Understanding and Improving Regression Test Selection in Continuous Integration

August Shi, Peiyuan Zhao, Darko Marinov
Department of Computer Science
University of Illinois at Urbana-Champaign, USA
{awshi2,pzhao12,marinov}@illinois.edu

***Abstract*—Developers rely on regression testing in their continuous integration (CI) environment to find changes that introduce regression faults. While regression testing is widely practiced, it can be costly. Regression test selection (RTS) reduces the cost of regression testing by not running the tests that are unaffected by the changes. Industry has adopted module-level RTS for their CI environment, while researchers have proposed class-level RTS.**

In this paper, we compare module- and class-level RTS techniques in a cloud-based CI environment, Travis. We also develop and evaluate a hybrid RTS technique that combines aspects of the module- and class-level RTS techniques. We evaluate all the techniques on real Travis builds. We find that the RTS techniques do save testing time compared to running all tests (RetestAll), but the percentage of time for a full build using RTS (76.0%) is not as low as found in previous work, due to the extra overhead in a cloud-based CI environment. Moreover, we inspect test failures from RetestAll builds, and although we find that RTS techniques can miss to select failed tests, these test failures are almost all flaky test failures. As such, RTS techniques provide additional value in helping developers avoid wasting time debugging failures not related to the recent code changes. Overall, our results show that RTS can be beneficial for the developers in the CI environment, and RTS not only saves time but also avoids misleading developers by flaky test failures.

***Index Terms*—regression test selection, continuous integration, flaky tests**

I. Introduction

Developers rely on regression testing to quickly detect regression faults introduced by their code changes. Nowadays, regression testing is commonly performed in continuous integration (CI): after every push to the repository, a CI server, typically in the cloud, builds and tests the code [20]. While regression testing is important and widely-practiced, it has two major problems. First, it can be time-consuming due to running many tests after every change and having frequent changes, e.g., as reported by Google [25]. Also, running regression testing on CI servers in the cloud incurs a monetary cost for the cloud resources, e.g., as reported by Microsoft [19]. Second, regression testing suffers from *flaky tests* [7], [24], [36], which can pass or fail non-deterministically regardless of the code changes, e.g., as reported by Facebook [17]. With flaky tests, a developer cannot trust a new test failure to indicate a regression fault in the recent code changes that the developer should debug.

Regression test selection (RTS) can reduce the costs of regression testing. RTS runs only a subset of the regression test suite—the tests that are affected by the changes [34]. An RTS technique tracks the dependencies among the tests and code entities (e.g., modules, classes, or methods) and selects to run only the tests whose (direct or indirect) dependencies changed. RTS aims to run fewer tests, speeding up regression testing. Researchers have proposed many RTS techniques that perform selection at different granularity levels of dependencies [14], [15], [18], [23], [26], [27], [28], [32], [34], [38]. Researchers have also reported RTS to be effective in experiments, measured by the percentage of tests selected and testing time saved. Large companies have adopted RTS to speed up their regression testing [12], [13], [17], [19], [25]. For industry, important metrics are the overall build time and the quality of the test outcomes (i.e., if test failures reveal real faults).

While both industry and research use and study RTS techniques, there is a gap in the granularity level of RTS they use. A typical (object-oriented) software project is organized hierarchically into modules that contain classes that contain methods that contain statements; there can be also dependencies among projects. In *industry*, RTS has progressed from coarser- to finer-grain dependencies, from running all tests (*RetestAll*) to tracking dependencies among project *modules*[1]. Modern systems in industry [12], [13], [17], [19] commonly use *module*-level dependencies, track changes made to project modules, *select a subset of modules* that are affected by the changes, and then run *all tests within the selected modules* [13], [25]. In *research*, the progress has been from finer- to coarser-grain dependencies, from traditional work using statements [34] to using methods [38] to the most recent work reporting *class*-level dependencies to be more effective than finer-grain dependencies [15], [23], [37].

We aim to understand how module- and class-level RTS techniques compare in a real CI environment: should everyone adopt module-level RTS used by large companies, should they adopt class-level RTS proposed by researchers, or is the ideal trade-off in the middle? While module-level RTS has a very small overhead to analyze what modules are affected by the changes, the coarse-grained dependency tracking and test selection (all tests within affected modules) can select more tests than class-level RTS selects (only the affected test classes, not all tests, within affected modules). While class-level RTS can select fewer tests, it has two issues stemming from tracking dependencies (only) on classes: (1) it can potentially miss to

[1]We use the term "module" following the Maven build system for Java, but other regression testing, CI, or build systems use other terms, e.g., "target".

select affected tests, e.g., due to changes to non-source code files like configuration files, whereas module-level RTS finds such changes to affect entire modules and then selects all tests within such modules; (2) it requires extra analysis time to determine the affected test classes, compared to the time to determine only the affected modules, so the overall time for class-level RTS can be higher despite selecting fewer tests.

To evaluate module- and class-level RTS techniques in a CI environment, we would ideally compare techniques in general and not specific tools. Some metrics, such as the number of selected tests, are mostly determined by the technique, but a key metric that developers care about—the total build time—is determined by the tool. We thus compare specific tools and carefully analyze results to draw general conclusions about techniques. We start with GIB [2], a module-level RTS tool, and Ekstazi [15], a class-level RTS tool.

We also implement a new RTS tool, *GIBstazi*, that simply combines both module- and class-level RTS. GIBstazi first uses GIB to quickly select what modules are affected by the changes, and then only on those modules applies Ekstazi to select affected tests. If a change is in a non-source-code file that is not tracked by class-level RTS, GIBstazi defaults to GIB behavior and selects all the tests within the affected modules. Moreover, after our preliminary experiments show that GIB out-of-the-box would almost always selects all modules based on the changes, we make enhancements to better filter changes that tests should not be affected by. Our subsequent evaluation of GIB uses these enhancements as the default configuration. By combining GIB and Ekstazi, GIBstazi aims to select and run fewer tests than GIB, leading to faster testing, but not necessarily faster than Ekstazi. However, because GIBstazi defaults to GIB behavior due to non-source-code changes, GIBstazi can be safer (i.e., not miss to select some affected tests) than Ekstazi.

To evaluate real build times that developers would see in practice for the three RTS techniques, we utilize Travis CI [5], the most popular cloud-based CI service for open-source projects [20]. We evaluate the techniques on a diverse set of open-source Java projects already configured to use Travis; whenever the developer pushes some change, Travis triggers one (or more) build job(s). We replay the build jobs of each project by running all the tests (RetestAll) and using each of the three RTS techniques. For each technique run on each job, we collect three metrics relative to RetestAll: the percentage of tests selected, the percentage of time to run the selected tests, and the percentage of time overall to build the job. On the 22 projects and the 935 build jobs we replayed, we find that all three RTS techniques on average save time over RetestAll on Travis: GIB, Ekstazi, and GIBstazi take 79.7% 76.0%, and 77.4%, respectively, of the total RetestAll build time. These percentages for total build time in CI are higher than previously reported (60%–70%) for local, non-CI environments [15], [23], [37]. Moreover, these percentages are much higher than suggested by just the percentage of tests selected, e.g., 30.6% for Ekstazi.

