FEDERAL UNIVERSITY OF PERNAMBUCO

Pós-Graduação em Ciência da Computação

JEANDERSON BARROS CÂNDIDO

TEST SUITE PARALLELIZATION IN OPEN-SOURCE PROJECTS: A STUDY ON ITS USAGE AND IMPACT

RECIFE

2018

Jeanderson Barros Cândido

TEST SUITE PARALLELIZATION IN OPEN-SOURCE PROJECTS: A STUDY ON ITS USAGE AND IMPACT

A M.Sc. Dissertation presented to the Center for Informatics of Federal University of Pernambuco in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Advisor: Marcelo Bezerra d'Amorim

RECIFE
2018

Catalogação na fonte
Bibliotecário Jefferson Luiz Alves Nazareno CRB 4-1758

C217t Cândido, Jeanderson Barros.
Test suite parallelization in open-source projects: a study on its usage and impact / Jeanderson Barros Cândido . – 2018.
52 f.: fig., tab.

Orientador: Marcelo Bezerra d`Amorim.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. Cin. Ciência da Computação. Recife, 2018.
Inclui referências.

1. Engenharia de software. 2. Teste de software. 3. Paralelismo de testes. I. d`Amorim, Marcelo Bezerra. (orientador). II. Título

005.1 CDD (22. ed.) UFPE-MEI 2018-73

Jeanderson Barros Cândido

Test Suite Parallelization in Open-Source Projects: A Study on Its Usage and Impact

A M.Sc. Dissertation presented to the Center for Informatics of Federal University of Pernambuco in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Approved on: April 10th, 2018

COMMITTEE MEMBERS

_____

Prof. Fernando José Castor de Lima Filho
Centro de Informática / UFPE

_____

Prof. Eduardo Magno Lages Figueiredo
Departamento de Ciência da Computação / UFMG

_____

Prof. Marcelo Bezerra d'Amorim
Centro de Informática/UFPE
(Advisor)

I dedicate this thesis to all my family, friends and professors who gave me the necessary support to get here.

ACKNOWLEDGEMENTS

I would not make this far without the help and support from some people that have followed me in this journey. I am a very fortunate person for having such wonderful parents. They have always trusted me and gave me the necessary support and freedom to chase my dreams. I dedicate this work to my family for being always present in my file, in the bad and good moments, and always will be my reference. Everything that I've accomplished so far would not be possible without their unconditional support, and I will never be able to express my gratitude into words.

My interest for Software Engineering and research appeared while I was still an undergraduate student in Computer Science at UFCG, in Campina Grande. I had excellent professors but two of them had a huge influence in my career and encouraged me to apply in graduate school at UFPE. Thank you very much to professors Tiago Massoni and Rohit Gheyi for giving me opportunity to talk and mentoring me during my staying at UFCG.

I am very grateful for the friends that I made at CIn-UFPE. In special, my academic brothers Luis Melo, Igor Simões, and Davino Junior – the last was a close friend from our group. You were my family while I was at UFPE. I am very thankful for the many funny moments and hangouts we had during that moment, not to mention the uncountable late hours in the lab helping each other. I wish all the success in the world for you guys because you are very passionate and hard workers. You deserve it! I am thankful for my advisor, Marcelo d'Amorim, for accepting me and walking beside me during this journey. Since the beginning, he was present and always encouraged me to give my best. Thank you very much to professor Leopoldo Teixeira for staying with me in the beginning while Marcelo was at Georgia Tech, and also for the many conversations followed by a tasty cup of coffee. I am also thankful to the all the members of GENTES for all the lessons and learning experiences from the many presentations. Thanks to FACEPE for funding my work.

Finally, I am very grateful to my girlfriend for supporting and staying by my side during this moment of my life, to the friends from UFCG, to the friends from Lar dos Estudante, and to my friends from Cidade Viva for the funny moments. I am thankful to God for all opportunities and for everything that I have been living.

To follow the path, look to the master, follow the master, walk with the master, see through the master, become the master.

—UNKNOWN AUTHOR

ABSTRACT

Testing is a costly process but essential in the development process. Complex systems may contain long-running test suites. Dealing with high testing costs remains an important problem in Software Engineering despite being under active research for years. Test suite parallelization is an important approach to address this problem, given the popularity of multi-core processors and native suport from testing frameworks and build systems. This work reports our findings on the usage and impact of test suite parallelization in open-source projects. This study brings to light the benefits and burdens of that approach. It provides recommendations to practitioners and tool developers to speed up test execution. Considering a set of 468 popular Java projects we analyzed, we found that 24% of the projects contain costly test suites but parallelization features still seem underutilized in practice — only 19.1% of costly projects use parallelization. The main reported reason for adoption resistance was the concern to deal with concurrency issues. Results suggest that, on average, developers prefer high predictability than high performance in running tests.

Keywords: Software Engineering. Software Testing. Test Parallelization

RESUMO

Teste de software é um processo custoso mas essencial no processo de desenvolvimento. Sistemas complexos podem demandar várias horas para executar toda bateria de testes. Por tanto, estratégias para mitigar os custos de teste continuam sendo um tema importante na Engenharia de Software. Paralelismo de suites de teste é uma abordagem importante para lidar com este problema, dada a popularidade de processadores multi-core e suporte nativo de bibliotecas de teste e sistemas de build. Este trabalho reporta nossas descobertas a respeito da utilização e impacto de paralelismo de suites de teste em projetos de código aberto. Este estudo destaca os benefícios e desafios desta abordagem e provê recomendações para programadores e desenvolvedores de ferramentas para acelerar a execução de testes. Considerando um conjunto de 468 projetos populares desenvolvidos em Java, nós vimos que 24% destes projetos possuem suites de testes que demandam um alto custo de tempo de execução, porém o uso de paralelismo de suites de teste ainda é subutilizado na prática — apenas 19.1% destes projetos utilizam paralelismo. O principal motivo de resistência para a adoção desta abordagem é o receio de lidar com problemas de concorrência. Os resultados sugerem que, em média, desenvolvedores preferem alta previsibilidade ao invés de alta performance na execução de testes.

Palavras-chave: Engenharia de Software. Teste de Software. Paralelismo de Testes

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

SIMD	Simple Instruction Multiple Data
CI	Continuous Integration
CUT	Cloud Unit Testing
GPU	Graphics Processing Unit
JVM	Java Virtual Machine
C0	Sequential
C1	Sequential Classes with Parallel Methods
C2	Parallel Classes with Sequential Methods
C3	Parallel Classes with Parallel Methods
FC0	Forked JVMs with Sequential Methods
FC1	Forked JVMs with Parallel Methods

CONTENTS

1

INTRODUCTION

Dealing with high testing costs has been an important problem in software engineering research and industrial practice. As software evolves, it is likely that the size of the test suites increases, as well. Developers often add new tests to check bug fixes and to evaluate new features added to the code base [Pinto et al., 2012]. Consequentially, the demand of time and computing resources to test the software may also increase. In the limit, this increase becomes impractical and requires efficient strategies to overcome the costs of testing.

Several approaches have been proposed in the research literature to address the regression testing problem, with the focus mainly on test suite minimization, prioritization, and selection [Yoo and Harman, 2012]. These techniques assist regression testing either by discovering a subset of relevant tests to execute (i.e., test minimization and test selection) or reordering test scheduling to increase early fault detection (i.e., test case prioritization). In industry, the focus has been mainly on distributing the testing workload on different machines. Evidence of this approach are the Google TAP system [Google Engineering Tools, 2011; Google TechTalks, 2010] and the Microsoft CloudBuild system [Schulte and Prasad, 2013], which provide distributed environments to efficiently build massive amounts of code and run tests. Another popular approach is to build in-house server clusters to distribute testing workloads. For example, as of August 2013, the test suite of the Groupon PWA system, which powers the `groupon.com` website, included over 19K tests. To run all those tests under 10m, Groupon used a cluster of 4 computers with 24 cores each [Kim et al., 2013]. At large organizations, the alternative of renting cloud services [Clutch, 2018] or even building proprietary infrastructures for running tests is a legitimate approach to mitigate the regression testing problem. For these cases, the use of commodity hardware is an attractive solution for running tests. However, for projects with modest or nonexistent budgets and yet relatively heavy testing workloads, this solution may not be economically viable.

The proliferation of multi-core CPUs and the increasing popularization of testing frameworks and build systems, which today provide mature support for parallelization, enable speedups

through increased CPU usage. These two elements — demand for cost-effective test execution and supply of relatively inexpensive testing infrastructures — inspired us to investigate test suite parallelization in practice. Note that parallelization is complementary to other approaches to mitigate testing costs such as (safe) test selection [Rothermel and Harrold, 1997; Gligoric et al., 2015] and continuous integration [Saff and Ernst, 2003].

1.1 Research Methodology

The main purpose of our study is to understand the usage and impact of test parallelization in real projects. We conduct our investigation according to four dimensions of analysis. To understand the aspect of usage, we propose the dimensions *feasibility* and *adoption*. Similarly, to understand the impact of test parallelization, we propose the dimensions *speedup* and *tradeoffs*.

