COMPARING MODULE- AND CLASS-LEVEL REGRESSION TEST SELECTION IN CONTINUOUS INTEGRATION

BY

PEIYUAN ZHAO

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

Adviser:

Professor Darko Marinov

ABSTRACT

Developers rely on regression testing in their continuous integration (CI) environment to find if their changes introduce faults. While regression testing is widely practiced, it is very costly. Regression test selection (RTS) reduces the cost of regression testing by running only the tests that are affected by the changes. Industry has adopted module-level RTS running in a CI environment. Researchers have recently proposed class-level RTS but did not evaluate it in a CI environment.

In this work, we compare module-level and class-level RTS techniques in a CI environment. Specifically, we evaluate GIB, a module-level RTS technique, and Ekstazi, a class-level RTS technique, on Travis CI, a popular CI service for open-source projects. We also propose a hybrid RTS technique, GIBstazi, that combines aspects of the module-level and class-level RTS techniques. GIBstazi is safer than Ekstazi because GIBstazi tracks non-source-code changes while Ekstazi does not. We evaluate all the techniques on Travis CI using real, historical commits that were built on Travis CI.

Our results show that all the RTS techniques do save testing time compared to running all tests (RetestAll) in a CI environment, but the savings in the end-to-end time for a full build is not as high as shown in previous work in a local environment running with a dedicated machine due to the extra overhead in a CI environment. Moreover, we are the first to inspect test failures from RetestAll builds. We find that these failures are often not selected by the RTS techniques, but almost all test failures missed by the RTS techniques are from so called flaky tests, which can pass or fail nondeterministically regardless of the code changes. The overall results show that RTS can be beneficial for the developers, hybrid GIBstazi gives the best trade-off, and RTS not only saves time but also avoids misleading developers by flaky test failures.

To my parents, for their love and support.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Darko Marinov, for his guidance and advice. I am very fortunate to get a chance to learn from Darko.

I would also like to thank the colleagues from Mir group, including August Shi, Alex Gyori, and Muhammad Suleman Mahmood, for the collaboration.

Last, I would like to thank my friends at Daya Tech for the encouragement and support.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

1.1 REGRESSION TESTING

Regression testing is a type of software testing that retests the software that has been modified. The main idea is to run the regression test suite after every change to quickly detect regression faults introduced by the code changes. If a test fails while it was passing before, the test failure should indicate a fault introduced by the change that the developer should debug. Regression testing is commonly automated by running the regression test suite in a continuous integration (CI) environment. After every change made to the code repository, a CI server, typically in the cloud, builds and tests the code [17]. While regression testing is important and well-practiced, the process can be time-consuming due to the large size of the regression test suite, and due to the potentially frequent code changes [24]. Furthermore, as regression testing is run on CI servers in the cloud, there is also a huge monetary cost for using the cloud resources [16].

1.2 REGRESSION TEST SELECTION (RTS)

Regression test selection (RTS) can reduce the cost of regression testing. The goal of RTS is to run fewer tests, speeding up the regression testing process. RTS aims to only select a subset of the tests in the regression test suite that are affected by the change [31] and thus the tests that are not affected by the change will not be run. A RTS technique tracks the dependencies between the tests and code entities (e.g., methods, classes, or modules) and selects to run the tests whose dependencies changed. Researchers have proposed many different RTS techniques that perform test selection at different granularity levels of dependencies [13–15, 21, 25–27, 30–32]. Researchers have found RTS to be quite effective, in terms of both percentage of tests selected and time saved. Furthermore, industry has also adopted the use of RTS to speed up their regression testing efforts [11, 12, 24].

While both the industry and the academia use and study RTS techniques, there is a divide in the level of granularity of RTS used between the two. In *industry*, it is common to use a coarse-grained granularity of dependencies by tracking *modules* in the project, selecting to run all tests within the modules that are affected by the changes [12, 24]. Module-level RTS has been shown to be effective in industry, which relies on CI. The reason for the effectiveness is mainly due to the small overhead necessary in figuring out what modules are actually affected by the changes. However, the coarse-grained dependency tracking and

coarse-grained level of tests being selected (all tests within an affected module) can lead to a much larger number of tests selected to run than those selected to run by class-level RTS. In *academia*, researchers have traditionally studied finer-grained levels of dependencies, such as statements or methods [32]. Recent work has found *class*-level dependencies to be the most effective in terms of time savings [14, 21]. While class-level RTS can select to run fewer tests than module-level RTS, by tracking dependencies to only classes, class-level RTS can potentially be unsafe, failing to select tests that are affected by changes to non-class files, missing to run tests that reveal regression faults. Furthermore, researchers evaluated the effectiveness of class-level RTS using a dedicated machine, where testing is done on the same machine over many revisions, as opposed to building and testing on a CI service that performs testing on different machines in the cloud.

1.3 OVERVIEW OF WORK

We evaluate and compare module-level and class-level RTS techniques in a CI environment. To the best of our knowledge, no prior work has studied how these two types of RTS techniques compare against each another in a CI environment for regression testing. Evaluating RTS techniques in a CI environment also allows us to use the RTS tools along with the actual build commands used by the developers. By doing this evaluation, our work better simulates the usage of the tools in real world than the prior studies. As representatives of the two RTS techniques, we use GIB [19], a module-level RTS tool, and Ekstazi [14], a class-level RTS tool, for our evaluation.

Furthermore, we implement a RTS tool, called *GIBstazi*, that combines both module-level and class-level RTS. The idea of GIBstazi is to use GIB to first quickly filter out what modules are affected by the changes, and then only apply Ekstazi on the affected modules to determine what tests need to be run. If any of the changes are made to non-source-code files that cannot be tracked by class-level RTS, GIBstazi defaults to GIB behavior and runs all the tests within the affected modules. With this combination, GIBstazi aims to run fewer tests than GIB, leading to faster testing, though not necessarily faster than Ekstazi. However, because GIBstazi defaults to GIB behavior due to non-source-code changes that are not tracked by Ekstazi, GIBstazi can be safer than Ekstazi.

We utilize Travis CI [4], a CI service available for open-source projects, as the common environment for our evaluation of the three RTS techniques. We evaluate on open-source Java projects that build using Travis CI; we replay their historical builds both by running all their tests, called RetestAll, and while using the three RTS tools. We measure the percentage of tests selected by the RTS techniques as well as the percentage of time it takes to select

and run the selected tests, all with respect to RetestAll. We choose 22 open-source projects and their 1444 historical jobs for our evaluation, but RetestAll and all three RTS tools pass on 980 jobs. On these 22 projects and 980 jobs, we found that all three RTS techniques, on average, provide savings in time over RetestAll, with GIB running 79.15%, Ekstazi running 76.38%, and GIBstazi running 78.76% of RetestAll time.

An important question in RTS is whether selection misses some real test failure. After all, it is trivial to have a super fast RTS technique by not running any test, but it would miss all potential failures. When we replayed historical builds for RetestAll, we found builds with test failures. Ideally, testing should run all tests that fail to signal to the developers that there is a fault in the code due to the changes. We found that the RTS techniques actually fail to select many of these failed tests from RetestAll runs, with up to 32.29% of the failed tests not selected to run by GIBstazi! Missing to select these failing tests is seemingly bad. However, our inspection of the test failures showed that almost all the test failures from RetestAll are from so called flaky tests, which can pass or fail nondeterministically regardless of the code changes. As such, it can be beneficial that a RTS technique does *not* run these tests and does *not* give false alarms to developers about having to debug the faults that are not due to the code changes.

In summary, the contributions in this work are:

- **Implementing GIBstazi:** We implement a hybrid, module- and class-level, RTS technique as a Maven extension.
- **Deploying RTS on Travis CI:** We describe how to deploy three types of RTS techniques in a cloud CI environment, on Travis CI.
- **Empirical Evaluation:** We empirically evaluate module-level, class-level, and a hybrid RTS technique with real-world setup in CI.
- **Failure Analysis:** We analyze failed tests from RetestAll runs and evaluate how well RTS techniques select to run such failed tests.

Overall results show several important findings about RTS. First, RTS can be beneficial for the developers, providing better results than the default RetestAll, in CI environments. Second, hybrid GIBstazi gives the best trade-off, because it works faster than GIB and only somewhat slower than Ekstazi, but it is safer than Ekstazi. Third, RTS not only saves time but also avoids misleading developers by flaky test failures.

CHAPTER 2: BACKGROUND

In this chapter, we first introduce Maven, the build system that the RTS tools integrate with. We then introduce Travis CI, the CI service used in this study. We finally describe the RTS techniques evaluated in this study.

2.1 MAVEN

Maven is a build automation tool used primarily for Java projects [3]. Maven allows a project to build using its project object model (POM) and a set of plugins that are shared by all projects using Maven, providing a uniform build system. POM is an XML representation of a Maven project held in a file named `pom.xml`. Developers can add dependencies, plugins, and extensions to the project by modifying the corresponding sections in the `pom.xml` file.

In Maven, a multi-module project is defined by a parent POM referencing multiple modules. A module in Maven is simply a collection of source code, tests, necessary resources, and POM that represents a substantial component of a larger system, defined by the developer. Each module in a multi-module project has its own POM, and there is a root POM that represents the whole project. When a module is built, the classes are compiled and tests are run. By default, Maven builds the modules sequentially and the build order of the modules in a project is determined based on the dependency relationships among the modules, which are declared in the POM files. By default, Maven stops the entire build execution immediately if any error occurs while building a particular module, i.e., the later modules are not run at all. Such default "fail fast" behavior provides faster feedback to developers if any failure or error occurs, but it can miss to run some failures from later modules.

2.2 TRAVIS CI

Continuous integration (CI) is the development practice of merging in small code changes frequently. The goal of CI is to provide rapid feedback on changes made by developers. Travis CI is a CI platform that supports software development by automatically building and testing code changes, providing immediate feedback on the changes. Travis CI is the most widely used CI service for open-source projects [18]. Travis CI is closely integrated with GitHub, the most popular web-based hosting service for open-source projects [5, 8–10].