To understand the effectiveness of RTS techniques with respect to test failures that happen during RetestAll, we collect the test outcomes (passes and failures) for each technique and systematically inspect failures. Some test failures during regression testing, namely those from flaky tests, are *undesirable* [7], [17], [24], [36]. Because RTS may miss test failures from RetestAll (in our experiments, it missed 39 failures), we analyzed whether RTS missed desirable or undesirable test failures. We find that almost all the test failures (38 out of 39) from RetestAll that any RTS technique missed are undesirable, flaky test failures.

While RTS techniques are *not* explicitly designed to avoid flaky test failures, we empirically find that RTS is highly beneficial for avoiding flaky test failures, a benefit of RTS not previously reported [14], [15], [18], [23], [26], [27], [28], [32], [34], [38]. We also apply the RTS techniques on 19 failed pull request jobs that are not flaky (confirmed through reruns) from five of our projects. We find that the RTS techniques do not miss any failed test in these 19 jobs, even though class-level RTS could miss to select affected tests due to non-source code changes.

In summary, the contributions of this paper are:

- **Hybrid RTS Technique:** We develop a simple, hybrid module- and class-level RTS technique.
- **Empirical Evaluation:** We empirically evaluate the module-level, class-level, and hybrid RTS techniques in CI; no prior work compared such RTS techniques in CI.
- **Failure Analysis:** We are the first to analyze test failures from RetestAll runs in the context of RTS. We find almost all failures are due to flaky tests, showing how RTS techniques by chance mitigate the flaky test problem.

From our results, we recommend developers to use the hybrid GIBstazi RTS technique, because it provides the best trade-off: it works faster than GIB and only somewhat slower than Ekstazi, but it is safer than Ekstazi. Moreover, our analysis of failed tests reveals that RTS techniques provide additional, although not directly targeted, benefits in helping developers avoid flaky test failures.

II. Continuous Integration (CI) Background

We describe how developers use CI systems to build and test their code, with an emphasis on CI systems in the cloud. We use Travis [5] as an example to help illustrate the concepts. Travis is widely used [21] and integrates well with projects hosted on GitHub, the most popular platform for open-source projects [6], [9], [10], [11]. We use Travis in our evaluation and introduce some Travis terminology here.

When a developer pushes a commit to a repository, the push triggers a *build* on the CI servers to compile and test the code for that commit. Developers can specify the exact commands to be run on the CI server; for Travis, these commands are in the `.travis.yml` file. Travis also allows developers to configure the build to run multiple *jobs*. Each job is configured with different environment variables or even different build commands; all jobs from one build run for the same commit. Each job is scheduled on its own *clean* virtual machine in the cloud. Hence, we focus our evaluation *per job*.

When a job starts on a remote CI server in the cloud, artifacts from prior jobs will not be on the machine. As such, CI needs to either recompile the project from scratch before

running any tests or rely on some way of saving compiled artifacts from prior jobs in a persistent datastore and copying them over for the new job. Most Travis setups recompile the code from scratch. A Travis job is divided into multiple phases that are executed sequentially. If a phase fails, the job fails early, and the later phases are not run. The `install` phase typically compiles the project code but does *not* run the tests. For example, the default command in the `install` phase for building a Maven project is `mvn install -DskipTests=true -Dmaven.javadoc.skip=true -B -V`, which purposely skips tests, but installs all the compiled artifacts. On Travis, it is possible to save certain artifacts across jobs, specified in the `cache` section of the `.travis.yml`. Such artifacts are copied onto the virtual machine before the job starts, and they are uploaded onto a separate, persistent server after the `before_cache` phase, which occurs near the end of the job.

After code has been properly compiled, CI can start running tests. In Travis, tests usually run in the `script` phase, which is executed after (but not immediately after) the `install` phase. The default command in the `script` phase for Maven projects is `mvn test -B`. Since the `script` phase occurs after the `install` phase that should compile all necessary artifacts, the `script` phase commands need not recompile code.

Finally, CI gives feedback to the developer about the status of each job after it completes. Travis assigns a status to each completed job: *pass* means all phases ran successfully, *fail* means the `script` phase ran unsuccessfully, and *error* means some other phase ran unsuccessfully (usually the `install` phase, suggesting compilation failed).

III. Regression Test Selection (RTS) Techniques

We describe the three RTS techniques that we evaluate. They track dependencies and perform selection at different granularity levels: module-level, class-level, and a hybrid module- and class-level. Prior work found RTS at the class-level to outperform finer granularity such as method-level [15], [23], [37], so we do not evaluate finer-grained RTS techniques.

A. Module-level RTS

Developers use modules to group related project parts, and module-level RTS detects changes at the level of modules. When module-level RTS detects some changed module(s), it finds all the *affected* modules by computing the transitive closure of the changed modules in the module dependency graph specified by the developers. Module-level RTS then runs all the tests within all affected modules.

We use Gitflow Incremental Builder (GIB) as a module-level RTS tool [2]. GIB is a Maven extension that can perform module-level RTS. GIB relies on Git to determine the code changes. Given two Git commits, GIB diffs the two commits to determine what files changed. GIB then maps the changed files back to the Maven modules to determine changed modules. GIB finds the affected modules through the module dependency graph obtained from parsing the `pom.xml` Maven build files provided by the developers. GIB then runs the provided Maven command (e.g., `mvn install` or `mvn test`) only on the affected modules. GIB was originally intended for incremental building, but as long as the Maven command includes some testing, GIB effectively performs module-level RTS by running all the tests within only affected modules. As module-level RTS is conceptually rather simple, we believe GIB is a representative tool.

Enhancing GIB In our preliminary experiments, we use GIB out-of-the-box on 423 commits from open-source projects from our later evaluation. We run GIB using the command `mvn validate`, which just checks the structure of the modules in the Maven project without compiling or running tests, to check how many modules GIB selects based on the changes. We find that in 65% of these commits GIB selects *all* the modules in the project. Overall, GIB selects over 70% of all modules in all the commits. As such, we add two key enhancements to default GIB to improve it for RTS. Our subsequent evaluation uses GIB with these enhancements as the default.

Our enhancements (1) configure GIB in a reasonable way for RTS and (2) extend GIB with a new feature. GIB uses Git to determine what files changed, but not all changed files affect Java test outcomes. For example, if the only change is to the top-level README file, GIB computes that a change to a file in the root of the project affects all modules, thereby running all tests, even though changing a README file is unlikely to actually affect a test outcome. To avoid such over-selection of tests, we configure GIB to exclude certain files, using the following regex based on file names we have seen commonly changing but are unlikely to affect test outcomes:

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

Our regex works well generally, but developers using GIB should tailor the regex further for their projects.

While our regex filters out many cases where file changes would lead to unnecessary test runs, our initial experiments found many other cases where changes to the Maven `pom.xml` files lead to a large number of test runs. In general, changes to `pom.xml` can affect tests, e.g., by changing a project's library dependencies [16]. However, we observed many changes to `pom.xml` files that did not change the dependencies, e.g., some changes simply update the project's own version number, which should not affect test outcome. We extended GIB to check if any dependency of a module within the project changed between runs. Our enhancement stores in a separate file, `classpathfile`, the names of all the dependencies for each module (but no version number if a dependency is a module in the current project). Before each run, if `classpathfile` exists, GIB compares the contents of the file with the dependencies of each module to see if any dependencies changed. If not, GIB ignores changes to `pom.xml`.