The dimension *feasibility* measures the potential of parallelization to reduce testing costs. In the limit, parallelization would be fruitless if all projects had short-running test suites or if the execution cost was dominated by a single test case in the suite. The dimension *adoption* evaluates how often existing open-source projects use parallelization schemes and how developers involved in costly projects (not using test suite parallelization) perceive this technology. It is important to measure resistance of practitioners to the technology and to understand their reasons. The dimension *speedup* evaluates the observed impact of parallelization in running times. Finally, the dimension *tradeoffs* evaluates the relationship between speedups obtained with parallelization and issues that arise when running tests in parallel, most notably test flakiness [Luo et al., 2014; Bell et al., 2015]. A "flaky" test is a test with non-deterministic outcome (i.e., may pass or fail) under the same circumstances. There are several causes that may cause this non-deterministic behavior, including data races on test execution. In the following, we present our research questions grouped by dimensions, and we elaborate how each research question contributes to the dimensions of our study:

FEASIBILITY OF TEST SUITE PARALLELIZATION

- **RQ1** *How prevalent are time-consuming test suites?*
- **RQ2** *How is time distributed across test cases?*

ADOPTION OF TEST SUITE PARALLELIZATION

- **RQ3** *How popular is test suite parallelization?*
- **RQ4** *What are the main reasons that prevent developers from using test suite parallelization?*

SPEEDUPS OF TEST SUITE PARALLELIZATION

- **RQ5** *What are the speedups obtained with parallelization (in projects that actually use it)?*
- **RQ6** *How test execution scales with the number of available CPUs?*

TRADEOFFS OF TEST SUITE PARALLELIZATION

- **RQ7** *How parallel execution configurations affect testing costs and flakiness?*

On research question RQ1, we are interested in understanding the occurrence of costly test suites in popular open-source projects. Those projects are relevant to our study because they help us to better understand how maintainers approach the cost of testing. In addition, they allow us to investigate the challenges and benefits of applying test parallelization. On research question RQ2, we are interested in understanding how each test case contributes to the total cost of execution. Recall that parallelizing test execution would be ineffective if a sufficiently small subset of tests dominate the test workload. Research question RQ3 helps us to measure the usage of test parallelization in practice and RQ4 addresses the barriers by the perspective of the developers. Research question RQ5 evaluates the impact of parallelization and RQ6 evaluates how the underlying hardware influences the speedups obtained. Finally, research question RQ7 addresses the relationship between speedup and the rate of failures manifested by the non-deterministic execution of tests. We analyze those failures to give a direction towards the conception of techniques aimed at the safe scheduling of tests.

1.2 Statement of the Contributions

This work reports an empirical study we conducted to analyze the usage and impact of test parallelization to speed up testing in open-source projects. We provide recommendations to practitioners and developers of new techniques and tools aiming to speed up test execution with parallelization. We summarize the main contributions of this work in the following:

1. Feedback from developers about the major concerns on adopting test parallelization;
2. Empirical evidence supporting the importance of parallelization to speed up test execution;
3. Evaluation of different parallelization schemes on test suites from real projects;
4. Guidelines to adopt test parallelization and directions improve existing tooling support.

This work was developed in collaboration with my colleague Luis Melo under the supervision of Prof. Marcelo d'Amorim. We published our results in the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017) [Candido et al., 2017]. The artifacts we produced as result of this study, including supporting scripts and the full list of projects, are publicly available in our website:

`https://jeandersonbc.github.io/testsuite-parallelization/`

1.3 Outline

The rest of this work is structured as follows. Chapter 2 presents an overview of the parallel execution of test suites and elaborates the support of parallelization features on testing frameworks and build systems. Chapter 3 presents our methodology to find subjects to conduct the study and describes our data set. Chapter 4 presents the setup we used and elaborates the methodology and results of our experiments according to the dimensions presented. Chapter 5 discusses the lessons we learned and provides recommendations to practitioners interested in parallelization and tool developers. Chapter 6 presents the threats to validity of this work and chapter 7 discusses related works. Finally, chapter 8 concludes this dissertation and elaborates future work.

2

BACKGROUND

In this chapter, we explain the main concepts used in our work. Initially, in Section 2.1, we provide an overview of test parallelization and define the scope of our study. In Section 2.2 and Section 2.3, respectively, we explain how testing frameworks and build systems enable parallel execution of tests.

2.1 Overview

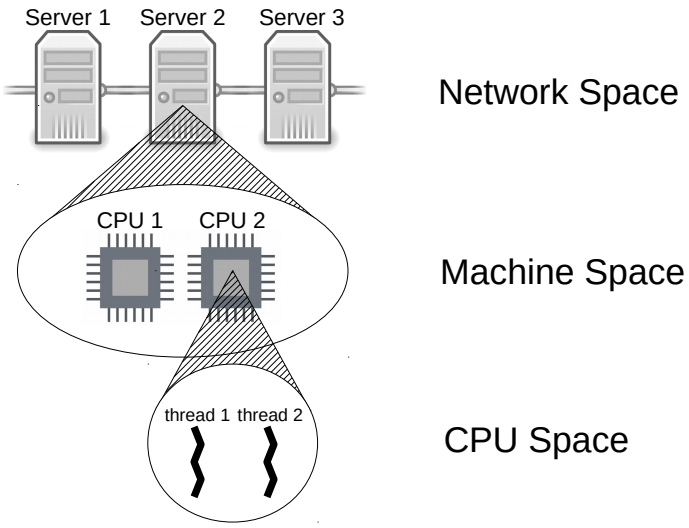
It is intuitive to reasoning about test execution as a linear process where tests run one after the other; however, they can run in parallel at different levels: from distributed environments in the cloud to multithreaded execution within a single machine. Figure 1 illustrates these different levels of test parallelization. The highest level indicates parallelism obtained through different machines on the network space. For instance, using several machines from a cloud service to offload test execution. The lower levels (i.e., machine and CPU spaces) denote parallelism obtained within a single machine. This form of parallelism is enabled through build systems and testing frameworks. For instance, on the machine space, the developer could configure the build system to spawn multiple processes proportional to the number of cores available in the CPU to run the tests. Similarly, on the CPU space, tests could be executing on different threads within a process.

It is important to note that a variety of build systems and testing frameworks [JUnit, 2018; NUnit, 2018; TestNG, 2018] provide today support for parallel test execution as to benefit from the available power of popular multi-core processors. Forking operating system processes to run test jobs is the basic mechanism of build systems to obtain parallelism at the machine space. For Java-based build systems, such as Maven and Gradle, this amounts to spawning a JVM, on a given CPU, to handle a test job and aggregating results when jobs finish. Multithreaded execution is another mechanism to support test parallelization. Popular testing frameworks support the use of multiple threads to run tests at the CPU space rather than a single

thread. Furthermore, notice that these different levels of parallelism (i.e., network, machine, and CPU) are complementary: the lower levels leverage the computing power of server nodes whereas the highest level leverages the aggregate processing power of a network of machines.

In this work, we focus on low-level parallelism, where computation can be offloaded at different CPUs within a machine and at different threads within each process. In the following, we elaborate relevant features of testing frameworks and build systems for parallelization. We focus on Java, Maven, and JUnit in this work but the discussion can be generalized to other languages and tools.

Figure 1: Levels of parallelism.



2.2 Testing Frameworks

Popular testing frameworks may provide different ways to enable parallel execution. For instance, JUnit provides a test runner with parallelization enabled while TestNG offers an API to annotate classes to run in parallel. Despite the different forms to enable parallel execution, the resulting effect is equivalent. In the following, we show the general choices to control parallelism within a Java Virtual Machine (JVM).

- **Sequential (C0).** No parallelism is involved.
- **Sequential Classes with Parallel Methods (C1).** This configuration corresponds to running test classes sequentially, but running test methods from those classes concurrently.
- **Parallel Classes with Sequential Methods (C2).** This configuration corresponds to running test classes concurrently, but running test methods sequentially.

- **Parallel Classes with Parallel Methods (C3).** This configuration runs test classes and methods concurrently.

Notice that an important aspect in deciding which configuration to use (or in designing new test suites) is the possibility of race conditions on shared data during execution. Data sharing can occur, for example, through a state that is reachable from statically-declared variables in the program or through variables declared within the scope of the test class or even through resources available on the file system and the network [Luo et al., 2014]. Considering data race avoidance, configuration C1 is preferable over C2 when it is clear that test methods in a class do not manipulate shared state, which can be challenging to determine [Bell et al., 2015]. Similarly, C2 is preferable over C1 when it is clear that several test methods in a class perform operations involving shared data. Configuration C3 does not restrict scheduling orderings. Consequently, this configuration is more likely to manifest data races during execution when tests change the state of some shared resource. Note that speedups obtained with a particular configuration depend on several factors, including the test suite size and distribution of test methods per class. For instance, considering a small number of long test classes (i.e., a large number of test methods per class), the configuration C1 is more like to achieve a higher speedup over C2 since more tests could be executing at the same time in parallel, given enough processing power.

2.3 Build Systems

Recall from Section 2.1 that forking processes to run test jobs is the foundation for test parallelization enabled by build systems. The list below shows the choices to control parallelism through the build system.

- **Forked JVMs with Sequential Methods (FC0).** The build system spawns multiple JVMs with this configuration, assigning a partition of the set of test classes to each JVM. Test classes and methods run sequentially within each JVM.
- **Forked JVMs with Parallel Methods (FC1).** With this configuration, the build system forks multiple JVMs, assigning a partition of the set of test classes to each JVM as FC0 does. Each JVM run test methods concurrently, as C1 does.

Conceptually, the configuration *Parallel Classes with Sequential Methods* (C2) and FC0 are similar: test classes run in parallel and test methods sequentially. Similarly, the configuration *Parallel Classes with Parallel Methods* (C3) and FC1 are conceptually similar as test methods from multiple test classes are running at the same time. Although threads introduce

less overhead compared to operating system processes, the configurations with forked JVMs provides an implicit memory-access protection to the tests since each process has access only to their corresponding memory space. Maven offers an option to reuse JVMs that can be used to attenuate the potentially high cost of spawning new JVM processes on every test class (if reuse is enabled). Although this option is enabled by default, in practice, it is common to disable this functionality to achieve test isolation: after the execution of a test class, the corresponding JVM terminates and the build system starts a new JVM to handle the next test class. As discussed in the previous section (see Section 2.2), the design of the tests plays an important role when deciding which configuration better suits for the test execution.

Note from the listing that forking can only be combined with configuration C1 (see Section 2.2) as Maven made the design choice to only accept one test class at a time per forked process. In theory, it could be possible to combine forking with other options of multithreaded execution but existing build tools do not provide this option out-of-the-box. Maven provides these features through its test plugin Maven Surefire [Apache, 2017a]. Maven Surefire is a plugin responsible for activities related to testing during the build cycle, and it allows one to configure the test execution with different settings (e.g., testing library, library version, tests to execute by default, and configuring parallel execution).

