Z`). An application module contains all the code needed to create a binary but does not contain any tests. A test module on the other hand contains only test code, so test modules are not part of the final software released to customers. In the rest of this chapter, we use the terms *build node* to represent an application module and *test node* to represent a test module, as we view them as nodes in a dependency graph.

In the figure, a directed edge from one node to another indicates that the former node depends on the latter node. For example, the directed edge from `C` to `A` indicates that `C` is dependent on `A`. Similarly, directed edges can exist between test nodes and build nodes, e.g., `Y` depends on `D`. In this case, either tests in `Y` directly test code from `D`, or tests in `Y` utilize helper utility functions from `D`. We do not allow build nodes to depend on test nodes. The dashed line from a test node to the

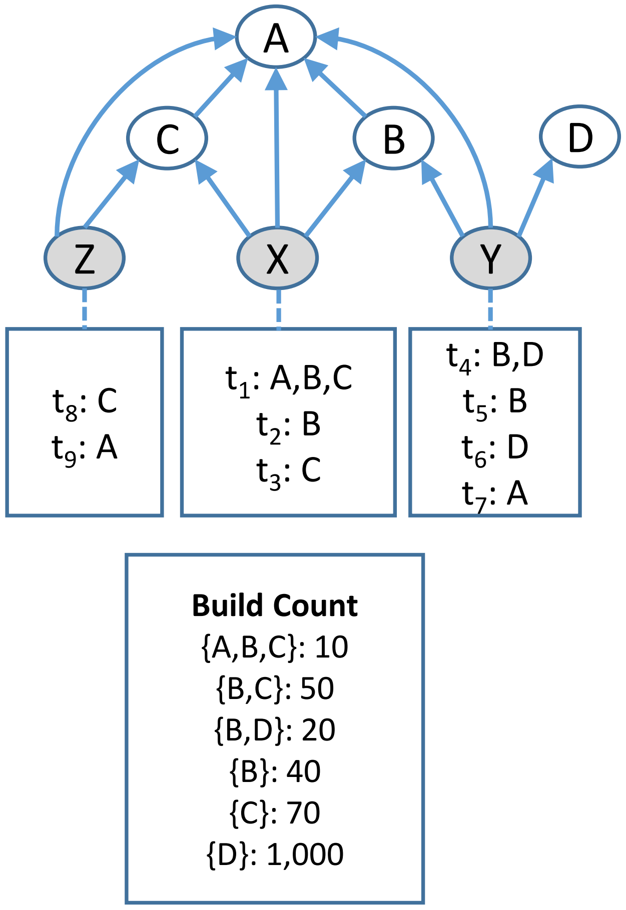

Figure 3.1: An example dependency graph with build nodes `A`, `B`, `C`, and `D`; test nodes `X`, `Y`, and `Z`, actual dependencies for tests t_1 through t_9, and the build count (number of times built) for sets of build nodes

box with test names indicates that the test node contains those tests. For example, the dashed line from `X` to the box with tests t_1, t_2, and t_3 means `X` contains those three tests. Furthermore, within the box, the listed nodes after the ":" for each test are the actual dependencies the test needs when executing, e.g., t_1 uses build nodes `A`, `B`, and `C`.

When a change happens to the code in a build node, the build node is built, and all build nodes and test nodes transitively dependent on that build node are built as well. Whenever a test node is built, all the tests inside it are also executed. For example, when test node `Z` is built, tests t_8 and t_9 are executed. The dependencies in a project's dependency graph control the order in which nodes are built. For example, `X` cannot be built before nodes `A`, `B`, and `C` are built.

Suppose that a developer makes a change to code in build node `D`. As such, `D` is built, and test node `Y`, which depends on `D`, is built as well, and all the tests in `Y` are executed. However, as shown in Figure 3.1, tests t_5 and t_7 do not actually depend on D, and they need not be executed when D changes. These two tests are executed because they happen to be placed in Y. This inefficiency is even worse if the dependencies change more frequently. The table "Build Count" in Figure 3.1 shows the number of times the exact set of build nodes are built together, e.g., the set of build nodes {A,B,C} are built 10 times together. We see that, in this example, build node D by itself is built 1,000 times. As Y depends on D, it is built at least 1,000 times, hence all tests in test node Y are executed at least 1,000 times. To be more precise, the tests are executed 1,120 times, because these tests also need to be executed when D is built in combination with other build nodes and when B is built as well, resulting in additional 120 times (i.e., the tests are executed when sets {A,B,C}, {B,C}, {B,D}, and {B} are built).

3.2 PROBLEM STATEMENT

Let $\mathcal{B}$ be the set of all *build nodes* for the project. Let $\mathcal{N}$ be the set of all *test nodes* for the project. Let $\mathcal{T}$ be the set of all the *tests* for the project, contained in the set of test nodes. Let Π be a partitioning of $\mathcal{T}$, and for a test node n, let $\Pi(n) \subseteq \mathcal{T}$ be the set of tests contained in n. Given that Π is a partitioning of $\mathcal{T}$, $\Pi(n_1) \cap \Pi(n_2) = \emptyset$ for distinct test nodes n_1 and n_2. For each test t, let $TDeps(t) \subseteq \mathcal{B}$ be all the build nodes that t depends on – these are the build nodes t needs to both compile and to execute. Whenever a test node n is built, all the tests in $\Pi(n)$ are executed.

Consider a sequence of builds during a given range of time R (say, over a six-month period). There are two factors that influence the total number of test executions for a project during R: (1) the count of the number of times test nodes are built over R (given the incremental nature of a build, not every test node in a project is built in a given build), and (2) the number of tests executed when building a test node.

Our solution changes the partitioning Π of tests by moving tests from existing test nodes to other (possibly new) test nodes. To compute the reduction in the number of test executions, we first need to compute the number of times test nodes are built (the test nodes' build counts) after a change in partitioning Π without actually replaying the builds during R.

3.2.1 Computing Build Count for a Test Node

Let us assume we have collected the build count for all the build nodes and test nodes during a past range of time R. To determine the build count of a test node n after moving tests across test nodes, one possibility is to simply reuse the build count of n collected in that range of time R. However, simply reusing the build count is inaccurate for two main reasons. First, after moving tests between test nodes, there can be changes to the dependencies of each existing test node. Second, new test nodes can be added, which do not have any existing build count information. Recall the example in Figure 3.1. If test t_8 is moved out of `Z`, that movement removes the dependency that test node `Z` has on `C`. This modified test node cannot be expected to have the same build count as before, since a test node is built when at least one of its dependencies change. Similarly, if t_7 is moved out of `Y` and into a brand new test node that depends only on `A`, we would not have any collected build count for this new test node.

However, in most cases we can accurately compute the build count of a given test node entirely in terms of the build count of the build nodes in $\mathcal{B}$ and the dependencies of the tests within the given test node. We make the following assumptions during the range of time R:

1. The dependencies of individual tests themselves do not change.
2. Tests are changed, added or removed to the test nodes only in conjunction with another change to a dependent build node, i.e., a change does not affect test nodes only.

From our experience with CloudBuild, we believe these assumptions hold for most test nodes. For the first assumption, given that the dependencies in question are modules in the project as opposed to finer-grained dependencies such as lines or files, it is unlikely that there are drastic changes that lead to changes in dependencies at the level of modules. For the second assumption, we find that it is very rare for developers to be only changing test code; they change code in build nodes much more frequently and changes made to test nodes are in response to those build node changes. As we explain below, given these assumptions, we can compute precisely when a test node n will be built, namely whenever at least one dependency in $\mathit{TDeps}(t)$ for any test t in n is built. Although these assumptions may not hold for a very small fraction of test nodes during the time frame R, their effect is negligible when R is sufficiently large (e.g., a few months).

We define the dependencies of a test node n as the union of the dependencies of the tests contained in n given a partitioning Π.

Definition 3.1. For a given test node n, the set of build nodes that n depends on under Π is defined as:

$$\mathit{NDeps}(n, \Pi) \doteq \{b \in \mathcal{B} \mid \exists t \in \Pi(n) \text{ such that } b \in \mathit{TDeps}(t)\} \tag{3.1}$$

Given our assumptions about test nodes, a test node n is built *if and only if* a build node in $\mathit{NDeps}(n, \Pi)$ is built.

For a subset $\mathcal{B}' \subseteq \mathcal{B}$, let $BC_R(\mathcal{B}')$ denote the number of builds where only the build nodes in $\mathcal{B}'$ are built together. The box in Figure 3.1 titled "Build Count" shows the number of times each subset of build nodes is built, e.g., the exact subset {`A`, `B`, `C`} was built 10 times in the range of time. Given our assumptions of test nodes not changing, a test node n is built in a build iff any of its dependencies is built. We can thus compute the number of times a test node n is built by summing up the build counts of $BC_R(\mathcal{B}')$ where $\mathcal{B}'$ intersects with $NDeps(n, \Pi)$.

Definition 3.2. For a given test node n, the computed number of times the nodeis built during a range of time R is defined as:

$$NodeCount_R(n, \mathit{\Pi}) \doteq \sum_{\{\mathcal{B}' \subseteq \mathcal{B} \mid \mathcal{B}' \cap NDeps(n, \mathit{\Pi}) \neq \emptyset\}} BC_R(\mathcal{B}') \tag{3.2}$$

Since the set of all builds in R can be partitioned by using the distinct subsets $\mathcal{B}'$ as identifiers, we count each build exactly once in the above equation.

3.2.2 Number of Test Executions in a Project

Our metric for the cost of testing is the number of tests that are executed over R. As such, our definition of the testing cost for a test node is related to the number of tests in the test node and the number of times the test node would be built:

Definition 3.3. The number of test executions for a test node n within a range of time R is the product of the number of tests in n and the build count of n within R:

$$NodeCost_R(n, \mathit{\Pi}) \doteq |\mathit{\Pi}(n)| \times NodeCount_R(n, \mathit{\Pi}) \tag{3.3}$$

The total number of test executions for testing the entire project would then be the sum of the number of test executions for each test node.

Definition 3.4. The number of test executions $Cost_R(\mathcal{N}, \mathit{\Pi})$ needed for building all test nodes $\mathcal{N}$ in a project within a range of time R is defined as:

$$Cost_R(\mathcal{N}, \mathit{\Pi}) \doteq \sum_{n \in \mathcal{N}} NodeCost_R(n, \mathit{\Pi}) \tag{3.4}$$

3.2.3 Reducing Test Executions

Our goal is to reduce the number of test executions in a project. While we could formulate the problem to allow placement of tests to any test node irrespective of the initial placement Π, the resulting ideal placement of tests could be very different from the original placement of tests.

Such a placement would then result in too many suggested test movements for the developers to implement, and it is not practical to invest huge effort (although one-time) in achieving this ideal placement of tests.

We consider the option of *splitting* a test node n into two test nodes $\{n, n'\}$ ($n' \notin \mathcal{N}$) where a subset of tests from n are moved to n'. As such, we can constrain the number of suggested test movements to report to developers. We define Π^n (over $\mathcal{N} \cup \{n'\}$) to be identical to Π at $\mathcal{N} \setminus \{n, n'\}$, and the disjoint union $\Pi^n(n) \uplus \Pi^n(n') = \Pi(n)$, meaning that Π^n only moves a subset of tests (possibly empty) from n to n'.

Our test placement problem can now be restated as the following decision problem:

$$\text{Does there exist } n \in \mathcal{N} \text{ such that } Cost_R(\mathcal{N}, \mathit{\Pi}) > Cost_R(\mathcal{N} \cup \{n'\}, \mathit{\Pi}^n) + c? \tag{3.5}$$

where c is a constant value representing the minimal number of test executions to reduce.

The threshold c acts as a knob for controlling suggestions, where a split is suggested only when the reduction in number of test executions is worth the overhead of the developer implementing the suggestion. Essentially, c represents the minimal return-on-investment that can be expected by a developer to implement the suggested split. With multiple such n that reduce the number of test executions by at least c, the one that provides the highest reduction can be chosen first.

3.3 TESTOPTIMIZER

Our technique, called TestOptimizer, provides a practical solution to reduce the number of wasteful test executions. More specifically, TestOptimizer produces a ranked list of suggestions for moving tests that help reduce a large number of wasteful test executions with minimal test movements. Our technique also allows developers to specify a threshold c in terms of the number of test executions that should be reduced for a suggestion. Using our suggestions, developers can also incrementally address the problem, thereby avoid investing a one-time huge effort. The suggestions also include additional recommendations where tests can be moved into existing test nodes, in case such test nodes already exist in the dependency graph. We first present how TestOptimizer splits a single test node into two test nodes and then explain how to use the single-split algorithm to deal with all test nodes in a given project.

3.3.1 Splitting a Test Node

Given a test node, we seek to find a subset of tests that can be moved to a new test node, resulting in the highest reduction in number of test executions. We use an iterative greedy algorithm to find

Algorithm 3.1: Splitting a test node

```
Input: Test node $n$
Input: Partitioning $\Pi$
Input: Range of time $R$
Output: New partitioning of tests $\Pi^n$
Output: New test node $n'$
1  $groups \leftarrow NewMap()$;
2  for each $t \in \Pi(n)$ do
3      $deps \leftarrow TDeps(t)$;
4      if $\neg groups.ContainsKey(deps)$ then
5          $groups[deps] \leftarrow \emptyset$;
6      end
7      $groups[deps] \leftarrow groups[deps] \cup \{t\}$;
8  end
9  $n' \leftarrow NewNode()$;
10 $\Pi^n \leftarrow NewPartition(\Pi, n')$;
11 repeat
12     $tests \leftarrow \emptyset$;
13     $maxCost \leftarrow Cost_R(\{n, n'\}, \Pi^n)$;
14     for each $deps \in groups.Keys$ do
15         $\Pi' \leftarrow \Pi^n$;
16         $RemoveTests(\Pi', n, groups[deps])$;
17         $AddTests(\Pi', n', groups[deps])$;
18         $newCost \leftarrow Cost_R(\{n, n'\}, \Pi')$;
19         if $newCost < maxCost$ then
20             $tests \leftarrow groups[deps]$;
21             $maxCost \leftarrow newCost$;
22         end
23     end
24     if $tests \neq \emptyset$ then
25         $RemoveTests(\Pi^n, n, tests)$;
26         $AddTests(\Pi^n, n', tests)$;
27     end
28 until $tests = \emptyset$;
29 return $\Pi^n, n'$
```

such subset of tests that can be moved from the given test node.

Algorithm 3.1 shows the individual steps. Given a test node n, the loop in lines 2 to 8 iterates over the individual tests in that test node. For each test t, our algorithm gets the build nodes the test depends on using $TDeps(t)$. If an existing group of tests already shares the same set of build nodes as dependencies, the test is added to that group; otherwise, a new group is created.

Next, the algorithm simulates moving groups of tests into another (initially empty) test node so as to identify the group that results in the highest reduction. At line 9, the algorithm makes a new test node n', and at line 10, the algorithm makes a new partitioning Π^n that includes n'. The loop in lines 14 to 23 iterates through the groups and simulates moving each group of tests from n to n'. The simulation is done by using a temporary partitioning Π' that starts as Π^n but is modified by using $RemoveTests$ to remove a group of tests from n and then using $AddTests$ to add the same group of tests to n'. The algorithm chooses the group of tests whose movement results in the highest reduction in the number of test executions. The outer loop (lines 11 to 28) greedily chooses to move that group of tests from n into n' (lines 24 to 27) by modifying that new partitioning Π^n. The outer loop iterates until there are no groups that help reduce the number of test executions. When the loop terminates, the algorithm returns the new partitioning Π^n and the new test node n'. In case n cannot be split, Π^n will map n' to an empty set. Our algorithm moves groups of tests instead of individual tests in each iteration due to two reasons. First, moving groups of tests instead of individual tests can make the search faster. Second, since the group of tests all share the same dependencies, these tests are likely related to each other and so the algorithm should place them in the same test node.

In the example dependency graph shown in Figure 3.1, consider the given test node n as `Y`. Table 3.1 shows the result after each iteration of the outer loop (lines 11 to 28). Initially, the first loop (lines 2 to 8) identifies four groups of tests, shown as g_1 to g_4. The initial number of test executions computed using the $Cost_R$ function is $4,480$. At the end of the first iteration of the outer loop, group g_4 is selected as the group that results in the highest reduction in the number of test executions, resulting in $3,370$ test executions. Note that g_4 includes the test t_7 that depends only on `A`, which is the root node of the dependency graph. Therefore, it is clearly evident that a large number of wasteful test executions is due to the test t_7. After one more iteration, the algorithm returns the partitioning that maps test node n to tests in groups g_1 and g_3 and maps new test node n' to tests in groups g_2 and g_4; this results in a final $2,480$ test executions.

3.3.2 Handling all Test Nodes

Given a set of test nodes, TestOptimizer applies Algorithm 3.1 to each test node to find a split that results in the highest reduction. In case the algorithm returns a partitioning where the new

Table 3.1: Steps in splitting test node Y

Iteration	Node Y	Node Y_1	# Test Execs
1	$g_1 : \{t_4\}$ $g_2 : \{t_5\}$ $g_3 : \{t_6\}$ $g_4 : \{t_7\}$		4,480
2	$g_1 : \{t_4\}$ $g_2 : \{t_5\}$ $g_3 : \{t_6\}$	$g_4 : \{t_7\}$	3,370
3	$g_1 : \{t_4\}$ $g_3 : \{t_6\}$	$g_4 : \{t_7\}$ $g_2 : \{t_5\}$	2,480

test node is mapped to no tests, TestOptimizer considers that the test node under analysis cannot be split further. In each iteration, TestOptimizer finds the test node that has the best split, i.e., producing the highest reduction in the number of test executions. TestOptimizer then updates the overall partitioning with the returned partitioning for that best split and adds the new test node into the set of all test nodes. TestOptimizer repeats these steps until there exists no test node that can be split any further. At the end, TestOptimizer returns all suggestions ranked in descending order based on their reductions in number of test executions. If there are any new test nodes that share the exact same dependencies as another test node (new or existing), TestOptimizer also makes the suggestion to combine the two test nodes into one.

TestOptimizer also allows developers to specify thresholds for a split. From Section 3.2.3, this threshold value is c, representing the minimal number of test executions by which a split must reduce. The threshold is implemented by adding a condition to line 19 in Algorithm 3.1. This condition can help eliminate trivial suggestions that may not be worthwhile of the effort required to implement the suggestion.

Returning to our running example, consider the threshold c as 100 test executions. TestOptimizer first splits Y, creating new test node Y_1, following the steps as shown in Table 3.1. TestOptimizer next splits Z, creating new test node Z_1, where Z now only includes t_8, and Z_1 includes t_9. TestOptimizer finally splits Y_1, creating new test node Y_2; Y_1 now includes t_5, and Y_2 includes t_7. TestOptimizer then terminates, because it cannot split any test node that reduces the number of test executions by at least the threshold value 100. TestOptimizer also suggests the two test nodes Y_2 and Z_1, which contain tests t_7 and t_9, respectively, can be combined into one test node A_1, because both the tests share exactly the same dependency. Figure 3.2 shows the final dependency graph after applying our suggestions, Overall, for this example, TestOptimizer reduces the number of test executions to $2,250$.