2.2.1 Builds and Jobs

Whenever a code push or pull request happens to a code repository in GitHub that is configured to build on Travis CI, Travis CI will start a *Travis build*. A Travis build is simply a group of *Travis job*s. A Travis job is an automated process that clones the code repository, checks out the corresponding commit, builds and tests the project in a configurable virtual environment, and potentially deploys the software. By default, one Travis build has one Travis job, and Travis CI provides a default build environment and a default set of commands to build and test the software. Developers can configure Travis CI to create multiple Travis jobs to run different commands in different environments by configuring the job matrix in the `.travis.yml` file, the configuration file for Travis CI in the code repository. For example, the developers may want to build and test the software on different operating systems, e.g., Linux and Mac OS X, with different versions of Java, e.g., Java 8 and Java 9. All the Travis jobs in the same Travis build run with the same commit of the code.

2.2.2 Phases

A Travis job can be further divided into multiple phases that are executed sequentially. The two most important phases are the `install` phase and the `script` phase. The purpose of the `install` phase is to download necessary dependencies, compile the project code, and create executable artifacts. For example, the default command in the `install` phase for building a Java Maven project is `mvn install -DskipTests=true -Dmaven.javadoc.skip=true -B -V`, which purposely skips testing but installs all the compiled artifacts. The `script` phase is executed after (but not immediately after) the `install` phase. As the `script` phase occurs after the `install` phase and all necessary artifacts should be compiled, it is not necessary for the `script` phase commands to recompile the code. The purpose of the `script` phase is to run tests. For example, the default command in the `script` phase for Maven-based Java projects is `mvn test -B`.

The reason for having separate phases for compiling and testing is that when a Travis job is started, the source code is cloned into a *fresh* virtual environment, without any of the artifacts from the previous build. In fact, given the cloud nature of the Travis CI, it is not guaranteed that even the same underlying hardware is used for every job. As such, compilation is not incremental in a Travis CI setup and everything is usually compiled from scratch for every single job.

2.2.3 Caching

Travis CI can cache contents in between builds to speed up the build process. Travis CI fetches the cache for every build, including builds for branches and pull requests. If a branch does not have its own cache, Travis CI fetches the cache of the repository's default branch. When a Travis build has multiple Travis jobs, multiple caches are created, one for each job, if different jobs are using different operating systems, language runtime versions, and environment variables.

Developers can specify the directories to cache in between builds in the `cache` section of the `.travis.yml`. Such artifacts are copied into the virtual environment before the `install` phase starts, and they are uploaded onto a separate, persistent server during the `cache` phase, which happens after the `script` phase. Travis CI also has the `before_cache` phase if developers would want to run commands before uploading the new cache archive. For example, the dependency management utility may write log files into the directory developers are caching. Developers may not want these log files to affect the cache so they can run commands to delete these log files in the `before_cache` phase.

2.2.4 Build Results

Travis CI determines the build result of a Travis job based on the execution result of the `script` phase. A Travis job is marked as 'passed' if all the commands in the `script` phase succeeded and is marked as 'failed' if any command in the `script` phase failed. Typically, a 'failed' result indicates certain test failures. A Travis job is marked as 'errored' if the `script` phase is not executed due to a failure in an earlier phase or due to a system crash or timeout. Often, an 'errored' result indicates an issue during the `install` phase, suggesting that the compilation failed. Travis CI then assigns a result to the entire Travis build based on the build results of all the Travis jobs in the Travis build. A Travis build is marked as 'passed' if all the Travis jobs are 'passed', 'errored' if any Travis job is 'errored', and 'failed' otherwise.

2.3 RTS TECHNIQUES

As mentioned earlier, a RTS technique tracks the dependencies between tests and the code under test. The entities in the code under test that the tests depend on can be tracked at different levels of granularity, e.g., statements, methods, or classes, depending on the RTS technique. When developers make changes to some code under test, a RTS technique maps the change to the entities that it tracks and finds the tests that depend on those changed

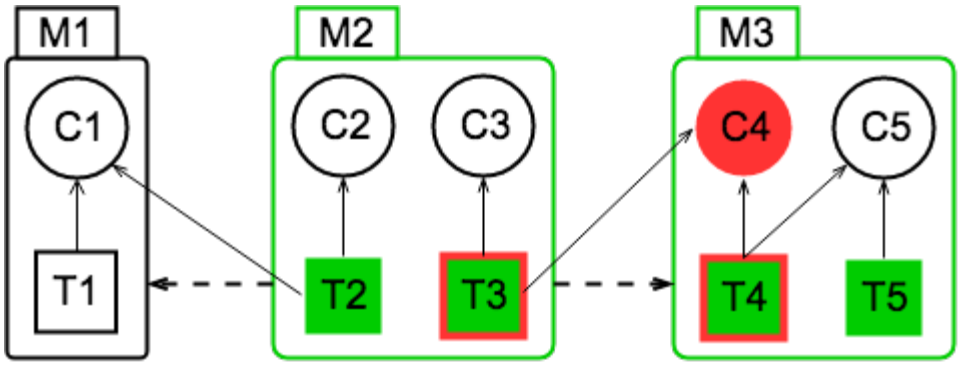

Figure 2.1: An example to illustrate RTS at different levels of granularities

entities. Different RTS techniques perform selection at different levels of granularities with respect to the tracked dependencies. A RTS technique is *safe* if it selects all the tests that are affected by the change.

In this section, we describe the three RTS techniques that we evaluate. We consider (1) GIB, which operates at the granularity of modules, (2) Ekstazi, which operates at the granularity of classes, and (3) GIBstazi, a hybrid of both GIB and Ekstazi. While Ekstazi was evaluated in prior research [14, 21], it was done only on dedicated servers and not using CI services, which means the experiment targets may not be built and tested in a way how the developers using CI would build and test. To the best of our knowledge, GIB was not previously evaluated on any platform, and GIBstazi is our original contribution. For each technique, we point out how to set it up in Travis CI.

2.3.1 Example

Figure 2.1 is a simple example to illustrate the different selections at different levels of granularities. In this example project, there are three modules (M1, M2, M3), five classes (C1, C2, C3, C4, C5), and five tests (T1, T2, T3, T4, T5). Classes and tests are displayed to be contained within the modules they belong in. Arrows with solid lines denote the dependencies between tests and classes. Arrows with dotted lines denote the dependencies between modules. For example, M2 depends on M1 because T2 depends on C1, and T2 is in M2, while C1 is in M1. In this example, consider that the developer made a change to C4 (highlighted in red). By default, without any RTS, RetestAll runs all five tests. We show the selection result of different techniques in the following subsections.

2.3.2 GIB

Gitflow Incremental Builder (GIB) is an open-source tool for performing module-level RTS [19]. GIB is a Maven extension that can be integrated into any Java project that uses Maven as the build system. GIB relies on Git to determine the code changes. Given two commits, GIB performs the *diff* to determine what files changed between the two commits. GIB then maps the files back to the modules they belong to through the directory structure and determines that those are *changed modules*. In addition to those changed modules, GIB analyzes the POM of the project to determine the modules in the upwards transitive closure of the changed modules. All these modules in the closure are selected by GIB as the *affected modules*. GIB then applies the specified test commands to the affected modules selected by GIB to run the tests in these affected modules. As we can see, GIB can also be used for incremental building, as long as we run build commands on the affected modules.

In Figure 2.1, GIB selects four tests to run. Because C4 is changed, M3 is a changed module and selected first, and then M2 is selected because M2 depends on M3 (affected modules are highlighted in green). Thus all the tests in M2 and M3 (T2, T3, T4, T5) are selected to run, while T1 is skipped.

When using GIB on Travis CI, it is necessary to configure GIB to compare against the commit of the previous build to determine the code changes. Travis CI does provide this information in the `TRAVIS_COMMIT_RANGE` environment variable. Another important point to consider is how compilation interacts with using GIB for RTS. When GIB determines what modules should be selected, it inherently assumes that the dependencies of those modules have already been built and are available. If not, compilation would fail due to missing dependencies. When using a dedicated machine for building the project, this is not an issue because it is expected that the previous builds are built on the same machine and the artifacts are kept around. On Travis CI, though, each build starts in a fresh environment and any compiled artifacts from the previous build are not available. However, Travis CI operates with a separate phase, the `install` phase, that is meant to perform compilation. For GIB to perform RTS on Travis, the `install` phase must run to completion, without using GIB. (Other build systems such as Bazel [1] can cache and reuse the compiled artifacts but require a different setup from the default Travis CI.) After all the code is compiled, GIB can be used in the later `script` phase for testing to actually perform module-level RTS.

Enhancing GIB

For our experiments, we have (1) configured GIB in a reasonable way for RTS and (2) extended GIB with a new feature. GIB uses Git to track file changes and what modules are affected by those changes, but not all file changes actually affect Java test behaviors. For example, if the only change made is to the top-level README file, GIB would assume that a change to a file in the root of the project affects all the modules, thereby running all tests, even though it is extremely unlikely any change to a README file would actually result in changes to test behavior. To avoid such over-selection of tests, we configure GIB to exclude certain file patterns that are unlikely to affect test behavior by using the following regex:

```
.*\.apt$|.*\.txt$|.*\.md$.|*\.html$|.*\.rst$|.*\.scss$|.*\.css$|.*\.png$|.*\.py$|
|.*README.*|.*\.jpg$|.*\.jpeg$|.*NOTICE$|.*\.git.*|site.xml|index.xml|checkstyle.xml
```

We construct these filters based on our best knowledge of what files will not affect test behaviors in the general case. Developers using GIB for their projects would want to create more specific filters, e.g., they would not ignore changes to the files that are used as test resources, even if those files match some of the general ignorable extensions.