B. Class-level RTS

Class-level RTS tracks dependencies at the class level. First, it maps each test class to the classes that the test depends on. Then, if a class changes, class-level RTS selects all test classes that depend on the changed class. The dependencies of each test class can be computed dynamically [15] or statically [23].

We use Ekstazi [14], [15] as a class-level RTS tool. Ekstazi is a Maven plugin that performs dynamic class-level RTS. Ekstazi instruments the code under test to obtain which classes each test[2] depends on. Ekstazi also tracks checksum values for each `.class` file (compiled from a source Java file). After a project change, Ekstazi first waits for Maven to compile source files to `.class` files, then uses the stored checksums to determine which classes actually changed, and finally selects the tests that depend on the changed classes as per the stored dependency mapping. Both the mapping from tests to dependencies and the class checksums are stored within `.ekstazi` directories, one for each module.

C. Hybrid Module- and Class-level RTS

Module-level RTS can select many more tests than class-level RTS, because module-level RTS selects *all* tests within all affected modules, even if many such tests may not be affected by the changes. We propose a hybrid module- and class-level RTS that simply combines elements of both. It first uses module-level analysis to determine the affected modules, and then uses class-level analysis on the affected modules to select individual tests. However, if a change is to a non-source-code file (not specified in the exclude regex), e.g., a `.json` file that may be a test input, the hybrid technique defaults back to module-level RTS and selects all tests in the affected modules, being safer than class-level RTS (which does not track changes to non-source-code files).

We implement our hybrid technique in a tool called *GIBstazi*. GIBstazi builds upon GIB to determine affected modules, and for each such module, GIBstazi applies Ekstazi to select tests within the module. If any change is to non-source-code files, GIBstazi defaults back to GIB and selects all the tests within the affected modules. For each module, GIBstazi selects either (1) no tests (if the module is not affected), (2) all tests (if some non-source-code file changed), or (3) the same tests as Ekstazi. We expect GIBstazi to select fewer tests than GIB but more than Ekstazi; the time savings from GIBstazi should also be between GIB and Ekstazi. GIBstazi is a fork of GIB, publicly available on GitHub [3].

IV. Experimental Setup

We describe how we select the projects for our experiments and the commits for each project. We then describe how we configure to run the different RTS techniques for each project's commits on Travis. Finally, we describe how we collect the job results for our evaluation. The collected job logs and our results are publicly available [30].

A. Projects

Since the RTS tools we use are for the Maven build system [4], our evaluation requires Maven projects. In addition, given that GIB and GIBstazi operate at the module level, we need Maven projects that are multi-module. Moreover, we need these projects to build on Travis. We query GitHub to get the top 1000 popular Java projects ranked by stars, and then we filter to obtain only multi-module Maven projects. Finally, we filter for projects that use Travis, resulting in 105 projects.

TABLE I: Filtering of projects for our evaluation

Total starting Maven projects from GitHub	1000
Multi-module Maven projects on Travis	105
Projects whose build takes longer than 10 minutes	46
Projects with tests and replayable with RTS tools	22

Further, we want to evaluate on projects whose builds are sufficiently long-running such that a developer may want to use RTS in the first place. For each of the 105 projects, we query Travis for the latest 20 builds, average the build times, and select projects that took on average longer than 10 minutes to build, resulting in 46 projects. The build times reported by Travis represent the overall time the project takes to *build* and not just the times for *testing*. In particular, some of these projects only compile code on Travis and intentionally skip tests. Since we are evaluating RTS techniques, we want the projects that run at least *some* tests during the `script` phase on Travis. From the 46 projects, we keep the projects that run tests on Travis, and we further keep only the projects that can run with all three RTS tools (e.g., GIB requires Java 8), resulting finally in 22 projects. Some of the projects have testing time shorter than 10 minutes, and the average testing time per job is 9.9 minutes (Section V-A). Table I summarizes the filtering.

We collect revisions for each of the 22 projects for rerunning on Travis. We collect these revisions from actual prior Travis builds. In contrast, recent work on RTS [15], [23] selected the revisions as *sequential commits* from the master branch in the GitHub repository of each project. However a single Travis build corresponds to a push from the developer, and the code changes between two pushes can correspond to several commits in the repository.

For each project, we collect from Travis 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 such that replaying these historical commits later on gives the same code changes between each build as observed by the developers when using Travis for these builds.

B. Replaying with RTS

We replay the commits collected for each project on Travis for all RTS techniques, including RetestAll. For each technique, we create a new GitHub account and fork the projects into the account, and then for each commit of a project, do the following four steps: (1) checkout the commit (specifically with "`git checkout $sha .`" using '`.`' to not create a detached branch); (2) modify the `pom.xml` and `.travis.yml` files to use a specific RTS technique on the project when run on Travis; (3) modify the `pom.xml` and `.travis.yml` files further for our experimental purposes, to count tests run and measure time for running, with these modifications being the same for all techniques, including RetestAll; and (4) recommit the files after the modifications as a new, fresh commit and push it to our forked repository on GitHub, triggering the build, and thus

[2]By "test" we mean "test class". Ekstazi selects test classes that each can have several test methods. We count tests at the level of test classes as well.

one or more jobs, on Travis. (The very first commit that we recommit for each of the three RTS techniques selects all tests as in RetestAll, but the later commits use RTS.) We describe next the specific modifications for each RTS technique for step (2) and the general modifications for evaluation purposes for step (3). We aim for smallest necessary modifications to minimize risk of affecting the build process in each project.

1) GIB: We modify the project's top-level `pom.xml` file to include the GIB 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 Travis environment variable `TRAVIS_COMMIT_RANGE` provides these two commit SHAs.

We modify the `.travis.yml` file's `cache` section to save the `classpathfile` generated (Section III-A). The cache is needed to share data between jobs because Travis runs each job on a fresh virtual machine. We also configure `.travis.yml` to *not* use GIB during the `install` phase, as the entire project must build from scratch, and using GIB in this phase could prevent certain modules from being compiled. We disable GIB in the `before_install` phase (which occurs right before the `install` phase) and then enable it in the `before_script` phase (which occurs right before the `script` phase).

2) Ekstazi: We modify the project's top-level `pom.xml` file to include the Ekstazi Maven plugin; we use version `4.6.3` in our evaluation. We modify the `.travis.yml`'s `cache` section to save in between jobs one combined `.ekstazi` directory with metadata for all modules. We further add in the `before_script` phase the commands to copy the cached `.ekstazi` directories to each module in the project for the `script` phase to use for testing, and we add in the `before_cache` phase the commands to combine the updated `.ekstazi` directories after the tests finish. These `.ekstazi` directories can be much bigger than the `classpathfile` cached by GIB, and caching these directories is a necessary extra overhead to use Ekstazi in a cloud-based CI environment.

3) GIBstazi: We modify the `pom.xml` and `.travis.yml` files the same way as necessary for both GIB and Ekstazi individually, i.e., configuring to add the GIBstazi extension, and configuring `.travis.yml` to cache between jobs both the `classpathfile` and `.ekstazi` directories while also copying them appropriately.