To illustrate the usage of parallelism configurations in practice, Figure 2 shows a fragment of a Maven configuration file, known as *pom.xml*, with the *Forked JVMs with Parallel Methods* (FC1) configuration defined. With this configuration, Maven forks one JVM per core (`forkCount` parameter) and uses five threads (`threadCount` parameter) to run test methods (`parallel` parameter) within each forked JVM. Maven reuses created JVMs on subsequent forks when execution of a test class terminates (`reuseFork` parameter). By changing the values on the configuration, it is possible to use other configurations mentioned on this chapter.

Figure 2: Configuration Forked JVMs with Parallel Methods (FC1) on Maven.

```
1  <plugin>
2      <groupId>org.apache.maven.plugins</groupId>
3      <artifactId>maven-surefire-plugin</artifactId>
4      <configuration>
5          <forkCount>1C</forkCount>
6          <reuseForks>true</reuseForks>
7          <parallel>methods</parallel>
8          <threadCount>5</threadCount>
9      </configuration>
10 </plugin>
```

3

OBJECT OF ANALYSIS

To conduct our study, we considered open-source projects from Github, a popular hosting service with more than 78 million projects [Github, 2018]. We used Github's search API [Github Developer, 2018] to identify projects that satisfy the following criteria:

1. Java is the primary language[1];
2. the project has at least 100 stars;
3. the latest update was on or after January 1st, 2016;
4. the `readme` file contains the string "*mvn*".

We focused on Java for its popularity. Although there is no clearcut limit on the number of Github stars [Github Help, 2018] to define relevant projects, we observed that one hundred stars was enough to eliminate trivial subjects. The third criteria serves to skip projects without recent activity. The fourth criteria is an approximation to find Maven projects. We focused on Maven for its popularity on Java projects. Important to highlight that, as of now, the Github's search API can only reflect contents from repository statistics (e.g., number of forks, main programming language); it does not provide a feature to search for projects containing certain files (e.g., pom.xml) in the directory structure. Figure 3 illustrates the query to the Github API as an HTTP request. The result set is sorted in descending order of stars.

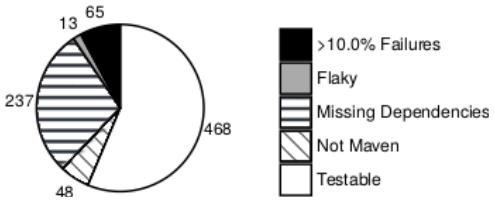
Figure 3: Query to the Github API for projects that (1) use Java, (2) contains at least 100 stars, (3) has been updated on January 1st, 2016 (or later), (4) contains the string "*mvn*" in the `readme` file.

```
1 https://api.github.com/search/repositories?q=language:java
2     +stars:>=100+pushed:>=2016+mvn%20in:readme&sort=stars
```

[1] In case of projects in multiple languages, the Github API considers the predominant language as the primary language.

We used the following methodology to select projects for analysis. After obtaining the list of potential projects from Github, we filtered those containing a *pom.xml* file in the root directory. Then, considering this set of Maven projects, we executed the tests for three times to discard those projects with issues in the build file and non-deterministic results observed from sequential executions. As of August 25th 2017, our search criteria returned a total of 831 subjects. From this set of projects, 48 projects were not Maven or did not have a *pom.xml* in the root directory, 237 projects were not considered because of environment incompatibility (e.g., missing DBMS), 13 projects were discarded because of test flakiness. Recall that a "flaky" test is a test that may pass or fail under the same circumstances [Luo et al., 2014]. As some of our experiments consist of running tests on different threads, we ignored these projects as it would be impractical to identify whether a test failed due to a race condition or some other source of flakiness. From the remaining 533 projects with deterministic results, we eliminated 65 projects with 10% or more failing tests as to reduce bias. For the remaining projects with failing tests, we used the JUnit's `@Ignore` annotation to ignore failing tests. Our final set of subjects contains 468 projects. Figure 4 summarizes our sample set.

Figure 4: Distribution of projects: from the initial sample of 831 projects, we ignored 48 projects without Maven support, 237 with missing dependencies, 13 projects with flaky tests, and 65 projects had at least 10% of failing tests. We considered 468 projects to conduct our study.



To run our experiments, we used a Core i7-4790 (3.60 GHz) Intel processor machine with eight virtual CPUs (four cores with two native threads each) and 16GB of memory, running Ubuntu 14.04 LTS Trusty Tahr (64-bit version). We configured our workstation to only run essential services as to avoid noise from unrelated operating system events. The machine was dedicated to our experiments and we accessed it via SSH. In addition, we configured the `isolcpus` option from the Linux Kernel [Kernel.org, 2018] to isolate six virtual CPUs to run our experiments, leaving the remaining CPUs to run operating system processes [Unix StackExchange Community, 2018]. The rationale for this decision is to prevent context-switching between user processes (running the experiment) and OS-related processes. We used Java 8 and Maven 3.3.9 to build projects and run test suites. To process test results and generate plots we used Python, Bash, R and Ruby. All source artifacts are publicly available for replication on our website (`https://jeandersonbc.github.io/testsuite-parallelization/`). This includes supporting scripts and the full list of projects.

4

EVALUATION

In this chapter, we elaborate the evaluation of our experiments. We describe the methodology, execution, and results for each research question presented in Section 1.1. Research questions are grouped by dimension of analysis and each dimension is grouped in a dedicated section.

4.1 Feasibility

This dimension evaluates the opportunities for test parallelization. Specifically, we are interested in time-consuming test suites and in evaluating how the time cost is distributed across test cases. In the limit, parallelization would be fruitless if all projects had short-running test suites or if the execution cost was dominated by a single test case in the suite.

4.1.1 How prevalent are time-consuming test suites?

To evaluate prevalence of projects with time-consuming test suites, we considered the 468 projects (Figure 4) identified in Chapter 3. Figure 5 illustrates the script we used to measure time. We took the following actions to isolate our environment from measurement noise. In addition to using a dedicated workstation with isolated CPUs (see last paragraph in page 23), we observed that some test tasks called test-unrelated tasks (e.g., *javadoc* generation and static analyses) that could interfere in our time measurements. To address that potential issue, we inspected Maven execution logs from a sample including a hundred projects prior to running the script from Figure 5. The tasks we found were ignored from execution (lines 1-4). Also, to make sure our measurements were fair, we compared timings corresponding to the sequential execution of tests using Maven with that obtained with JUnit's default `JUnitCore` runner, invoked from the command line. Results were very close.

The main loop (lines 6-15) of the script in Figure 5 iterates over the list of subjects and invokes Maven multiple times (lines 8-11). It first makes all dependencies available locally (line

Figure 5: Bash script to measure time cost of test suites. For each subject, we fetch all dependencies, compile the source and test files, and execute the tests in offline mode ignoring non-related tasks.

```
1  MAVEN_SKIPS="-Drat.skip=true -Dmaven.javadoc.skip=true \
2      -Djacoco.skip=true -Dcheckstyle.skip=true \
3      -Dfindbugs.skip=true -Dcobertura.skip=true \
4      -Dpmd.skip=true -Dcpd.skip=true"
5
6  for subj in $SUBJECTS; do
7      cd $SUBJECTS_HOME/$subj
8      mvn clean dependency:go-offline
9      mvn test-compile install -DskipTests $MAVEN_SKIPS \
10         &> compile.log
11     mvn test -o -fae $MAVEN_SKIP &> testing.log
12     cat testing.log \
13         | grep --text "\[INFO\] Total time:" \
14         | tail -n 1
15 done
```

Figure 6: Distribution of running times per cost group.

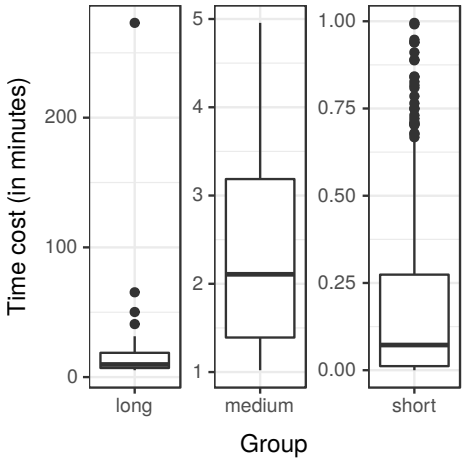
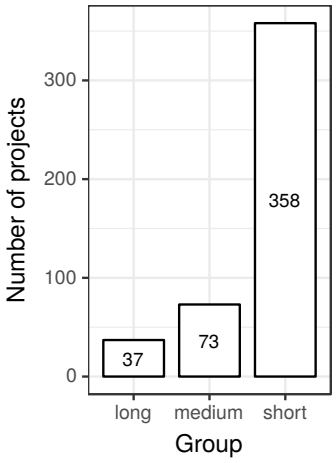


Figure 7: Number of projects in each cost group



8), compiles the source and test files (line 9), and then runs the tests in offline mode as to skip the package update task, enabled by default (line 11). After that, we used a regular expression on the output log to find elapsed times (line 12-14).

We ran the test suite for each subject three times, reporting averaged execution times in three ranges: tests that run within a minute (short), tests that run in one to five minutes (medium), and tests that run in five or more minutes (long). Figure 7 shows the number of projects in each group. As expected, long and medium projects do not occur as frequently as short projects. However, they do occur in relatively high numbers. Figure 6 shows the distribution of execution time of test suites in each of these groups. Note that the y-ranges are different. The distribution associated with the short group is the most unbalanced (right skewed). The test suites in this group ran in 15 or less seconds for over 75% of the cases.

Considering the groups medium and long, however, we found many costly executions. Nearly 75% of the projects from the medium group take 3.5 or more minutes to run and nearly 75% of the projects from the long group take around 20 minutes to run. We found cases in the long group were execution takes more than 50 minutes to complete, as can be observed from the outliers in the boxplot.