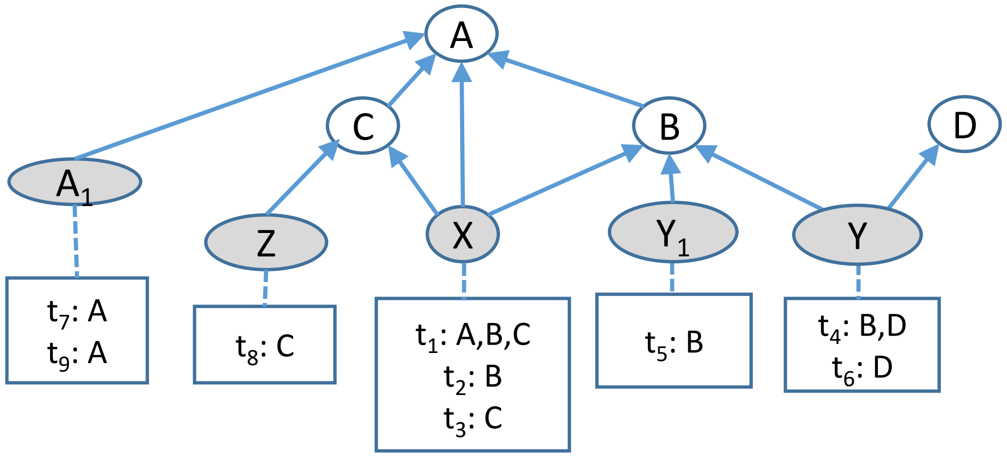

Figure 3.2: Final dependency graph after applying our suggestions

3.4 IMPLEMENTATION

We implement TestOptimizer as a prototype on top of CloudBuild [67].

3.4.1 Code Coverage

For our implementation, we target tests written using the Visual Studio Team Test (VsTest) framework [2], as the majority of tests in CloudBuild are executed using VsTest. Since TestOptimizer requires actual dependencies of each individual test, we use the Magellan code coverage tool [8] to collect those dependencies. In particular, we first instrument the binaries in a project, where a binary corresponds to a build node, using Magellan. We then execute the tests on the instrumented binaries and save a coverage trace for each test. The coverage trace includes all blocks that are executed by the test. We map these blocks to their original binaries and therefore to the build nodes to construct the set of actual dependencies for each test. Our implementation also handles the special constructs such as `AssemblyInitialize` and `ClassInitialize` that have specific semantics in VsTest. For example, `AssemblyInitialize` is executed only once when running all the tests in a test node, but the binaries exercised should be propagated as dependencies to all the tests in that test node.

3.4.2 Test Node Affinity

We use *affinity* to refer to the set of build nodes that are intended to be tested by a test node. Affinity helps avoid suggesting moving tests that are already in the right test node and also helps improve the scalability of our technique. Ideally, one would ask the developers to provide the set of build nodes that each test node is intended to test. However, as it is infeasible to ask the developers to spend time labeling every single test node, we instead develop a heuristic to automatically compute the affinity for any given test node n.

First, we compute the dependencies $\mathit{NDeps}(n, \Pi)$ of the test node. Next, we compute $\Gamma(b, n)$ of each $b \in \mathit{NDeps}(n, \Pi)$ as the number of tests in n that covers b during their execution.

$$\Gamma(b, n) = |\{t \in \Pi(n) \mid b \in \mathit{TDeps}(t)\}| \tag{3.6}$$

Finally, let $\Gamma^{Max}(n) = \max_{\{b \in \mathit{NDeps}(n,\Pi)\}} \Gamma(b, n)$, the maximum Γ value among all build nodes in n. We compute the affinity as:

$$\mathit{Affinity}(n) = \{b \in \mathit{NDeps}(n, \Pi) \mid \Gamma(b, n) = \Gamma^{Max}(n)\} \tag{3.7}$$

Once the affinity is computed, any test that does not exercise all dependencies in *Affinity*(n) is considered to be amenable for movement, called an *amenable test*. In the context of Algorithm 3.1, affinity can be implemented by modifying line 2 to skip tests that exercise at least all dependencies in *Affinity*(n). Our experimental results show that affinity helps exclude a large number of tests from our analysis, and it also provides more suggestions that are more likely to be accepted by developers. Furthermore, we find that developers tend to agree with the affinity computed for each test node (Section 3.5.4), as our heuristic identifies the affinity correctly for ≈99% of the test nodes in our evaluation projects.

3.4.3 Output

TestOptimizer generates an HTML report with all suggestions for test movements. The report displays metrics concerning the number of test executions with the current placement of tests and the build count for the current test nodes. For each test node with amenable tests, the report suggests how to split the test node to reduce the number of test executions. Furthermore, the report also suggests any existing test nodes where the tests can be moved into, where these existing test nodes share the exact same dependencies as the tests to be moved. The report ranks the suggestions starting with the test nodes that achieve the highest reductions. This ranked list can help developers in prioritizing their effort. The report also includes the test nodes that were found to not contain any tests amenable for movements as to give the developer a more complete picture. Anonymized reports for all evaluation projects (including the running example) are publicly available[2].

[2]`http://mir.cs.illinois.edu/awshi2/testoptimizer`

Table 3.2: Statistics of evaluation projects used in our evaluation

Evaluation project	**KLOC**	**# Build Nodes**	**# Test Nodes**	**# Tests**	**# Build Count Entries**
ProjA	468	431	135	7,228	297,370
ProjB	926	416	90	5,870	127,030
ProjC	1,437	574	111	7,667	810,030
ProjD	3,366	1,268	184	12,918	6,522,056
ProjE	29,058	8,540	173	17,110	34,486,308
Overall	35,255	11,229	693	50,793	42,242,794
Average	7,051	2,245	138	10,158	8,448,558

3.5 EVALUATION

In our evaluation, we address the following three research questions:

RQ1 How many test executions can be saved by applying TestOptimizer suggestions?

RQ2 How much time does TestOptimizer take to run?

RQ3 What is the developer feedback for the suggestions of TestOptimizer?

The goal of RQ1 is to evaluate how effective TestOptimizer suggestions are at saving test executions. The goal of RQ2 is to evaluate how scalable TestOptimizer is when run on large projects, particularly at the size of projects in Microsoft. The goal of RQ3 is to evaluate whether TestOptimizer suggestions are actually acceptable for developers and to better understand reasons for why the suggestions were needed in the first place.

3.5.1 Experimental Setup

We apply our technique on five medium to large proprietary projects that use CloudBuild as the underlying build system. Table 3.2 shows the statistics of our evaluation projects. For confidentiality reasons, we refer to our evaluation projects as ProjA, ProjB, ProjC, ProjD, and ProjE. The row "Overall" is the sum of all the values in each column. The row "Average" is the arithmetic mean of all the values in each column. All evaluation projects primarily use C# as the main programming language, but also include some code in other languages such as C++ or Powershell. Column 2 shows the size of C# code in each evaluation project. As shown in the table, our evaluation projects range from medium-scale (468 KLOC) to large-scale projects (29,058 KLOC). Column 3 shows the number of build nodes, and Column 4 shows the number of test nodes. Column 5 shows the number of (manually written) tests in each evaluation project.

Table 3.3: Results of applying TestOptimizer on our evaluation projects

Project	# Amenable Test Nodes	# Amenable Tests	# Moved Tests	# New Test Nodes	# Orig Test Execs (millions)	Reduced Test Execs #	Reduced Test Execs All %	Reduced Test Execs Amenable %	Time to Analyze (min)
ProjA	20	1,343	1,047	25	9.9	635,815	6.4	10.6	12.9
ProjB	5	328	291	4	2.4	46,393	2.0	5.5	0.4
ProjC	15	364	333	12	17.1	779,523	4.6	13.9	11.9
ProjD	15	358	312	13	63.6	2,377,208	3.7	7.8	315.4
ProjE	30	1,553	1,250	28	18.3	1,498,045	8.2	16.5	994.3
Overall	85	3,946	3,233	82	111.2	5,336,984	4.8	10.2	1,335.0
Average	17	789	646	16	22.2	1,067,396	5.0	10.9	267.0

For each evaluation project, we collect historical data about the number of times sets of build nodes are built in a previous range of time. CloudBuild maintains this information about each build (including the details of build nodes and test nodes that are scheduled) in a SQL Server database. Using this database, we compute the "Build Count" information for sets of build nodes during the time period of 180 days starting from Feb 1^{st}, 2016. Column 6 shows the number of table entries in the database for each evaluation project, where each entry is a set of build nodes for the evaluation project and the number of times those build nodes were built together. From Figure 3.1, this number would correspond to the number of entries in the box titled "Build Count" for each evaluation project. The historical data is over a period of 180 days for all evaluation projects, except ProjE. We could not collect 180 days worth of historical data for ProjE, our largest evaluation project, due to out-of-memory errors, since our tool performs all computations in memory. For ProjE, we use 30 days worth of historical data (from the same start date).

We configure TestOptimizer to split a test node only if the reduction is at least $2,000$ test executions (setting the threshold value c to be $2,000$). We use this value based on our discussions with the developers of some of our evaluation projects. Finally, although we compute the reduction based on the historical data over a range of time R in the past, we present to developers the results as potential savings for a future range of time R (as future savings matter more to developers), under the assumption that the past is a good indicator for the future.

3.5.2 RQ1: Savings in Test Executions

Table 3.3 presents the results showing the reduction in terms of number of test executions after applying TestOptimizer. Once again, the row "Overall" is the sum of all the values in each column, and the row "Average" is the arithmetic mean of all the values in each column.

Column 2 shows the number of test nodes where TestOptimizer finds amenable tests, based on affinity (Section 3.4.2). Test nodes with amenable tests are *amenable test nodes*. Column 3 shows the number of amenable tests. The percentage of amenable test nodes range from 5.6% (5 / 90 in ProjB) to 17.3% (30 / 173 in ProjE). These results indicate that developers often ensure that tests

Table 3.4: Results when considering spurious dependencies

	Spurious Deps		**# Test**	**Reduced Test Execs**	
Project	**# Test Nodes**	**# Deps**	**Execs (millions)**	**#**	**%**
ProjA	8	12	10.0	796,250	7.9
ProjB	35	44	2.4	82,498	3.4
ProjC	36	43	20.4	4,842,269	20.0
ProjD	68	116	69.7	8,497,928	12.2
ProjE	115	190	24.2	7,445,681	30.8
Overall	262	405	126.7	21,664,626	17.1
Average	52	81	25.3	4,332,925	14.9

are placed in the correct test node. However, the fact that there are tests in incorrect test nodes suggests that developers can still make mistakes as to where the tests belong to, especially if they lack a global view of the project, so there is still room for improvement. Up to 3,946 tests across all evaluation projects can be moved as to reduce the number of test executions.

Column 4 shows the number of tests TestOptimizer suggests to move, meaning their movement can provide a substantial reduction in the number of test executions. Column 5 shows the number of new test nodes that need to be created for the tests to be moved into (not including any existing test nodes that already exactly share the dependencies of the tests to be moved). Column 6 shows the number of test executions (in millions) for the original placement of tests based on historical data. Columns 7-9 show the reductions in number of test executions if developers implement the suggestions. Column 7 shows the reduction in terms of number of test executions, while Column 8 shows the percentage of reduction when compared against the original number of test executions (Column 6 from Table 3.2). However, these columns show the reduction relative to the number of test executions of *all* test nodes in the evaluation project; there are many test nodes that TestOptimizer finds as non-amenable test nodes. Column 9 also shows percentages but compared against the number of test executions concerning *only* the amenable test nodes, essentially compared against only the test nodes that TestOptimizer can actually suggest movements from. When considering only the amenable test nodes, we see the percentage of reduction in number of test executions is higher compared with the reductions based on all the test nodes. In the case of these percentages, the row "Overall" is computed as the total reduction in number of test executions across all evaluation projects over the total number of original test executions. As such, the overall percentage reduction in number of test executions for amenable tests is 10.2%.

Spurious Dependencies In our results, we find that some of the test nodes have additional developer-specified dependencies that are not required by *any* tests inside that test node. We refer to such dependencies as *spurious dependencies*. The primary reason for spurious dependencies could be due to the evolution of the application code. More specifically, developers could have originally specified a test node to have some dependencies, but after code changes the tests were moved around so that the test node no longer depends on some of those developer-specified dependencies. When a spurious dependency is built, the dependent test node is built unnecessarily, causing a number of wasteful test executions. A spurious dependency can be simply removed and all the tests within the test node will still run properly. Given TestOptimizer, spurious dependencies are the build nodes declared as dependencies in the build specification for a test node but are not covered by the test node's tests. Table 3.4 shows the number of test nodes that have spurious dependencies in each evaluation project (Column 2) along with the total number of spurious dependencies those test nodes depended on (Column 3).

Although detecting spurious dependencies is not a core contribution, it is an additional advantage of our technique. The reduction in the number of test executions after both moving tests and removing spurious dependencies is the reduction developers would actually obtain. Table 3.4 shows the effects of having spurious dependencies and the reduction in the number of test executions for each evaluation project. Column 4 shows the number of test executions (in millions) for each evaluation project using developer-specified dependencies for each test node. The number of test executions is higher than the values shown in Table 3.3, as they include the effects of these spurious dependencies. Columns 5 and 6 show the reduction in number of test executions for each evaluation project after the suggested test movements from TestOptimizer relative to the number of test executions obtained using the developer-specified dependencies. We also see cases where spurious dependencies seem to be a big problem.

To give an estimate of how the reduction in the number of test executions maps to savings in terms of machine time, we calculate the average time (154 ms) across all the tests of the five evaluation projects. Using this average test execution time and the reduction of number of test executions, our results show that TestOptimizer can help save 38.6 days of machine time.

A1: TestOptimizer can help reduce millions of test executions (up to 2,377,208) among our evaluation projects. By considering only the amenable test nodes, these reductions range from 5.5% to 16.5% among our evaluation projects, overall 10.2% across all evaluation projects. However, when we consider the effects of spurious dependencies, TestOptimizer can reduce the number of test executions by 21.7 million (17.1%) across all evaluation projects.

3.5.3 RQ2: Time to Run

In Table 3.3, Column 10 shows the time (in minutes) taken by TestOptimizer to analyze the historical data and to run through its algorithm to suggest movements for each evaluation project.

Table 3.5: Feedback from developers of four evaluation projects

Project	Sugg.	Accepted Sugg.	Valid/Rejected Sugg.	Invalid Sugg.
ProjA	20	16	3	1
ProjB	5	4	1	0
ProjC	5	5	0	0
ProjD	15	13	0	2
Overall	45	38	4	3
Average	11	9	1	1

Our results show that the analysis time ranges from 0.4 minutes up to 994.3 minutes ($\approx$16 hours). TestOptimizer takes on average 267.0 minutes per evaluation project, while overall 1,335.0 minutes across all evaluation projects. TestOptimizer's running time seems to be a factor of the number of build nodes, test nodes, tests, and the amount of historical data available. We find this time can still be reasonable as we envision our technique to be run infrequently. We also envision TestOptimizer can be run incrementally by only analyzing newly added tests as to suggest to developers the best test node to place new tests; TestOptimizer can work quickly when analyzing only a small number of tests. Furthermore, TestOptimizer is still a prototype, and it can be improved by implementing lazy loading of data and caching previous computations to avoid repeated calculations of build count.

A2: TestOptimizer takes on average 267.0 minutes (4.5 hours) per evaluation project, and overall 1,335.0 minutes (22.3 hours) across all evaluation projects. Such times are reasonable considering our intent is to run TestOptimizer infrequently, suggesting one-time test movements.

3.5.4 RQ3: Developer Feedback

Regarding RQ3, we approached the developers of our evaluation projects to receive their feedback on suggested test movements. We received feedback from the developers of four of our evaluation projects: ProjA, ProjB, ProjC, and ProjD. Table 3.5 presents the results for each of these four evaluation projects. Column 2 shows the number of test movement suggestions reported by our tool, which is counted by the number of test nodes where our tool found tests amenable for movement. Column 3 shows the number of suggestions accepted by developers, and they intend to implement the suggestions. Column 4 shows the number of suggestions that the developers considered as valid, but they do not intend to implement the suggestions. Finally, Column 5 shows the number of suggestions that the developers considered as invalid. As shown in our results, 84.4% of the suggestions were indeed accepted by the developers. Furthermore, among the 16 accepted suggestions in ProjA, developers already implemented six of the suggestions.

Overall, the developer feedback is highly encouraging. The feedback also helps us understand how code evolution resulted in wasteful test executions in the case of some of the test nodes where TestOptimizer suggests test movements. For example, a developer informed us that some application code was moved from one build node to a new build node as a part of a major refactoring. However, the related test code was not moved into the relevant test node. Since the original test node still had a dependency on the new build node, tests continued to execute properly, but the tests would also execute when the build system builds any other dependency the original test node had. After analyzing our suggestions, the developer felt that our technique could also be quite helpful finding a better organization of the tests. This response is encouraging, because it demonstrates TestOptimizer's ability in addressing issues beyond wasteful test executions. Another feedback was that developers may not be aware of an existing test node better suited for their tests, especially with many developers and many test nodes in the project. Therefore, developers tend to place tests in some test node they are familiar with; they later add more dependencies to that test node, eventually leading to wasteful test executions. Developers also appreciate the idea that our suggestions can help break edges from test nodes to build nodes in the dependency graph, thereby reducing the build time along with the test execution time (breaking edges prevents test nodes from being built, which itself takes some time in the build system). The developers we approached also asked that we provide these reports at regular intervals (e.g., once per week), so they can monitor and improve the health of their project.

Accepted We present the two common scenarios under which developers tend to accept our suggestions (Column 3 in Table 3.5). We use the dependency graph shown in Figure 3.3 as an illustrative example. In the figure, test node `X` is dependent on build nodes `B` and `C`. The figure also shows the build counts of `B` and `C`, where the build count of `C` is much greater than the build count of `B`. The figure shows that the test node `X` includes 100 tests, where most of them are dependent on `B` and only a few (just one in this example) depend on `C`. The primary issue is that most of the tests in `X` are wastefully executed due to the high build count of `C`. In this scenario, our technique typically suggests to move the tests from `X` into a test node that is dependent only on `B`. We notice that developers tend to accept our suggestions in these scenarios, since they help reduce a large number of test executions in test nodes such as `X`.