4) Modifications for all techniques for experiments: For our evaluation, we need extra modifications to report tests selected and time taken. At the end of the `script` phase, we add commands to report how many tests are run by counting the number of Surefire report files generated that each represent a test run. We also add commands in the `script` phase to report how much time the `script` phase takes to run. Timing the `script` phase, where testing is meant to be performed, we can simulate running RTS "locally", without including the times for compiling code from scratch or downloading dependencies; we refer to the time measured in this phase as *test time*. We next remove from `.travis.yml` the entire `notifications` phase, which is used to notify developers of the job status; we do not want to spuriously notify developers concerning our replaying of their jobs. Removing this phase does not disrupt the compile and testing process in the previous `install` and `script` phases, and because we remove this phase for all techniques, including RetestAll, our timing comparison is consistent as well. Finally, we modify `.travis.yml` to not run any jobs with Java versions below Java 8, because GIB requires Java 8.

TABLE II: Basic statistics about projects used in evaluation, including distribution of pass/fail/error statuses for RetestAll

ID	Project	# Jobs	Pass	Fail	Error
P1	SonarSource/sonarqube	38	19	19	0
P2	elasticjob/elastic-job-lite	19	19	0	0
P3	apache/rocketmq	19	0	19	0
P4	alibaba/dubbo	18	18	0	0
P5	aws/aws-sdk-java	19	18	1	0
P6	brianfrankcooper/YCSB	19	18	1	0
P7	apache/incubator-skywalking	19	19	0	0
P8	antlr/antlr4	170	169	1	0
P9	vavr-io/vavr	5	5	0	0
P10	Graylog2/graylog2-server	1	1	0	0
P11	javaparser/javaparser	19	19	0	0
P12	languagetool-org/languagetool	18	17	1	0
P13	druid-io/druid	83	69	14	0
P14	killbill/killbill	47	0	47	0
P15	apache/storm	84	76	8	0
P16	iluwatar/java-design-patterns	19	14	5	0
P18	google/guava	35	35	0	0
P17	javaee-samples/javaee7-samples	335	334	1	0
P19	prestodb/presto	172	171	1	0
P20	apache/incubator-pulsar	13	7	4	2
P21	apache/flink	211	209	2	0
P22	Tencent/angel	14	7	7	0
	SUM	1377	1244	131	2

C. Collecting Job Logs

Replaying each commit starts jobs on Travis. After each job finishes, we download its log from Travis for analysis. We further consider only the jobs where we can successfully parse from the logs the number of tests run and the test time in the `script` phase. Jobs may not finish properly for several reasons, such as compilation errors (so tests are not even run) or strict timeouts maintained by Travis. Moreover, we consider only the jobs after the first commit for each project, because for the first commit all RTS techniques select all tests (there is no change yet), and we want to measure the effectiveness of RTS in the steady state, after changes have happened. Finally, we do not analyze any project where any of the RTS tools we use consistently crashes for all the jobs due to internal tool errors. In total we collect 1377 jobs across 22 projects.

D. Statistics of Jobs

Table II shows the distribution of the jobs that we collected across the 22 projects from our evaluation. We label each project with an ID that we use later and show the project's slug from GitHub. We also show the number of jobs we analyze for each project, classified as *pass*, *fail*, or *error* based on the job status reported by Travis for RetestAll. The overall number of jobs with status *pass*, *fail*, and *error* are 1244 (90.3%), 131 (9.5%), and 2 (0.2%), respectively. For the two jobs with the *error* status, we find it due to an unsuccessful phase that occurs *after* the `script` phase, i.e., after tests have run, so even in such cases, we can still collect information about the tests selected to run and the time for testing.

Recall that jobs having status *fail* in Travis does not necessarily mean that tests failed but that the `script` phase failed, which may not be due to test failures. For example, project P3 has all of its jobs with status *fail*, but we find that the reason is due to the `script` phase including a step that tries to deploy artifacts to another server, which we cannot access. Tests pass before this step, but because the deploy step is in the `script` phase, Travis marks the entire job as *fail*.

V. Results

We aim to answer the following two research questions:

RQ1: How do different RTS techniques compare in terms of tests selected, test time, and total build time in CI?

RQ2: How well does RTS select failing tests in CI?

A. RQ1: Tests Selected, Test Time, and Total Time

We first evaluate RTS techniques for *all* Travis jobs in our experiments, regardless of the job status. For each job, we compute the percentage of tests selected, test time, and total time of each RTS technique relative to RetestAll. We also compute the arithmetic mean of these percentages for all jobs in each project, and finally we compute the overall arithmetic mean of these averages per project. Overall, GIB, Ekstazi, and GIBstazi, respectively, select 59.1%, 35.2%, and 42.8% of the tests, take 86.6%, 65.5%, and 59.4% of the test time, and take 77.3%, 77.9%, and 72.2% of the total time.

Surprisingly, GIBstazi appears to be the fastest technique, unlike our initial expectations. However, we find many jobs passing for some technique with the corresponding jobs failing for RetestAll. As a result, some jobs even exceed 100% as the percentage of tests run by the RTS techniques relative to RetestAll, appearing as if RTS runs more tests than available! The reason for this anomaly is that test failures occur in the middle of job execution. (Many of these failures are flaky tests, as we discuss in Section V-B.) By default, when a test fails in a multi-module Maven project, Maven stops early, skipping all modules that come after the module with the failed test(s). As such, the remaining tests that should have been run are not actually run. Our tooling counts the number of tests that are actually run, so it ends up not counting all the tests the technique would have run had there been no test failure. While these numbers reflect what a developer would actually observe on Travis, they do not allow us to properly answer RQ1.

To provide a fairer comparison of the RTS techniques and RetestAll, we focus on only the jobs where RetestAll and all three RTS techniques have status *pass*, i.e., jobs where all tests that should be run are actually run. Table III shows the results for these 935 passing jobs, with an average of 51.9 jobs per project for 18 projects. We do not show the four projects that have no jobs where RetestAll and all three RTS techniques pass. The columns under "RetestAll" show the number of tests, the test time, and the total job time, all averaged across all jobs for each project, and then across the projects in the final row. The columns under "GIB", "Ekstazi", and "GIBstazi" show the average percentage of each metric relative to RetestAll for each respective RTS technique. The "AVG" row is the arithmetic mean of the values in each column.

We see from the final row that Ekstazi now outperforms both GIB and GIBstazi in terms of selecting the fewest tests and having the shortest test time and total time. GIBstazi, on the other hand, outperforms GIB in terms of all three metrics. Performing a series of Wilcoxon paired signed-rank tests for the tests selected, test time, and total time among all pairs of the three techniques, we find statistically significant differences ($p < 0.01$) for the percentage of tests selected and test time, but no such differences for the total time.

Overall, the trend between GIB, Ekstazi, and GIBstazi in terms of tests selected, test time, and total time now matches our initial expectations. However, there are jobs where the trend does not hold, so we inspect them in more detail.

1) Ekstazi Selects More Tests: We find 64 jobs distributed across eight projects where Ekstazi runs more tests than GIB or GIBstazi. We sample a job from each of these projects, as it is likely a characteristic of the project that leads to Ekstazi running more tests. We examine the job logs, the diffs between the job's commits, and the job configurations. Overall, we find four different causes.

Non-Default Runners. In P6, P15, and P16, we find jobs where GIB skips modules where Ekstazi runs tests. These projects have tests that do not use the default JUnit4 runner: TestNG, JUnit `Enclosed` runner, or JUnit Jupiter (new in JUnit5). Ekstazi incorrectly runs all tests using these runners, regardless of changes[3]. GIB and GIBstazi (correctly) find that a module is unaffected and do not run any test.