It is important to note that we underestimated running times as we missed test modules not enabled for execution in the root *pom.xml*. For instance, the project `apache.maven-surefire` runs all unit tests in a few seconds. According to our criteria, this project is classified as short but a closer look reveals that only smoke tests are executed in this project by default. In this project, integration and system tests, which take longer to run, are only accessible via custom parameters, which we do not handle in our experimental setup. We enabled such parameters for this specific project and observed that testing time goes to nearly 30 minutes. For simplicity, we considered only the tests executed by default. From the 468 testable projects, 400 successfully executed all tests and 68 reported some test failure. From these 68 subjects (39, 22, and 7, from the groups short, medium, and long, respectively) only 11 subjects have more than 5% of failing tests (7.3% on average).

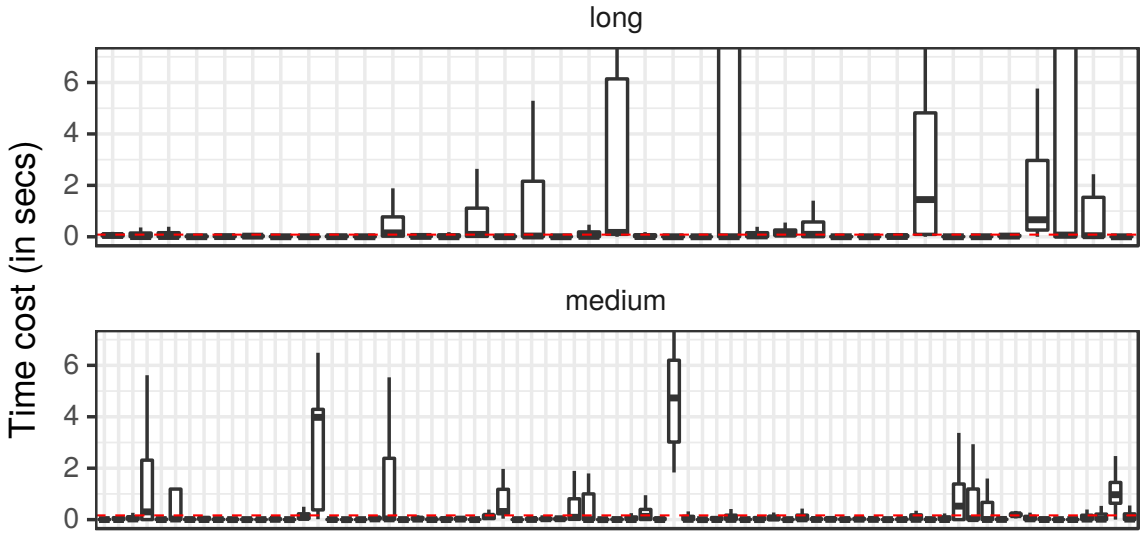
Answering RQ1: *We conclude that time-consuming test suites are relatively frequent in open-source projects. We found that 24% of the 468 projects we analyzed (i.e., nearly 1 in every 4 projects) take at least 1 minute to run and 8% of them take at least 5 minutes to run.*

4.1.2 How is time distributed across test cases?

Section 4.1.1 showed that medium and long-running projects are not uncommon, accounting for nearly 24% of the 468 projects we analyzed. Research question RQ2 measures the distribution of test costs in test suites of the 110 costly projects identified previously (i.e., medium and long groups). In the limit, if cost is dominated by a single test from a large test suite, it is unlikely that parallelization will be beneficial as a test method is the smallest working unit in test frameworks.

Figure 8 shows the time distribution of individual test cases per project. We observed that the average median times (see dashed horizontal red lines) were small, namely 0.08s for medium projects and 0.16s for long projects, and the standard deviations associated with each distribution were relatively low. High values of σ are indicative of CPU monopolization. We found only a small number of those. The highest value of σ occurred in `uber_chaperone`,

Figure 8: Distribution of test case time per project.



a project from the long group. This project contains only 26 tests, 17 of which take less than 0.5s to run, one of which takes nearly 3s to run, two of which take nearly 11s to run, four of which takes on average 3m to run, and two of which take ∼8m to run. For this project, 98.4% of the execution cost is dominated by 20% of the tests; without these two costly tests this project would have been classified as short-running. We did not find other projects with such extreme time monopolization profile. Project `facebookarchive_linkbench` is also classified as long-running and has the second highest value of σ. For this project, however, cost is distributed more smoothly across 98 tests, of which 8 (8.1%) take more than 1s to run with the rest of the tests running faster.

Figure 9: Size of test suites analyzed.

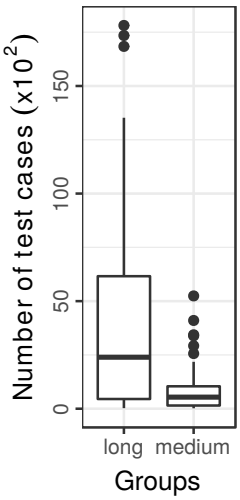


Figure 10: Size versus running time of test suites.

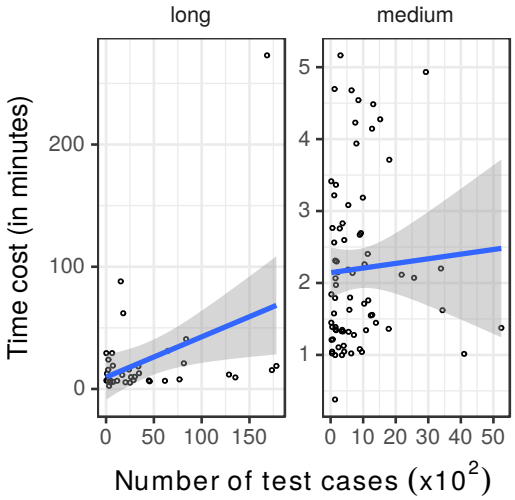


Figure 9 shows the difference in the distribution of test suite sizes across groups. This figure indicates that long projects have a higher median and much higher average number of test

cases. Furthermore, we noted a strong positive correlation between running time and number of test on projects in the long group. Considering the medium group, the correlation between two variables was weak. Figure 10 illustrates the regression lines between these the variables test suite cost and number of test cases. To sum, we observed that for projects with long-running test suites running time is typically justified by the number of test cases as opposed to the cost of individual test cases.

> **Answering RQ2:** *Overall, results indicate that projects with a very small number of tests monopolizing end-to-end execution time were rare. Time most often is distributed evenly across test cases.*

4.2 Adoption

This dimension evaluates the popularity of parallelization schemes in open source projects, and how developers perceive this technology.

4.2.1 How popular is test suite parallelization?

To answer this research question, we used projects from the medium and long groups where parallelization can be more helpful. We used dynamic and a static approaches to find manifestations of parallelism. We discuss results obtained with these complementary approaches in the following.

4.2.1.1 Dynamic checking

To find dynamic evidence of parallelism, we ran the test suites from our set of 110 projects to output all key-value pairs of Maven parameters. To that end, we used the option `-X` to produce debug output and the option `-DskipTests` to skip execution of tests. We skipped execution of tests as we observed from sampling that only bootstrapping the Maven process suffices to infer which parallel configuration modes it uses to run the tests. It is also important to point that we used the default configurations specified in the project.

We inferred parallel configurations by searching for certain configuration parameters in log files. According to Maven's documentation [Apache, 2017a], a parallel configuration depends either on (1) the parameter `parallel` to define the parallelism mode within a JVM followed by the parameter `threadCount` or (2) the parameter `forkCount`[1] to define the num-

[1]This parameter is named `forkMode` in old versions of Maven Surefire.

ber of forked JVMs. As such, we captured, for each project, all related key-value pairs of Maven parameters and mapped those pairs to one of the possible parallelization modes. For instance, if a given project contains a module with the parameter `<forkCount>1C</forkCount>`, the possible classifications are *Forked JVMs with Sequential Methods* (FC0) or *Forked JVMs with Parallel Methods* (FC1), depending on the presence and the value of the parameter `parallel`. If the parameter `parallel` is set to `methods`, the detected mode will be FC1.

Large projects may contain several test suites distributed on different Maven modules potentially using different configurations. For those cases, we collected the Maven output from each module discarding duplicates so as to avoid inflating counts for configuration modes that appear in several modules of the same project. For instance, if a project contains two modules using the same configuration, we counted only one occurrence.

Considering our set of 110 projects, we found that only 13 of those projects had parallelism enabled by default, with only configurations *Parallel Classes with Sequential Methods* (C2), *Parallel Classes with Parallel Methods* (C3), and FC0 being used. Configurations C3 and FC0 were the most popular among these cases. Note that these results under-approximate real usage of parallelism as we used default parameters in our scripts to spawn the Maven process. That decision could prevent execution of particular test modules. Table 1 shows the 13 projects we identified where parallelism is enabled by default in Maven. Column "*Subject*" indicates the name of the project, column "*# of modules*" indicates the fraction of modules containing tests that use the configuration of parallelism mentioned in column "*Mode*". We note that, considering these projects, the modules that do not use the configuration cited use the sequential configuration C0. For example, three modules (=28-25) from Log4J2 use sequential configuration. It came as a surprise the observation that no project used distinct configurations in their modules.

4.2.1.2 Static checking

Given that the dynamic approach cannot detect parallelism manifested through the default configuration of projects, we also searched for indications of parallelism in build files. We parsed all *pom.xml* files under the project's directory and used the same approach as in our previous analysis to classify configurations. We noticed initially that our approach was unable to infer the configuration mode for cases where the decision depends on the input (e.g., `<parallel>${parallel.type}</parallel>`). For these projects, the tester needs to provide additional parameters in the command line to enable parallelization (e.g., `mvn test -Dparallel.type=classesAndMethods`). To handle those cases, we considered all possible values for the parameter (in this case, `${parallel.type}`). It is also important to note that this approach is not immune to false negatives, which can occur when *pom.xml* files are en-

Table 1: Subjects with parallel test execution enabled by default.