While these exclusions filter out many cases where file changes can lead to unnecessary test runs, we found that there were still many cases where changes to the `pom.xml` files, the POM for a Maven project, lead to a large number of test runs. While it is necessary to track changes to `pom.xml`, where a change can lead to true changes in dependencies in the project, we observed that often the only change to `pom.xml` files in a project did not affect the dependencies. For example, some changes would simply update the developer's email address, which should not affect test behavior. Indeed, anytime any external dependencies do not change, then test behavior is unlikely to change as well, and so changes to `pom.xml` that do not change external dependencies should be fine to ignore. We added the logic that checks if any dependency of any module within the project changed between runs to GIB. If there is no change to dependencies, then GIB ignores changes to `pom.xml` (by adding `pom.xml` to the list of files to exclude). Furthermore, we ignore version changes if there is an intra-project dependency between the project's own modules, as a common change is an update of the version number of all modules within the same project.

For tracking dependency changes among modules, GIB parses the Maven project to determine the dependencies of each module. It encodes this information into a separate file called `classpathfile`. Before each run, if such a file exists, GIB compares the contents of the file with the dependencies of each module to see if any of the dependencies changed. If not, `pom.xml` can be filtered out. For this feature to work on Travis CI, `classpathfile` file has to be preserved across builds and restored at the start of the `script` phase. This storing and

restoring of the file is achieved by the caching feature in Travis CI described in 2.2.2.

2.3.3 Ekstazi

Ekstazi implements a class-level RTS technique [13, 14]. Ekstazi is a Maven plugin that can be integrated into any Java project that builds using Maven. Ekstazi instruments the code under test to obtain which classes each test depends on.[1]

In Figure 2.1, Ekstazi selects two tests to run. Because C4 is changed, Ekstazi iterates through all the tests in this project and selects tests that depend on C4. In the end, only T3 and T4 are selected to run by Ekstazi (framed in red); note that these two tests come from different modules.

Prior work [14] has shown that Ekstazi is quite effective compared to using a finer granularity, such as methods. The trade-off of tracking fewer dependencies, in both analysis time to find what tests to run and in the overhead from running instrumented code under test, is beneficial when considering the overall time of RTS, even if a greater number of tests is being selected. In addition to the dependency mapping from test to classes, Ekstazi also tracks checksum values for each `.class` file (the compiled file form for a source Java file, representing a Java class). When a change is made, after all the source files have been compiled to `.class` files, Ekstazi uses the stored checksum to determine which classes actually changed. Once Ekstazi knows what classes changed, it selects for running the tests that depend on those changed classes, found from the stored dependency mapping. Both the mapping from tests to dependencies and the checksum of the classes are stored within `.ekstazi` directories, one for each Maven module in the project. Ekstazi relies on the files within the `.ekstazi` directories to determine what tests to select.

On Travis, the artifacts from the previous build are not by default preserved in the virtual environments used for Travis CI. As such, for Ekstazi to work on Travis CI, `.ekstazi` directories must be preserved across builds and restored before testing is performed in the `script` phase. Caching and restoring these `.ekstazi` directories is an extra overhead necessary for Ekstazi to work in a CI environment.

2.3.4 GIBstazi

As we can see, GIB can select to run a large number of tests, often much larger than the number of tests selected by Ekstazi. Many tests selected by GIB may be actually unaffected

[1]When we write "test", we mean "test class" not "test method". Ekstazi selects to run test classes, and a test class can contain several test methods. In this work, we count tests at the level of test classes as well.

by the changes and do not need to be run. We implement GIBstazi, a combination of GIB and Ekstazi, to improve on GIB in selecting fewer tests, and to improve on the safety of Ekstazi in selecting more tests when non-Java files change. GIBstazi first relies on GIB to determine what modules are affected by the change. For those affected modules, GIBstazi applies Ekstazi to determine what tests within each module need to be run.

In Figure 2.1, GIBstazi also selects two tests to run. Since C4 is changed, GIBstazi selects two modules, M2 and M3, as the affected modules. GIBstazi then iterates through the tests in M2 and M3 and selects tests that depend on C4. Thus only T3 and T4 are selected to run by GIBstazi. Comparing with Ekstazi, which iterates through all the tests in the project, GIBstazi does not check if T1 depends on the change.

Comparing with GIB, GIBstazi should select fewer tests due to the use of Ekstazi. Comparing with Ekstazi, Ekstazi can be unsafe in its selection as it does not track files outside of the compiled class files from source code. For example, if a developer updated a `.json` file that is used as test input, Ekstazi would not select any tests to run since it does not track what tests depend on such non-source-code file, even though the change could potentially affect test behavior. As such, we implement GIBstazi to check if the changed files are source code files only. If all the changes are to source code files, GIBstazi uses Ekstazi to perform selection safely on the modules selected by GIB. If some changes are to non-source-code files, GIBstazi defaults back to GIB module-level RTS and selects to run all the tests within the affected modules. For a given module, GIBstazi can either select no tests (if the module is not affected), all the tests (if non-source-code files changed), or the same tests as Ekstazi. We expect GIBstazi to select some number of tests that is between the number selected by GIB and Ekstazi, and the time savings for regression testing from using GIBstazi should also fall in between GIB and Ekstazi.

As GIBstazi is a combination of both GIB and Ekstazi, for one to use GIBstazi on Travis CI, it requires all the necessary components from GIB and Ekstazi. GIBstazi needs to preserve the `classpathfile` file for the purpose of the enhancements we applied for GIB across builds. GIBstazi also needs to preserve the `.ekstazi` directories across builds since Ekstazi requires such files for selection.

CHAPTER 3: EXPERIMENTAL SETUP

In this chapter, we describe our experimental setup. We describe how we selected the target projects used in our experiments along with the commits used for each project. We then describe how we configured to run the different RTS techniques for each project's commits on Travis CI. We also describe how we shadowed the project's current commits with GIBstazi on Travis CI.

3.1 PROJECTS

To determine what projects to use in our evaluation, we start by querying GitHub for the projects that use Java as the primary programming language. We obtain the top 1000 Java projects on GitHub ranked by stars, a measurement of popularity. We then further filter out the projects that use Maven as the build system because the RTS tools that we evaluate are for the Maven build system. In addition, given that GIB and GIBstazi perform selection at the granularity of module-level, we need the target projects to be multi-module. Furthermore, we need the projects that are using Travis CI. In total, we get 105 Java Maven multi-module projects that build on Travis CI.

Further, we want to evaluate these RTS tools on the projects with sufficiently long-running builds such that a developer may want to use RTS in the first place to save some test time. For each of the 105 projects, we query Travis CI for the latest 20 builds, average the build times, and select projects that took on average longer than 10 minutes to build. Such filtering results in 46 projects. The build times reported by Travis CI represent the overall time the project takes to build and not just the times for testing. In particular, some of these projects only compile the code on Travis CI and intentionally skip tests. As we are evaluating RTS techniques, we want the projects that run tests on Travis CI. We further keep only the projects that are compliant with all three RTS tools (e.g., GIB requires the projects to use Java 8). We finally get 22 projects for our evaluation.

Java Maven multi-module projects using Travis CI	105
Projects with build time longer than 10 min	46
Projects with tests and replayable with RTS tools	22

Table 3.1: Filtering of Projects

3.2 COMMITS

After we selected the target projects, we need to collect the revisions for each of them for rerunning on Travis CI. Recent studies on RTS [14,21] selected the revisions as *all sequential commits* from the master branch of each project. However, not all these commits are built on Travis CI because a single Travis build corresponds to a push from the developer, and a single push may include several commits. As such, the size of the changes between builds on Travis CI can be generally larger than the size of changes between sequential commits, as used in prior studies. Due to our focus on evaluating RTS on Travis CI, we collect the revisions for our experiments by querying Travis CI for the commits that triggered the builds on Travis CI. For each project, we collect the commit SHAs associated with the latest 20 push builds on the master branch. We collect these SHAs in the order in which they actually happened on Travis CI such that replaying these historical commits later on gives the same code changes between each build as observed by the developers when using Travis CI for these builds.

3.3 REPLAYING WITH RTS TOOLS

We replay the commits we collected for each project on Travis CI for RetestAll and all RTS techniques. For each technique and RetestAll, we create a new GitHub account and fork the projects into the account. Next, for each commit of each project, we checkout the commit, make necessary modifications to the `.travis.yml` and `pom.xml` files in order to run the appropriate RTS technique when building on Travis CI, and finally recommit the state of the code as a new commit and push to GitHub, triggering the build on Travis CI for our own forked repository.

For each RTS technique, we aim to make the smallest amount of changes necessary to the `.travis.yml` and `pom.xml` files. We describe the general modifications made to `.travis.yml` here and describe the tool-specific modifications of `.travis.yml` and `pom.xml` in the following subsections.

We modify the `.travis.yml` file as follows. At the end of the `script` phase, we add extra commands to count how many tests are run during the build process, by counting the number of Surefire report files generated[1], where each file represents a test run. We also add steps in the `script` phase to record how much time the `script` phase takes to run. By timing this phase, where testing is meant to be performed, we can simulate a "local" build

[1]We have since discovered that this is not a reliable way to count the number of tests actually run because Surefire can generate some report files even if a test class did not run any test method.

using RTS, where times for downloading dependencies and compiling code are not included. We next remove altogether the `notifications` section from the `.travis.yml`. This section is used to notify developers of the status of the build, and we do not want to notify the developers when replaying historical builds. Removing `notifications` should not disrupt the compiling and testing process in the previous `install` phase and `script` phase. Because we uniformly remove the `notifications` section from all techniques, including RetestAll, the timing comparison should be consistent as well. Finally, we modify the `.travis.yml` to not run any jobs with Java versions below Java 8, because GIB requires Java 8. These modifications should not disrupt the building and testing behavior in the `install` and `script` phase needed for evaluating RTS, and because we uniformly make these modifications for all techniques, including RetestAll, our comparison between these techniques should be consistent as well.

3.3.1 GIB

For GIB, we first modify the `pom.xml` file of the project to add the GIB as a Maven extension. We configure the extension to compare the differences between two Git commit SHAs, where the first is the commit SHA of the previous build and the second is the current SHA. The two commit SHAs can be obtained using the Travis environment variable `TRAVIS_COMMIT_RANGE`. Furthermore, we apply the exclude regex as described in Section 2.3.2.