Non-Deterministic Compilation. In P4 and P17, we find that compiling even the same commit twice in a row results in different compiled `.class` files. P4 uses `cobertura`, which creates instrumented classes on which Ekstazi finds test dependencies; `cobertura`'s instrumentation is non-deterministic and does not always create the same final `.class` file for the same source file. P17 automatically generates some source files as part of the build process, but the generation is non-deterministic. Specifically, the order of the methods in the generated source files can differ between runs, which in turn results in different compiled `.class` files; Ekstazi relies on the `.class` files to not change if the developer makes no changes to source code, arguing that comparing `.class` files is more robust than comparing source files [14]. In this scenario, Ekstazi finds spurious changes and runs too many tests.

Incompatible with GIB. In P8 and P13, we notice that the job configuration for the jobs where Ekstazi runs more tests is set to navigate into a specific module to run only its tests (effectively `cd module; mvn test`). GIB assumes the root of the project starts from the current module and is unaware that the current module is part of a larger Maven project. Thus, GIB does not determine that the current module is affected by changes from the other modules, and it skips the current module altogether. Therefore, GIB is running *too few* tests in these cases. Developers using GIB (and GIBstazi) need more in-depth changes to their specific job configurations for GIB to work correctly in these cases; specifically, they should *not* navigate into a module and instead run from the root using the Maven's `-pl` option to specify the module.

[3]We confirmed via private communication with the developers of Ekstazi.

TABLE III: Tests selected and time savings from using RTS across *only passed* jobs

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	19	2.0	3.9	5.1	100.0	102.8	100.9	100.0	101.1	97.0	100.0	99.2	97.2
P2	19	158.4	1.4	6.5	24.4	39.1	78.3	7.1	58.9	97.8	24.4	40.4	70.3
P4	18	161.9	5.6	8.7	57.3	61.4	74.4	11.6	94.6	121.7	29.1	55.7	80.9
P5	15	179.0	7.4	9.3	100.0	103.6	103.4	16.6	49.8	60.6	100.0	109.7	109.7
P6	18	31.1	7.0	10.7	61.5	61.5	69.7	44.6	25.8	62.0	43.6	56.1	74.4
P7	19	98.7	4.1	10.8	17.5	29.8	80.4	2.5	37.5	56.7	13.7	31.5	79.1
P8	169	12.9	10.0	11.8	10.5	16.0	33.7	21.7	35.1	50.1	10.4	16.9	35.5
P9	5	140.0	10.9	11.8	100.0	86.9	101.4	38.7	71.0	75.7	59.0	75.6	97.2
P10	1	177.0	7.3	12.9	100.0	109.9	115.3	0.0	36.9	70.0	100.0	154.4	131.7
P11	19	177.0	7.0	14.9	91.1	94.1	83.1	24.1	59.2	50.9	26.5	58.9	46.0
P12	11	400.1	12.7	16.2	7.7	8.7	22.5	0.8	4.6	21.6	7.7	13.0	29.6
P13	38	198.3	12.8	20.8	33.5	48.4	72.1	25.1	49.8	73.0	7.5	29.5	67.6
P15	47	53.2	5.1	14.9	47.4	61.8	91.6	5.8	69.4	93.9	26.0	63.7	91.0
P16	12	319.2	6.7	22.1	25.9	31.9	40.9	55.8	87.6	94.0	17.5	28.5	41.5
P17	330	8.9	3.7	22.2	32.1	45.7	102.4	20.8	41.7	93.4	31.8	53.7	102.7
P18	35	494.1	16.8	20.2	100.0	97.5	98.1	55.8	62.5	83.3	56.1	61.6	69.2
P19	24	12.4	25.1	29.5	100.0	98.6	99.3	100.0	100.9	101.4	100.0	99.3	99.9
P21	136	366.5	30.7	32.1	49.1	64.9	66.9	19.5	62.7	65.2	40.0	68.1	70.2
AVG	51.9	166.1	9.9	15.6	58.8	64.6	79.7	30.6	58.3	76.0	44.1	62.0	77.4

Job Timeout. In P21, for a job where Ekstazi runs more tests, we find that the changes should not actually affect the module where Ekstazi runs tests, so Ekstazi runs *too many* tests. We find that the immediately prior job for Ekstazi times out, so the cache for Ekstazi dependencies is not updated for the subsequent job. Thus, Ekstazi compares the next commit with the commit *two* (rather than one) before it, finding more changes than GIB and GIBstazi find by comparing the next commit with the one before it. Essentially Ekstazi finishes running the tests from the prior job in the subsequent job (which does not time out). This example demonstrates how much Ekstazi depends on completing prior runs. Timeouts can occur more often on CI machines in the cloud, out of the developers' control, so this example also demonstrates issues with using Ekstazi in such a CI environment.

Trying to better understand differences between the *techniques* rather than differences due to tool engineering, we further filter out jobs where Ekstazi runs more tests than the other techniques. We obtain the same trends between the three techniques, with Ekstazi seeming even better. Overall, for GIB, Ekstazi, and GIBstazi, respectively, the average percentages of tests selected are 62.6%, 25.9%, and 47.4%; test times are 68.1%, 55.2%, and 66.7%; and total times are 82.1%, 74.8%, and 80.2% (not shown in tables due to space limits).

2) Ekstazi Runs Slower: Even when we consider only the jobs where Ekstazi runs no more tests than the other techniques, we still find jobs where Ekstazi test time is longer. Overall, for GIB and Ekstazi, respectively, the average percentages of tests selected are 62.6% and 25.9%, and test times are 68.1% and 55.2%; the difference is much higher in tests selected than in test time. We examine several of these jobs and find two reasons why Ekstazi runs slower.

Overhead of Ekstazi Instrumentation. Especially noticeable in jobs where Ekstazi runs no more tests than GIB, Ekstazi test time is longer primarily due to the runtime overhead of extra instrumentation Ekstazi needs to track dependencies; GIB requires no dynamic analysis. Note that in such cases GIBstazi also runs roughly the same as Ekstazi, as GIBstazi also relies on the same instrumentation.

Overhead of Ekstazi Requiring Compilation. We find jobs where Ekstazi spends a lot of time to determine that it need not run tests in some modules, while GIB and GIBstazi quickly determine to run very few modules. For example, in P4, we find a job where Ekstazi and GIBstazi run no tests. GIBstazi determines this rather fast, as it first selects very few modules, and those modules have no tests to run. However, Ekstazi has to analyze each and every module to determine that no tests should be run in it. While this analysis is generally rather fast, e.g., 2–4 seconds per module, P4 is a project with over 60 modules, so the time adds up. Furthermore, P4 is configured to run other plugins in each module, such as `cobertura` instrumentation, adding even more time per module. GIBstazi's skipping of all the unaffected modules leads to substantial speedup against Ekstazi.