Group	*Subject*	*# of modules*	*Mode*
Long	Apache Flink	66/74	FC0
Long	Apache Log4J2	25/28	FC0
Long	Apache Mahout	8/9	FC0
Medium	Apache OpenNLP	4/4	FC0
Medium	BounceStorage Chaos	1/1	C2
Medium	Eclipse Californium	2/20	C2
Long	Hazelcast Jet	6/7	FC0
Long	Jankotek MapDB	1/1	C3
Long	Javaslang	3/3	C3
Medium	Jcabi Github	1/1	C3
Long	Vavr-io Vavr	3/3	C3
Medium	Yegor256 Rultor	1/1	C3
Medium	Yegor256 Takes	1/1	C3

capsulated in jar files or files downloaded from the network. Consequently, this approach complements the the dynamic approach. Overall, we found 14 projects manifesting parallelism with this approach. Compared to the set of projects listed in Table 1, we found four new projects, namely: `Google Cloud DataflowJavaSDK` (using configuration C3), `Mapstruct` (using configuration FC0), `T-SNE-Java` (using configuration FC0), and `Urbanairship Datacube` (using configuration C3). Curiously, we also found that project `Jcabi`, `Rultor`, and `Takes` were not detected using this methodology. That happened because these projects loaded a *pom.xml* file from a jar file that we missed. Considering the static and dynamic methods together, we found a total of 17 distinct projects using parallelism, corresponding to the union of the two subject sets.

Answering RQ3: *Results indicate that test suite parallelization is underused. Overall, only 15.45% of costly projects (17 out of 110) use parallelism.*

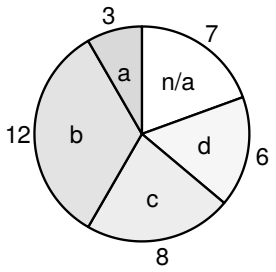
4.2.2 What are the main reasons that prevent developers from using test suite parallelization?

To answer this research question we surveyed developers involved in a selection of projects from our benchmark with time-consuming test suites. The goal of the survey is to better comprehend developer's attitude towards the use of parallelism as a mechanism to speedup regression testing. We surveyed developers from a total of 89 projects. From the initial list of 110 projects, we discarded 11 projects that we knew a priori used parallelization, and 10 projects where we could not find developer's emails from commit logs. From this list of projects, we mined potential participants for our study. More precisely, we searched for developer's name and email from the last 20 commits to the corresponding project repository. Using this approach, we identified a total of 297 eligible participants. Finally, we sent plain-text e-mails, containing the survey, to those developers. In total, 38 developers replied but we discarded three replies with subjective answers. Considering projects covered by the answers, a total of 36 projects (61.29% of the total) were represented in those replies. Note that multiple developers on each project received emails. In one specific case, one developer worked in multiple projects, and we consider it as a different answer. We sent the following set of questions to developers:

1. *How long does it take for tests to run in your environment? Can you briefly define your setup?*
2. *Do you confirm that your regression test suite does *not* run in parallel?*
3. *Select a reason for not using parallelization:*
    a) *I did not know it was possible.*
    b) *I was concerned with concurrency issues.*
    c) *I use a continuous integration server.*
    d) *Some other reason. Please elaborate.*

Considering question 1, we confirmed that execution time was compatible with the results we reported in Section 4.1.1. Furthermore, 12 of the participants indicated the use of Continuous Integration (CI) to run tests, with 4 of these participants reporting that test suites are modularized and those modules are tested independently in CI servers through different parameters. Those participants explained that such practice helps to reduce time to observe test failures, which is the goal of speeding up regression testing. A total of 6 participants answered that they do run tests in their local machines. Note, however, that CI does not preclude low-level parallelization. For example, installations of open-source CI tools (e.g., Jenkins CI [2018];

Figure 11: Summary of developer's answers to survey question 3.



Travis CI [2018]) in dedicated servers would benefit from running tests faster through low-level test suite parallelization.

Considering question 2, the answers we collected indicated, to our surprise, that six of the 36 projects execute tests in parallel. This mismatch is justified by cases where neither of our checks (static or dynamic) could detect presence of parallelism. A closer look at these projects revealed that one of them contained a *pom.xml* file encapsulated in a jar file (similar case as reported in Section 4.2.1.2), in one of the projects the participant considered that distributed CI was a form of parallelism, and in four projects the team preferred to implement parallelization instead of using existing features from the testing framework and the build system — in two projects the team implemented concurrency control with custom JUnit test runners and in two other projects the team implemented concurrency within test methods. Note that, considering these four extra cases (ignored two distributed CI cases), the usage of parallelization increases from 15.45% to 19.1%. We do not consider this change significant enough to modify our conclusion about practical adoption of parallelization (RQ3).

Considering question 3, the distribution of answers was as follows. A total of 8.33% of the 36 developers who answered the survey did not know that parallelism was available in Maven (option "a"), 33.33% of developers mentioned that they did not use parallelism concerned with possible concurrency issues (option "b"), 16.67% of developers mentioned that continuous integration suffices to provide timely feedback while running only smoke tests (i.e., short-running tests) locally (option "c"), and 16.67% of developers who provided an alternative answer (option "d") mentioned that using parallelism was not worth the effort of preparing the test suites to take advantage of available processing power. A total of 19.45% of participants did not answer the last question of the survey. The pie chart in Figure 11 summarizes the distribution of answers.

Answering RQ4: *Results suggest that dealing with concurrency issues (i.e., the extra work to organize test suite to safely explore concurrency) was the principal reason for developers not investing in parallelism. Other reasons included availability of continuous integration services and unfamiliarity with the technology.*

4.3 Speedups

Given the presence of projects with parallelization enabled by default identified on the dimension *Adoption* (see Section 4.2.1), in this dimension, we evaluate the speedups obtained with parallelization.

4.3.1 What are the speedups obtained with parallelization (in projects that actually use it)?

To answer RQ5, we considered the 13 subjects from our benchmark that use parallelization *by default* (see Table 1). We compared running times of test suites with enabled parallelization, as configured by project developers, and without parallelization. It is important to note that there are no observed failures in either execution. Table 2 summarizes results. Lines are sorted by project names. Columns "*Group*" and "*Subject*" indicate, respectively, the cost group and the name of the project. Column "T_s" shows sequential execution time and column "T_p" shows parallel execution time. Column "T_s/T_p" shows speedup or slowdown. As usual, a ratio above 1x indicates speedup and a ratio below 1x indicates slowdown.

Results show that, on average, parallel execution was 3.53 times faster compared to sequential execution. Three cases worth special attention: `Apache Log4J2`, `BounceStorage Chaos`, and `Yegor256 Takes`. We note that parallel execution in `Apache Log4J2` was ineffective. We found that Maven invokes several test modules in this project but the test modules that dominate execution time run sequentially by default. This was also the case for the highlighted project `Eclipse Californium`. No significant speedup was observed in `BounceStorage Chaos`, a project with only three test classes, of which one monopolizes the bulk of test execution time. This project uses configuration *Parallel Classes with Sequential Methods* (C2), which runs test classes in parallel but runs test methods, declared in each class, sequentially. Consequently, speedup cannot be obtained as the cost of the single expensive test class cannot be broken down with the selected configuration. Finally, the speedup observed in project `Yegor256 Takes` was the highest amongst all projects. This subject uses configu-

Table 2: Speedup (or slowdown) of parallel execution (T_p) over sequential execution (T_s). Default parallel configuration of Maven is used. Highest slowdown/speedup appears in gray color.

Group	*Subject*	T_s	T_p	T_s/T_p
Medium	Apache Flink	11.79m	2.57m	4.59x
Long	Apache Log4J2	8.24m	8.21m	1.00x
Long	Apache Mahout	27.38m	18.15m	1.51x
Medium	Apache OpenNLP	1.30m	0.55m	2.36x
Medium	BounceStorage Chaos	1.51m	1.47m	1.03x
Medium	Eclipse Californium	1.45m	1.40m	1.04x
Long	Hazelcast Jet	8.26m	3.67m	2.25x
Long	Jankotek MapDB	10.06m	8.58m	1.17x
Medium	Javaslang	2.18m	1.82m	1.20x
Medium	Jcabi Github	2.76m	0.30m	9.20x
Long	Vavr-io Vavr	3.26m	2.25m	1.45x
Medium	Yegor256 Rultor	2.30m	0.27m	8.52x
Medium	Yegor256 Takes	2.00m	0.19m	10.53x
Average				3.53x

ration *Parallel Classes with Parallel Methods* (C3) and contains 419 test methods distributed nearly equally among 148 test classes with a small number of test methods. Furthermore, several methods in those classes are time-consuming. As result, the CPUs available for testing are kept occupied for the most part during test execution.

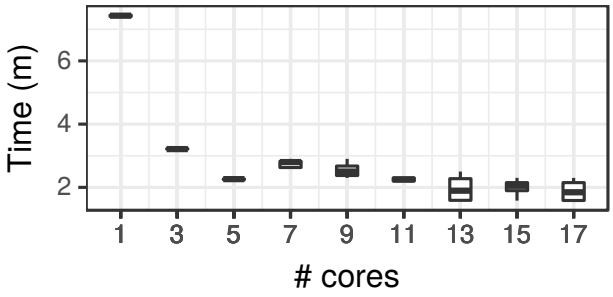
> **Answering RQ5:** *Considering the machine setup we used, the average speedup observed with default configurations of parallelization was 3.53x.*

4.3.2 How test execution scales with the number of available CPUs?

This experiment evaluates the impact of making a growing number of CPUs available to the build system for testing. For this reason, we used a different machine, with more cores, compared to the one described in Chapter 3. We used a Xeon E5-2660v2 (2.20GHz) Intel processor machine with 80 virtual CPUs (40 cores with two native threads each) and 256GB of memory, running Ubuntu 14.04 LTS Trusty Tahr (64-bit version). This experiment spawns a growing number of JVMs in different CPUs, using parallel configuration *Forked JVMs with Sequential*

Methods (FC0). We selected subject MapDB in this experiment as it represents the case of a long-running test suite (see Table 2) with test cases distributed across many test classes – 194. Recall that a test class is the smallest unit that can be used to spawn a test job on a JVM and that we have no control over which test classes will be assigned to which JVM that the build system forks. Figure 12 shows the reduction in running times as more CPUs contribute to the execution. We ran this experiment for a growing number of cores 1, 3, ..., 39. The plot omits results beyond 17 cores as the tendency for higher values is clear. We noticed that improvements are marginal after three cores, which is the basic setup we used in other experiments. This saturation is justified by the presence of a single test class, `org.mapdb.WALTruncat`, containing 15 test cases that take over two minutes to run.