Besides modifying the `pom.xml` file, we have to make additional modifications to the `.travis.yml` file. Recall from Section 2.3.2 that we use the `classpathfile` file to track these dependencies to help decide when it is unnecessary to consider changes to a project's `pom.xml`, which can trigger all tests in the project to run. We configure the `.travis.yml` to cache the `classpathfile` for the project across builds. Also, we configure the `.travis.yml` to *not* use GIB during the `install` phase, as the code must build from scratch, and we do not want GIB to to prevent certain modules from being compiled in this phase. We disable GIB at the beginning of the `before_install` phase (the phase Travis executes right before the `install` phase) and then enable GIB at the end of the `before_script` phase (the phase Travis executes right before the `script` phase).

3.3.2 Ekstazi

For Ekstazi, we first configure the `pom.xml` file to include the Ekstazi Maven plugin; we use version `4.6.3` [2] in our evaluation. We also configure the `.travis.yml` to cache the `.ekstazi` directories that Ekstazi generates to track the test dependencies. We then add in the `before_script` phase the commands to copy in the cached `.ekstazi` directories into their

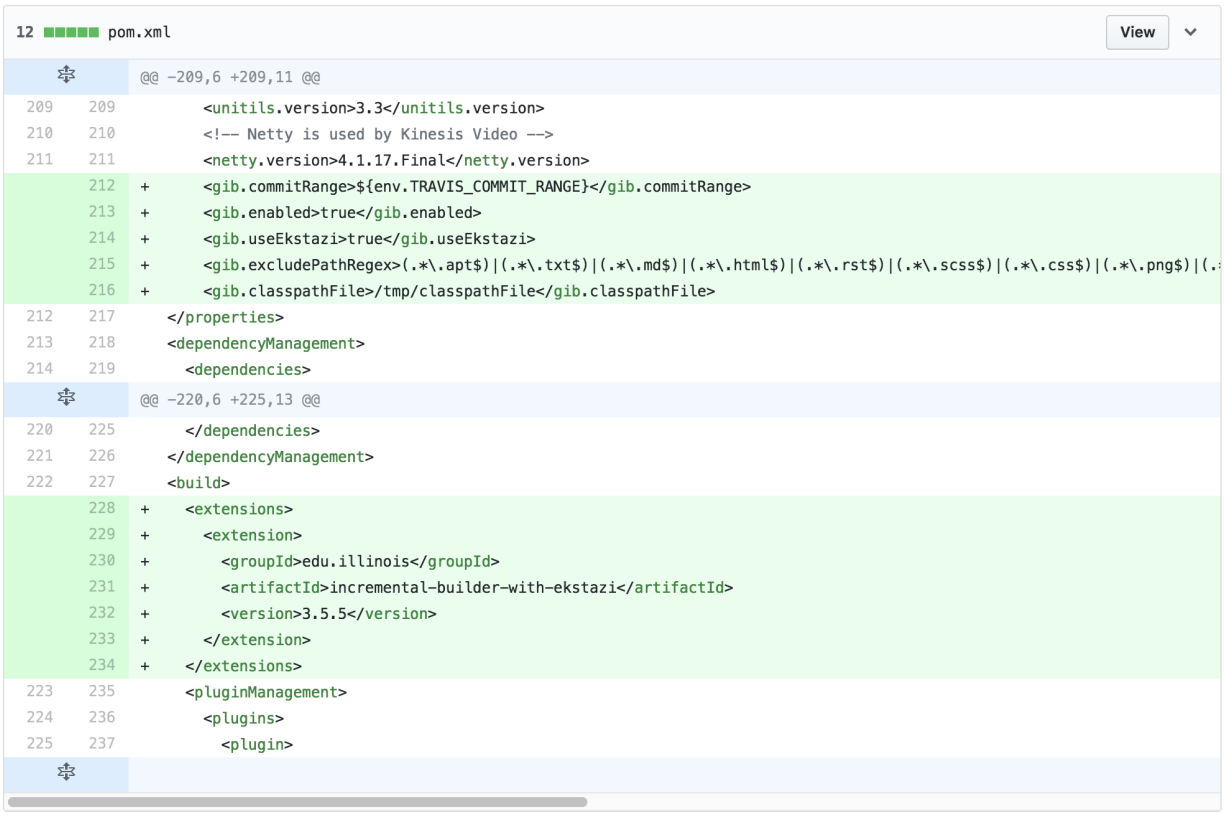

Figure 3.1: An example of changes made to `pom.xml` to enable GIBstazi

desired places in the project directory for the `script` phase to use for testing, and we add in the `before_cache` phase the commands to save the updated `.ekstazi` directories to the cache after the tests finish.

3.3.3 GIBstazi

For GIBstazi, the modifications we make to the `pom.xml` and `.travis.yml` files are a combination of those we make for GIB and Ekstazi. In Figure 3.1, we show an example of modifications made to `pom.xml` to enable GIBstazi. We modify the `pom.xml` file to use the GIBstazi extension, configured similarly as with GIB but also to use Ekstazi in the affected modules. We modify the `.travis.yml` to cache both the `classpathfile` and the `.ekstazi` directories, along with the same extra steps for restoring and updating the files for the `script` phase.

3.4 COLLECTING JOB LOGS

Replaying historical commits with our modifications results in new Travis jobs run on Travis CI. After these jobs finish, we download the job logs from Travis CI for analysis. As a basic sanity check, we only consider the jobs where we could parse the logs successfully for indication of the number of tests run and the time the testing in the `script` phase took. It is possible for jobs to not finish properly, such as due to compilation errors (so tests are not even run) or due to strict timeout limits by Travis CI. We cannot properly compare any numbers pertaining to tests selected or testing time in such cases. In total, across all 22 projects, we collected 1444 jobs for our evaluation.

3.5 SHADOWING WITH GIBSTAZI

We also shadow the recent commits from the projects, replaying the current builds with GIBstazi. We track the builds on the master branch from the projects as they are triggered by developers, and we replay those new builds when they are triggered. First, we fork each project into a new account for shadowing. We set up a cron job scheduled to run once an hour. When the cron job is scheduled to run, it queries Travis CI for each project for any new builds that occurred since the last time the cron job was run. If there were any new builds for a project, the cron job pulls the commits corresponding to those builds into the shadowing fork of the project and replays them like we do for the historical builds. We perform these runs only for GIBstazi, because GIBstazi strikes a good balance among all three RTS techniques: it provides better time savings over GIB and is designed to be safer than Ekstazi.

CHAPTER 4: RESULTS

In this chapter, we first show the results from our replay of historical builds with RTS. We compare the number of tests selected and the time for testing with the RTS techniques against RetestAll. We also analyze the test failures from RetestAll and inspect whether the RTS techniques select to run the failed tests. We finally show the results from our shadow runs of ongoing Travis CI builds (in contrast to replay runs of historical builds).

4.1 STATISTICS OF JOBS

Table 4.1 shows the distribution of jobs that we collected across the 22 projects used in our evaluation. We label each project with an ID that we use in later tables and show the project's slug name from GitHub. The jobs are sorted by the total build time, shown later. We also show the number of jobs we analyzed for each project and the percentage of those jobs classified as 'passed', 'failed', or 'errored', based on the job status reported by Travis CI for RetestAll. The overall number of jobs with status 'passed', 'failed', and 'errored' are 1308 (89.87%), 134 (9.97%), and 2 (0.15%), respectively.

We have almost no 'errored' jobs; such status typically means compilation failed in the `install` phase, so no testing would have been done, and we could not collect any information about tests. However, in the two 'errored' jobs, we find that the jobs errored in the `deploy` phase that occurs after the `script` phase, i.e., after tests have run. In such cases, we are still able to collect information about the tests selected to run and the time for testing. (In retrospect, we could have changed `.travis.yml` to not try the deployment.)

Another point to note is that jobs having status 'failed' does not necessarily mean that tests failed. In Travis CI, status 'failed' means that the `script` phase failed, which may not be due to test failures. For example, project apache/rocketmq (P3) has all of its jobs with status 'failed', but we found that the reason is due to the developers including in their Maven build command run in their `script` phase a step that tries to deploy artifacts to another server, which we do not have access to. Tests can be passing before this step, but because the deploy step is in the `script` phase, the entire job is marked as 'failed'.

ID	Project	Jobs	Passed	Failed	Errored
P1	SonarSource/sonarqube	40	20	20	0
P2	elasticjob/elastic-job-lite	20	20	0	0
P3	apache/rocketmq	19	0	19	0
P4	alibaba/dubbo	19	19	0	0
P5	aws/aws-sdk-java	20	19	1	0
P6	brianfrankcooper/YCSB	20	18	2	0
P7	apache/incubator-skywalking	20	20	0	0
P8	antlr/antlr4	179	178	1	0
P9	vavr-io/vavr	5	5	0	0
P10	Graylog2/graylog2-server	2	2	0	0
P11	javaparser/javaparser	20	20	0	0
P12	languagetool-org/languagetool	19	18	1	0
P13	druid-io/druid	88	73	15	0
P14	killbill/killbill	47	0	47	0
P15	apache/storm	89	81	8	0
P16	iluwatar/java-design-patterns	20	15	5	0
P17	google/guava	37	37	0	0
P18	javaee-samples/javaee7-samples	352	351	1	0
P19	prestodb/presto	179	178	1	0
P20	apache/incubator-pulsar	13	7	4	2
P21	apache/flink	221	219	2	0
P22	Tencent/angel	15	8	7	0
	SUM	1444	1308	134	2

Table 4.1: Basic statistics about projects used in evaluation, including distribution of passed/failed/errored jobs for RetestAll