Another common scenario where developers often accepted our suggestion is when there already exist a test node with the exact same set of dependencies where the tests can be moved. In this scenario, the effort involved in implementing the suggestion is minimal, i.e., developers just need to copy-paste the tests into the existing test node.

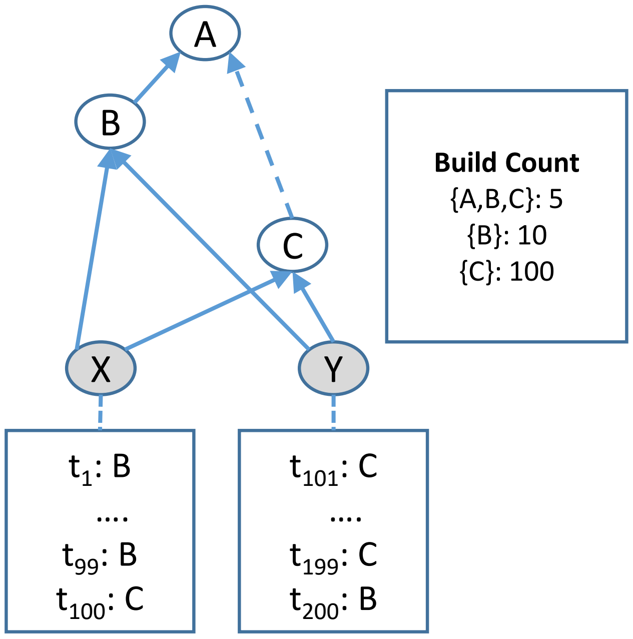

Figure 3.3: An example dependency graph illustrating the scenario where developers tend to accept our suggested movements

Valid/Rejected We present the common scenario under which developers considered that the suggestions are valid, but not willing to implement those suggestions (Column 4 of Table 3.5). We use the same dependency graph in Figure 3.3 as an illustrative example, this time focusing on test node `Y` instead. In `Y`, the majority of the tests have a dependency on `C` instead of `B`. Although this scenario could result in wasteful test executions with respect to tests such as t_{200}, it is not as substantial (since only a few tests are affected) compared to what occurs for tests in `X`. Due to the lower benefit in implementing the suggestion, developers seemed reluctant in moving such tests, especially when there is no existing test node the tests can simply be moved into.

Invalid Regarding the suggestions that are considered as invalid (Column 5 of Table 3.5), we find that they are primarily due to implementation issues. TestOptimizer relies on code coverage to collect test traces, in this case using Magellan. In case a test exercises a binary via only a constant or refers to a class using constructs such as `typeof`, we notice that Magellan does not collect the necessary dependency, a limitation in the code coverage tool. Due to such missing dependencies, our tool suggests invalid movements that were rejected by the developers. In the future, we plan to explore further how to collect dependencies in these scenarios.

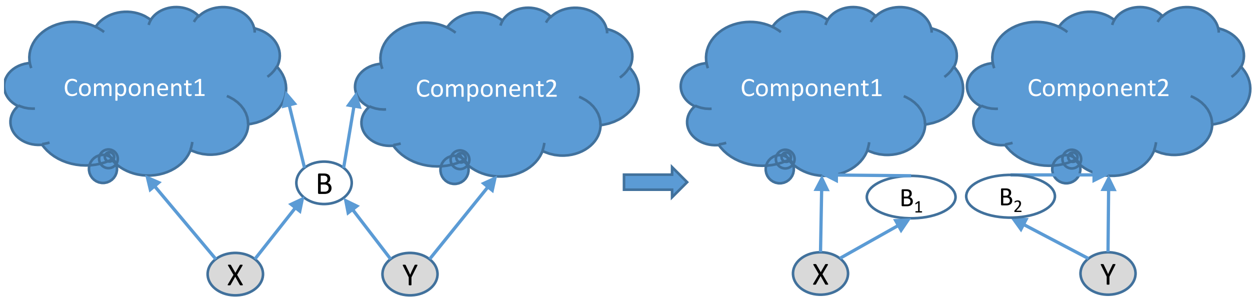

Figure 3.4: An example scenario where our suggestion leads to a higher-level refactoring of a build node

Interesting Scenario We finally present an interesting scenario where our suggestion led to a higher-level refactoring of a build node. Figure 3.4 shows two major components in a project, where each component contains several build nodes. There are two test nodes `X` and `Y`, which are intended to test build nodes in Component1 and Component2, respectively. However, due to the build node `B`, changes in Component1 can trigger tests in `Y`, and similarly changes in Component2 can trigger tests in `X`. In an attempt to implement our suggestion in `X`, instead of splitting `X`, the developer split the build node `B` into two build nodes $\texttt{B}_1$ and $\texttt{B}_2$. The resulting dependency graph is shown in the figure, where the split helped remove dependencies between the build nodes and test nodes. This feedback is very encouraging, because our suggestions are able to help developers make insightful decisions in removing major unnecessary dependencies in the project.

A3: Developers generally approved of TestOptimizer's suggestions, accepting 84.4% of the suggestions; developers already implemented six of the 16 accepted suggestions in ProjA.

3.5.5 Discussion

Formulation Given that our formulation of the problem of reducing the number of test executions over a range of time involves defining cost metrics for a given dependency graph, we could have formulated the problem instead as an optimization problem instead of a decision problem (i.e., does there exist a test node to split that achieves substantial reduction). The goal of the optimization problem would then be to determine an optimal placement of tests within test nodes as to achieve a minimal cost.

We initially experimented with such a formulation, encoding the entire problem into a set of SMT equations and then using Z3 [12], a state-of-the-art constraint solver, to solve for an optimal solution. However, in our initial experiments with such a formulation, we faced some challenges that lead to our eventual decision to change the formulation and use a greedy algorithm instead.

First, we encountered scalability issues with using Z3. While Z3 could very quickly output the solution for an optimal placement of tests for our small, toy examples, Z3 could not operate at the scale of the dependency graphs we eventually need for our evaluation on Microsoft evaluation projects. Running Z3 on the dependency graphs from our evaluation projects would take several hours and often would time out, leading to no results[3]. While TestOptimizer can be run infrequently and offline, the time to run using Z3 was still too long, especially when a time out occurs and Z3 cannot output any intermediate results to at least reduce some number of test executions (not necessarily the most optimal). Z3 is one of the most effective constraint solvers available, so we believe any other solver would lead to the same issues.

Second, we find that developers really appreciate a technique that can give them intermediate steps to take that make progress towards an optimal solution. Developers do not have time to perform all the necessary movements to achieve the minimal number of test executions, and they would prefer to make some changes incrementally, whenever they have time. As such, a model using a constraint solver to output the optimal solution at once does not give them any feedback on which tests to move first as to achieve the best reduction. The iterative greedy algorithm does provide this information. Furthermore, when we presented final optimal placements to developers, they were rather confused for certain placements; they would spend time trying to reason *why* TestOptimizer would suggest those movements. As such, we found they preferred a step-by-step process that shows them more information as to why TestOptimizer would suggest tests to move.

Cost Metric Instead of using the number of test executions as the cost metric to reduce, we could use overall time. Such a cost metric per test node would involve the expected running time for all tests contained in a test node *plus a fixed time* per test node to represent the overhead in setting up a test node (so such a constant can limit the creation of too many new test nodes).

We initially experimented with using such a cost metric, but we encountered misunderstandings concerning what the results actually entail when we approached developers. By telling developers about the time savings the suggestions would provide, they assumed that the time savings are *end-to-end* time savings. In CloudBuild, the nodes are not built sequentially but rather in parallel, so the moment an affected node's dependencies are available (and cloud resources are available), that node can be built. The end-to-end time is not the sum of the time to build all the affected nodes but rather the time for the longest path through the affected nodes in the dependency graph. Developers assumed this end-to-end time, but TestOptimizer actually aims to reduce the *total* time to build all affected nodes. This time to build all affected nodes represents the total machine time, which is effectively the monetary cost for building and testing. Given this clash in expectations

[3]In the process, our experiments actually served as benchmarks that helped the Z3 developers implement new strategies for improving the performance of Z3. Unfortunately, those improvements still were not enough to get Z3 to scale to our evaluation projects.

involving time, we find that presenting cost in terms of number of test executions is a good balance in the goal TestOptimizer aims to achieve and what developers would also want to reduce.

Comparison against Finer-Grained RTS A build system such as CloudBuild essentially performs RTS at a coarser granularity level of modules using dependency graphs. Although previous finer-grained RTS techniques can further reduce the number of test executions by doing the selection on tests in each test node, such finer-grained techniques are not as practical due to the following reasons. First, existing techniques assume that tests are executed at the end of the build. Therefore, even if there exists a single test (in a test node) that is affected by the change, the entire test node still needs to be built. In contrast to that, TestOptimizer can give suggestions that help entirely skip the build of that test node as well, if TestOptimizer can suggest some test movements that remove dependencies from the test node. Second, existing techniques require storage and maintenance of metadata, such as code coverage per individual test, which needs to be continuously updated along with the changes in the underlying modules. This aspect adds non-trivial overhead in the case of millions of test executions. Instead, TestOptimizer does not require any additional information beyond what the build system already saves.

3.6 THREATS TO VALIDITY

Our evaluation of TestOptimizer is on five Microsoft projects, so the results may not generalize to other projects. TestOptimizer explicitly works only for projects that use a build system like CloudBuild, where code and tests are divided into modules, and each is built separately on their own machine with only the developer-specified module-level dependencies. However, we see several companies other than Microsoft using a similar build system, such as in Facebook Buck [36] or Google Bazel [9], so TestOptimizer is applicable in those settings as well.

We compute the actual dependencies for each individual test dynamically using code coverage. While code coverage can get us exactly what each test executes and therefore needs at runtime, it may miss some dependencies if the test execution itself is nondeterministic. However, as we collect code coverage at the module level, even if some tests are nondeterministic, it is rather unlikely the tests are nondeterministic to the point that the code coverage tool misses to collect an entire module dependency for a test.

3.7 SUMMARY

In this chapter, we present TestOptimizer, a technique that helps reduce wasteful test executions due to suboptimal placement of tests. We formulate the problem of wasteful test executions and develop an algorithm for reducing the number of wasteful test executions. We implement our technique in a prototype tool on top of Microsoft's CloudBuild. We evaluate the effectiveness of TestOptimizer on five proprietary projects from Microsoft. The results show that TestOptimizer can reduce 21.7 million test executions (17.1%) across all our evaluation projects. Furthermore, developers of four of our evaluation projects accepted and intend to implement 84.4% of our suggestions; developers have already implemented some of these suggestions as well. Beyond saving machine resources, the reduction in test executions can also help reduce the developer effort in triaging the test failures, if these irrelevant tests are flaky.

CHAPTER 4: AUTOMATICALLY FIXING ORDER-DEPENDENT FLAKY TESTS

Aside from the cost of regression testing, another challenge facing regression testing is the presence of flaky tests. This chapter presents a framework for fixing order-dependent flaky tests, which were found to be a prominent type of flaky tests in open-source projects [128]. Section 4.1 formalizes the problem of order-dependent tests. Section 4.2 presents the iFixFlakies framework for automatically fixing order-dependent tests. Section 4.3 presents our experimental setup for evaluating iFixFlakies. Section 4.4 presents our evaluation results. Section 4.5 presents threats to validity. Finally, Section 4.6 concludes and summarizes the chapter.

4.1 FORMALIZATION OF ORDER-DEPENDENT TESTS

Order-dependent tests are flaky tests whose results can differ depending on the order in which the tests run. An order-dependent test consistently passes when run in one order but then consistently fails when run in a different order [118, 190].

Let $\mathcal{T}$ be the set of all tests[1] in the test suite. A *test order* is a sequence of a *subset* of tests from $\mathcal{T}$. For a test order O that has a test $t \in \mathcal{T}$, let $\text{run}_t(O)$ be the result of the test t when run in the test order O; the result can be either $PASS$ or $FAIL$ consistently. We ignore other flaky tests that have results $PASS$ and $FAIL$ when rerun in the same test order due to other sources of nondeterminism. We use $\text{run}(O)$ to refer to the result of the *last* test in O. We use $[t]$ to denote a test order consisting of just one test t, and use $O + O'$ to denote the concatenation of two test orders O and O' in a new test order consisting of the tests in O followed by the tests in O'.

Definition 4.1. A test $t \in \mathcal{T}$ has a **passing test order** or a **failing test order** O if $\text{run}_t(O) = PASS$ or $\text{run}_t(O) = FAIL$, respectively.

Definition 4.2. An **order-dependent test** $t \in \mathcal{T}$ has a passing test order O and a failing test order $O' \neq O$.

We classify an order-dependent test into one of two types: victim or brittle. We also classify other tests related to order-dependent tests into three different *roles*: polluter, cleaner, and state-setter.

4.1.1 Victim

A *victim* is an order-dependent test that consistently passes when run by itself in isolation from

[1]When we say *test*, for Java we mean *test method* as defined in JUnit.

```java
// Victim (in ShutdownListenerManagerTest class)
@Test
public void assertIsShutdownAlready() {
  shutdownListenerManager.new InstanceShutdownStatusJobListener().
    dataChanged("/test_job/instances/127.0.0.1@-@0", Type.NODE_REMOVED, "");
  verify(schedulerFacade, times(0)).shutdownInstance();
}
// Polluter (also in ShutdownListenerManagerTest class)
@Test
public void assertRemoveLocalInstancePath() {
  JobRegistry.getInstance().registerJob("test_job",
    jobScheduleController, regCenter);
  shutdownListenerManager.new InstanceShutdownStatusJobListener().
    dataChanged("/test_job/instances/127.0.0.1@-@0", Type.NODE_REMOVED, "");
  verify(schedulerFacade).shutdownInstance();
}
// Cleaner (in FailoverServiceTest class)
@Test
public void assertGetFailoverItems() {
  JobRegistry.getInstance().registerJob("test_job",
    jobScheduleController, regCenter);
  ... // 12 more lines
  JobRegistry.getInstance().shutdown("test_job");
}
```

Figure 4.1: Example victim, polluter, and cleaner from `elasticjob/elastic-job-lite`

other tests, and yet there exists a failing test order in which the test fails when run with one or more other tests.

Definition 4.3. An order-dependent test $v \in \mathcal{T}$ is a **victim** if run$([v]) = PASS$.

The reason why a victim fails in a failing test order is that there is at least one test that runs before the victim, and these tests "pollute" the state (e.g., global variable, file system, network [190]) on which the victim depends. We call such state-polluting tests *polluters*. Note that a polluter can consist of multiple tests, where the combination of running those tests in a certain order leads to the victim failing. More specifically, a polluter is a test order that leads the victim to fail when run before the victim. If a polluter consists of more than one test, then no suborder of the polluter leads the victim to fail, instead all tests of a polluter must run in that order before the victim for the victim to fail.

Definition 4.4. A test order (with one or more tests) P is a **polluter** for a victim v if run$(P+[v]) = FAIL$.

Figure 4.1 shows an example (identified by iFixFlakies) of a victim and a polluter from project `elasticjob/elastic-job-lite` [26]. The polluter is the test `assertRemoveLocalInstancePath` (or `PT` for short), because it starts the instance (Line 11) but does not shut it down at the end of the run. By not shutting down the instance, the victim is the test `assertIsShutdownAlready` (or `VT` for short) that fails on Line 6, which checks whether an instance of a class variable has been shut down. `VT` passes by itself or in test orders where a polluter that starts the instance, like `PT`, is run after `VT`.

A victim may not fail even when a polluter is run before it, as long as a cleaner is run between the two. Intuitively, a *cleaner* is a test order that resets the state polluted by a polluter; when the cleaner is run after a polluter and before its victim, the victim passes.

Definition 4.5. A test order C is a **cleaner** for a polluter P and its victim v if run$(P + C + [v]) = PASS$.

An example of a cleaner is also shown in Figure 4.1. The test `assertGetFailoverItems` (or `CT` for short) is a cleaner for `PT` and `VT`, because Line 23 of `CT` shuts down the instance that `PT` starts and `VT` checks. Therefore, even if `PT` runs before `VT`, as long as `CT`'s Line 23 successfully executes before `VT`, `VT` passes. We can fix `VT` by inserting the statement from this line of `CT` at the end of `PT`, or by inserting this line into a teardown method (annotated with `@After` in JUnit) in `ShutdownListenerManagerTest` that runs after every test. We sent this change as a pull request to the developers of `elasticjob/elastic-job-lite`, which they accepted [27].

4.1.2 Brittle

In contrast to a victim, an order-dependent test is a *brittle* if the test consistently fails when run by itself in isolation, and yet there exists a passing test order in which the test passes when one or more tests are run before the brittle.

Definition 4.6. An order-dependent test $b \in \mathcal{T}$ is a **brittle** if run$([b]) = FAIL$.

Intuitively, because a brittle fails in isolation and yet has a passing test order, then its passing test order must contain one or more tests that set up the state for the brittle to pass. We refer to a test order that sets up the state for a brittle as a *state-setter*.

Definition 4.7. A test order S is a **state-setter** for a brittle b if run$(S + [b]) = PASS$.

Figure 4.2 shows an example identified by iFixFlakies as a brittle and its corresponding state-setter from project `wildfly/wildfly` [33]. The test `testPermissions` (or `BT` for short) is a brittle, because it fails when run by itself, due to an `AccessControlException`. The test `testBind` (or `ST` for short) is a state-setter for `BT`, because running `ST` and then `BT` is enough to make `BT` pass.

```
 1 // Brittle (in WritableServiceBasedNamingStoreTestCase class)
 2 @Test
 3 public void testPermissions() throws Exception {
 4   ...
 5   final String name = "a/b";
 6   final Object value = new Object();
 7   try {
 8     ...
 9     store.bind(new CompositeName(name), value);
10   }
11   ...
12   assertEquals(value, testActionWithPermission(JndiPermission.ACTION_LOOKUP,
13     permissions, namingContext, name));
14 }
15 // State-setter (also in WritableServiceBasedNamingStoreTestCase class)
16 @Test
17 public void testBind() throws Exception {
18   final Name name = new CompositeName("test");
19   final Object value = new Object();
20   ... // 6 more lines
21   assertEquals(value, store.lookup(name));
22 }
```

Figure 4.2: Example brittle and state-setter from `wildfly/wildfly`

iFixFlakies finds that the `store.lookup` call of `ST` on Line 21 is the only method call that `BT` needs in order to pass. `store` is a test class variable that is initialized by the setup method (annotated with `@Before` in JUnit, meaning it runs before every test method is run) of the test class. When `store.lookup` is invoked before Line 12, `BT` passes. When we proposed this fix to the developers of `wildfly/wildfly`, they quickly accepted our fix and clarified that it works because the lookup call causes "the `WildFlySecurityManager.<clinit>` to run" and running this class constructor resolves the `AccessControlException` of `BT` [34].