Figure 12: Scalability.



Answering RQ6: *Results suggest that execution FC0 scales with additional cores but there is a bound on the speedup that one can get related to how well the test suite is balanced across test classes.*

4.4 Tradeoffs

This dimension assesses the impact of using distinct parallel configurations on test flakiness and speedup. Increased parallelism can increase resource contention leading to concurrency issues such as data races across dependent tests [Luo et al., 2014; Bell et al., 2015]. Flakiness and speedup are contradictory forces that could influence the decision of practitioners about which parallel configuration should be used for testing. Note that Section 4.3.1 evaluated speedup in isolation.

4.4.1 How parallel execution configurations affect testing costs and flakiness?

To answer this research question, we selected 15 different subjects from our dataset. To select subjects, we sorted projects whose test suites run in 1m or more by decreasing order of execution time and selected the first fifteen projects that use JUnit 4.7 or later. The rationale for this criteria is to ensure compatibility with parallel configuration since older versions of JUnit does not support parallel testing. We ran the test suites from the selected projects against all configurations described in Chapter 2, and compared their running times and rate of test flakiness. We used the sequential execution configuration (C0) as the comparison baseline in this experiment. We ran each project on each configuration for three times. Overall, we needed to reran test suites 270 times, 18 times (3x6 configurations) on each project. Given the low standard deviations observed in our measurements, we considered three reruns reasonable for this experiment.

It is worth mentioning that we used custom JUnit runners as opposed to Maven to run the test suites with different parallel configurations (see Chapter 2). After carefully checking library versions for compatibility issues and comparing results with JUnit's we observed that several of Maven's executions exposed problems. For example, Maven incorrectly counts the number of test cases executed for some of the cases where test flakiness are observed. To address those issues we implemented custom test runners for configurations *Sequential Classes with Parallel Methods* (C1), *Parallel Classes with Sequential Methods* (C2), and *Parallel Classes with Parallel Methods* (C3) and, for configurations *Forked JVMs with Sequential Methods* (FC0) and *Forked JVMs with Parallel Methods* (FC1), we implemented a bash script that coordinates the creation of JVMs and invokes corresponding custom runners. So to faithfully reflect Maven's behavior in our scripts, we carefully analyzed the source code [Apache, 2018] of the Maven Surefire plugin. We implemented test runners using the `ParallelComputer` class from JUnit [JUnit, 2017].

We used Maven log files to identify test classes to run and used the Maven dependency plugin [Apache, 2017b] to build the project's classpath (with the command `mvn dependency:build-classpath`). Once we find the tests suite to run and the corresponding classpath, we invoke the test runners mentioned above on them. We configured this experiment to run at most three JVMs in parallel. Recall that in our setup (see Chapter 4), we limited our kernel to use only three cores and reserved one core for OS-related processes. To ensure that our experiments terminate (recall that deadlock or livelock could occur) we used the `timeout` command [Linux Man Page, 2018] configured to dispatch a *kill* signal if test execution exceeds a given time limit. Finally, we save each execution log and stack traces generated from JUnit to collect the execution time, the number of failing tests, and to diagnose outliers in our results.

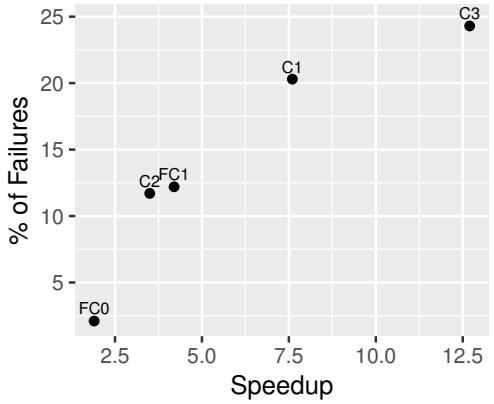
Table 3: Speedup versus Flakiness ($\%_{\text{fail}}$). Configuration *C0* denotes the comparison baseline. Columns *T* and *N* indicate time and number of tests, respectively. Other columns show speedup and percentage of failing tests in different configurations, compared to *C0*.

Subject	*C0*		*C1*		*C2*		*C3*		*FC0*		*FC1*	
	T	*N*	speedup	$\%_{\text{fail}}$	speedup	$\%_{\text{fail}}$	speedup	$\%_{\text{fail}}$	speedup	$\%_{\text{fail}}$	speedup	$\%_{\text{fail}}$
Activiti	5.9m	2,029	72.1x	96.3%	1.5x	6.9%	75.9x	96.3%	2.9x	6.6%	3.1x	8.0%
AWS SDK Java Core	3.7m	847	2.0x	2.2%	2.5x	2.8%	3.7x	4.0%	1.9x	0.2%	3.5x	3.1%
Bucket4J	3.0m	110	2.5x	0%	1.3x	0.9%	4.2x	1.8%	1.3x	0%	3.7x	0%
Facebook Linkbench	6.1m	98	1.0x	0%	1.6x	1.0%	1.0x	0%	1.7x	0%	1.6x	0%
GoogleCloud Dataflow	1.6m	3,345	1.2x	1.7%	2.7x	1.0%	0.8x	5.4%	0.8x	1.7%	0.8x	1.7%
INRIA spoon	2.3m	1,042	1.2x	28.8%	2.7x	77.2%	1.6x	56.6%	1.8x	0%	1.8x	29.0%
Jcabi Github	2.6m	634	2.1x	0%	17.7x	0%	28.8x	0%	2.0x	0%	2.9x	0%
JCTools Core	3.6m	690	4.5x	0%	3.6x	0%	18.0x	0%	2.8x	0%	9.0x	0%
MapDB	8.2m	5,324	1.5x	0%	2.7x	0%	4.8x	0%	1.7x	1.0%	3.4x	1.0%
Moquette	3.7m	169	4.6x	65.6%	3.4x	33.0%	12.3x	78.0%	2.5x	22.5%	9.3x	69.4%
Spring Cloud Function	2.8m	168	6.4x	77.4%	1.3x	0.6%	6.6x	79.2%	1.1x	0%	2.9x	32.7%
Stream Lib	2.1m	149	0.9x	0%	2.2x	0%	2.4x	0.7%	2.7x	0%	3.6x	0%
Stripe Java	4.3m	302	4.8x	6.3%	3.3x	7.3%	21.5x	15.0%	2.7x	0%	8.6x	11.6%
TabulaPDF Java	2.4m	186	7.2x	0.5%	1.1x	0%	7.2x	2.7%	1.0x	0%	7.2x	1.6%
Urban Airship Datacube	8.3m	36	1.9x	25.0%	4.7x	44.4%	1.9x	25.0%	1.0x	0%	1.9x	25.0%
Average	4.0m	1,006.6	7.6x	20.3%	3.5x	11.7%	12.7x	24.3%	1.9x	2.1%	4.2x	12.2%

Table 3 summarizes results ordered by subject's name. Values are averaged across multiple executions. We did not report standard deviations as they are very small in all cases. As to identify the potential causes of flakiness, we inspected the exceptions reported in execution logs. We found that, in most of the cases, flakiness was caused by race conditions: approximately 97.5% of the failures were caused by a `null` dereference and 1.6% were caused by concurrent access on unsynchronized data structures. Cases of likely broken test dependencies were not as prevalent as race conditions (0.8% of the total): `EOFException` (0.2%), `FileSystemAlreadyExistsException` (0.2%), and `BufferOverflowException` (0.4%). Results suggest that anticipating race conditions to schedule test executions would have higher impact compared to breaking test dependencies using a tool such as ELECTRICTEST [Bell et al., 2015].

The projects with flakiness in all configurations were `AWS SDK`, `Activiti`, `GoogleCloud`, and `Moquette`. It is worth highlighting the unfortunate case of `Moquette`, which manifested more than 20% flaky tests in every configuration. Considering time, it is noticeable from the averages, perhaps as expected, an increasing speedup from configuration *C1* to *C3* and from configuration *FC0* to *FC1*. It is also worth mentioning that some combinations manifested slowdown instead of speedup. Recall that parallel execution introduces the overhead of spawning and managing JVMs and threads. Overall, results show that 0% of flakiness have been reported in 30 of the 75 (=5x15) pairs of project and configuration we analyzed (40% of

Figure 13: Test failures versus speedups per configuration (on average).



the total). In for 2 of the 15 projects flakiness was not manifested in any combination pairs. We noticed with some surprise that the average speedup of configuration *C1* was higher compared to *FC1* indicating that it is not always the case that using more CPUs pays off. Important to note that the cost of spawning new JVMs can be significant in *FC1*. Figure 13 summarizes the average of test failures versus the speedup obtained.

> **Answering RQ7:** *Overall results indicate that the test suites of 40% of the projects we analyzed could be run in parallel without manifesting any flaky tests. In some of these cases, speedups were significant, ranging from 1x to 28.8x.*

5

LESSONS LEARNED

This work reports our finding on a study to evaluate impact and usage of test suite parallelization, enabled by modern build systems and testing frameworks. This study is important given the importance of speeding up testing and the support for test parallelization in modern build systems and testing frameworks. Note that test suite parallelization is complementary to the alternative approaches to speedup testing (see Chapter 2). In the following, we discuss the lessons learned based on our observations, and we provide recommendations for developers with existing code base:

5.1 Refactor tests for load balancing

Forked JVMs scale better with the number of cores when the test workload is balanced across testing classes. To balance the workload, automated user-oblivious refactoring can help in scenarios where developers are not willing to change test code but have access to machines with a high number of cores. In addition, it is recommended to avoid an unbalanced distribution of time for the tests. On the *feasibility* dimension (see Section 4.1), our results show that test cases are typically short-running, typically taking less than half a second to run. Furthermore, we found that only in rare cases few test cases monopolize the overall time to run a test suite.