3) Non-Source-Code Changes: In most cases, GIBstazi runs at least as many tests as Ekstazi because GIBstazi defaults to GIB behavior when there are non-source-code changes. To estimate the potential impact of such changes on RTS, we measure how much tests depend on non-source-code files. For each project, we first use `fabricate` [1], a tool that traces what files are accessed when executing a command (in our case, running `mvn test`), on the latest commit of the project. We record the non-source-code files used by the tests. We find many file names with extensions such as `.json` or `.properties`, and many files under `test/resources`, suggesting these files are used as inputs or configuration for tests. We then measure how many commits in the project made changes to any of these files. While one project had no commit that changed any of the dependencies found using `fabricate`, the other projects had on average 7.5% commits with such a change. Thus, Ekstazi has a relatively high risk to miss selecting a test affected by non-source-code changes.

B. RQ2: Test Failure Analysis

RTS aims to select only tests affected by changes. A key question is whether RTS misses to select some tests that fail due to the changes. If RTS misses such a *real* test failure, the

TABLE IV: Percentage of test failures selected or not selected by RTS

Project	# Failed Tests	GIB %			Ekstazi %			GIBstazi %		
		Selected	Not Selected	Unknown	Selected	Not Selected	Unknown	Selected	Not Selected	Unknown
apache/flink	2	100.0	0.0	0.0	50.0	50.0	0.0	50.0	50.0	0.0
apache/incubator-pulsar	13	53.9	46.2	0.0	84.6	0.0	15.4	38.5	46.2	15.4
apache/storm	7	71.4	28.6	0.0	57.1	42.9	0.0	71.4	28.6	0.0
aws/aws-sdk-java	1	100.0	0.0	0.0	0.0	100.0	0.0	100.0	0.0	0.0
brianfrankcooper/YCSB	2	50.0	50.0	0.0	50.0	50.0	0.0	50.0	50.0	0.0
druid-io/druid	10	70.0	30.0	0.0	20.0	70.0	10.0	20.0	70.0	10.0
iluwatar/java-design-patterns	5	0.0	100.0	0.0	20.0	60.0	20.0	0.0	100.0	0.0
javaee-samples/javaee7-samples	7	0.0	100.0	0.0	0.0	14.3	85.7	0.0	100.0	0.0
killbill/killbill	47	100.0	0.0	0.0	100.0	0.0	0.0	100.0	0.0	0.0
languagetool-org/languagetool	1	0.0	100.0	0.0	0.0	100.0	0.0	0.0	100.0	0.0
prestodb/presto	1	0.0	100.0	0.0	0.0	100.0	0.0	0.0	100.0	0.0
SUM/AVG	96	72.9	27.1	0.0	69.8	19.8	10.4	64.6	32.3	3.1

developer could miss a regression fault, defeating the purpose of regression testing. However, some failures are due to flaky tests and undesirable, so it would be beneficial to miss such failures. RTS is *not* explicitly designed to avoid flaky tests, but it can miss them by chance. In our study, we find several test failures in RetestAll, so we check whether the RTS techniques select those tests, and whether those failures are desirable.

Table IV shows the number of test failures from RetestAll and the breakdown of percentage of those tests selected by the RTS techniques. We obtain the test failures by parsing the Travis logs using tools from TravisTorrent [8]. Unfortunately, the parsing cannot successfully identify test failures for all failed jobs due to project-specific "noise" in the logs (e.g., not printing test progress in the default format). Also, just the Travis status of the job itself does not necessarily indicate there are test failures, as other Travis phases could mark the job as *fail* for other issues, so such jobs have no failed tests in the logs. In total, we find 11 projects with test failures we can parse out, leading to a total of 96 test failures. We categorize those test failures for each RTS technique into three categories: selected, not selected, or unknown. We use unknown if we cannot tell from the logs whether the test would have been selected, which happens if another test in a module before the relevant test fails in the RTS run, leading to an early failure that skips running the tests in later modules.

We inspect *all* the test failures from RetestAll. For 29 test failures, we find that the job with the test failure was immediately preceded by a job that had the same test failure [22]. When RTS does not select a test, the outcome of that not selected test is *not* necessarily pass but the same outcome from the prior run (may be fail), i.e., the failed outcome of the test is known from before. For example, if the developer makes no change related to fixing a test failure, there is no need to run the unaffected test again for the same known failure. An RTS tool can simply copy the prior outcome of the tests from the prior job if the tests are not run, allowing the developer to consistently see the test failures that have not yet been fixed.

We examine in detail the remaining 67 test failures where the test passed in the prior job. We find only *one* real failure and all others due to *flaky tests* that can pass or fail non-deterministically even on the same code [17], [24], [36]. Most flaky test failures are not related to code changes and are undesirable as they do not reveal real faults due to code changes [7]. We confirm that these tests are flaky in two ways. For some tests, we find that one of the RTS techniques did run the test and yet the test passed. For the remaining tests, we rerun the same job for RetestAll up to six times and check that the test passes in any rerun. In summary, all test failures are flaky except for one, from brianfrankcooper/YCSB, which consistently fails in those reruns. Examining the test logs and code changes, we determine this one test failure to represent a real fault, but we also find that *all* the RTS techniques selects this test, so none of them misses this regression fault.

The failed jobs we have used in evaluation so far are from pushes to the master branch, so their failures are expected to be due to flaky tests if the developers follow good CI practices and only merge in pull requests with passing jobs. However, failures from pull request jobs may more likely be real failures from attempting to merge. We apply RTS techniques on failed pull requests to examine whether the techniques miss real test failures. We first obtain recent (up to 10) failed pull requests from our evaluation projects. We then rerun their jobs six times in the RetestAll configuration to confirm that the failed tests fail consistently, resulting in 37 failed jobs. We then replay these failed jobs with Ekstazi, resulting in 19 successful replays across five projects; Ekstazi fails to run in 18 jobs due to the JVM setting. We find that Ekstazi does not miss any real test failures in these 19 pull request jobs, and by extension both GIB and GIBstazi also would not miss any test failures.

In sum, our inspection shows that all tests that failed in RetestAll for pushes to the master branch but are not selected by any RTS technique are either failing from before (so the test outcome is already known) or due to flaky tests (so do not reveal real regressions). As such, it is actually beneficial for RTS to *not* select these failed tests from RetestAll: the higher the percentage of failed tests *not* selected, the *better*. GIBstazi does not select such tests at a higher percentage than the other two techniques. Even if all the test failures categorized as unknown for Ekstazi are actually not selected, its percentage of not selected tests would be 30.2%, still lower than the not selected percentage for GIBstazi, 32.3%. If we consider pull request jobs where we confirm that test failures are not flaky, the RTS techniques do not fail to select any failed test.

C. Shadowing Projects

In our experiments with replaying, we are unable to successfully replay many commits due to significant differences between the environment used for the jobs back when they

TABLE V: Time savings from RTS on shadowed jobs

Project	# Jobs	Original Time (m)	Shadowed Time (%)
aws/aws-sdk-java	8	8.9	114.0
apache/incubator-skywalking	15	11.8	72.4
google/error-prone	4	15.7	70.5
languagetool-org/languagetool	93	16.0	53.9
javaparser/javaparser	6	16.2	52.9
alibaba/dubbo	4	18.2	56.4
linkedin/pinot	36	18.3	42.1
google/guava	5	24.3	75.4
iluwatar/java-design-patterns	2	25.9	83.5
SUM/AVG	173	17.3	69.0

were originally triggered and the current environment, e.g., the differences in external dependencies a project needs. For example, one of the projects we failed to replay is google/error-prone, which has a dependency on a `SNAPSHOT` version of JUnit. The JUnit developers can overwrite this version with new changes, so the dependency name does not uniquely determine its content. One such change was to the API of certain methods that google/error-prone in an earlier commit relies on. Since the `SNAPSHOT` version in the central repository was updated, our replay uses this latest version, leading to compilation errors. As such, we do not use google/error-prone in our evaluation on historical commits.