Both cleaners (for victims) and state-setters (for brittles) help make order-dependent tests pass when they run in certain test orders. Hence, we refer to cleaners and state-setters as *helpers*. Our insight is that these helpers already contain logic to set the state for their corresponding order-dependent tests. Therefore, a patch to fix an order-dependent tests would involve changing the test code to include said logic.

Note that for a "good" patch that fixes an order-dependent test, we have three main requirements for the test after applying the patch. First, the test must still cover the same logic as before. Second, the test must have the same result regardless of the order in which it is run. Third, the test must not run substantially slower than before. Such requirements rule out simple "patches", such as removing the order-dependent test entirely (which goes against the first requirement that the test covers the same logic). Besides changing the test code itself through patches, there are other ways

```
1  def iFixFlakies(odtest, passingorder, failingorder):
2    odtype, polluters, cleaners = minimize(odtest, passingorder, failingorder)
3    patches = []
4    if odtype == VICTIM:
5      for polluter in polluters:
6        for cleaner in cleaners[polluter]:
7          patches += [patch(polluter + [odtest], cleaner)]
8    else: # odtype == BRITTLE
9      for statesetter in polluters:
10       patches += [patch([odtest], statesetter)]
11   return patches
```

Figure 4.3: Pseudo-code for the overall process of iFixFlakies

to address the problem of order-dependent tests, such as enforcing a specific test order. We discuss some of these other potential fixes for order-dependent tests in Section 4.4.4.

4.2 IFIXFLAKIES

We present iFixFlakies to automatically recommend patches for order-dependent tests with helpers. Figure 4.3 shows the pseudo-code for the overall process. iFixFlakies takes as input an order-dependent test, a passing test order, and a failing test order. By Definition 4.2, each order-dependent test has at least one passing and one failing test order. Several automated approaches exist for detecting order-dependent tests and their corresponding test orders [69, 118, 190], which can provide all these inputs for iFixFlakies. iFixFlakies has two main components: Minimizer and Patcher. iFixFlakies first calls Minimizer (Line 2) to get the type of the order-dependent test, the minimized polluters/state-setters, and the minimized cleaners. Based on the type of the order-dependent test, iFixFlakies then calls Patcher to create patches corresponding to each helper for the order-dependent test (lines 7 and 10).

Prior to developing iFixFlakies, we attempted to manually fix some order-dependent tests using just their passing and failing test orders. In this manual process, we found it difficult to *understand* why each test fails, let alone *fix* the test. However, as part of this process, we found ourselves manually searching for the polluter, cleaner, and state-setter tests for order-dependent tests, which is what inspired Minimizer. Once we realized the importance of the helpers and how they can be used as the basis for patches, we developed Patcher. Overall, we find that the manual steps that we undertook could be automated by a tool, leading to iFixFlakies, and such automation can save developers' time for fixing order-dependent tests.

```
1  def minimize(odtest, passingorder, failingorder):
2    isolation = run([odtest])
3    # Run in isolation multiple times to confirm it is order-dependent
4    for i in range(RERUN):
5      if not isolation == run([odtest]):
6        raise Exception("Incorrectly␣classified␣as␣order-dependent")
7
8    # Passing in isolation means victim, failing means brittle
9    if isolation == PASS:
10     odtype = VICTIM
11     startingorder = failingorder
12     expected = FAIL
13   else: # isolation == FAIL
14     odtype = BRITTLE
15     startingorder = passingorder
16     expected = PASS
17
18   polluters = set() # State-setters for brittles
19   cleaners = {}     # Empty map from polluters to cleaners
20
21   # Get minimal test order that causes odtest to match expected result
22   prefix = startingorder[0:indexOf(odtest, startingorder)]
23   while run(prefix + [odtest]) == expected:
24     polluter = deltadebug(prefix, lambda o: run(o + [odtest]) == expected)
25     polluters.add(polluter)
26     if odtype == VICTIM:
27       cleaners[polluter] = findcleaners(odtest, polluter,
28                                         passingorder, failingorder)
29     # If not configured to find everything, stop
30     if not FIND_ALL:
31       break
32     prefix.remove(polluter)
33   return odtype, polluters, cleaners
```

Figure 4.4: Pseudo-code for finding minimal test orders

4.2.1 Minimizer

Minimizer aims to find the minimal subsequence[2] of tests, called *minimal test order*, from a passing test order or a failing test order to make the order-dependent test pass or fail, respectively. The minimal test order is "1-minimal", meaning removing any test from the minimal test order will no longer satisfy the criterion [79, 184].

Figure 4.4 shows the pseudo-code for Minimizer. The input is an order-dependent test and its two test orders. As shown in Lines 4-6, Minimizer first checks whether the order-dependent test consistently passes or fails by itself, rerunning the test `RERUN` number of times (default is 10). If the

[2]The term "subsequence" refers to a potentially non-consecutive subset of elements in relation to the original ordering.

test consistently passes or fails, it is likely order-dependent. This check should not be needed when the input test is correctly classified as order-dependent, but our evaluation finds that we incorrectly classified three tests in our previous work [118]. If the test is truly order-dependent, the isolation result determines whether it is a victim or a brittle (Lines 9-16).

Next, Minimizer proceeds to delta debug [79, 184] the prefix to find the minimal test order (Line 24). Delta debugging iteratively splits a sequence of elements to find a smaller subsequence that satisfies a criterion. Our general delta debugging method takes two parameters: (1) the sequence to start delta debugging and (2) the criterion (in the form of a function) to check the current subsequence validity at each iteration. For Line 24, a subsequence is valid when running it before the order-dependent test matches the expected result for that test. The delta debugging output is a minimal test order representing a polluter for a victim or a state-setter for a brittle; ideally the polluter or state-setter consists of only one test. The search for finding a polluter or a state-setter is the same, so our code assigns the final result to the variable named `polluter`, but it is actually a state-setter if the order-dependent test is a brittle.

In practice, after Line 24, a developer would proceed to find a cleaner for the polluter if the order-dependent test was a victim, or proceed to Patcher if the order-dependent test was a brittle. However, for the sake of our experimental evaluation, we introduce the option to find *all* polluters or state-setters from these test orders. If the `FIND_ALL` option is set (Line 30), Minimizer proceeds to find more polluters or state-setters by first removing the found polluter or state-setter and then continuing with the loop that calls delta debugging again (Lines 23-32). The process stops when running the prefix before the order-dependent test no longer matches the expected result. Our experimental evaluation (Section 4.4.2) shows that finding more polluters or state-setters does not provide substantial benefits in terms of patching order-dependent tests, so in practice one can just use the first handful of found tests of each type.

Finding Cleaners After finding a polluter for a victim, Minimizer proceeds to find cleaners (Lines 26-27 of Figure 4.4). Figure 4.5 shows the `findcleaners` method. It takes as input a victim, a polluter for the victim, a passing test order, and a failing test order. The returned cleaners make the victim pass when they are run between the polluter and the victim.

First, `findcleaners` determines *cleaner candidates*, which are test orders that are potentially cleaners. `findcleaners` finds cleaner candidates using the passing and/or failing test order, depending on the index of the polluter and victim in these test orders. For the passing test order, if the victim is run after the polluter, then a cleaner must be among the tests that run between the polluter and victim, so these tests in between become a cleaner candidate (Lines 4-7). For the failing test order, a cleaner can be run before the polluter or after the victim, so tests that run before the polluter or after the victim both become cleaner candidates (Lines 9-12). Finding a cleaner is

```
 1 def findcleaners(victim, polluter, passingorder, failingorder):
 2   # Determine cleaner candidates from passing and failing orders
 3   candidates = []
 4   polluterpos = indexOf(polluter, passingorder)
 5   victimpos = indexOf(victim, passingorder)
 6   if polluterpos < victimpos:
 7     candidates += [passingorder[polluterpos + 1:victimpos]]
 8 
 9   polluterpos = indexOf(polluter, failingorder)
10   victimpos = indexOf(victim, failingorder)
11   candidates += [failingorder[0:polluterpos]]
12   candidates += [failingorder[victimpos + 1:len(failingorder)]]
13 
14   # Add all tests as single candidates
15   candidates += [[c] for c in failingorder]
16 
17   # Filter out candidates to find actual cleaners
18   cleaners = []
19   for c in candidates:
20     if run(polluter + c + [victim]) == PASS:
21       # If not configured to find everything, just return the first one
22       if not FIND_ALL:
23         return [deltadebug(c, lambda o: run(polluter + o + [victim]) == PASS)]
24       cleaners += [c]
25 
26   # Minimize the cleaners, so <polluter, cleaner, victim> passes
27   return unique(map(lambda c:
28                       deltadebug(c,
29                                  lambda o: run(polluter + o + [victim]) == PASS),
30                                  cleaners))
```

Figure 4.5: Pseudo-code for finding cleaners

crucial to enable automated search for a patch. To maximize the chance to find at least one cleaner, `findcleaners` also considers *every* individual test as a cleaner candidate, including even both the polluter and the victim (Line 15).

By considering every test as a cleaner candidate, `findcleaners` may even find a cleaner that JUnit would never run between the polluter and the victim. More specifically, when a polluter and victim are in the same class, `findcleaners` may find a cleaner consisting of tests from a different class than the polluter and victim; JUnit will never run this cleaner between the polluter and victim. `findcleaners` still searches for such cleaners, because their code can be used by Patcher.

For each cleaner candidate, `findcleaners` runs the polluter, the cleaner candidate, and then the victim, checking whether the victim passes in this test order. If the victim passes, then the cleaner candidate is an actual cleaner; `findcleaners` proceeds to delta debug the cleaner candidate to find the minimal test order (Line 23), with the delta debugging criterion being that running the polluter,

```
1 def patch(order, helpertests):
2   stmts = []
3   # Grab statements from helper methods, including setups and teardowns
4   for h in helpertests:
5     stmts += get_setup(h) + get_body(h) + get_teardown(h)
6 
7   # Create a method within the last helper's class with these statements
8   patchmethod = insert_new_method(test_class(helpertests[-1]))
9 
10  # Insert call to patchmethod at start of flaky test (last test in order)
11  insert_call_at_start(patchmethod, order[-1])
12 
13  # Delta debug statements such that the order (that was failing) can pass
14  minimalstmts = deltadebug(stmts, lambda s: patchmethod.setbody(s).compile()
15                                          and run(order) == PASS)
16 
17  patchmethod.setbody(minimalstmts)
18  return patchmethod
```

Figure 4.6: Pseudo-code for finding a patch

the subsequence from the cleaner, and the victim passes.

If the `FIND_ALL` option is not set, then the first cleaner found is returned. Otherwise, `findcleaners` checks the remaining cleaner candidates, for the set of all unique cleaners. We use this option to find all cleaners as part of our evaluation (Section 4.4); our results suggest that finding just a few cleaners suffices.

Minimizer takes the returned cleaners from `findcleaners` and adds them to a map from found polluters to found cleaners (Line 27 in Figure 4.4). The final return value for Minimizer is a tuple of (1) the type of the order-dependent test (victim or brittle), (2) the polluters or state-setters for the order-dependent test, and (3) the map from polluters to cleaners (empty if the order-dependent test is a brittle). These values in the returned tuple are then used by the next component, the Patcher.

4.2.2 Patcher

Patcher automatically recommends patches for fixing an order-dependent test using code from helpers. Patcher takes as input (1) the minimal test order where the order-dependent test fails: for a victim, this order is the polluter followed by the victim, and for a brittle, this order is just the brittle; and (2) a helper for the order-dependent test (note that a helper can consist of multiple tests). Figure 4.6 shows the pseudo-code for Patcher.

First, Patcher obtains all of the statements from the tests in the helper (Line 5). These statements come from not just the body of the test itself but also from all the setup and teardown methods associated with the test class of the test. We use JavaParser [31], a library for parsing Java

```
 1   // Victim (in ShutdownListenerManagerTest class)
 2   @Test
 3   public void assertIsShutdownAlready() {
 4    // Call to patch method
 5    new FailoverServiceTest().patch();
 6    ...
 7   }
 8 + // Starting patch method (in FailoverServiceTest class)
 9 + public void patch() {
10 +   // statements from @BeforeClass or @Before
11 +   ...
12 +   // 13 statements from cleaner, assertGetFailoverItems
13 +   ...
14 +   JobRegistry.getInstance().shutdown("test_job");
15 +   // statements from @AfterClass or @After
16 +   ...
17 + }
```

Figure 4.7: Starting code of Patcher for example in Figure 4.1

source code, to obtain these statements. Patcher keeps these statements in the order that JUnit runs them in (i.e., statements in `@Before` run first, then statements in the test, and lastly, statements in `@After`). More specifically, `get_setup` obtains the statements from the setup methods (annotated with `@BeforeClass` or `@Before` in the test class or super-classes), `get_body` obtains all statements in the helper test's body, and `get_teardown` obtains statements from the teardown methods (annotated with `@AfterClass` or `@After` in the test class or super-classes). If the helper test method's `@Test` annotation is parameterized with the optional `expected` [32], which indicates that the test expects a particular exception to be thrown for it to pass, then `get_body` also wraps the statements from the test in an appropriate try-catch block.

Next, Patcher adds code to run the helper code before the order-dependent test in two steps. First, Patcher creates an empty method, referred to as the `patch` method, to store all of the statements from the helper (Line 8). Second, Patcher inserts a call to the `patch` method at the start of the order-dependent test (Line 11). The inserted code creates an instance of the test class using the default constructor and uses that instance to call `patch`. Note that the code shows inserting this call at the start of the order-dependent test, but for a victim, the call can also be inserted at the end of the polluter. Users can configure Patcher to insert the patch at the beginning of the order-dependent test, or at the end of the polluter for victims.

Figure 4.7 shows an example of the starting code to be minimized by Patcher. This code is adapted from the example in Figure 4.1. Line 9 shows the declaration of the new `patch` method. The body of the `patch` method contains all of the statements from (1) the setup method of `FailoverServiceTest`, (2) the cleaner test body (`assertGetFailoverItems`), and (3) the tear-

```
1   // Victim (in ShutdownListenerManagerTest class)
2   @Test
3   public void assertIsShutdownAlready() {
4 +   // Call to patch method
5 +   new FailoverServiceTest().patch();
6     ...
7   }
8 + // Final patch method (in FailoverServiceTest class)
9 + public void patch() {
10 +   JobRegistry.getInstance().shutdown("test_job");
11 + }
```

Figure 4.8: Final code of Patcher for example in Figure 4.1

down method of `FailoverServiceTest`. The inserted line (Line 5) calls `patch` using a new instance of the helper's test class.

Finally, Patcher delta debugs the statements from the helper to find the minimal list of statements that can make the order-dependent test pass when run in the minimal test order (Line 14 of Figure 4.6); the minimal list of statements is also "1-minimal", and the finest granularity is at the level of statements as defined by JavaParser [31]. The delta debugging method is the same general one as in Minimizer, except this time it is minimizing the list of statements from the helper instead of test orders. The delta debugging criteria for Patcher are that the `patch` method compiles and that the inserted code makes the order-dependent test pass when run in the minimal test order. Patcher returns the `patch` method with the minimal list of statements for the order-dependent test to pass. Figure 4.8 shows the final code after Patcher runs (Line 10) for the example in Figure 4.1.

While the order-dependent test can already be fixed by inserting a call to the `patch` method at the start of the order-dependent test, a developer using iFixFlakies can choose to *inline* the statements from the `patch` method directly into the order-dependent test or into the polluter. In some cases, it may be trivial to just inline these statements into the order-dependent test body. However, in general, a developer should decide whether it is best to inline the statements of the helper into the order-dependent test or polluter, or leave them in a separate method. Factors that may influence the developer's decision include the applicability of the `patch` method to other tests and the data encapsulation of the `patch` method.

To further refine how statements invoked from helpers fix the order-dependent test, Patcher could potentially minimize and inline the statements of methods (indirectly) invoked by the helpers. By minimizing those statements, the developer can be given a patch that is much more specific to the cause of the flakiness. However, it can be difficult to inline statements from code further away from the helpers. Also, the number of statements in the final patch will likely increase when minimizing and inlining statements from methods invoked by helpers. As such, Patcher currently

Table 4.1: Breakdown of the 213 likely order-dependent tests from a public dataset [29]

# of tests	Category
22	in a class with `@FixMethodOrder`
49	`reuseForks` is set to false in `pom.xml`
3	non-order-dependent test
2	out-of-memory when run with iFixFlakies
137	truly order-dependent tests

does not minimize the statements of these invoked methods, and we leave such investigation for future work.

4.3 EVALUATION SETUP

In our prior work, we released a public dataset of flaky tests, including order-dependent tests [29, 118]. This dataset consists of 213 likely order-dependent tests from 38 Maven modules[3]; a Maven module consists of code and tests from the project that the developers organized to be built and run together. This dataset also has at least one passing and one failing test order for each order-dependent test.

We implement iFixFlakies as a plugin for Maven [35]. For each module in the Maven project, iFixFlakies takes as input order-dependent tests in the module to fix along with a passing test order and failing test order for each order-dependent test. iFixFlakies uses a custom JUnit test runner, the same from iDFlakies [118], to run the tests, so iFixFlakies currently recommends patches for only JUnit order-dependent tests in Maven-based projects.

Unfortunately, not all tests in the dataset are well suited for our goal of submitting patches to developers. First, 22 tests are in test classes annotated with `@FixMethodOrder`. This annotation tells JUnit to run the tests within that test class in a fixed order. Since the developers are already aware of the order-dependent tests in their test suite and have taken measures to address them, we omit these tests from our evaluation. Second, we also do not consider 49 tests from the dataset that are in modules that use the Maven Surefire parameter `reuseForks` to run each test class isolated in its own JVM. Such isolation removes many of the dependencies between tests and is another way used by developers to accommodate order-dependent tests.

We run iFixFlakies on all the remaining 142 purported order-dependent tests using the passing and failing test orders from our dataset. Some order-dependent tests have more than one passing test order and/or failing test order in the dataset, and we need only one of each for iFixFlakies, so we arbitrarily choose one of each test order to run iFixFlakies. We configure iFixFlakies to find

[3]This dissertation evaluates more order-dependent tests than evaluated in the original paper on iFixFlakies [162].

all polluters, cleaners, and state-setters for every order-dependent test. For each order-dependent test, we run iFixFlakies on Microsoft Azure with the virtual machine size `Standard_D11_v2`, which consists of 2 CPUs, 14GB of RAM, and 100GB of hard disk space.

Overall, we find 137 truly order-dependent tests. 3 tests were mis-classified as order-dependent (found to be non-order-dependent through our reruns) and, due to the large number of polluters and cleaners, 2 tests encounter out-of-memory errors from iFixFlakies. Table 4.1 shows the summary breakdown of the tests from the dataset.

4.4 EVALUATION

To evaluate the effectiveness and efficiency of iFixFlakies, we address the following research questions:

RQ1 What are the numbers of victims, brittles, polluters, cleaners, and state-setters found by iFixFlakies among test suites with order-dependent tests? How many order-dependent tests can iFixFlakies fix?