5.2 Incentivize forking

On the *tradeoffs* dimension (see Section 4.4), we observed that forked JVMs manifest lower rates of test flakiness in comparison to schemes with multithreaded execution enabled. For instance, in Forked JVMs with Sequential Methods (FC0), only 5 of 15 projects manifest flakiness and, excluding the extreme case of `Moquette` and `Activiti`, projects manifest flaky tests in low rates 0.23% to 1.70%. Overall, schemes based on multithreaded execution are likely to increase the rate of test flakiness caused by the parallel execution of tests (see Figure 13). The

configuration Forked JVMs with Sequential Methods (FC0) is the most conservative configuration among the available options. Based on this observation, developers of projects with long-running test suites should consider using that feature, which is available in modern build systems today (e.g., Maven).

5.3 Break test dependencies

Test flakiness is a central concern when running tests in parallel. Non-forked JVMs can achieve impressive speedups at the expense of sometimes impressive rates of flakiness. Dependent tests can be affected by different scheduling of test methods and classes. On the *tradeoffs* dimension (see Section 4.4), our results indicate that configurations that fork JVMs do not achieve speedups as high as other more-aggressive configurations, but they manifest much lower flakiness ratios. Breaking test dependencies to avoid flakiness and take full advantage of those options is advised for developers with a greater interest in efficiency. On dimension *speedups* (see Section 4.3) we observed several cases of projects that could improve significantly the execution by enabling multithreaded execution (e.g., configuration *Parallel Classes with Parallel Methods*). When designing new test suites, developers should consider test parallelization and be aware of dependencies.

5.4 Improve debugging for build systems

While preparing our experiments, we found scenarios where Maven's executions did not reflect corresponding JUnit's executions. Those issues can hinder developers from using parallel testing. Better debugging infrastructure is important to improve confidence and incentivize the adoption of test parallelization.

6

THREATS TO VALIDITY

In this chapter, we discuss the limitations of our study and our approach to handle them. In the following, we describe the external, internal, and construct threats to the validity of our results.

6.1 External Validity

External validity concerns the representativeness of our results to the population observed (in our case, open-source projects). The generalization of our findings is limited to our selection of projects, testing framework, and build system. To mitigate that issue, we selected subjects according to an objective criteria, described in Chapter 3. It remains to be evaluated the extent to which our observations would change when using different testing frameworks and build systems. In addition, some of the selected subjects contain failing tests. This is understandable since some projects may have an unstable revision on the latest commit by the time we downloaded the project. Test failures may reduce the testing time due to early termination or even inflate the time. For instance, a test could hang indefinitely for unavailable resources. To mitigate this threat, we eliminated subjects with flaky tests and filtered projects with at least 90% of the tests passing. Only 17% of our subjects have failing tests. We carefully inspected our rawdata to identify and ignore these failures with JUnit's `@Ignore` annotation.

6.2 Internal Validity

Internal validity concerns the consistency of our measurements. In practice, external factors may affect the causality of the observed results during our experiments. Our results could be influenced by unintentional mistakes made by humans who interpreted survey data and implemented scripts and code to collect and analyze the data. For instance, in some of our initial experiments, we observed that some background services on the operating system were

introducing noise in our results. To eliminate that noise, we deactivated several background services and configured our workstation to run only the necessary to keep the operating system and our experiments execution. In addition, we configured our kernel to isolate six virtual CPUs to run our experiments, leaving the remaining CPUs to run operating system processes (see Section 4.1.1). In our last experiment, we observed inconstancies when using Maven with parallel execution enabled (see Section 4.4.1). We developed JUnit runners to reproduce Maven's parallel configurations and implemented several scripts to automate our experiments (e.g., run tests and detect parallelism enabled by default in the subjects). All those tasks could bias our results. To mitigate those threats, the first two authors of this study validated/inspected each other to increase chances of capturing unintentional mistakes.

6.3 Construct Validity

We considered a number of metrics in this study that could influence some of our interpretations. For example, we measured number of test cases per suite, distribution of test costs in a suite, time to run a suite, etc. In principle, these metrics may not reflect the main problems associated with test efficiency.

7

RELATED WORK

Regression testing research has focused mostly on test suite minimization, prioritization, and selection [Yoo and Harman, 2012; Soetens et al., 2016]. Most of these techniques are unsound (i.e., they do not guarantee that fault-revealing tests will be considered for testing). For instance, test suite augmentation (i.e., addition of tests to an existing test suite) may significantly hinder the effectiveness of prioritization techniques [Lu et al., 2016]. Shi et al. [2015] investigated how test suite reduction and regression test selection perform individually and how the combination of both techniques (i.e., applying test selection from the test suite reduction) performs in open-source projects. Results revealed that test selection has higher fault-detection capability and better speedup performance than test prioritization. EKSTAZI [Gligoric et al., 2015; Çelik et al., 2017] is an example of a sound regression testing technique. It conservatively computes which tests have been impacted by file changes. A test is discarded for execution if it does not depend on any changed file dynamically reachable from execution. Important to note that regression testing techniques, including test selection, is complementary to test suite parallelization.

A test is said to be "*flaky*" when it yields non-deterministic results for the same code revision. Flakiness is a major concern in software testing and it has been reported and investigated by practitioners and researchers. Luo et al. [2014] conducted an extensive empirical study to understand the sources of flakiness and their implications. Two sources of flakiness observed in our experiments are flakiness caused by data races from the random scheduling of tests and flakiness caused by order dependency (i.e., tests assume some execution order due to data dependency). ELECTRICTEST [Bell et al., 2015] is a tool for efficiently detecting data dependencies across test cases. Dependency tracking is important to avoid test flakiness when parallelizing test suites. ELECTRICTEST observes reads and writes on global resources made by tests to identify these dependencies at low cost. Unfortunately, we were unable to consider ELECTRICTEST in our experiments because it is not publicly available due to intellectual property restrictions. Recent research proposed PRADET [Gambi et al., 2018], a tool for detecting

test dependencies based on data-flow analysis. Given a pre-defined execution order, PRADET infers an over-approximation of dependencies and refines those dependencies by rerunning tests out of order. It remains to be investigated the impact of PRADET to reduce flakiness in unrestricted test suite parallelization. DEFLAKER is another tool recently published for detecting flaky tests [Bell et al., 2018]. It tracks the coverage of the latest modifications in the code base and marks as "*flaky*" any failing test outside the coverage set. This technique is useful to detect flakiness upfront when the developer is continuously integrating changes to the code base. In addition to the technique, an interesting contribution of this work is an empirical evaluation on the effectiveness of rerunning tests to identify flakiness. Results revealed that simply rerunning a test immediately after its failure is not effective to witness test flakiness. In most cases, it was necessary to clean the working directory (e.g., regenerate resource files) and run the failing test in a fresh JVM. It remains for us to investigate the effectiveness of different rerun strategies for the subset of failing tests discovered in our experiments (see Table 3).

The use of the Simple Instruction Multiple Data (SIMD) design has been previously explored in research to accelerate test execution [d'Amorim et al., 2007, 2008; Kim et al., 2012; Nguyen et al., 2014; Rajan et al., 2014; Sen et al., 2015; Yaneva et al., 2017]. The SIMD architecture, as implemented in modern GPUs, for instance, allows the execution of a given instruction simultaneously against multiple data. For that reason, in principle, one test could be ran simultaneously against multiple inputs provided that multiple test inputs exist associated to that one test. Recent work [Rajan et al., 2014; Yaneva et al., 2017] explored that idea to speedup test execution of embedded software using graphic cards. Although benchmarks indicate superior performance compared to traditional multicore CPUs, the use of the technology in broader settings is limited. For example, execution of more general programs can violate the SIMD's lock-step assumption on the control-flow of threads. This violation would negatively affect performance. Furthermore, handling complex data is challenging in SIMD [d'Amorim et al., 2007, 2008]. The approach is promising when multiple input vectors exist for each test and the testing code heavily manipulates scalar data types. The datasets used in those papers satisfied those constraints.

Google [Google Engineering Tools, 2011; Google TechTalks, 2010] and Microsoft [Schulte and Prasad, 2013] have been creating distributed infrastructures to efficiently build massive amounts of code and run massive numbers of tests. Those scenarios bring different and challenging problems such as deciding when to trigger the build under multiple file updates [Memon et al., 2017]. Although such distributed systems are targeted at extremely large scale code and test bases, the same ideas can be applied to handle the build process of large, albeit not as large, projects. For example, Gambi et al. [2017] recently proposed Cloud Unit Testing (CUT), a tool to automatically parallelize JUnit tests on the cloud. The tool allows the developer to control

resource allocation and deal with the project specific test dependencies. Note that test suite parallelization is complementary to these high-level parallelism schemes.

Continuous Integration (CI) services, such as Travis CI [Travis CI, 2018], are becoming widely used in the open-source community [Hilton et al., 2016; Vasilescu et al., 2015]. Accelerating time to run tests in CI is important as to reduce the period between test report updates. Module-level regression testing [Vasic et al., 2017], for example, can be helpful in that setting. It is important to note that test failures are more common in CI compared to an overnight run or a local run, for instance. This can happen because of semantic merge conflicts [Brun et al., 2011], for instance. As such effect can impact developer's perception and tolerance towards failures, we are curious to know if developers would be willing to receive more frequent test reports at the expense of potentially increasing failure rates due to flakiness caused by parallelism.

8

CONCLUSIONS

Testing is an expensive process. Even in open-source development, complex projects exist and may contain long-running test suites. Despite all advances in regression testing research, dealing with high testing costs remains an important problem in Software Engineering. This work reports our findings on the usage and impact of test execution parallelization in open-source projects. Multicore CPUs are widely available today even on smartphones. Moreover, popular testing frameworks and build systems that capitalize on these machines provide mature support to exploit the underlying computing resources to speedup test execution.