To gain better understanding of RTS for RQ1 and RQ2, instead of replaying historical commits, we *shadow* the Travis builds from 40 projects (including some whose RetestAll jobs do not compile in our earlier experiments), similar to Bell et al. [7]: we set up automatic tracking of projects such that when their developers trigger Travis builds on the master branch, our setup replays those builds close in time to the triggered builds. One advantage of shadowing current builds is evaluating in an environment similar to when the builds are actually built (e.g., less likely to have out-of-date dependencies). Another advantage is that we can observe how well RTS helps the project for its current state, which may have different characteristics from the project state in the jobs we replayed before. Finally, shadowing allows us to better evaluate the feasibility of utilizing RTS in a realistic cloud-based CI environment as it stands currently.

To enable shadowing, we first fork each project into a new account. We then set up a cron job to query Travis once an hour for each project to check if some new builds occurred since the last time the cron job was run. If there is a new build, the cron job pulls the commits corresponding to the new build(s) into our shadowing fork and replays them with RTS. We perform these runs only for GIBstazi, because we find it to strike a good balance among all three RTS techniques, saving more time than GIB, being designed to be safer than Ekstazi, and having avoided the most flaky test failures. To limit our usage of Travis, we shadow for 20 days, obtaining results from 217 jobs of nine projects. We obtain shadowing results for two projects (google/error-prone and linkedin/pinot) for which we have no results from before due to the issues with replaying.

RQ1. Table V shows the total time for the original job and the percentage of that time taken by the shadowed version with GIBstazi. The table only shows these numbers for the jobs where both the original job and our shadowed job pass. GIBstazi runs 69.0% of the original time, similar to before.

RQ2. We inspect all cases where the original jobs fail and TravisTorrent [8] could parse out test failures from the logs. We obtain 24 test failures from three projects: languagetool-org/languagetool, google/guava, and iluwatar/java-design-patterns. GIBstazi selects to run 17 of the failed tests (70.8%) and does not select to run six of the failed tests (25.0%). We cannot tell for one test if GIBstazi selected it, due to a (flaky!) test failure from an earlier module.

For 15 test failures, we find that GIBstazi runs the failed test in the prior job, where it also failed, so the test failure would have been known. Of the remaining nine test failures, we confirm four to be flaky either from GIBstazi running the test and passing, or from our repeating the original job and observing the test passing. Excluding the one test failure that is skipped, we believe the remaining four test failures are all real test failures. GIBstazi selects to run three of those failed tests, so the developers would have noticed the failures. However, GIBstazi did *not* select to run one failed test. This test failure is from languagetool-org/languagetool, and further inspection shows that the change made was to a `.txt` file. The regex we use for GIBstazi configuration ignores changes to `.txt` files, but for a project like languagetool-org/languagetool, the `.txt` files are an integral part of tests. Thus, we made a configuration error [33] in GIBstazi, using generic filtering that applies to most projects. The developers of languagetool-org/languagetool should apply project-specific filtering to ensure such test failures are not missed. Note that Ekstazi would also have missed to select this failed test.

VI. Threats to Validity

Our results may not generalize beyond projects used in our study. We use a diverse set of projects from GitHub, the most popular service for hosting open-source projects. We choose as many projects that could satisfy our filtering requirements, which includes choosing only Java and Maven projects due to tool constraints. We focus on projects that take a relatively long time to build and test, which are projects where developers would want to use RTS to save regression testing time. We believe that the 22 projects used in our study are fairly representative of such projects.

The historical replays we perform are not exactly the same as if they had been run when the developers triggered the build. A particular problem are any external dependencies the developers used at the time of the build but are now no longer available or, even worse, changed the content and behavior while they still having the same name (e.g., `SNAPSHOT` dependencies). Replayed jobs from such builds can fail although they would have passed when the developers built. To alleviate this issue, we replay using not just the RTS techniques but also RetestAll, so we do not compare RTS against the RetestAll job that happened potentially a while ago with a drastically different setup. Moreover, we replay on Travis, the same environment the developers use, to more closely imitate how the developers build their code using CI. Finally, we use live shadowing and not just historical replays.

VII. Related Work

Regression Test Selection. RTS has been studied for several decades [34]. Researchers have proposed various different RTS techniques, selecting tests by tracking dependencies at different levels of granularities, ranging from precise control-flow edges [18], [28] to methods [38] to classes [15], [23], [37]. Recent work has emphasized the need for RTS to provide time savings in end-to-end regression testing. For example, Gligoric et al. proposed Ekstazi [15], which tracks dependencies at the class level and selects test classes as opposed to test methods, leading to a larger number of tests run compared to tracking at a finer granularity. However, the analysis for RTS at the class level is very quick, eventually leading to better time savings. Zhang proposed HyRTS [37] that tracks dependencies at both class and method levels. Zhang found that HyRTS outperformed class-level RTS in terms of tests selected but could not always outperform class-level RTS in time due to the costs of method-level dependency collection. Companies such as Google and Microsoft rely on even coarser-grained dependency tracking, at the module level, due to the even quicker analysis time [12], [13], [29]. This work compares module- and class-level RTS in a cloud-based CI environment.

Our work is quite similar to work by Vasic et al. [31] that 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. For one project on which they evaluated, they found that adding class-level RTS improves module-level RTS time by 65.26%. 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 and runs *no* tests within affected modules. As such, we find that GIBstazi improves over GIB much less, 62.0% versus 64.6% of RetestAll time, respectively. Furthermore we compare both module-level RTS and class-level RTS, as well as against GIBstazi, in a cloud-based CI environment, where every build starts fresh on a new machine; Vasic et al. evaluated Ekstazi# on a dedicated machine.

Continuous Integration. Continuous integration is widely used in industry. Recent work has studied why developers use CI and the benefits they experience [20], [21], [39]. One main reason for the rise in CI research is the increase in developers using CI, particularly with services such as Travis, which provides CI for free for open-source projects on GitHub. Moreover, Travis exposes the logs from the builds that occur on their servers, allowing ease of access to build results. TravisTorrent [8] provides a dataset of logs from Travis and also some tooling for parsing the logs. We utilize Travis for our evaluation, and we also use the TravisTorrent tooling to parse the logs for our analysis of failed tests.

Yu and Wang [35] recently studied the potential of RTS in a CI environment. They analyzed factors such as the commit frequency of projects and the size of the changes between commits to gauge how effective RTS could be in a CI environment. They also compared a class-level and a method-level *static* RTS technique against RetestAll. Our work differs from theirs in that we investigate a module-level, class-level, and hybrid module/class-level *dynamic* RTS techniques in a CI environment. We also evaluate in a cloud-based CI environment, Travis, which differs from Yu and Wang's choice of Jenkins, a CI environment on a local server that is more under the control of the developers. Finally, we further analyze the test failures that occur during the runs and compare how well the RTS techniques select the failing tests.