RQ2 What are the characteristics (e.g., size, uniqueness) of the patches generated by iFixFlakies?

RQ3 How much time does iFixFlakies take to find polluters, cleaners, state-setters, and patches?

The goal of RQ1 is to inform researchers and tool developers on which types of order-dependent tests and roles of tests are the most common so that they can be prioritized appropriately. With the main insight of iFixFlakies being to use helpers to propose patches for order-dependent tests, RQ1 also evaluates the frequency of tests that have helpers and therefore the applicability of our insight on order-dependent tests. The goal of RQ2 is to evaluate the effectiveness of the patches proposed by iFixFlakies and accepted pull requests created based on those patches. The goal of RQ3 is to evaluate the efficiency of iFixFlakies and thus how it could be integrated into a practical software development process.

4.4.1 RQ1: Characteristics of Tests

Table 4.2 shows some summary information about the projects and modules that contain at least one order-dependent test. For each module, the table lists the total number of tests, the number of order-dependent tests, and the breakdown of the number of victims and brittles among those order-dependent tests. Overall, we find that out of 137 order-dependent tests, 120 tests are victims and 17 tests are brittles, so most order-dependent tests are victims.

Table 4.2: Characteristics of the order-dependent tests (OD) in the projects used in our study

		Number of					Average number of		
ID	**Project Name - Module**	**tests**	**OD**	**victims**	**brittles**	**victims w/ cleaners**	**polluters per victim**	**cleaners per victim**	**state-setters per brittle**
M1	alibaba/fastjson	4,470	11	4	7	1	1.8	279.0	51.6
M2	alien4cloud/alien4cloud	14	1	1	0	1	1.0	1.0	n/a
M3	apache/incubator-dubbo - m1	110	4	4	0	4	2.5	6.8	n/a
M4	- m2	65	4	3	1	3	5.3	152.0	2.0
M5	- m3	21	1	1	0	1	1.0	2.0	n/a
M6	- m4	40	3	3	0	0	1.0	n/a	n/a
M7	apache/jackrabbit-oak	3,178	2	1	1	0	1.0	n/a	1.0
M8	apache/struts	61	4	4	0	4	1.0	16.0	n/a
M9	ctco/cukes	14	1	1	0	1	1.0	3.0	n/a
M10	dropwizard/dropwizard	80	1	1	0	1	2.0	16.0	n/a
M11	elasticjob/elastic-job-lite	511	6	6	0	5	1.0	35.8	n/a
M12	espertechinc/esper	49	1	1	0	0	1.0	n/a	n/a
M13	fhoeben/hsac-fitnesse-... [37]	269	1	0	1	n/a	n/a	n/a	4.0
M14	gooddata/GoodData-CL	8	1	0	1	n/a	n/a	n/a	4.0
M15	hexagonframework/s... [39]	48	1	0	1	n/a	n/a	n/a	42.0
M16	jfree/jfreechart	2,176	1	1	0	0	1.0	n/a	n/a
M17	jhipster/jhipster-registry	53	1	1	0	1	1.0	7.0	n/a
M18	kevinsawicki/http-request	163	28	28	0	28	1.0	1.0	n/a
M19	ktuukkan/marine-api	925	2	2	0	2	1.0	15.0	n/a
M20	openpojo/openpojo	1,185	5	5	0	5	3.0	9.0	n/a
M21	pholser/junit-quickcheck	11	1	0	1	n/a	n/a	n/a	1.0
M22	sonatype-nexus-... [38]	18	1	1	0	1	1.0	4.0	n/a
M23	spring-projects/spri... [40]	10	1	1	0	1	1.0	5.0	n/a
M24	spring-projects/spring-ws	119	2	2	0	2	1.0	16.0	n/a
M25	tbsalling/aismessages	44	2	2	0	0	1.0	n/a	n/a
M26	tools4j/unix4j	288	1	1	0	0	1.0	n/a	n/a
M27	undertow-io/undertow	79	1	1	0	1	4.0	12.0	n/a
M28	Wikidata/Wikid... [41] - m1	49	3	0	3	n/a	n/a	n/a	10.0
M29	- m2	23	2	2	0	2	1.0	1.0	n/a
M30	wildfly/wildfly	82	44	43	1	0	1.0	n/a	36.0
Total/Average per test		**14,163**	**137**	**120**	**17**	**64**	**1.3**	**20.0**	**28.3**

Table 4.2 also shows the average number of polluters per victim, cleaners per victim (that have cleaners), and state-setters per brittle that iFixFlakies finds. Each victim has at least one polluter. In the final row for averages, we show the averages computed per test (not per module). On average, we find 1.3 polluters per victim, with a total of 156 polluters for the 120 victims. Note that our search does *not* exhaustively find all polluters for a victim; the polluters that it finds depend on the position of the victim in the failing test order. On average across all victims, the position of a victim in its failing test order is 55.7% (i.e., a victim is just over the halfway position in the failing test order). 106 of the victims have just one polluter, while 14 victims have more than one polluter; the max number of polluters per victim is 6. While a polluter can consist of multiple tests that only when run together before the victim lead to it failing (Section 4.1), we find that only 4 polluters consist of more than one test. Because most polluters consist of only one test, it is practical to assume only one test pollutes the state for a victim, and future work on finding polluters may benefit from focusing on individual tests.

We hypothesized the existence of cleaners among the order-dependent tests in our prior work [118], and using iFixFlakies we find and show the actual number of cleaners per victim and polluter; different polluters for the same victim may have cleaners in common, but we report each cleaner separately per polluter for the same victim, because each one indicates a potential different patch for that victim. Also, the number we report for each module is the average number of cleaners per victims that have some cleaners (e.g., for `alibaba/fastjson`, the reported 279.0 is the number of cleaners for the single victim that has a cleaner). We find that 64 victims of the total 120 victims have at least one cleaner, so over half of all victims can be fixed using the code from their corresponding cleaners. Of these 64 victims, 32 have just one cleaner, while the remaining 32 have more than one cleaner. The average number of cleaners per the 64 victims with at least one cleaner is 20.0. In total, we find 1,282 cleaners for all 64 victims that at least one cleaner, where each cleaner consists of only one test. As described in Section 4.2.1, when iFixFlakies searches for cleaners, it considers *every* test as a potential cleaner, even when JUnit would not run such a test in between the polluter and victim. From the 64 victims with cleaners, we find seven with cleaners that JUnit would not run between the polluter and the victim. Interestingly, two of the cleaners are actually the polluters of a victim as well!

We also find that 13 victims have more than one polluter with cleaners. Interestingly, all polluters of these victims have exactly the same cleaners. Based on these results, a developer should use iFixFlakies to search for cleaners in just one polluter to know whether a victim likely contains a cleaner or not. Different cleaners can produce different patches, but we find that the numbers of statements produced by different cleaners are largely similar (Section 4.4.2).

Concerning state-setters, each brittle must have at least one state-setter, and we find a brittle has on average 28.3 state-setters. The 17 brittles have a total of 481 state-setters. Because all 17 brittles can be fixed using code from one of their state-setters, and 64 victims have cleaners, iFixFlakies can recommend patches for a total number of 81 tests, over half of the 137 truly order-dependent tests. In total, iFixFlakies finds 1,763 helpers to use to recommend patches for the 81 tests.

A1: Of the 137 order-dependent tests on which we evaluate iFixFlakies, 120 are victims and 17 are brittles. 64 of the victims have cleaners, so combined with the 17 brittles (that must have state-setters), iFixFlakies can recommend patches for 81 of the 137 order-dependent tests.

4.4.2 RQ2: Characteristics of Patches

Table 4.3 shows the characteristics of the patches that iFixFlakies recommends, with one patch per helper; we do not show rows for modules with no helpers, namely modules M6, M12, M16, M25, and M26. For each module, we show the average number of patches per order-dependent test in that module. We also show the average number of unique patches, based on statements, for each order-dependent test per module. For example, M8 (`apache/struts`) has 16.0 patches per order-

Table 4.3: Characteristics of patches recommended by iFixFlakies; for each module, averages per order-dependent test are shown

				First Patch		All Patches	
ID	# Patches	# Unique Patches	# Unique Patch Sizes	Avg. # Stmts	Avg. % Stmts from Original	Avg. # Stmts	Avg. % Stmts from Original
M1	80.0	8.9	1.9	1.1	21.4%	2.0	40.4%
M2	1.0	1.0	1.0	1.0	16.7%	1.0	16.7%
M3	6.8	2.2	1.0	7.0	65.8%	5.4	38.6%
M4	114.5	3.5	1.0	1.5	12.6%	1.0	8.2%
M5	2.0	2.0	2.0	5.0	71.4%	4.5	69.0%
M7	1.0	1.0	1.0	2.0	28.6%	2.0	28.6%
M8	16.0	2.0	2.0	2.0	13.3%	4.1	8.0%
M9	3.0	2.0	1.0	1.0	14.3%	1.0	14.3%
M10	16.0	8.0	4.0	2.0	25.0%	4.5	32.6%
M11	35.8	5.8	2.8	1.2	15.0%	1.5	15.3%
M13	4.0	2.0	1.0	2.0	25.0%	2.0	25.9%
M14	4.0	3.0	2.0	1.0	20.0%	2.0	22.2%
M15	42.0	16.0	10.0	4.0	66.7%	6.2	80.9%
M17	7.0	6.0	5.0	10.0	100.0%	7.6	83.9%
M18	1.0	1.0	1.0	1.0	16.7%	1.0	16.7%
M19	15.0	1.0	1.0	1.0	25.0%	1.0	22.4%
M20	27.0	7.8	1.0	1.0	12.9%	1.0	13.8%
M21	1.0	1.0	1.0	1.0	25.0%	1.0	25.0%
M22	4.0	4.0	3.0	4.0	80.0%	4.0	65.0%
M23	5.0	3.0	3.0	2.0	66.7%	6.2	60.0%
M24	16.0	9.0	5.5	9.0	90.9%	7.6	63.1%
M27	12.0	9.0	4.0	1.0	20.0%	2.7	32.5%
M28	10.0	2.7	2.0	1.7	17.2%	1.6	18.2%
M29	1.0	1.0	1.0	1.0	11.1%	1.0	11.1%
M30	36.0	8.0	4.0	13.0	86.7%	1.9	14.4%
Average	**21.8**	**3.6**	**1.7**	**2.0**	**25.9%**	**2.3**	**30.2%**

dependent test, but only 2.0 unique patches per order-dependent test. Overall, while iFixFlakies recommends, on average, 21.8 patches for each order-dependent test across all modules, only 3.6 are actually unique. The overall average in the final row is the average per test across all modules, not the (unweighted) average of averages per module (and some modules have more than one order-dependent test with a patch).

Table 4.3 also shows the average number of unique patch sizes among all patches for each order-dependent test per module; several patches with different statements can have the same number of statements. If the patch size is the most important for a good patch, then it suffices to find just one patch of a certain size instead of finding all the different patches of that size. With only 1.7 unique patch sizes per order-dependent test on average, many patches actually have the same size.

Table 4.3 also shows some statistics about the sizes of patches for only the first patch (from iFixFlakies trying the first cleaner of the first polluter or the first state-setter) and across all patches.

Table 4.4: Number of tests addressed by pull requests (PRs) based on iFixFlakies patches.

	# of Test Fixed by		
ID	**Pending PRs**	**Accepted PRs**	**Patcher**
M1	1	7	8
M2	0	1	1
M3	0	2	2
M4	0	4	4
M5	0	1	1
M7	1	0	1
M8	0	4	4
M9	1	0	1
M10	0	1	1
M11	0	5	5
M13	0	1	1
M14	1	0	1
M15	1	0	1
M18	28	0	28
M19	0	2	2
M20	4	1	5
M21	0	1	1
M22	1	0	1
M23	0	1	1
M24	2	0	2
M27	0	1	1
M28	0	3	3
M29	0	2	2
M30	0	1	1
Total	**40**	**38**	**78**

The table shows the average number of statements and the average percentage of the number of statements w.r.t. the number of statements in the original helper (Section 4.2.2). Across all patches, iFixFlakies recommends a patch with only 2.3 statements on average, and these statements comprise only 30.2% of the statements in the original patch method. In fact, of the 1,763 total patches, 1,147 (65.1%) contain just one statement! When we look into the spread of the patch sizes per order-dependent test, we find that, on average, each order-dependent test has around 90% of their patches with the same size, most often being the smallest size. For example, the average number of statements in the first patch (2.0) is almost equal to the average number of statements across all patches (2.3). The results suggest that iFixFlakies should search for a few helpers, but not all of them, because the majority of the helpers lead to the same size of patches.

Submitted Patches We submitted pull requests for 78 of the 81 order-dependent tests with helpers; 3 of the 81 had already been fixed before we submitted pull requests. Table 4.4 shows the breakdown of the tests corresponding to our pull requests. Developers already accepted pull requests for 38 tests.

While our pull requests are based on the patches generated by iFixFlakies, we sent patches for 33 exactly as iFixFlakies recommended, and the remaining 45 required small, manual changes. For 18 of the changed fixes, the change involved just putting shared code of different order-dependent tests into one setup/teardown method. For one test, a brittle, from our manual inspection, we find that the real problem with the test is actually a developer typo. Based on this inspection, we find it more appropriate to make the typo fix rather than include the extra code that iFixFlakies recommends. When we had to make changes to the patch for the pull requests, the effort was roughly 1-3 minutes per patch, mostly refactorings or simple changes to match the style of the existing code. Existing techniques and tools [45, 47, 166, 172] could help with such manual effort. We believe that developers using iFixFlakies could use such tools for more automation but still examine the patches and manually apply small changes if necessary. We make available the patches that iFixFlakies generates, a more detailed breakdown describing the changes that we made to the patches, and links to the corresponding pull requests on our website [30].

Because iFixFlakies fixes an order-dependent test using statements from a helper, the recommended patches may reduce the order-dependent test's fault-detection capability, i.e., make the test miss a fault. However, if a patch does reduce an order-dependent test's fault-detection capability, then the passing test order in which iFixFlakies (may have) found the helper could likely miss the fault as well. iFixFlakies assumes that each passing test order is correct, and the failing test order indicates a fault in the test code, not a fault in the code under test. We do not believe that the scenario where the failing test order indicates a fault in the code under test actually occurred in our evaluation, particularly because developers did not reject our pull requests to fix the order-dependent tests.

It should be noted that for M30 (`wildfly/wildfly`), iFixFlakies actually helps fix victims *without* cleaners as well! None of the 43 victims have a cleaner. However, they all share the same polluter, and that polluter is itself the single brittle that iFixFlakies finds. When we apply a recommended patch for the brittle, not only is the brittle fixed, but all of the victims are also fixed. This example showcases one of the complexities of order-dependent tests and how iFixFlakies can even help fix order-dependent tests that do not have helpers themselves. We do *not* count these 43 tests as fixed in our evaluation, because iFixFlakies fixes these tests indirectly.

A2: iFixFlakies recommends 21.8 patches per order-dependent test, with only 3.6 unique patches and 1.7 patch sizes per order-dependent test. Of the GitHub pull requests we sent for 78 order-dependent tests, developers have accepted pull requests for 38 tests, with none rejected so far.

Table 4.5: Average time in seconds that iFixFlakies takes; '*' denotes that the time includes finding some test(s) with no cleaner

	Test suite	Avg. time to find first				Avg. time to find all			
ID	**time**	**polluter**	**cleaner**	**state-setter**	**patch**	**polluters**	**cleaners**	**state-setters**	**patches**
M1	203	92	*523	42	299	113	*1,443	1,748	25,424
M2	1	17	9	n/a	76	17	48	n/a	76
M3	8	22	52	n/a	294	48	465	n/a	2,473
M4	206	143	178	21	130	389	7,089	39	54,856
M5	1	2	4	n/a	395	2	25	n/a	572
M6	3	19	*104	n/a	n/a	19	*104	n/a	n/a
M7	189	218	*5,710	50	416	218	*5,710	50	416
M8	4	13	11	n/a	411	13	159	n/a	14,478
M9	2	6	8	n/a	47	6	44	n/a	230
M10	7	36	16	n/a	714	57	589	n/a	4,960
M11	24	45	*293	n/a	258	45	*1,434	n/a	11,854
M12	11	28	*82	n/a	n/a	28	*82	n/a	n/a
M13	3	9	2	n/a	291	33	*8	n/a	619
M14	1	n/a	n/a	3	33	n/a	n/a	14	258
M15	1	n/a	n/a	20	114	n/a	n/a	485	5,018
M16	22	16	*1,695	n/a	n/a	16	*1,695	n/a	n/a
M17	3	10	75	n/a	419	10	216	n/a	493
M18	2	15	5	n/a	56	15	227	n/a	56
M19	1	14	5	n/a	50	14	675	n/a	756
M20	21	47	53	n/a	53	95	3,125	n/a	1,455
M21	7	n/a	n/a	12	76	n/a	n/a	12	76
M22	3	32	25	n/a	169	32	41	n/a	660
M23	1	20	19	n/a	93	20	85	n/a	406
M24	2	15	4	n/a	139	15	139	n/a	1,766
M25	1	5	*34	n/a	n/a	5	*34	n/a	n/a
M26	5	35	*535	n/a	n/a	35	*535	n/a	n/a
M27	17	32	79	n/a	232	109	1,025	n/a	2,617
M28	3	n/a	n/a	8	43	n/a	n/a	70	416
M29	1	2	3	n/a	36	2	24	n/a	36
M30	3	21	*114	11	463	21	*114	345	4,749
Average	**25**	**28**	**154**	**27**	**160**	**38**	**638**	**837**	**7,238**

4.4.3 RQ3: Performance

Table 4.5 shows the time iFixFlakies takes to find polluters, cleaners, and state-setters, along with the time to create patches. The table shows for each module the average time (across all order-dependent tests in the module) that iFixFlakies takes to find/create (1) the first polluter, cleaner, state-setter, and patch; and (2) all polluters, cleaners, state-setters, and patches. The time to create patches assumes that a helper has been found and does *not* include the time to find the helper. As a reference for the time taken by iFixFlakies, the table shows the time to run each module's test suite.

Effective use of iFixFlakies would only require finding the first polluter and cleaner (for a victim) or state-setter (for a brittle) so that iFixFlakies can recommend a patch (Section 4.4.2). If the victim has more than one polluter, then the time for the first cleaner is for the first cleaner of the first polluter. Similarly, the time to the first patch for such a victim is then the patch created from the first cleaner for the first polluter. If the victim has no cleaners, the table reports the time taken by iFixFlakies to search for cleaners for that first polluter, eventually not finding any; we mark such time with a '*' in the table. The overall average time to find the first polluter, cleaner, state-setter, and patch is 28, 154, 27, and 160 seconds, respectively. The overall averages are over all order-dependent tests, not over modules. Likewise, the overall average for running all tests in a module is "weighted" by the number of tests as well, so modules with more than one order-dependent test have their test suite time counted multiple times, once per each order-dependent test. Compared to this weighted average time to run all the tests, the time to find the first polluter, cleaner, state-setter, and patch is about 1.1x, 6.2x, 1.1x, and 6.4x the time to run all tests, respectively.