ID	RetestAll			GIB			Ekstazi			GIBstazi		
	Tests (#)	Test Time (m)	Total Time (m)	Tests (%)	Test Time (%)	Total Time (%)	Tests (%)	Test Time (%)	Total Time (%)	Tests (%)	Test Time (%)	Total Time (%)
P1	2.00	2.52	3.62	100.00	98.79	100.78	100.00	99.98	99.97	100.00	93.29	93.73
P2	158.25	1.43	6.57	28.15	41.71	78.52	11.66	66.41	98.43	28.15	46.55	73.17
P3	55.79	6.67	8.34	24.53	27.02	39.46	22.64	46.13	67.00	24.53	34.80	52.06
P4	162.05	5.60	8.81	59.76	60.41	73.48	14.26	94.91	123.12	32.87	58.55	81.23
P5	177.45	7.20	9.19	94.08	97.32	97.36	18.71	52.80	63.68	86.00	99.07	100.18
P6	30.60	6.73	10.42	60.62	58.23	67.70	47.88	31.55	65.48	44.28	53.24	72.63
P7	98.60	4.10	10.82	21.60	32.61	80.78	7.35	42.15	60.00	17.85	36.68	80.51
P8	12.85	9.92	11.79	15.86	18.87	30.42	24.38	35.10	46.51	15.99	21.50	33.60
P9	140.00	10.91	11.80	100.00	80.79	94.07	38.71	68.23	72.20	59.00	71.69	92.29
P10	177.00	7.37	13.26	100.00	106.00	108.93	50.00	88.12	94.91	100.00	146.49	125.96
P11	176.90	6.94	15.01	91.58	82.03	74.19	27.87	53.31	43.73	30.16	54.65	40.82
P12	399.47	12.52	15.58	26.02	36.20	45.69	6.92	16.77	33.41	12.66	14.55	32.11
P13	173.40	10.13	17.30	50.60	52.03	71.72	16.50	68.64	78.03	15.32	31.41	61.02
P14	159.87	10.69	18.09	110.87	386.83	101.73	80.10	76.87	87.33	128.05	120.17	113.86
P15	44.28	9.12	19.03	60.16	41.84	72.78	19.74	87.85	94.70	35.45	40.94	73.07
P16	315.05	6.60	19.55	26.22	31.52	49.06	57.48	90.07	96.17	21.17	31.88	52.39
P17	494.08	16.73	20.14	100.00	95.76	96.34	55.40	58.81	79.93	55.63	57.13	64.56
P18	8.89	3.67	22.23	35.99	44.06	98.62	23.71	33.47	88.96	35.41	47.65	98.65
P19	71.27	24.93	27.79	63.63	79.00	81.73	3.83	30.88	38.16	3.83	26.36	34.56
P20	157.85	30.20	32.42	61.35	70.66	72.29	103.46	95.41	95.83	57.26	51.77	54.23
P21	359.53	32.66	34.04	50.77	61.52	63.27	16.53	69.07	71.28	39.84	63.22	65.48
P22	79.53	36.51	39.65	31.43	33.86	38.95	0.00	3.79	12.35	0.00	1.78	10.12
AVG	123.19	13.72	21.15	58.14	67.78	75.93	24.90	53.52	70.24	38.43	46.39	66.95

Table 4.2: Tests selected and time savings from using RTS across *all* jobs

4.2 TESTS SELECTED AND TESTING TIME

Table 4.2 shows the results of replaying with RTS per project, where each project is referenced by ID. The columns under 'RetestAll' show, for each project, the average number of tests across all jobs, the average number of minutes the `script` phase took (testing time) across all jobs, and the average number of minutes each job took overall (total build time). The columns under 'GIB', 'Ekstazi', and 'GIBstazi' show, for each project and respective RTS technique, the average percentage of tests run, the percentage of the `script` phase time the technique took to run, and the percentage of the overall time the technique took to run, all normalized with respect to RetestAll. The final 'AVG' row is the weighted average of each column, computed using all jobs across all projects at once.

From the results, we see that the weighted average testing time for all three RTS techniques is lower than the testing time for RetestAll, with 67.78% for GIB, 53.52% for Ekstazi, and 46.39% for GIBstazi. We see that RTS can significantly improve testing time for the commits that are pushed in a CI environment. Moreover, we see that RTS helps improve the total build time as well, where GIB takes 75.93%, Ekstazi takes 70.24%, and GIBstazi takes 66.95% of the RetestAll total time. We observe that GIBstazi outperforms the other two RTS techniques in terms of time savings for both testing time and total time, but we discuss why the time measurements across *all* jobs are not reliable. We still show these times because they give an indication of what a developer may observe in practice.

When we consider the averages of number of tests selected across the RTS techniques, we see that Ekstazi ends up selecting the fewest percentage of tests. This observation makes sense, as Ekstazi operates at a finer granularity than GIB and GIBstazi. However, the time for Ekstazi is still larger than for GIBstazi despite selecting to run fewer tests.

It is important to note that Table 4.2 includes *all* jobs, some of which may be 'passed' for some technique and 'failed' for another. For example, we see some projects where the percentage of tests run by a RTS technique exceeds 100%, suggesting that RTS may be somehow running more tests than available! However, the reason for this anomaly, that RetestAll seemingly ran fewer tests than a RTS technique, is the test failures in the middle of the build execution in RetestAll, as we are using RetestAll as the baseline for comparison. (Most of these failures are flaky tests as we discuss in Section 4.5.) Generally, as mentioned in Section 2.1, when a test fails in a multi-module Maven project, the build fails early, and the tests in later modules that should have been run are not actually run. Our tooling counts the number of tests based on what was actually run, so we can end up not counting all the tests in the test suite for a project during a RetestAll run with a test failure.

ID	# Jobs	RetestAll			GIB			Ekstazi			GIBstazi		
		Tests (#)	Test Time (m)	Total Time (m)	Tests (%)	Test Time (%)	Total Time (%)	Tests (%)	Test Time (%)	Total Time (%)	Tests (%)	Test Time (%)	Total Time (%)
P1	20	2.00	3.97	5.10	100.00	100.71	99.75	100.00	100.08	96.49	100.00	97.50	96.14
P2	20	158.25	1.43	6.57	28.15	41.71	78.52	11.66	66.41	98.43	28.15	46.55	73.17
P3	0	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
P4	18	161.89	5.61	8.70	57.48	59.00	72.40	11.56	93.48	119.31	29.07	55.68	80.36
P5	16	179.00	7.36	9.35	100.00	104.97	103.76	21.79	53.47	63.15	100.00	113.03	111.52
P6	18	31.11	6.98	10.68	61.61	59.23	67.78	44.82	25.66	61.05	43.75	53.00	71.86
P7	20	98.60	4.10	10.82	21.60	32.61	80.78	7.35	42.15	60.00	17.85	36.68	80.51
P8	178	12.91	9.92	11.79	15.88	18.89	30.47	24.41	35.20	46.64	16.01	21.54	33.66
P9	5	140.00	10.91	11.80	100.00	80.79	94.07	38.71	68.23	72.20	59.00	71.69	92.29
P10	2	177.00	7.37	13.26	100.00	106.00	108.93	50.00	88.12	94.91	100.00	146.49	125.96
P11	20	176.90	6.94	15.01	91.58	82.03	74.19	27.87	53.31	43.73	30.16	54.65	40.82
P12	11	400.09	12.73	16.24	7.70	8.67	21.35	0.80	4.91	21.11	7.70	12.75	27.95
P13	40	192.90	12.58	20.77	36.28	38.09	62.19	12.08	62.41	74.19	9.86	22.21	56.94
P14	0	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
P15	50	53.72	5.09	14.89	52.53	65.29	89.72	15.52	76.69	93.17	31.09	64.31	90.50
P16	13	319.23	6.73	22.02	31.59	36.49	45.47	59.04	94.04	94.59	23.93	35.36	47.81
P17	36	494.00	16.72	20.11	100.00	95.53	96.17	56.78	59.68	80.92	57.01	58.08	65.39
P18	347	8.85	3.67	22.23	35.18	43.35	98.41	23.09	33.14	89.06	34.59	47.18	98.49
P19	25	11.88	24.69	28.94	100.00	99.43	99.88	100.00	101.17	101.35	100.00	100.56	100.97
P20	0	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
P21	141	366.18	30.86	32.19	42.43	66.25	68.01	14.71	61.50	63.94	35.41	69.65	71.61
P22	0	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
SUM/AVG	980	112.39	10.55	19.56	52.50	57.16	79.15	23.87	55.61	76.38	36.45	56.29	78.76

Table 4.3: Tests selected and time savings from using RTS across *only passed* jobs

4.3 COMPARING WITH ONLY PASSED BUILDS

To provide a fairer comparison between the effectiveness of the RTS technique with respect to RetestAll, we apply stricter filtering on the jobs. We only consider the jobs where RetestAll and all three RTS techniques are 'passed'. By looking at only 'passed' jobs, we get a better measure of time, ensuring that these are jobs without any test failures that could lead to early failing, skipping later tests that might actually be selected to run.

Table 4.3 shows the results for these passed jobs. The table format is similar to Table 4.2, but we add the number of jobs considered for each project. We now consider 980 jobs across all projects ('SUM/AVG' shows this total number for jobs). Some projects result in no jobs after this filtering, indicating that there are no jobs where RetestAll and all three RTS techniques pass; these are indicated by 0's under '# jobs' and N/A's in the remaining columns. With this filtering, we see the weighted average for the total time for RetestAll has dropped somewhat, from 21.15 minutes to 19.56 minutes. This drop can be expected because longer-running builds are more likely expected to fail.

We see from the summary row that the average testing and total time percentages are higher than in Table 4.2. Moreover, we see that Ekstazi now outperforms both GIB and GIBstazi in terms of both testing time and the total time, at 55.61% and 76.38%, respectively. This difference in time is somewhat expected, as Ekstazi also selects the fewest tests, on average 23.87% of RetestAll tests run. GIBstazi, on the other hand, outperforms GIB in terms of percentage of tests selected, testing time, and total time. The trend between GIB, Ekstazi, and GIBstazi now matches our initial expectations, laid out in Section 1.2.