There has been more work studying test failures on Travis (but not in combination with RTS as we do). Labuschagne et al. [22] studied how often regression testing on Travis reveals faults that developers fix. They queried Travis for the results of builds and focused on patterns of builds that toggle pass and fail 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 74% of the non-flaky failed builds were caused by a fault in the code under test, with the remaining due to incorrect/obsolete tests. They reported flaky tests to affect 13% of the failed builds they studied (although this percentage is an underestimate, as they considered historically failed builds that consistently passed during their reruns as non-flaky). 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. One reason for the different percentage of flaky tests is that they studied all builds (from the `master` branch and other branches, as well as pull-request builds) whereas we studied builds from the `master` branch (because many other builds cannot be replayed as they are not available).

VIII. Conclusions

Regression testing is widely practiced but costly, and RTS reduces the cost. Industry has adopted module-level RTS, while research has reported class-level RTS to be effective. We compare module- and class-level RTS in a cloud-based CI environment. We find that RTS techniques improve testing time over RetestAll in this environment, and GIBstazi, our new hybrid module- and class-level RTS technique, offers a good trade-off. Our investigation of test failures from RetestAll shows that the RTS techniques often miss to select some failed tests, but this happens to be desirable because these tests are flaky and not indicative of faults introduced by code changes. In sum, the results show that RTS offers benefit to developers, not only to reduce machine time but also to avoid false alarms from flaky tests.

Acknowledgments

We thank Tianyin Xu for his comments on an earlier draft of this paper and Qianyang Peng for providing some data on test failures for our evaluation. This work was partially supported by NSF grants. CCF-1421503, CNS-1646305, CNS-1740916, CCF-1763788, and OAC-1839010. We acknowledge support for research on regression testing and flaky tests from Facebook, Futurewei, Google, Microsoft, and Qualcomm.

REFERENCES

[1] fabricate. https://github.com/SimonAlfie/fabricate.

[2] gitflow-incremental-builder. https://github.com/vackosar/gitflow-incremental-builder.

[3] gitflow-incremental-builder with GIBstazi. https://github.com/august782/gitflow-incremental-builder.

[4] Maven. http://maven.apache.org/.

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

[6] A. Alali, H. Kagdi, and J. I. Maletic. What's a typical commit? A characterization of open source software repositories. In *ICPC*, pages 182–191, 2008.

[7] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov. DeFlaker: Automatically detecting flaky tests. In *ICSE*, pages 433–444, 2018.

[8] M. Beller, G. Gousios, and A. Zaidman. TravisTorrent: Synthesizing Travis CI and GitHub for full-stack research on continuous integration. In *MSR*, pages 447–450, 2017.

[9] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu. The promises and perils of mining Git. In *MSR*, pages 1–10, 2009.

[10] H. Borges, A. Hora, and M. T. Valente. Predicting the popularity of GitHub repositories. In *PROMISE*, pages 9:1–9:10, 2016.

[11] C. Brindescu, M. Codoban, S. Shmarkatiuk, and D. Dig. How do centralized and distributed version control systems impact software changes? In *ICSE*, pages 322–333, 2014.

[12] S. Elbaum, G. Rothermel, and J. Penix. Techniques for improving regression testing in continuous integration development environments. In *FSE*, pages 235–245, 2014.

[13] H. Esfahani, J. Fietz, Q. Ke, A. Kolomiets, E. Lan, E. Mavrinac, W. Schulte, N. Sanches, and S. Kandula. CloudBuild: Microsoft's distributed and caching build service. In *ICSE*, pages 11–20, 2016.

[14] M. Gligoric, L. Eloussi, and D. Marinov. Ekstazi: Lightweight test selection. In *ICSE DEMO*, pages 713–716, 2015.

[15] M. Gligoric, L. Eloussi, and D. Marinov. Practical regression test selection with dynamic file dependencies. In *ISSTA*, pages 211–222, 2015.

[16] A. Gyori, O. Legunsen, F. Hariri, and D. Marinov. Evaluating regression test selection opportunities in a very large open-source ecosystem. In *ISSRE*, pages 112–122, 2018.

[17] M. Harman and P. O'Hearn. From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis. In *SCAM*, pages 1–23, 2018.

[18] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for Java software. In *OOPSLA*, pages 312–326, 2001.

[19] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy. The art of testing less without sacrificing quality. In *ICSE*, pages 483–493, 2015.

[20] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, and D. Dig. Trade-offs in continuous integration: Assurance, security, and flexibility. In *ESEC/FSE*, pages 197–207, 2017.

[21] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *ASE*, pages 426–437, 2016.

[22] A. Labuschagne, L. Inozemtseva, and R. Holmes. Measuring the cost of regression testing in practice: A study of Java projects using continuous integration. In *ESEC/FSE*, pages 821–830, 2017.

[23] O. Legunsen, F. Hariri, A. Shi, Y. Lu, L. Zhang, and D. Marinov. An extensive study of static regression test selection in modern software evolution. In *FSE*, pages 583–594, 2016.

[24] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. An empirical analysis of flaky tests. In *FSE*, pages 643–653, 2014.

[25] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco. Taming Google-scale continuous testing. In *ICSE-SEIP*, pages 233–242, 2017.

[26] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *FSE*, pages 241–251, 2004.

[27] G. Rothermel, S. Elbaum, A. Malishevsky, P. Kallakuri, and B. Davia. The impact of test suite granularity on the cost-effectiveness of regression testing. In *ICSE*, pages 130–140, 2002.

[28] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *TOSEM*, 6(2):173–210, 1997.

[29] A. Shi, S. Thummalapenta, S. K. Lahiri, N. Bjorner, and J. Czerwonka. Optimizing test placement for module-level regression testing. In *ICSE*, pages 689–699, 2017.

[30] A. Shi, P. Zhao, and D. Marinov. Dataset for understanding and improving regression test selection in continuous integration. 2019. https://zenodo.org/record/3268234#.XR2RRYhKg2w.

[31] M. Vasic, Z. Parvez, A. Milicevic, and M. Gligoric. File-level vs. module-level regression test selection for .NET. In *ESEC/FSE*, pages 848–853, 2017.

[32] G. Xu and A. Rountev. Regression test selection for AspectJ software. In *ICSE*, pages 65–74, 2007.

[33] T. Xu, L. Jin, X. Fan, Y. Zhou, S. Pasupathy, and R. Talwadker. Hey, you have given me too many knobs! Understanding and dealing with over-designed configuration in system software. In *ESEC/FSE*, pages 307–319, 2015.

[34] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *STVR*, 22(2):67–120, 2012.

[35] T. Yu and T. Wang. A study of regression test selection in continuous integration environments. In *ISSRE*, pages 135–143, 2018.

[36] A. Zaidman and F. Palomba. Does refactoring of test smells induce fixing flaky tests? In *ICSME*, pages 1–12, 2017.

[37] L. Zhang. Hybrid regression test selection. In *ICSE*, pages 199–209, 2018.

[38] L. Zhang, M. Kim, and S. Khurshid. FaultTracer: A change impact and regression fault analysis tool for evolving Java programs. In *FSE*, pages 40:1–40:4, 2012.

[39] Y. Zhao, A. Serebrenik, Y. Zhou, V. Filkov, and B. Vasilescu. The impact of continuous integration on other software development practices: a large-scale empirical study. In *ASE*, pages 60–71, 2017.