On average, iFixFlakies takes 38, 638, 837, and 7,238 seconds to find/create all polluters, cleaners, state-setters, and patches, respectively; once again, '*' denotes time that includes searching for cleaners where there are none. Compared to the time to find/create just the first corresponding test/patch, the time is 1.5x, 25.5x, 33.5x, and 289.5x larger. The average time for finding all state-setters is particularly larger than the time for finding the first state-setter due to the large number of state-setters in M1 (`alibaba/fastjson`). The average time for creating all patches is also particularly larger due to the large number of helpers (one per patch).

Note that iFixFlakies performance can be improved, e.g., Patcher could modify the bytecode of the patch code in-memory [72, 123] to avoid compilation during delta debugging, or could instrument the code to allow turning statements on or off during delta debugging similar to meta-mutants [110, 173]. In general, considering the large amount of time to create all patches and there being fewer unique patches than all patches, developers should *not* use iFixFlakies to create all patches using all helpers for each order-dependent test; obtaining just a few should suffice.

Overall, the average end-to-end time for iFixFlakies to try to create a patch for an order-dependent test is 256 seconds; the end-to-end includes the time to find the first helper (including the time to find the first polluter for victims) and then to create the corresponding patch. This end-to-end time also includes the cases where an order-dependent test has no cleaner, and iFixFlakies spends time looking for it. If we split the order-dependent tests between those with and without helpers, the time to create a patch for an order-dependent test with a helper is 207 seconds, while the time to fail to create a patch for one without a helper is 325 seconds.

A3: iFixFlakies on average takes 28, 154, 27, and 160 seconds to find/create the first polluter, cleaner, state-setter, and patch, respectively, for an order-dependent test. The times are much larger when trying to consider all polluters, cleaners, state-setters, and patches. Furthermore, considering that there are also far fewer unique patches than all patches, we do not recommend using iFixFlakies to create all patches.

4.4.4 Discussion

Using iFixFlakies, we aim to fix order-dependent tests by modifying test code. However, there are potentially other ways to address the problem of order-dependent tests. One way is to enforce that tests run in a specific order (all the tests in a previously found passing test order) all the time. Enforcing the order thereby removes any possibility of failing test orders for order-dependent tests, and the only reason for a failure should be due to the changes developers make to the code. However, such an approach does not ensure tests can be run independently, which in turn makes other testing tasks more difficult. For example, test parallelization becomes more difficult if there is no guarantee that tests can have the correct result when they are run on any machine with any other tests; preventing order-dependent test failures when using test parallelization requires extra effort [119]. Debugging also becomes more difficult if tests must always run in a specific order, because developers must then run all the tests (at least up to the failing test) as they iteratively try to debug any test failures.

Another approach to addressing the problem of order-dependent tests is to always run tests isolated from one another, e.g., run each test in its own process. Most order-dependent tests are victims, which fail due to polluters, so isolation can naturally remove this state pollution between the polluter and the victim. However, isolation adds extra overhead. Bell and Kaiser [48] found that running JUnit tests in their own JVM processes adds on average a 618% overhead. Having such high overheads every time tests are run, e.g., after every change as in regression testing, adds tremendous cost to the development process.

4.5 THREATS TO VALIDITY

The results of our study concerning the frequency of victims, brittles, polluters, cleaners, and state-setters may not generalize to other projects. We attempt to mitigate this threat by using a dataset of popular and diverse projects from our prior work [118]. We generated this dataset of order-dependent tests using 13 projects from earlier work on flaky tests [50], and 150 Java projects deemed the most popular on GitHub [28] based on the number of stars that the projects have. Furthermore, iFixFlakies itself or tools that it uses (e.g., JavaParser [31]) may have faults that could have affected our results. We used extensive logging in iFixFlakies, and at least two people (among the dissertation author and collaborators) reviewed iFixFlakies's code and logs.

The metrics that we use to evaluate the patches that iFixFlakies creates, e.g., patch size and uniqueness, may not be the most important metrics for determining the quality of patches. Other important metrics include the time taken to run the patched-in code. The patches that iFixFlakies recommends may also not lead the order-dependent test to pass for test orders other than the failing test orders that iFixFlakies checks. To mitigate these two threats, we submitted pull requests for the patches that iFixFlakies recommends. So far, developers have already accepted pull requests for 38 order-dependent tests, and the rest are pending with none rejected.

4.6 SUMMARY

We present iFixFlakies, a framework for automated fixing of order-dependent tests. Our main insight for iFixFlakies is that test suites often have *helpers* whose code can help fix order-dependent tests. iFixFlakies searches for helpers and uses their code to propose relatively small patches for order-dependent tests. Our evaluation on 137 order-dependent tests from a public dataset shows that iFixFlakies can automatically recommend patches for 81 of 137 tests. The recommended patches are effective, with 65.1% of them having just one statement. Also, iFixFlakies is efficient, requiring only 207 seconds on average to produce the first patch for an order-dependent test with a helper. The effectiveness and efficiency of iFixFlakies show promise that it may be integrated into a practical software development process. We used patches recommended by iFixFlakies to open pull requests for 78 order-dependent tests (3 of the 81 had already been fixed); developers have already accepted pull requests for 38 tests, and the remaining ones are pending but none have been rejected.

CHAPTER 5: RELATED WORK

This chapter covers work related to traditional techniques to reduce the cost of regression testing (Section 5.1), continuous integration builds (Section 5.2), flaky tests (Section 5.3), and mutation testing (Section 5.4).

5.1 REGRESSION TESTING TECHNIQUES

Test-suite reduction, regression test selection, and test-case prioritization are traditional approaches to reducing the cost of regression testing [183]. We describe work in each of these three areas in more detail.

5.1.1 Test-Suite Reduction (TSR)

Test-suite reduction (TSR) is a well-studied research topic [183], and over the years researchers have proposed various ways to create reduced test suites [53, 57, 60, 61, 71, 78, 86, 92, 104, 109, 125, 129, 182, 183, 191] and to evaluate the effectiveness of TSR techniques [148, 151, 176, 177, 188]. All these studies used either seeded faults or mutants to evaluate fault-detection effectiveness. We find that the Failed-Build Detection Loss (FBDL) for a reduced test suite is much higher than its mutant-detection loss. Our definition of fault-detection effectiveness follows the approach from Wong et al. [176] and Rothermel et al. [151]; while they measured the percentage of *seeded faults* detected by the original test suite that the reduced test suite does not detect, we measure the percentage of *failed builds* where the reduced test suite does not detect all the faults detected by the original test suite. Since their experiments had one seeded fault per faulty program version, their evaluation matches our FBDL_S, where all test failures are mapped to the same fault. However, we have multiple mappings from test failures to faults in FFMap, allowing us to consider multiple faults in a build at a time, which is required when using test failures from real-world software evolution.

In our previous work [158], we studied the effects of software evolution on TSR. We measured the mutant-detection loss of the reduced test suite at an early, passed version where TSR is performed and the mutant-detection loss at a future, passed version. We found that the loss remains roughly the same, indicating TSR may be predictable, suggesting that the reduced test suite remains as effective as before even as software evolves. In this dissertation, we instead measure missed failed builds based on historical project build logs. Moreover, we evaluate whether the traditional TSR metrics are good *predictors* of the missed failed builds. Unlike prior work, we find that TSR is unpredictable and can lead to a large percentage of missed failed builds in the future. Other work has also studied prediction in the context of regression testing, but for regression test selection [94, 146, 147] as opposed to TSR.

Instead of removing tests from a test suite as in TSR, prior work has also proposed removing statements or minimizing test inputs as to reduce the cost of testing; such an approach is known as *test-case reduction*. Originally, reducing individual test cases was used for debugging purposes, such as proposed by delta debugging [184], to create smaller test cases that can reproduce a failure from the original test case. Groce et al. [79] later proposed such a reduction for speeding up regression testing by reducing the test case while still satisfying some test-requirements, such as coverage. We later followed up on Groce et al.'s work by proposing inadequate test-case reduction that reduces test cases to satisfy only a configurable percentage of the specified test-requirements [44].

5.1.2 Regression Test Selection (RTS)

Given a program change, regression test selection (RTS) techniques aim to select a subset of affected tests [66, 74, 93, 95, 141, 150, 183]. The key idea of prior RTS techniques is to maintain some metadata, such as statements covered by tests on the previous version, and leverage this metadata to select a subset of tests that are affected by the change. Most proposed techniques collected the metadata dynamically, i.e., instrumenting test runs to determine what parts of the code each test covers and using that information to determine what tests to run in the future [74, 121, 141, 150, 185]. We recently proposed STARTS [121, 122], a technique for performing RTS statically by determining the relationship between tests and classes being tested through a static analysis of the class-level dependency graph. However, a pure static analysis can potentially miss dependencies reachable through dynamic language features such as reflection; we later extended STARTS to account for reflection [161]. The number of different RTS techniques is large and growing. Rothermel and Harrold [149] established a framework for analyzing RTS techniques, and we recently proposed a framework to test implementations of RTS techniques [192].

While early work on RTS used fine-grained metadata such as what statements are covered by which tests, Harrold et al. [93] specify that RTS can be based on coarser-grained entities, such as methods, classes, or modules. Recent work by Gligoric et al. [73, 74] found dynamic RTS that tracks dependencies dynamically from test runs at the coarse granularity level of classes to be effective. We also similarly found class-level static RTS to be more effective than using finer granularities in our prior work comparing static RTS at different levels of granularity [121].

In our other prior work [165], we compared a class-level RTS technique against module-level RTS on open-source projects using the Maven build system on Travis CI, finding that the two have similar costs in terms of regression testing time while not missing to select tests that fail due to changes. Vasic et al. [175] also evaluated class-level and module-level RTS, but for .NET applications, and they proposed a hybrid technique that runs a class-level RTS technique on top of an incremental build system that does module-level RTS. Gyori et al. [83] studied class-level RTS compared against an even coarser-granularity of RTS, at the project level, where different open-source projects, each written by different teams of developers, depend on one another in a very large open-source ecosystem. They find that class-level RTS can provide additional opportunities for more efficient testing at this scale.

Researchers have also proposed RTS techniques that do not rely on directly analyzing changes and their relationship with the tests. Herzig et al. [98] proposed an RTS technique based on a cost model. Their technique dynamically skips tests when the expected cost of running a test exceeds the expected cost of not running the test. All tests are still run at least once before the code is released, so fault detection is merely delayed (but can potentially be more costly to fix). Machalica et al. [130] recently proposed selecting tests based on historical failure information. The goal is to not select all tests that *can* be affected by the changes, but rather to predict what tests would likely fail and select to run only those tests. Elbaum et al. [65] also proposed using historical information to select what tests to run in an industrial setting at Google, though they combine such an RTS technique with test-case prioritization.

5.1.3 Test-Case Prioritization (TCP)

Test-case prioritization (TCP) aims to run tests in a more optimized order, one that runs likely-to-fail tests (which detect faults) earlier [183]. Unlike TSR or RTS, TCP does not reduce the absolute cost of regression testing by running fewer tests. Instead, the idea behind TCP is that developers can early on observe what failures occur and begin to debug even as the remaining tests continue to run. As such TCP guarantees safety in terms of not missing to run a failing test that indicates a fault, because TCP eventually does run all tests.

Most prior work has implemented techniques based on test coverage (e.g., prioritizing tests that cover more) and diversity (e.g., ordering tests such that similar tests in terms of coverage are ordered later), and investigated characteristics that can affect TCP effectiveness such as test granularity or number of versions from original point of prioritization [97, 106, 124, 126, 127, 186]. Recent work showed that traditional coverage-based techniques may not be cost effective at running the tests that detect bugs earlier, because they tend to execute long-running tests first [58]. In lieu of coverage-based techniques, some prior work investigated TCP using information retrieval (IR) techniques. Saha et al. [155] proposed IR-based TCP that compares the textual similarity between recent changes and test code as to rank the tests, evaluating on Java projects. Mattis and Hirschfeld [131] also evaluated IR-based TCP similarly for Python projects. We later conducted a deeper analysis of IR-based TCP on Java projects by investigating the effectiveness of different configuration options for IR. We also also proposed hybrid TCP techniques that combine IR-based TCP with historical test information, such as previous test running times and historical test failure information; the hybrid techniques performed better than the individual techniques that formed that hybridization [143].

5.2 CONTINUOUS INTEGRATION (CI)

Researchers have recently studied developers' usage of continuous integration (CI) for both open-source and industry settings [100, 101]. Given CI's wide-spread usage, it becomes important to develop techniques that improve upon CI as a whole, in both testing and in the actual building of code after changes.

There is a growing body of work on techniques to improve the efficiency of builds [42, 132, 133]. Telea and Voinea [169] developed a tool that decomposes header files (in C/C++ code) to remove performance bottlenecks. Morgenthaler et al. [136] developed a tool that ranks libraries in a build based on their utilization rank to identify under utilized nodes that can be refactored. There has also been a body of work on predicting build outcomes without running the build [96, 107, 138, 180]. The goal of predicting a build outcome is to quickly determine if it is worthwhile to actually run the build, so then developers can entirely skip to run a build if it is predicted to pass (because passing builds would not indicate any faults in the code that developers would need to address).

Our work in TestOptimizer is closely related to work by Vakilian et al. [174] that attempts to decompose build nodes into two or more nodes to improve build times. Their work shares a similar goal with TestOptimizer, i.e., avoid unnecessary building of nodes when dependencies change. TestOptimizer substantially differs from their technique due to the following reasons. First, we focus on moving individual tests from their current test nodes into either other existing or new test nodes. We allow movements to existing test nodes, as opposed to Vakilian et al.'s technique that only creates new nodes. Second, because test nodes are essentially the leaf nodes in the dependency graph, we do not have to update dependencies for any child nodes. Third, Vakilian et al.'s technique does not take historical information into consideration, i.e., they assume all nodes are built in every build. In contrast, we take historical information into consideration to ensure that tests are moved away from test nodes that have a high historical build count. Finally, their technique identifies dependencies statically, whereas our technique uses dynamic analysis (code coverage), which is more precise. Although it would be ideal to evaluate both techniques on common evaluation projects, such an evaluation is not practical because both techniques use proprietary applications and target different technologies.

5.3 FLAKY TESTS

Luo et al. [128] reported the first extensive academic study of flaky tests; they categorized flaky tests by studying historical commits of fixes for flaky tests and found order-dependent tests to be among the top three most common categories. Gao et al. [70] studied flaky GUI tests, and they found tests that change the configurations for later-run tests, resulting in GUI order-dependent tests. Thorve et al. [170] studied flaky tests found in Android apps, leading to different root causes found in prior work. In our recent work [63], we similarly studied flaky tests in applications that use probabilistic programming or machine learning frameworks, finding that randomness is a bigger cause of flakiness in this domain.

5.3.1 Detecting Flaky Tests

In addition to studying flaky tests, researchers have also proposed techniques to detect different types of flaky tests early on, so developers know ahead of time that there can be failures due to these detected flaky tests. Bell et al. [50] proposed DeFlaker, a technique for detecting when a test failure is a flaky test failure by checking the coverage of failed tests against the recent changes. We previously proposed a technique NonDex [82, 159] for detecting tests that fail due to assuming deterministic implementations of nondeterministic specifications. We also proposed a technique FLASH [63] that detects flaky tests due to differences in numbers produced by random number generators by rerunning tests while controlling for the seeds to random number generators.

Focusing on just order-dependent tests, Zhang et al. [190] proposed DTDetector, which detects order-dependent tests through randomizing the test orders. We followed up and released iDFlakies [118], a framework for detecting order-dependent tests by randomizing test orders, similar to DTDetector. Along with the framework, we also released a dataset [29] of flaky tests found using iDFlakies; almost half of the flaky tests found are order-dependent tests, and we evaluate iFixFlakies using these tests.

Huo and Clause [103] studied tests whose assertions depend on input data not controlled by the tests themselves. They called these assertions “brittle”, inspiring our naming of brittles as tests with similar kinds of assertions[1]. The difference is that their brittle assertions *may* fail due to the tests using wrong input data that they do not control, while our brittles are tests that *always* fail when run in isolation (without a state-setter running before them). We previously proposed a technique, PolDet [84], for detecting tests that change shared state so the state at the end of their run differs from the state at the start of their run. In that work, we also called these tests “polluters”, and the polluters from this dissertation are similar in nature. The difference is that

[1]The term “brittle” test was also used to describe GUI tests that fail due to changes in the interface [102, 171, 181].

the polluters from that prior work *may* pollute the state so other tests (potentially future ones) fail, while our polluters in this dissertation *always* pollute the state for some existing victims. Bell et al. [49] proposed a technique, ElectricTest, to detect data dependencies between existing tests in a test suite, and Gambi et al. [69] followed up on ElectricTest with PraDet, which detects when dependencies between tests can actually lead to tests failing in different orders.

5.3.2 Debugging, Fixing, and Accommodating Flaky Tests

Besides detecting flaky tests, recent work has started investigated debugging, fixing, and accommodating flaky tests. Indeed, Harman and O'Hearn recently called for research to assume all tests are flaky and modify existing testing to take this assumption into account [91]. Our technique, iFixFlakies, is the first technique to automatically fix flaky tests, focusing specifically on order-dependent tests. Bell and Kaiser [48] proposed VMVM, a technique to tolerate order-dependent tests by restoring the state of the heap between test runs. VMVM adds instrumentation that re-initializes static fields shared between tests to isolate tests from one another with regards to their heap state when run in the same JVM. Muşlu et al. [137] proposed an even more extreme technique for isolation in that each test should not only run in a separate JVM but also in a fresh environment, e.g., a fresh file system. Bell et al. [50] also evaluated how various forms of isolation can help in test reruns to detect which test failures are due to flaky tests. Lam et al. [116] proposed RootFinder for determining the root causes of flaky tests at Microsoft. Building on that work, Lam et al. [117] later proposed a technique to fix flaky tests due to async waits by searching for suitable times to wait as to reduce the chance of flakiness. Finally, as developers may not want to fix known order-dependent tests in their test suite due to the benefits they can provide (sharing resources between tests without having to reset), we recently proposed enhancing regression testing techniques to take known test dependencies into account [119].

5.4 MUTATION TESTING

Mutation testing is commonly used in research to evaluate the quality of testing [105]. Researchers have reported strong correlations between mutants and real faults [46, 112]. Researchers have utilized mutants to generate tests [68, 142] and evaluate testing techniques [79, 126, 158, 164], such as for test-suite reduction as we do in this dissertation.