4.4 COMPARISON BETWEEN EKSTAZI AND GIBSTAZI

While Ekstazi compared against GIBstazi has smaller average percentages of both testing time and total time, we do observe individual jobs where Ekstazi surprisingly selects to run more tests than GIBstazi. We inspected these jobs to determine why Ekstazi selected more tests than GIBstazi.

These jobs came from eight projects. We sampled a job from each of these projects, as it is likely a characteristic of the project that leads to Ekstazi selecting more tests. We checked out the corresponding commit for the sampled job and attempted to run Ekstazi on the same commit twice, without making any changes. Ideally, when no change was made, Ekstazi should always select to run no tests. For four of the projects we could no longer rebuild the project and reproduce Ekstazi selecting more tests. For the remaining projects, when we ran Ekstazi twice, Ekstazi would always run some tests.

In two of the projects (P6 and P16), we found that the test suite uses a test runner other than the standard JUnit4, e.g., TestNG or JUnit Jupiter (new in JUnit5). When such test runners are used, Ekstazi selects to run all the tests regardless of changes[1]. In contrast, GIBstazi can (correctly) find that a module is unaffected and would not run any test. In another project (P13), we found that even if Ekstazi determines not to run some tests, Surefire still generates the log files indicating the test was (attempted to) run, so we count such log files. An example is when some test classes have no test methods to run (e.g., an abstract test class). However, in such cases, while the number of tests actually run may be incorrect (one test class may be run with zero test methods, so it should be more precisely counted as zero test classes), the overall timing is not largely affected, because there is no actual test to run and take time.

For the remaining project (P18), we found that compiling the same commit twice in a row results in different compiled class files. Further inspection showed that some source files are automatically generated as part of the build process and then later compiled into class files. The automatic source generation is nondeterministic. Specifically, the order of the methods in the generated source files can differ between builds, resulting in a different generated source each time, which in turn results in a different compiled class file for that source, which eventually triggers Ekstazi to select the tests that depend on these automatically generated classes. So even if there is no change to any non-generated source file, the compiled class file can differ. Ekstazi inherently relies on the compiled classes to not change if the developer makes no changes to source code; an argument is made that comparing class files is more robust than comparing source files [13]. However, if the compiled classes do change even if there are no changes to the non-generated source files, then Ekstazi would find spurious changes and run more tests than necessary.

For all of these projects, the reason that GIBstazi ran fewer tests than Ekstazi is due to GIB first selecting affected modules. If the tests that Ekstazi selects and runs spuriously are not from one of the affected modules, the tests would not be selected by GIBstazi because their module would not be considered in the first place. As such, GIBstazi can actually provide faster regression testing time than Ekstazi, in addition to being safer in some cases.

Aside from GIBstazi being faster than Ekstazi in the cases where GIBstazi selects to run fewer tests, we find that GIBstazi can be faster even when the two select the same number of tests due once again to the module selection that happens first. For example, in P13, we found a job where both Ekstazi and GIBstazi select to run no tests. For GIBstazi, the (selection and) testing process is rather fast, because first no modules are selected, and in

[1]We confirmed this via private communication with Milos Gligoric, the main developer of Ekstazi.

ID	# Failed Tests	GIB			Ekstazi			GIBstazi		
		Selected	Not Selected	Unknown	Selected	Not Selected	Unknown	Selected	Not Selected	Unknown
P5	1	100.00	0.00	0.00	0.00	100.00	0.00	100.00	0.00	0.00
P6	2	50.00	50.00	0.00	50.00	50.00	0.00	50.00	50.00	0.00
P12	1	0.00	100.00	0.00	0.00	100.00	0.00	0.00	100.00	0.00
P13	10	70.00	30.00	0.00	20.00	70.00	10.00	20.00	70.00	10.00
P14	47	100.00	0.00	0.00	100.00	0.00	0.00	100.00	0.00	0.00
P15	7	71.43	28.57	0.00	57.14	42.86	0.00	71.43	28.57	0.00
P16	5	0.00	100.00	0.00	20.00	60.00	20.00	0.00	100.00	0.00
P18	7	0.00	100.00	0.00	0.00	14.29	85.71	0.00	100.00	0.00
P19	1	0.00	100.00	0.00	0.00	100.00	0.00	0.00	100.00	0.00
P20	13	53.85	46.15	0.00	84.62	0.00	15.38	38.46	46.15	15.38
P21	2	100.00	0.00	0.00	50.00	50.00	0.00	50.00	50.00	0.00
SUM/AVG	96	72.92	27.08	0.00	69.79	19.79	10.42	64.58	32.29	3.12

Table 4.4: Percentage of test failures selected or not selected by RTS

turn no tests are run. However, Ekstazi has to analyze every module to determine that no tests should be run in any of the modules. While usually this analysis is rather fast, on the order of 3–4 seconds per module, P13 is a project with over 20 modules, such that GIBstazi skipping all modules first leads to significant speedup compared against Ekstazi analyzing each and every module, for a total of over 60-80 seconds.

4.5 TEST FAILURES

RTS aims to select to run only the tests that are affected by changes. However, a crucial question is whether RTS is safe, i.e., whether it can potentially miss to select tests that are actually affected by changes. If a test that should have been selected due to the change is missed, the developer could miss a regression fault, defeating the purpose of regression testing in the first place. In our study, we found several test failures from RetestAll, and we looked into whether or not the RTS techniques miss to select to run those tests.

Table 4.4 shows the number of test failures we find from RetestAll and the breakdown of what percentage of those tests are selected to run by the different RTS techniques. We obtained the test failures by parsing the job logs using tools provided by TravisTorrent [7]. Unfortunately, the parsing could not successfully parse out test failures for all 'failed' job logs due to project-specific "noise". For example, some projects run tests with the "quiet" option. Furthermore, looking at just the Travis CI status of the job actually does not necessarily indicate there were test failures, as other issues could have caused the job to be marked as 'failed', so such jobs have no failed tests to parse. In total, we found 11 projects with test failures we could parse out, leading to a total of 96 test failures. We categorize those test failures for each RTS technique for each project into three categories: selected, not selected, or unknown. We categorize a test failure as unknown if we cannot tell from the logs whether

the test would have been selected to run or not; such a case can happen if another test in a module before the module with the relevant test fails in the RTS run, leading to an early failure of the entire run that skips running the tests in the later modules. As such, we cannot tell if the RTS technique would have selected to run the test since it is skipped due to early failure.

From the table, we see that all the RTS techniques do *not* select to run many tests that fail when run with RetestAll, which is *seemingly* bad. On average, the RTS techniques do not select a rather large percentage of the failed tests, e.g., GIBstazi does not select on average 32.29%. This percentage is rather high, considering that by not selecting to run these tests, the developer would not see the failure and not know if there are regression faults in the most recent code changes.

We inspected all 96 test failures from RetestAll. As a first step, we found that for 29 of the test failures, the job with those test failures was immediately preceded by a job that had the same test failures. From a RTS perspective, even if RTS does not select to run some tests, the outcome of the not selected tests is *not* deemed to pass; instead, the outcome of the not selected tests is copied from the prior runs. In other words, the failed status of the test is still known from before. For example, if the developer does not make a change related to fixing the test failure, there is no need to run the unaffected test again to reveal the same known failure. A RTS tool could even literally copy the same outcome of the tests from a previous job if the tests are not run, allowing the developer to consistently see the test failure when the developer has not yet fixed it, but there is no need to actually spend resources to run such tests. (Ekstazi has an option `forcefail` to run all failing tests from prior runs, even if they are unaffected by the change.)

Of the remaining 67 test failures where the test passed in the immediately preceding job, we found that *all but one* of them are due to flaky tests. Flaky tests are tests that can pass or fail nondeterministically on the same code [6, 23]. As such, flaky test failures are not related to code changes and do not reveal real faults necessarily due to code changes. We confirmed that these tests are flaky in two ways. For some of the tests, we confirmed by finding that one of the RTS techniques did run the test and yet the test passed. For the remaining tests, we reran the same job for RetestAll up to five times and found one of those times the test passes. Only one of these test failures, from P6, consistently occurred in those five reruns. From examining the test logs, we believe that this test failure does represent a real fault, but we found that all the RTS techniques selected to run this test, so the regression fault would not have been missed.

From our inspection, we find then that all tests with failures that are not selected to run by any RTS technique were either failing before and are not affected by the changes, so the

(fail) result is already known without running, or were due to flaky tests, which do not reveal real faults due to code changes. As such, it is actually beneficial for RTS to *not* select to run these flaky tests found to have failed with RetestAll. From this perspective, the higher the percentage of failed tests not selected is actually *better*. We see that GIBstazi does not select such tests at a percentage of 32.29%, higher than the other techniques.

4.6 SHADOWING PROJECTS

In our experiments, we failed to replay many of the historical builds due to significant differences between the environment used for the builds back when they were originally built and the current environment on Travis CI. One particular reason is that Travis CI keeps updating the default build environment. For example, goldmansachs/gs-collections relies an older version of Maven to build while the default Maven version in the virtual environment provided by Travis CI has been updated through the years; as such this project is excluded from the evaluation. Another reason is the differences in external dependencies a project needs. For example, one of the projects we originally tried to replay is google/error-prone, which has a dependency on a SNAPSHOT version of JUnit. The JUnit developers can overwrite the SNAPSHOT version with the new changes, so the name of the dependency does not uniquely determine its content. One such change they made was to change the API of certain methods that the google/error-prone code relies on. Since the SNAPSHOT version in the central repository was overwritten with these new changes, our replay would end up using this latest SNAPSHOT version, which does not have the same content as what the historical build used, leading to compilation errors. As such, google/error-prone was not one of the projects we could use in our evaluation of historical builds.