Overall, test parallelization is underused in practice. From a set of 468 popular Java projects hosted on Github, we observed that 24% of the projects contain costly test suites. Surprisingly, only 19.1% of costly projects used parallelization. The main reported reason for adoption resistance was the concern to deal with concurrency issues. When developers do not design tests to run in parallel upfront, high reliability is preferable over high performance in test execution. Tests may become unreliable when they have some expected order of execution or when tests access shared resources. It would be impractical to distinguish whether a failure occurred by a legitimate fault or by the non-deterministic scheduling of tests. Despite some resistance observed from practitioners, our results suggest that parallelization can be used in many cases without sacrificing reliability. Projects with costly test suites and parallelization enabled by default could achieve a speedup of 3.53x on average compared to sequential execution. In addition, we were able to achieve significant speedups, ranging from 1x to 28.8x, in several projects with different parallelization schemes without manifesting flakiness.

Test parallelization is a legitimate approach to reduce the costs of testing, and it is complementary to other approaches (e.g., regression test techniques and distributed execution). More research needs to be done to improve automation to safely optimize parallel execution. For instance, developers could greatly benefit from techniques for refactoring test suites for better load balancing. Parallelization schemes based on forking processes are promising since they provide better isolation (i.e., tests run on their own JVM) and scale to the number of available

cores. While schemes based on multithreaded execution can achieve impressive speedups, data races are likely to occur since tests may change the state of shared objects. It is still necessary to investigate the effectiveness of rerunning failing tests in multithreaded schemes, and how it affects speedups. More sophisticate approaches may consider lightweight analysis to support safe scheduling of tests.

REFERENCES

L. S. Pinto, S. Sinha, and A. Orso, "Understanding myths and realities of test-suite evolution," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 33:1–33:11. [Online]. Available: http://doi.acm.org/10.1145/2393596.2393634

S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Software Testing Verification and Reliability*, vol. 22, no. 2, pp. 67–120, Mar. 2012.

Google Engineering Tools, "Testing at the speed and scale of google," June 2011, accessed on: 2018-01-05. [Online]. Available: http://google-engtools.blogspot.com.br/2011/06/testing-at-speed-and-scale-of-google.html

Google TechTalks, "Tools for continuous integration at google," October 2010, accessed on: 2018-01-07. [Online]. Available: http://www.youtube.com/watch?v=b52aXZ2yi08

W. Schulte and C. Prasad, "Taking control of your engineering tools," *Computer*, vol. 46, no. 11, pp. 63–66, Nov. 2013. [Online]. Available: http://dx.doi.org/10.1109/MC.2013.337

C. H. P. Kim, D. Marinov, S. Khurshid, D. Batory, S. Souto, P. Barros, and M. D'Amorim, "Splat: Lightweight dynamic analysis for reducing combinatorics in testing configurable systems," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 257–267. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491459

Clutch, "List of popular hosting cloud services," 2018, accessed on: 2018-02-01. [Online]. Available: https://clutch.co/cloud

G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *ACM Trans. Softw. Eng. Methodol.*, vol. 6, no. 2, pp. 173–210, Apr. 1997. [Online]. Available: http://doi.acm.org/10.1145/248233.248262

M. Gligoric, L. Eloussi, and D. Marinov, "Practical regression test selection with dynamic file dependencies," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: ACM, 2015, pp. 211–222. [Online]. Available: http://doi.acm.org/10.1145/2771783.2771784

D. Saff and M. D. Ernst, "Reducing wasted development time via continuous testing," in *Proceedings of the 14th International Symposium on Software Reliability Engineering*, ser. ISSRE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 281–. [Online]. Available: http://dl.acm.org/citation.cfm?id=951952.952340

Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 643–653. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635920

J. Bell, G. Kaiser, E. Melski, and M. Dattatreya, "Efficient dependency detection for safe java test acceleration," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 770–781. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786823

J. Candido, L. Melo, and M. d'Amorim, "Test suite parallelization in open-source projects: A study on its usage and impact," in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. Piscataway, NJ, USA: IEEE Press, 2017, pp. 838–848. [Online]. Available: http://dl.acm.org/citation.cfm?id=3155562.3155667

JUnit, "Junit web site," 2018, accessed on: 2018-02-13. [Online]. Available: http://junit.org

NUnit, "Nunit web site," 2018, accessed on: 2018-02-13. [Online]. Available: http://www.nunit.org

TestNG, "Testng web site," 2018, accessed on: 2018-02-13. [Online]. Available: http://testng.org

Apache, "Maven surefire plugin," 2017, accessed on: 2018-02-13. [Online]. Available: http://maven.apache.org/surefire/maven-surefire-plugin/

Github, "About github," 2018, accessed on: 2018-02-14. [Online]. Available: https://github.com/about

Github Developer, "Github search api," 2018, accessed on: 2018-02-13. [Online]. Available: http://developer.github.com/v3/search/

Github Help, "About stars," 2018, accessed on: 2018-02-13. [Online]. Available: https://help.github.com/articles/about-stars/

Kernel.org, "Kernel linux options," 2018, accessed on: 2018-01-05. [Online]. Available: https://www.kernel.org/doc/html/latest/admin-guide/kernel-parameters.html?highlight=isolcpu

Unix StackExchange Community, "Using `isolcpus`," 2018, accessed on: 2018-01-05. [Online]. Available: http://unix.stackexchange.com/questions/326579/how-to-ensure-exclusive-cpu-availability-for-a-running-process

Jenkins CI, "Build great things at any scale," 2018, accessed on: 2018-01-05. [Online]. Available: https://jenkins.io/

Travis CI, "Build apps with confidence," 2018, accessed on: 2018-01-05. [Online]. Available: https://travis-ci.org/

Apache, "Maven surefire source repository," 2018, accessed on: 2018-01-05. [Online]. Available: http://svn.apache.org/viewvc/maven/su\refire/trunk/

JUnit, "Parallelcomputer (junit api)," 2017. [Online]. Available: http://junit.org/junit4/javadoc/4.12/org/junit/experimental/ParallelComputer.html

Apache, "Maven dependency plugin," 2017, accessed on: 2018-02-13. [Online]. Available: https://maven.apache.org/plugins/maven-dependency-plugin/

Linux Man Page, "timeout - run a command with a time limit," 2018. [Online]. Available: http://linux.die.net/man/1/timeout

Q. D. Soetens, S. Demeyer, A. Zaidman, and J. Pérez, "Change-based test selection: An empirical evaluation," *Empirical Softw. Engg.*, vol. 21, no. 5, pp. 1990–2032, Oct. 2016.

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 *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 535–546. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884874

A. Shi, T. Yung, A. Gyori, and D. Marinov, "Comparing and combining test-suite reduction and regression test selection," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 237–247. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786878

A. Çelik, M. Vasic, A. Milicevic, and M. Gligoric, "Regression test selection across JVM boundaries," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, 2017, pp. 809–820. [Online]. Available: http://doi.acm.org/10.1145/3106237.3106297

A. Gambi, J. Bell, and A. Zeller, "Practical test dependency detection," in *Proceedings of the 2018 IEEE Conference on Software Testing, Validation and Verification*, ser. ICST 2018, 2018. [Online]. Available: http://jonbell.net/publications/pradet

J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "Deflaker: Automatically detecting flaky tests," in *Proceedings of the 2018 International Conference on Software Engineering*, ser. ICSE 2018, 2018. [Online]. Available: http://jonbell.net/publications/deflaker

M. d'Amorim, S. Lauterburg, and D. Marinov, "Delta execution for efficient state-space exploration of object-oriented programs," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ser. ISSTA '07. New York, NY, USA: ACM, 2007, pp. 50–60. [Online]. Available: http://doi.acm.org/10.1145/1273463.1273472

——, "Delta execution for efficient state-space exploration of object-oriented programs," *IEEE Trans. Softw. Eng.*, vol. 34, no. 5, pp. 597–613, Sep. 2008. [Online]. Available: http://dx.doi.org/10.1109/TSE.2008.37

C. H. P. Kim, S. Khurshid, and D. Batory, "Shared execution for efficiently testing product lines," in *2012 IEEE 23rd International Symposium on Software Reliability Engineering*, Nov 2012, pp. 221–230.

H. V. Nguyen, C. Kästner, and T. N. Nguyen, "Exploring variability-aware execution for testing plugin-based web applications," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 907–918. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568300

A. Rajan, S. Sharma, P. Schrammel, and D. Kroening, "Accelerated test execution using gpus," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: ACM, 2014, pp. 97–102. [Online]. Available: http://doi.acm.org/10.1145/2642937.2642957

K. Sen, G. Necula, L. Gong, and W. Choi, "Multise: Multi-path symbolic execution using value summaries," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 842–853. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786830

V. Yaneva, A. Rajan, and C. Dubach, "Compiler-assisted test acceleration on gpus for embedded software," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017. New York, NY, USA: ACM, 2017, pp. 35–45. [Online]. Available: http://doi.acm.org/10.1145/3092703.3092720

A. M. Memon, Z. Gao, B. N. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, "Taming google-scale continuous testing," in *39th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP 2017, Buenos Aires, Argentina, May 20-28, 2017*, 2017, pp. 233–242. [Online]. Available: https://doi.org/10.1109/ICSE-SEIP.2017.16

A. Gambi, S. Kappler, J. Lampel, and A. Zeller, "Cut: Automatic unit testing in the cloud," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017. New York, NY, USA: ACM, 2017, pp. 364–367. [Online]. Available: http://doi.acm.org/10.1145/3092703.3098222

M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: ACM, 2016, pp. 426–437. [Online]. Available: http://doi.acm.org/10.1145/2970276.2970358

B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov, "Quality and productivity outcomes relating to continuous integration in github," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 805–816. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786850

M. Vasic, Z. Parvez, A. Milicevic, and M. Gligoric, "File-level vs. module-level regression test selection for .net," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, 2017, pp. 848–853. [Online]. Available: http://doi.acm.org/10.1145/3106237.3117763

Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Proactive detection of collaboration conflicts," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 168–178. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025139