There has been much work in developing new tools to perform mutation testing, such as developing tools for general programming languages like Java [19, 110], for an intermediate representation like LLVM bitcode [56, 87, 88, 89], or even a universal mutator that aims to provide mutation testing across a wide variety of programming languages [80].

Prior work has also investigated how to improve mutation testing efficiency or effectiveness. Offutt et al. [139] proposed reducing the number of mutants by finding only a subset of mutation operators that are sufficient to evaluate mutation score. Just et al. [111] proposed tracking state infection and propagation at runtime to reduce the time for running tests on mutants. Zhang et al. [187] leveraged TSR and TCP for faster mutation testing. We previously proposed combining concepts from approximate computing and mutation testing to improve on both [75]. Building on ideas from that work, we later evaluated using transformations from approximate computing as new mutation operators [90]. Mutation testing also can suffer from flaky tests, which can mislead developers concerning the quality of their test suites, so we proposed techniques to mitigate the effects of flaky tests on mutation testing [157].

CHAPTER 6: CONCLUSIONS AND FUTURE WORK

Regression testing is an important part of the software development process, but it suffers from two main challenges: (1) regression testing is costly, and (2) regression test suites contain flaky tests. The cost of regression testing is due to large number of tests that have to be run after every change, and changes happen frequently, so a large amount of machine time is spent on just running tests. Flaky tests can nondeterministically pass or fail when run on the same version of code regardless of any changes, so flaky test failures can mislead developers into thinking there are faults in their changes that they need to debug. This dissertation addresses these challenges through test-suite transformations that can reduce the cost of regression testing and fix flaky tests.

First, we evaluate a traditional approach to reduce cost by transforming test suites through test-suite reduction (TSR). TSR removes tests from the test suite that are redundant with others based on heuristics such as the code coverage of individual tests. While TSR was proposed over two decades ago, prior work has always evaluated TSR using seeded faults and on a single version of software. Such evaluation does not indicate how well a reduced test suite performs into the future after software evolution. Our evaluation of TSR on real-world projects with real software evolution and test failures shows that reduced test suites computed from traditional TSR techniques are not as effective on future versions of the projects, and existing TSR metrics are not good predictors of the effectiveness of reduced test suites.

Second, we propose a different test-suite transformation that moves tests between modules in projects that use modern module-based build systems. Such a build system allows developers to distribute their builds and testing, but can lead to inefficiencies in testing due to suboptimal placement of tests within modules. We propose TestOptimizer to suggest optimal test movements between modules, and our evaluation on five Microsoft projects shows that the suggestions can result in a reduction of 21.7 million test executions (17.1%) across those projects.

Third, to address flaky tests, we propose iFixFlakies, a framework for automatically fixing order-dependent tests, a prominent type of flaky tests. iFixFlakies recommends patches for order-dependent tests based on the insight that often the logic for fixing these individual tests can be found among other tests in the test suite. We sent pull requests for 78 order-dependent tests, and developers accepted pull requests fixing 38 order-dependent tests while the rest remain pending.

6.1 FUTURE WORK

In this section, we describe potential future work building upon topics in this dissertation.

Better heuristics for determining redundant tests: Our evaluation finds that traditional TSR techniques based on code coverage are not effective at creating reduced test suites that detect faults in future builds. Developing new metrics to determine redundancy could lead to reduced test suites that that are less likely to miss faults in future builds and therefore more acceptable to developers.

Better predictors of FBDL: Our evaluation finds that the current metrics researchers use to evaluate reduced test suites are not good predictors of these test suites' FBDL in future builds. New metrics that better predict FBDL can help developers make the decision of whether to use reduced test suites or not.

Incremental TestOptimizer suggestion: Instead of running TestOptimizer on the whole test suite, we can make TestOptimizer incrementally suggest ideal placements for newly added tests.

Split build nodes: TestOptimizer splits test nodes by suggesting a better placement of tests. However, our evaluation finds that these suggestions sometimes lead to developers modifying build nodes instead, creating better dependency graphs for future builds. Interesting future work would be to extend TestOptimizer to suggest how to split build nodes.

Automatic refactoring of nodes: While TestOptimizer suggests placement of tests, it relies on developers to actually implement the suggestions. Our experience at Microsoft showed that developers ultimately desire a tool that can automatically implement those suggestions as well, so future work can explore large-scale automated refactoring techniques [178] that apply those suggestions.

Automatic fixing of different types of flaky tests: iFixFlakies specifically targets order-dependent tests. However, there are many different types of flaky tests with different causes such as async wait, concurrency, unordered collections, random number generators, etc. [128]. Given the success of iFixFlakies, it should be possible to develop different techniques targeting specific types of flaky tests. Eventually, if we have a suite of techniques targeting all kinds of flaky tests, we can help developers completely eliminate flaky tests from their test suites.

Computational science and reproducibility: Code developed by scientists for computational science experiments, such as for computational physics, currently has challenges in reproducing results. Our prior work has found that part of the challenge comes from scientists not being trained in software engineering practice [113]. However, the domain of code they are developing may be prone to nondeterminism that leads to difficulties in reproducing results, akin to flaky tests. We should take our work on flaky tests in traditional software to help scientists with their challenges in reproducibility.

Software continues to be an essential part of our daily lives, and as long as we develop software, we will need software testing. Regression testing is a crucial part of the software development process, and this dissertation presented techniques that improve regression testing. In the future, we expect to see further advancements to reduce the cost of regression testing, make testing more reliable, improve developer productivity, and overall lead to higher quality software.

REFERENCES

[1] "JUnit and Java 7," http://intellijava.blogspot.com/2012/05/junit-and-java-7.html, 2012.

[2] "Visual Studio Team Test," https://msdn.microsoft.com/en-us/library/ms379625.aspx, 2012.

[3] "Maintaining the order of JUnit3 tests with JDK 1.7." https://coderanch.com/t/600985/engineering/Maintaining-order-JUnit-tests-JDK, 2013.

[4] "VRaptor commit 021d10b7," https://github.com/caelum/vraptor4/commit/021d10b7, 2013.

[5] "VRaptor commit 49742a2d," https://github.com/caelum/vraptor4/commit/49742a2d, 2013.

[6] "VRaptor commit b2437ab1," https://github.com/caelum/vraptor4/commit/b2437ab1, 2013.

[7] "VRaptor Travis CI build #15235447," https://travis-ci.org/caelum/vraptor4/builds/15235447, 2013.

[8] "Magellan code coverage framework," http://research.microsoft.com/en-us/news/features/magellan.aspx, 2016.

[9] "Bazel," https://bazel.build, 2017.

[10] "FASTBuild," http://www.fastbuild.org/docs/home.html, 2017.

[11] "Rapid release at massive scale," https://engineering.fb.com/web/rapid-release-at-massive-scale, 2017.

[12] "Z3 theorem prover," https://z3.codeplex.com, 2017.

[13] "Design patterns implemented in Java," https://github.com/iluwatar/java-design-patterns, 2018.

[14] "Docker," https://www.docker.com, 2018.

[15] "JSONLD-JAVA," https://github.com/jsonld-java/jsonld-java, 2018.

[16] "Maven Git commit ID plugin," https://github.com/ktoso/maven-git-commit-id-plugin, 2018.

[17] "Multiclass classification," https://en.wikipedia.org/wiki/Multiclass_classification, 2018.

[18] "PIT mutation operators," http://pitest.org/quickstart/mutators, 2018.

[19] "PIT mutation testing," http://pitest.org, 2018.

[20] “Redline Smalltalk,” https://github.com/redline-smalltalk/redline-smalltalk, 2018.

[21] “Spring Data JDBC generic DAO implementation,” https://github.com/nurkiewicz/spring-data-jdbc-repository, 2018.

[22] “Swagger Maven Plugin,” https://github.com/kongchen/swagger-maven-plugin, 2018.

[23] “Travis-CI,” https://travis-ci.org, 2018.

[24] “Travis Docker image,” https://hub.docker.com/r/travisci, 2018.

[25] “VRaptor,” https://github.com/caelum/vraptor4, 2018.

[26] “Elastic-Job,” https://github.com/elasticjob/elastic-job-lite, 2019.

[27] “Elastic-Job pull request 592,” https://github.com/elasticjob/elastic-job-lite/pull/592, 2019.

[28] “GitHub,” https://github.com, 2019.

[29] “iDFlakies: Flaky test dataset,” https://sites.google.com/view/flakytestdataset, 2019.

[30] “iFixFlakies framework,” https://sites.google.com/view/ifixflakies, 2019.

[31] “JavaParser,” http://javaparser.org, 2019.

[32] “JUnit expected annotation,” https://junit.org/junit4/javadoc/4.12/org/junit/Test.html, 2019.

[33] “WildFly application server,” https://github.com/wildfly/wildfly, 2019.

[34] “WildFly bug report,” https://issues.jboss.org/browse/WFLY-11323, 2019.

[35] “Apache Maven project,” https://maven.apache.org, 2020.

[36] “Buck: A high-performance build tool,” https://buck.build, 2020.

[37] “HSAC-fitnesse-fixtures,” https://github.com/fhoeben/hsac-fitnesse-fixtures, 2020.

[38] “Nexus repository helm,” https://github.com/sonatype-nexus-community/nexus-repository-helm, 2020.

[39] “spring-data-ebean,” https://github.com/hexagonframework/spring-data-ebean, 2020.

[40] “Spring Data Envers,” https://github.com/spring-projects/spring-data-envers, 2020.

[41] “Wikidata toolkit,” https://github.com/Wikidata/Wikidata-Toolkit, 2020.

[42] B. Adams, R. Suvorov, M. Nagappan, A. E. Hassan, and Y. Zou, “An empirical study of build system migrations in practice: Case studies on KDE and the Linux kernel,” in *International Conference on Software Maintenance*, 2012, pp. 160–169.

[43] A. Alali, H. Kagdi, and J. I. Maletic, “What’s a typical commit? A characterization of open source software repositories,” in *International Conference on Program Comprehension*, 2008, pp. 182–191.

[44] M. A. Alipour, A. Shi, R. Gopinath, D. Marinov, and A. Groce, “Evaluating non-adequate test-case reduction,” in *International Conference on Automated Software Engineering*, 2016, pp. 16–26.

[45] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, “Learning natural coding conventions,” in *International Symposium on Foundations of Software Engineering*, 2014, pp. 281–293.

[46] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, “Using mutation analysis for assessing and comparing testing coverage criteria,” *IEEE Transactions on Software Engineering*, vol. 32, no. 8, pp. 608–624, 2006.

[47] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke, “Automated software transplantation,” in *International Symposium on Software Testing and Analysis*, 2015, pp. 257–269.

[48] J. Bell and G. Kaiser, “Unit test virtualization with VMVM,” in *International Conference on Software Engineering*, 2014, pp. 550–561.

[49] J. Bell, G. Kaiser, E. Melski, and M. Dattatreya, “Efficient dependency detection for safe Java test acceleration,” in *International Symposium on Foundations of Software Engineering*, 2015, pp. 770–781.

[50] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, “DeFlaker: Automatically detecting flaky tests,” in *International Conference on Software Engineering*, 2018, pp. 433–444.

[51] M. Beller, G. Gousios, and A. Zaidman, “TravisTorrent: Synthesizing Travis CI and GitHub for full-stack research on continuous integration,” in *Mining Software Repositories*, 2017, pp. 447–450.

[52] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu, “The promises and perils of mining Git,” in *Mining Software Repositories*, 2009, pp. 1–10.

[53] J. Black, E. Melachrinoudis, and D. Kaeli, “Bi-criteria models for all-uses test suite reduction,” in *International Conference on Software Engineering*, 2004, pp. 106–115.

[54] H. Borges, A. Hora, and M. T. Valente, “Predicting the popularity of GitHub repositories,” in *International Conference on Predictive Models and Data Analytics in Software Engineering*, 2016, pp. 9:1–9:10.

[55] C. Brindescu, M. Codoban, S. Shmarkatiuk, and D. Dig, “How do centralized and distributed version control systems impact software changes?” in *International Conference on Software Engineering*, 2014, pp. 322–333.

[56] T. T. Chekam, M. Papadakis, and Y. Le Traon, "Mart: A mutant generation tool for LLVM," in *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (Tool Demonstrations Track)*, 2019, pp. 1080–1084.

[57] J. Chen, Y. Bai, D. Hao, L. Zhang, L. Zhang, and B. Xie, "How do assertions impact coverage-based test-suite reduction?" in *International Conference on Software Testing, Verification, and Validation*, 2017, pp. 418–423.

[58] J. Chen, Y. Lou, L. Zhang, J. Zhou, X. Wang, D. Hao, and L. Zhang, "Optimizing test prioritization via test distribution analysis," in *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 656–667.

[59] T. Y. Chen and M. F. Lau, "Heuristics towards the optimization of the size of a test suite," in *International Conference on Software Quality Management*, 1995, pp. 415–424.

[60] ——, "A new heuristic for test suite reduction," *Journal of Information and Software Technology*, vol. 40, no. 5-6, pp. 347–354, 1998.

[61] ——, "A simulation study on some heuristics for test suite reduction," *Journal of Information and Software Technology*, vol. 40, no. 13, pp. 777–787, 1998.

[62] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *IEEE Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[63] S. Dutta, A. Shi, R. Choudhary, Z. Zhang, A. Jain, and M. Sasa, "Detecting flaky tests in probabilistic and machine learning applications," in *International Symposium on Software Testing and Analysis*, 2020, pp. 211–224.

[64] P. Duvall, S. M. Matyas, and A. Glover, *Continuous Integration: Improving Software Quality and Reducing Risk.* Addison-Wesley Professional, 2007.

[65] S. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments," in *International Symposium on Foundations of Software Engineering*, 2014, pp. 235–245.

[66] E. Engström, M. Skoglund, and P. Runeson, "Empirical evaluations of regression test selection techniques: A systematic review," in *International Symposium on Empirical Software Engineering and Measurement*, 2008, pp. 22–31.

[67] H. Esfahani, J. Fietz, Q. Ke, A. Kolomiets, E. Lan, E. Mavrinac, W. Schulte, N. Sanches, and S. Kandula, "CloudBuild: Microsoft's distributed and caching build service," in *International Conference on Software Engineering Companion*, 2016, pp. 11–20.

[68] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," in *International Symposium on Software Testing and Analysis*, 2010, pp. 147–158.

[69] A. Gambi, J. Bell, and A. Zeller, "Practical test dependency detection," in *International Conference on Software Testing, Verification, and Validation*, 2018, pp. 1–11.

[70] Z. Gao, Y. Liang, M. B. Cohen, A. M. Memon, and Z. Wang, "Making system user interactive tests repeatable: When and what should we control?" in *International Conference on Software Engineering*, 2015, pp. 55–65.

[71] J. Geng, Z. Li, R. Zhao, and J. Guo, "Search based test suite minimization for fault detection and localization: A co-driven method," in *Search-Based Software Engineering*, 2016, pp. 34–48.

[72] A. Ghanbari, S. Benton, and L. Zhang, "Practical program repair via bytecode mutation," in *International Symposium on Software Testing and Analysis*, 2019, pp. 19–30.

[73] M. Gligoric, L. Eloussi, and D. Marinov, "Ekstazi: Lightweight test selection," in *International Conference on Software Engineering (Tool Demonstrations Track)*, 2015, pp. 713–716.

[74] ——, "Practical regression test selection with dynamic file dependencies," in *International Symposium on Software Testing and Analysis*, 2015, pp. 211–222.

[75] M. Gligoric, S. Khurshid, S. Misailovic, and A. Shi, "Mutation testing meets approximate computing," in *International Conference on Software Engineering (New Ideas and Emerging Results Track)*, 2017, pp. 3–6.

[76] Google, "Avoiding flakey tests," http://googletesting.blogspot.com/2008/04/tott-avoiding-flakey-tests.html, 2008.

[77] R. Gopinath, C. Jensen, and A. Groce, "Code coverage for suite evaluation for developers," in *International Conference on Software Engineering*, 2014, pp. 72–82.

[78] A. Gotlieb and D. Marijan, "FLOWER: Optimal test suite reduction as a network maximum flow," in *International Symposium on Software Testing and Analysis*, 2014, pp. 171–180.

[79] A. Groce, M. A. Alipour, C. Zhang, Y. Chen, and J. Regehr, "Cause reduction for quick testing," in *International Conference on Software Testing, Verification, and Validation*, 2014, pp. 243–252.

[80] A. Groce, J. Holmes, D. Marinov, A. Shi, and L. Zhang, "An extensible, regular-expression-based tool for multi-language mutant generation," in *International Conference on Software Engineering (Tool Demonstrations Track)*, 2018, pp. 25–28.

[81] J. P. Guilford, *Fundamental Statistics in Psychology and Education*. McGraw-Hill, 1973.

[82] A. Gyori, B. Lambeth, A. Shi, O. Legunsen, and D. Marinov, "NonDex: A tool for detecting and debugging wrong assumptions on Java API specifications," in *International Symposium on Foundations of Software Engineering (Tool Demonstrations Track)*, 2016, pp. 993–997.

[83] A. Gyori, O. Legunsen, F. Hariri, and D. Marinov, "Evaluating regression test selection opportunities in a very large open-source ecosystem," in *International Symposium on Software Reliability Engineering*, 2018, pp. 112–122.

[84] A. Gyori, A. Shi, F. Hariri, and D. Marinov, "Reliable testing: Detecting state-polluting tests to prevent test dependency," in *International Symposium on Software Testing and Analysis*, 2015, pp. 223–233.

[85] R. G. Hamlet, "Testing programs with the aid of a compiler," *IEEE Transactions on Software Engineering*, vol. 3, no. 4, pp. 279–290, 1977.

[86] D. Hao, L. Zhang, X. Wu, H. Mei, and G. Rothermel, "On-demand test suite reduction," in *International Conference on Software Engineering*, 2012, pp. 738–748.

[87] F. Hariri and A. Shi, "SRCIROR: A toolset for mutation testing of C source code and LLVM intermediate representation," in *International Conference on Automated Software Engineering (Tool Demonstrations Track)*, 2018, pp. 860–863.

[88] F. Hariri, A. Shi, H. Converse, D. Marinov, and S. Khurshid, "Evaluating the effects of compiler optimizations on mutation testing at the compiler IR level," in *International Symposium on Software Reliability Engineering*, 2016, pp. 105–115.

[89] F. Hariri, A. Shi, V. Fernando, S. Mahmood, and D. Marinov, "Comparing mutation testing at the levels of source code and compiler intermediate representation," in *International Conference on Software Testing, Verification, and Validation*, 2019, pp. 114–124.

[90] F. Hariri, A. Shi, O. Legunsen, M. Gligoric, S. Khurshid, and S. Misailovic, "Approximate transformations as mutation operators," in *International Conference on Software Testing, Verification, and Validation*, 2018, pp. 285–296.