To better observe how well RTS performs on the actual builds, we *shadow* the builds, replaying the current builds much closer to when they are actually triggered (as described in Section 3.5). One advantage of shadowing current builds over the historical replaying is that we evaluate the build in an environment close to when they are actually built (e.g., less of a chance of there being issues with out-of-date dependencies). Another advantage is that we can observe how well RTS could help the developers for the current state of their project, which may have had different characteristics from the project state in the historical builds we replayed.

We shadowed the projects for 20 days and obtained results from nine projects, resulting in a total of 217 jobs (but not all passed). Some of the projects we have results for shadowing, namely google/error-prone and linkedin/pinot, are projects for which we have no historical build results due to the issues with replaying we mentioned earlier.

Project	# Jobs	Original Time (m)	Shadowed Time (%)
alibaba/dubbo	4	18.17	56.39
aws/aws-sdk-java	8	8.85	114.00
apache/incubator-skywalking	15	11.76	72.40
javaparser/javaparser	6	16.20	52.88
languagetool-org/languagetool	93	15.97	53.90
iluwatar/java-design-patterns	2	25.92	83.52
google/guava	5	24.29	75.36
google/error-prone	4	15.73	70.49
linkedin/pinot	36	18.32	42.13
SUM/AVG	173	16.17	57.20

Table 4.5: Time savings from RTS on shadowed jobs

Table 4.5 shows a comparison between the original Travis CI jobs and our shadowed jobs with GIBstazi. In the table, we show for each project the number of those jobs where both the original job and our shadowed version of the job passed, as we want to make a fair comparison as we did in Section 4.3. As such, we analyze a total of 173 jobs. We show the total, end-to-end time (not just testing time), because this time is most relevant to the developers that directly observe it, for the original job as reported by Travis CI, and we show what percentage of that time did the shadowed version with GIBstazi take. We find that GIBstazi still provides time savings over original build times, with a weighted average of 57.20% of the original time.

We also inspected the original jobs that failed where we could parse the logs for test failures using TravisTorrent [7]. We parsed out 24 test failures from those original jobs, all from languagetool-org/languagetool, google/guava, and iluwatar/java-design-patterns. GIBstazi selected to run 17 of the failed tests (70.83%), and GIBstazi did not select to run six of the failed tests (25.00%). We could not tell for one of them if GIBstazi would have selected the test or not, due to a (flaky) test failure from an earlier module.

We inspected these 24 test failures the same way we inspected test failures from the historical replayed builds. For 15 test failures, we found the same test failed in the prior job, so the status remains the same even if GIBstazi did not select to run that test. We also confirmed that GIBstazi did run the failed test in the prior job, so the status would have been known. Of the remaining test failures, we confirmed four to be flaky from either GIBstazi running the test and passing, or from our replaying the original build and observing the test passing. Of the remaining four test failures (not including the unknown one), we believe they are all real test failures. GIBstazi selected to run three of those failed tests, so the developers would have noticed the failures.

However, GIBstazi did *not* select to run one remaining failed test. This test failure is from languagetool-org/languagetool, and further inspection showed that the change made was to a `.txt` file. Due to the regex we used, we ignored changes made to all `.txt` files, but for a project like languagetool-org/languagetool, their `.txt` files are an integral part of their code and tests. As such, there is a misconfiguration in GIBstazi because we applied general filtering that works for many but not all projects. The actual developers of languagetool-org/languagetool should apply project-specific filtering to ensure such test failures do not get missed. Note that Ekstazi, as it tracks only class files, would also have missed to select this failed test as well.

CHAPTER 5: LIMITATION

While GIB and GIBstazi showed positive results in terms of time savings, there is a limitation in these module-level RTS tools. By default, as mentioned in Section 2.1, Maven builds the modules sequentially and stops the project build execution if any module failed. However, when developers introduce multiple regression faults across modules, this "fail fast" feature of Maven may lead to different build results when using different RTS tools.

In Figure 5.1, we show four versions of an example project representing the evolution of the project and how different RTS tools and RetestAll give different build results at version 4. There are three modules (A, B, C) in this example project. The Maven build order of these modules is A, B, C because B and C depend on A (denoted with arrows). We highlight a module in green if there are tests selected by the RTS tool in the module and all these tests passed. We highlight a module in red if there are tests selected by the RTS tool in the module but some of these tests failed. We do not highlight a module if there is no test executed in the module (no test is selected by the RTS tool or the module is not built because earlier modules failed).

At version 1, developers made changes to all three modules and the changes did not introduce any fault to the project. Thus, regardless of the RTS tool the developers are using, all the tests that are selected to run in these modules passed. Version 1 of this project is in a stable state where no test failed. At version 2, developers made changes to all three modules again and the changes introduced a fault to each of the three modules. Tests in module A are run first and some of them failed. Maven stopped the build process and thus the builds of later modules (B and C) are not executed. At version 3, developers changed module A again to fix the fault found in version 2. Now, tests in module A passed but tests in module B failed and module C is again not built since Maven stopped the build process. As we can see, GIB, GIBstazi, Ekstazi, and RetestAll give the same build results for each of versions 1, 2, and 3.

At version 4, developers changed module B to fix the fault found in version 3 (though the fault was actually introduced in version 2). RetestAll simply selects all the tests in each module to run. Tests in module A and B passed but tests in module C failed. Ekstazi iterates through all the modules to determine what tests to select. No test in module A is selected because no change has been made in module A since the last execution of Ekstazi in module A. Some tests in module B are selected by Ekstazi and those tests passed. Some tests in module C are selected by Ekstazi but those tests failed. RetestAll and Ekstazi give 'failed' build results because the fault in module C introduced in version 2 is discovered by

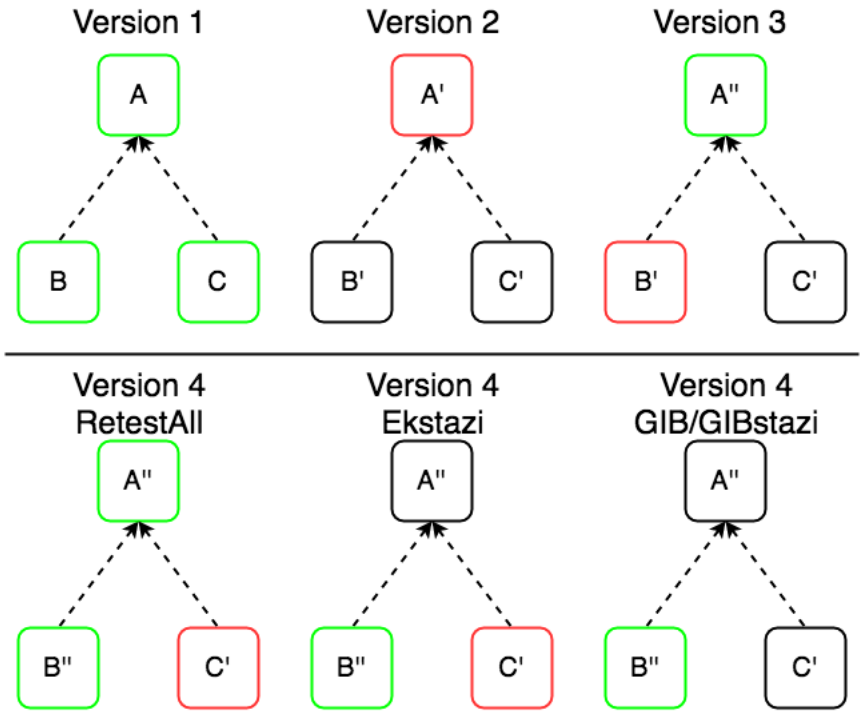

Figure 5.1: An example of different build results when using different RTS tools

running some tests in module C.

However, GIB and GIBstazi give 'passed' build results because they use Git to determine changed modules and then use Maven dependency information to determine affected modules. At version 4, module B is the only affected module because only module B was changed between version 3 and version 4 and no other module depends on module B. GIB then selects all tests in module B while GIBstazi applies Ekstazi in module B and selects some tests in module B. No test in module A or C is selected to run when using GIB or GIBstazi and developers get a 'passed' build result.

As we can see, due to the "fail fast" feature of Maven, tests in a module are executed only if the tests are selected by the RTS tool and all other modules built earlier passed. RetestAll and Ekstazi can progressively find faults introduced by changes when Maven skipped the build of later modules due to test failures in an earlier module because RetestAll and Ekstazi always try to process all the modules in each build. GIB and GIBstazi however, only process the modules affected by the most recent change. If an affected module failed, other affected modules are not tested right away, and they will only be tested when they are selected as affected modules in future versions. This limitation of GIB and GIBstazi may lead to different build results compared with RetestAll and Ekstazi.

CHAPTER 6: THREATS TO VALIDITY

Our conclusions about the benefits of RTS may not generalize beyond the projects used in our study. However, we use a diverse set of projects from GitHub, which is the most popular service for hosting open-source projects. We chose as many projects as we found to satisfy our filtering requirements. We focus on projects that take relatively long time to build and test, because we are concerned with projects that represent those where developers would want to use RTS to save regression testing time. We believe that our eventual choice of 22 projects used in our study are fairly representative of such projects.

While we replay historical builds from the projects in our study, the replays are not exactly the same as if they were run when the developers actually started the build in the past. One problem in particular would be if any external dependencies the developers relied on at the time of the build are now no longer available, or even worse changed the content and behavior while they still use the same name (e.g., the `SNAPSHOT` dependencies in Maven). When we replay such builds, they would fail although they would have passed back when the developers built. To alleviate this issue, we replay not just using the RTS techniques but also RetestAll, so we do not compare against the result of the build that potentially happened a long time ago with a drastically different setup. Furthermore, we do all the replaying and shadowing on Travis CI, the same environment that the developers build their code, as to more closely imitate how the developers build their code using CI services.

We enhance GIB and GIBstazi with filters to ignore files that should not affect test behaviors. It is possible that our filters may be ignoring too many files, i.e., some files that tests could potentially depend on. We constructed such filters based on our own experience and best knowledge of what types of files that we believe would generally not affect test behaviors.

CHAPTER 7: RELATED WORK

7.1 RTS

Regression test selection has been studied for several decades [13–15, 21, 25–27, 30–32]. Researchers have proposed various RTS techniques, selecting tests by tracking dependencies at different levels of granularities, ranging from precise control-flow edges [15, 27] to methods [32] to classes [14, 21]. Most of the early work on RTS focused on techniques that can select as few tests as possible from the regression test suite, but recent work has emphasized the need for RTS to provide time savings in end-to-end regression testing, not just in test numbers. For example, Gligoric et al. proposed Ekstazi [14], which tracks dependencies at the class level and selects test classes as opposed to test methods, leading to a larger number of tests actually being run compared to tracking at a finer granularity. However, the dependency analysis at the class level is very quick, eventually leading to better time savings despite running more tests. Companies such as Google and Microsoft rely on even coarser-grained dependency tracking, at the module level, due to the even quicker analysis time to determine what tests need to be run [11, 12, 28]. In our work, we compare module-level and class-level RTS in a CI environment.

Our work is most similar to recent work by Vasic et al. [29]. Vasic et al. created Ekstazi#, a tool that performs class-level RTS like Ekstazi for the .NET framework. Vasic et al. also evaluated running Ekstazi# on top of an incremental build system Concord, which inherently performs module-level RTS. As such, our hybrid RTS technique GIBstazi follows the ideas introduced by Ekstazi# and Concord. However, GIBstazi differs from their combination in that when changes are not source-code related, GIBstazi defaults back to GIB behavior, running *all* tests within affected modules, thereby being safer than just running Ekstazi (or Ekstazi#), which does not track those changes, running *no* tests within affected modules. Furthermore, in our work, we compare both module-level RTS and class-level RTS, as well as against GIBstazi, in a CI environment, where every build starts with a clean slate on a separate, non-dedicated machine. In contrast, Vasic et al. evaluate Ekstazi# using a dedicated machine with incremental builds.

7.2 CI

Continuous integration is widely used in industry and studied in research. Recent work has studied why developers use CI and the benefits they experience [17, 18, 33]. One main reason

for the rise in research on CI is the increased usage of CI, particularly with services such as Travis CI, which provides free CI service for open-source projects on GitHub. Furthermore, Travis CI exposes the build logs, allowing ease of access to information concerning building and testing results after developers push changes to their projects. TravisTorrent [7] provides a dataset of build logs from Travis CI, and it also provides the tooling for parsing the logs. We utilize Travis CI to replay historical builds of projects, and we also use TravisTorrent tooling to parse the logs for our analysis of failed tests.

There has been more work studying test failures on Travis CI (but not in combination with RTS as we do). Labuschagne et al. [20] studied how often regression testing on Travis CI reveals faults that developers fix. They query Travis CI for the results of builds and focused on patterns of builds that toggle 'passed' and 'failed' outcomes, indicating where a change caused an originally passing build to start to fail, followed by changes that lead to the build passing again. They found that 74% of the non-flaky failed builds were caused by a fault in the code under test, while the remaining are due to incorrect or obsolete tests. They also found flaky tests to affect 13% of the failed builds they studied. We also find flaky tests in our study, although we find that almost all the test failures from RetestAll in our study are flaky test failures, which are safe, and even preferable, to not select for RTS. One reason for the different percentage of flaky tests is that they study all builds (from `master` branch and other branches, as well as push builds and pull-request builds) whereas we study builds only from the `master` branch because many other builds cannot be replayed as the branches/commits are not available, especially for pull requests.

CHAPTER 8: CONCLUSIONS AND FUTURE WORK

Regression testing is widely practiced, particularly in continuous integration environments, but it can be quite costly. Regression test selection (RTS) is a way to reduce the cost of regression testing. Industry has adopted module-level RTS, while research has found class-level RTS to be very effective. In this work, we compare module-level RTS and class-level RTS in a continuous integration environment. We find that RTS techniques improve testing time over RetestAll, and the hybrid module-level and class-level RTS technique, which we call GIBstazi, offers a good trade-off. The savings from RTS on the end-to-end time for the entire build in a continuous integration service are not as high as shown in previous work that evaluated using a dedicated server. We also investigated test failures from RetestAll and found that the RTS techniques often do not select to run failed tests; however, our closer inspection revealed that most of these tests are flaky and not indicative of faults introduced by code changes. The results show that RTS offers benefits to developers not only to reduce machine time but also to avoid false alarms from flaky tests and wasting time debugging non-existent problems in the code changes.

In the future, we plan to improve GIBstazi to resolve the issue discussed in Chapter 5. GIBstazi should provide the option to forcefully run the modules that are selected but skipped in previous builds even if these modules are not affected by new changes. We also plan to fix the Surefire report counting issue discussed in Section 3.3. We also plan to evaluate more RTS techniques, such as STARTS [22], a static class-level RTS tool. In addition, we plan to replay more jobs multiple times to get more valid data and minimize flakiness, and we also plan to shadow more projects in order to promote the adoption of GIBstazi.

REFERENCES

[1] Bazel. `https://bazel.build/`.

[2] Ekstazi. `http://ekstazi.org/`.

[3] Maven. `http://maven.apache.org/`.

[4] Travis-CI. `https://travis-ci.org/`.

[5] Alali, A., Kagdi, H., and Maletic, J. I. What's a typical commit? A characterization of open source software repositories. In *ICPC* (2008).

[6] Bell, J., Legunsen, O., Hilton, M., Eloussi, L., Yung, T., and Marinov, D. DeFlaker: Automatically detecting flaky tests. In *ICSE* (2018).

[7] Beller, M., Gousios, G., and Zaidman, A. TravisTorrent: Synthesizing Travis CI and GitHub for full-stack research on continuous integration. In *MSR* (2017).

[8] Bird, C., Rigby, P. C., Barr, E. T., Hamilton, D. J., German, D. M., and Devanbu, P. The promises and perils of mining Git. In *MSR* (2009).

[9] Borges, H., Hora, A., and Valente, M. T. Predicting the popularity of GitHub repositories. In *PROMISE* (2016).

[10] Brindescu, C., Codoban, M., Shmarkatiuk, S., and Dig, D. How do centralized and distributed version control systems impact software changes? In *ICSE* (2014).

[11] Elbaum, S., Rothermel, G., and Penix, J. Techniques for improving regression testing in continuous integration development environments. In *FSE* (2014).

[12] Esfahani, H., Fietz, J., Ke, Q., Kolomiets, A., Lan, E., Mavrinac, E., Schulte, W., Sanches, N., and Kandula, S. CloudBuild: Microsoft's distributed and caching build service. In *ICSE* (2016).

[13] Gligoric, M., Eloussi, L., and Marinov, D. Ekstazi: Lightweight test selection. In *ICSE Demo* (2015).

[14] Gligoric, M., Eloussi, L., and Marinov, D. Practical regression test selection with dynamic file dependencies. In *ISSTA* (2015).

[15] Harrold, M. J., Jones, J. A., Li, T., Liang, D., Orso, A., Pennings, M., Sinha, S., Spoon, S. A., and Gujarathi, A. Regression test selection for Java software. In *OOPSLA* (2001).

[16] Herzig, K., Greiler, M., Czerwonka, J., and Murphy, B. The art of testing less without sacrificing quality. In *ICSE* (2015).

[17] Hilton, M., Nelson, N., Tunnell, T., Marinov, D., and Dig, D. Trade-offs in continuous integration: Assurance, security, and flexibility. In *ESEC/FSE* (2017).

[18] Hilton, M., Tunnell, T., Huang, K., Marinov, D., and Dig, D. Usage, costs, and benefits of continuous integration in open-source projects. In *ASE* (2016).

[19] Kosar, V. gitflow-incremental-builder. `https://github.com/vackosar/gitflow-incremental-builder`.

[20] Labuschagne, A., Inozemtseva, L., and Holmes, R. Measuring the cost of regression testing in practice: A study of Java projects using continuous integration. In *ESEC/FSE* (2017).

[21] Legunsen, O., Hariri, F., Shi, A., Lu, Y., Zhang, L., and Marinov, D. An extensive study of static regression test selection in modern software evolution. In *FSE* (2016).

[22] Legunsen, O., Shi, A., and Marinov, D. STARTS: STAtic Regression Test Selection. In *ASE Demo* (2017).

[23] Luo, Q., Hariri, F., Eloussi, L., and Marinov, D. An empirical analysis of flaky tests. In *FSE* (2014).

[24] Memon, A., Gao, Z., Nguyen, B., Dhanda, S., Nickell, E., Siemborski, R., and Micco, J. Taming Google-scale continuous testing. In *ICSE-SEIP* (2017).

[25] Orso, A., Shi, N., and Harrold, M. J. Scaling regression testing to large software systems. In *FSE* (2004).

[26] Rothermel, G., Elbaum, S., Malishevsky, A., Kallakuri, P., and Davia, B. The impact of test suite granularity on the cost-effectiveness of regression testing. In *ICSE* (2002).

[27] Rothermel, G., and Harrold, M. J. A safe, efficient regression test selection technique. *TOSEM* (1997).

[28] Shi, A., Thummalapenta, S., Lahiri, S. K., Bjorner, N., and Czerwonka, J. Optimizing test placement for module-level regression testing. In *ICSE* (2017).

[29] Vasic, M., Parvez, Z., Milicevic, A., and Gligoric, M. File-level vs. module-level regression test selection for .NET. In *ESEC/FSE* (2017).

[30] Xu, G., and Rountev, A. Regression test selection for AspectJ software. In *ICSE* (2007).

[31] Yoo, S., and Harman, M. Regression testing minimization, selection and prioritization: A survey. *STVR* (2012).

[32] Zhang, L., Kim, M., and Khurshid, S. FaultTracer: A change impact and regression fault analysis tool for evolving Java programs. In *FSE* (2012).

[33] Zhao, Y., Serebrenik, A., Zhou, Y., Filkov, V., and Vasilescu, B. The impact of continuous integration on other software development practices: A large-scale empirical study. In *ASE* (2017).