[91] M. Harman and P. O'Hearn, "From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis," in *International Working Conference on Source Code Analysis and Manipulation*, 2018, pp. 1–23.

[92] M. J. Harrold, R. Gupta, and M. L. Soffa, "A methodology for controlling the size of a test suite," *ACM Transactions on Software Engineering Methodology*, vol. 2, no. 3, pp. 270–285, 1993.

[93] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi, "Regression test selection for Java software," in *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2001, pp. 312–326.

[94] M. J. Harrold, D. Rosenblum, G. Rothermel, and E. Weyuker, "Empirical studies of a prediction model for regression test selection," *IEEE Transactions on Software Engineering*, vol. 27, no. 3, pp. 248–263, 2001.

[95] M. J. Harrold and M. L. Soffa, "An incremental approach to unit testing during maintenance," in *International Conference on Software Maintenance*, 1988, pp. 362–367.

[96] F. Hassan and X. Wang, "Change-aware build prediction model for stall avoidance in continuous integration," in *International Symposium on Empirical Software Engineering and Measurement*, 2017, pp. 157–162.

[97] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y. L. Traon, “Comparing white-box and black-box test prioritization,” in *International Conference on Software Engineering*, 2016, pp. 523–534.

[98] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy, “The art of testing less without sacrificing quality,” in *International Conference on Software Engineering*, 2015, pp. 483–493.

[99] K. Herzig and N. Nagappan, “Empirically detecting false test alarms using association rules,” in *International Conference on Software Engineering*, 2015, pp. 39–48.

[100] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, and D. Dig, “Trade-offs in continuous integration: Assurance, security, and flexibility,” in *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2017, pp. 197–207.

[101] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, “Usage, costs, and benefits of continuous integration in open-source projects,” in *International Conference on Automated Software Engineering*, 2016, pp. 426–437.

[102] C. Hoagland, “Fixing the brittleness problem with GUI tests,” https://www.stickyminds.com/articles/fixing-brittleness-problem-gui-tests, 2014.

[103] C. Huo and J. Clause, “Improving oracle quality by detecting brittle assertions and unused inputs in tests,” in *International Symposium on Foundations of Software Engineering*, 2014, pp. 621–631.

[104] D. Jeffrey and N. Gupta, “Improving fault detection capability by selectively retaining test cases during test suite reduction,” *IEEE Transactions on Software Engineering*, vol. 33, no. 2, pp. 108–123, 2007.

[105] Y. Jia and M. Harman, “An analysis and survey of the development of mutation testing,” *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[106] B. Jiang, Z. Zhang, W. K. Chan, and T. H. Tse, “Adaptive random test case prioritization,” in *International Conference on Automated Software Engineering*, 2009, pp. 233–244.

[107] X. Jin and F. Servant, “A cost-efficient approach to building in continuous integration,” in *International Conference on Software Engineering*, 2020, pp. 13–25.

[108] D. S. Johnson, “Approximation algorithms for combinatorial problems,” *Journal of Computer and System Sciences*, pp. 256–278, 1974.

[109] J. A. Jones and M. J. Harrold, “Test-suite reduction and prioritization for modified condition/decision coverage,” in *International Conference on Software Maintenance*, 2001, pp. 92–102.

[110] R. Just, “The Major mutation framework: Efficient and scalable mutation analysis for Java,” in *International Symposium on Software Testing and Analysis*, 2014, pp. 433–436.

[111] R. Just, M. D. Ernst, and G. Fraser, “Efficient mutation analysis by propagating and partitioning infected execution states,” in *International Symposium on Software Testing and Analysis*, 2014, pp. 315–326.

[112] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, “Are mutants a valid substitute for real faults in software testing?” in *International Symposium on Foundations of Software Engineering*, 2014, pp. 654–665.

[113] M. Krafczyk, A. Shi, A. Bhaskar, D. Marinov, and V. Stodden, “Scientific tests and continuous integration strategies to enhance reproducibility in the scientific software context,” in *International Workshop on Practical Reproducible Evaluation of Computer Systems*, 2019, pp. 23–28.

[114] B. Kunjummen, “JUnit test method ordering,” http://www.java-allandsundry.com/2013/01/junit-test-method-ordering.html, 2013.

[115] A. Labuschagne, L. Inozemtseva, and R. Holmes, “Measuring the cost of regression testing in practice: A study of Java projects using continuous integration,” in *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2017, pp. 821–830.

[116] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta, “Root causing flaky tests in a large-scale industrial setting,” in *International Symposium on Software Testing and Analysis*, 2019, pp. 101–111.

[117] W. Lam, K. Muşlu, H. Sajnani, and S. Thummalapenta, “A study on the lifecycle of flaky tests,” in *International Conference on Software Engineering*, 2020, pp. 1471–1482.

[118] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, “iDFlakies: A framework for detecting and partially classifying flaky tests,” in *International Conference on Software Testing, Verification, and Validation*, 2019, pp. 312–322.

[119] W. Lam, A. Shi, R. Oei, S. Zhang, M. D. Ernst, and T. Xie, “Dependent-test-aware regression testing techniques,” in *International Symposium on Software Testing and Analysis*, 2020, pp. 298–311.

[120] W. Lam, S. Zhang, and M. D. Ernst, “When tests collide: Evaluating and coping with the impact of test dependence,” University of Washington Department of Computer Science and Engineering, Tech. Rep. UW-CSE-15-03-01, 03 2015.

[121] O. Legunsen, F. Hariri, A. Shi, Y. Lu, L. Zhang, and D. Marinov, “An extensive study of static regression test selection in modern software evolution,” in *International Symposium on Foundations of Software Engineering*, 2016, pp. 583–594.

[122] O. Legunsen, A. Shi, and D. Marinov, “STARTS: STAtic Regression Test Selection,” in *International Conference on Automated Software Engineering (Tool Demonstrations Track)*, 2017, p. 949.

[123] X. Li and L. Zhang, "Transforming programs and tests in tandem for fault localization," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 92:1–92:30, 2017.

[124] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *IEEE Transactions on Software Engineering*, vol. 33, no. 4, pp. 225–237, 2007.

[125] J.-W. Lin and C.-Y. Huang, "Analysis of test suite reduction with enhanced tie-breaking techniques," *Journal of Information and Software Technology*, vol. 51, no. 4, pp. 679–690, 2009.

[126] Y. Lu, Y. Lou, S. Cheng, L. Zhang, D. Hao, Y. Zhou, and L. Zhang, "How does regression test prioritization perform in real-world software evolution?" in *International Conference on Software Engineering*, 2016, pp. 535–546.

[127] Q. Luo, K. Moran, and D. Poshyvanyk, "A large-scale empirical comparison of static and dynamic test case prioritization techniques," in *International Symposium on Foundations of Software Engineering*, 2016, pp. 559–570.

[128] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *International Symposium on Foundations of Software Engineering*, 2014, pp. 643–653.

[129] X.-y. Ma, B.-k. Sheng, and C.-q. Ye, "Test-suite reduction using genetic algorithm," in *International Conference on Advanced Parallel Processing Technologies*, 2005, pp. 253–262.

[130] M. Machalica, A. Samylkin, M. Porth, and S. Chandra, "Predictive test selection," in *International Conference on Software Engineering, Software Engineering in Practice*, 2019, pp. 91–100.

[131] T. Mattis and R. Hirschfeld, "Lightweight lexical test prioritization for immediate feedback," *Programming Journal*, vol. 4, pp. 12:1–12:32, 2020.

[132] S. McIntosh, B. Adams, and A. E. Hassan, "The evolution of Java build systems," *Empirical Software Engineering Journal*, vol. 17, pp. 578–608, 2012.

[133] S. McIntosh, M. Nagappan, B. Adams, A. Mockus, and A. Hassan, "A large-scale empirical study of the relationship between build technology and build maintenance," *Empirical Software Engineering Journal*, vol. 20, pp. 1587–1633, 2014.

[134] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, "Taming Google-scale continuous testing," in *International Conference on Software Engineering, Software Engineering in Practice*, 2017, pp. 233–242.

[135] J. Micco, "The state of continuous integration testing @Google," https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/45880.pdf, 2017.

[136] J. D. Morgenthaler, M. Gridnev, R. Sauciuc, and S. Bhansali, "Searching for build debt: Experiences managing technical debt at Google," in *International Workshop on Managing Technical Debt*, 2012, pp. 1–6.

[137] K. Muşlu, B. Soran, and J. Wuttke, "Finding bugs by isolating unit tests," in *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2011, pp. 496–499.

[138] A. Ni and M. Li, "Cost-effective build outcome prediction using cascaded classifiers," in *Mining Software Repositories*, 2017, pp. 455–458.

[139] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," *ACM Transactions on Software Engineering Methodology*, vol. 5, no. 2, pp. 99–118, 1996.

[140] A. J. Offutt, J. Pan, and J. M. Voas, "Procedures for reducing the size of coverage-based test sets," in *International Conference on Testing Computer Software*, 1995, pp. 111–123.

[141] A. Orso, N. Shi, and M. J. Harrold, "Scaling regression testing to large software systems," in *International Symposium on Foundations of Software Engineering*, 2004, pp. 241–251.

[142] M. Papadakis and N. Malevris, "Automatic mutation test case generation via dynamic symbolic execution," in *International Symposium on Software Reliability Engineering*, 2010, pp. 121–130.

[143] Q. Peng, A. Shi, and L. Zhang, "Empirically revisiting and enhancing IR-based test-case prioritization," in *International Symposium on Software Testing and Analysis*, 2020, pp. 324–336.

[144] L. S. Pinto, S. Sinha, and A. Orso, "Understanding myths and realities of test-suite evolution," in *International Symposium on Foundations of Software Engineering*, 2012, pp. 33:1–33:11.

[145] N. Rachatasumrit and M. Kim, "An empirical investigation into the impact of refactoring on regression testing," in *International Conference on Software Maintenance*, 2012, pp. 357–366.

[146] D. S. Rosenblum and E. J. Weyuker, "Predicting the cost-effectiveness of regression testing strategies," in *International Symposium on Foundations of Software Engineering*, 1996, pp. 118–126.

[147] ——, "Using coverage information to predict the cost-effectiveness of regression testing strategies," *IEEE Transactions on Software Engineering*, vol. 23, no. 3, pp. 146–156, 1997.

[148] G. Rothermel, S. Elbaum, A. Malishevsky, P. Kallakuri, and B. Davia, "The impact of test suite granularity on the cost-effectiveness of regression testing," in *International Conference on Software Engineering*, 2002, pp. 130–140.

[149] G. Rothermel and M. J. Harrold, “A framework for evaluating regression test selection techniques,” in *International Conference on Software Engineering*, 1994, pp. 201–210.

[150] ——, “A safe, efficient regression test selection technique,” *ACM Transactions on Software Engineering Methodology*, vol. 6, no. 2, pp. 173–210, 1997.

[151] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong, “An empirical study of the effects of minimization on the fault detection capabilities of test suites,” in *International Conference on Software Maintenance*, 1998, pp. 34–43.

[152] G. Rothermel, M. J. Harrold, J. von Ronne, and C. Hong, “Empirical studies of test-suite reduction,” *Journal of Software Testing, Verification and Reliability*, vol. 12, no. 4, pp. 219–249, 2002.

[153] D. Saff and M. D. Ernst, “Reducing wasted development time via continuous testing,” in *International Symposium on Software Reliability Engineering*, 2003, pp. 281–292.

[154] ——, “An experimental evaluation of continuous testing during development,” in *International Symposium on Software Testing and Analysis*, 2004, pp. 76–85.

[155] R. K. Saha, L. Zhang, S. Khurshid, and D. E. Perry, “An information retrieval approach for regression test prioritization based on program changes,” in *International Conference on Software Engineering*, 2015, pp. 268–279.

[156] W. Schulte and C. Prasad, “Taking control of your engineering tools,” *IEEE Computer*, vol. 46, no. 11, pp. 63–66, 2013.

[157] A. Shi, J. Bell, , and D. Marinov, “Mitigating the effects of flaky tests on mutation testing,” in *International Symposium on Software Testing and Analysis*, 2019, pp. 112–122.

[158] A. Shi, A. Gyori, M. Gligoric, A. Zaytsev, and D. Marinov, “Balancing trade-offs in test-suite reduction,” in *International Symposium on Foundations of Software Engineering*, 2014, pp. 246–256.

[159] A. Shi, A. Gyori, O. Legunsen, and D. Marinov, “Detecting assumptions on deterministic implementations of non-deterministic specifications,” in *International Conference on Software Testing, Verification, and Validation*, 2016, pp. 80–90.

[160] A. Shi, A. Gyori, S. Mahmood, P. Zhao, and D. Marinov, “Evaluating test-suite reduction in real software evolution,” in *International Symposium on Software Testing and Analysis*, 2018, pp. 84–94.

[161] A. Shi, M. Hadzi-Tanovic, L. Zhang, D. Marinov, and O. Legunsen, “Reflection-aware static regression test selection,” *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 187:1–187:29, 2019.

[162] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov, “iFixFlakies: A framework for automatically fixing order-dependent flaky tests,” in *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 545–555.

[163] A. Shi, S. Thummalapenta, S. K. Lahiri, N. Bjørner, and J. Czerwonka, “Optimizing test placement for module-level regression testing,” in *International Conference on Software Engineering*, 2017, pp. 689–699.

[164] A. Shi, T. Yung, A. Gyori, and D. Marinov, “Comparing and combining test-suite reduction and regression test selection,” in *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2015, pp. 237–247.

[165] A. Shi, P. Zhao, and D. Marinov, “Understanding and improving regression test selection in continuous integration,” in *International Symposium on Software Reliability Engineering*, 2019, pp. 228–238.

[166] D. Silva, R. Terra, and M. T. Valente, “Recommending automated extract method refactorings,” in *International Conference on Program Comprehension*, 2014, pp. 146–156.

[167] G. Soares, B. Catao, C. Varjao, S. Aguiar, R. Gheyi, and T. Massoni, “Analyzing refactorings on software repositories,” in *Brazilian Symposium on Software Engineering*, 2011, pp. 164–173.

[168] A. Srivastava and J. Thiagarajan, “Effectively prioritizing tests in development environment,” in *International Symposium on Software Testing and Analysis*, 2002, pp. 97–106.

[169] A. Telea and L. Voinea, “A tool for optimizing the build performance of large software code bases,” in *European Conference on Software Maintenance and Reengineering*, 2008, pp. 323–325.

[170] S. Thorve, C. Shrestha, and N. Meng, “An empirical study of flaky tests in Android apps,” in *International Conference on Software Maintenance and Evolution (New Ideas and Emerging Results Track)*, 2018, pp. 534–538.

[171] S. Thummalapenta, P. Devaki, S. Sinha, S. Chandra, S. Gnanasundaram, D. D. Nagaraj, and S. Sathishkumar, “Efficient and change-resilient test automation: An industrial case study,” in *International Conference on Software Engineering*, 2013, pp. 1002–1011.

[172] N. Tsantalis and A. Chatzigeorgiou, “Identification of move method refactoring opportunities,” *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 347–367, 2009.

[173] R. H. Untch, A. J. Offutt, and M. J. Harrold, “Mutation analysis using mutant schemata,” in *International Symposium on Software Testing and Analysis*, 1993, pp. 139–148.

[174] M. Vakilian, R. Sauciuc, J. D. Morgenthaler, and V. Mirrokni, “Automated decomposition of build targets,” in *International Conference on Software Engineering*, 2014, pp. 123–133.

[175] M. Vasic, Z. Parvez, A. Milicevic, and M. Gligoric, “File-level vs. module-level regression test selection for .NET,” in *International Symposium on Foundations of Software Engineering (Tool Demonstrations Track)*, 2017, pp. 848–853.

[176] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur, "Effect of test set minimization on fault detection effectiveness," in *International Conference on Software Engineering*, 1995, pp. 41–50.

[177] W. E. Wong, J. R. Horgan, A. P. Mathur, and A. Pasquini, "Test set size minimization and fault detection effectiveness: A case study in a space application," in *International Computer Software and Applications Conference*, 1997, pp. 522–529.

[178] H. Wright, D. Jasper, M. Klimek, C. Carruth, and Z. Wan, "Large-scale automated refactoring using ClangMR," in *International Conference on Software Maintenance*, 2013, pp. 548–551.

[179] J. Wuttke, K. Muşlu, S. Zhang, and D. Notkin, "Test dependence: Theory and manifestation," University of Washington, CSE, Tech. Rep. UW-CSE-13-07-02, 07 2013.

[180] Z. Xie and M. Li, "Cutting the software building efforts in continuous integration by semi-supervised online auc optimization," in *International Joint Conference on Artificial Intelligence*, 2018, pp. 2875–2881.

[181] R. Yandrapally, S. Thummalapenta, S. Sinha, and S. Chandra, "Robust test automation using contextual clues," in *International Symposium on Software Testing and Analysis*, 2014, pp. 304–314.

[182] S. Yoo and M. Harman, "Pareto efficient multi-objective test case selection," in *International Symposium on Software Testing and Analysis*, 2007, pp. 140–150.

[183] ——, "Regression testing minimization, selection and prioritization: A survey," *Journal of Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[184] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[185] L. Zhang, "Hybrid regression test selection," in *International Conference on Software Engineering*, 2018, pp. 199–209.

[186] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei, "Bridging the gap between the total and additional test-case prioritization strategies," in *International Conference on Software Engineering*, 2013, pp. 192–201.

[187] L. Zhang, D. Marinov, and S. Khurshid, "Faster mutation testing inspired by test prioritization and reduction," in *International Symposium on Software Testing and Analysis*, 2013, pp. 235–245.

[188] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid, "An empirical study of JUnit test-suite reduction," in *International Symposium on Software Reliability Engineering*, 2011, pp. 170–179.

[189] L. Zhang, S.-S. Hou, J.-J. Hu, T. Xie, and H. Mei, “Is operator-based mutant selection superior to random mutant selection?” in *International Conference on Software Engineering*, 2010, pp. 435–444.

[190] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin, “Empirically revisiting the test independence assumption,” in *International Symposium on Software Testing and Analysis*, 2014, pp. 385–396.

[191] H. Zhong, L. Zhang, and H. Mei, “An experimental study of four typical test suite reduction techniques,” *Journal of Information and Software Technology*, vol. 50, no. 6, pp. 534–546, 2008.

[192] C. Zhu, O. Legunsen, A. Shi, and M. Gligoric, “A framework for checking regression test selection tools,” in *International Conference on Software Engineering*, 2019, pp. 430–441.

[193] C. Ziftci and J. Reardon, “Who broke the build?: Automatically identifying changes that induce test failures in continuous integration at Google scale,” in *International Conference on Software Engineering*, 2017, pp. 113–122.