2018 IEEE 29th International Symposium on Software Reliability Engineering

A Study of Regression Test Selection in Continuous Integration Environments

Ting Wang, Tingting Yu
Department of Computer Science
University of Kentucky, Lexington, KY, 40506, USA
twa222@uky.edu, tyu@cs.uky.edu

***Abstract*—Continuous integration (CI) systems perform the automated build, test execution, and delivery of the software. CI can provide fast feedback on software changes, minimizing the time and effort required in each iteration. In the meantime, it is important to ensure that enough testing is performed prior to code submission to avoid breaking builds. Recent approaches have been proposed to improve the cost-effectiveness of regression testing through techniques such as regression test selection (RTS). These approaches target at CI environments because traditional RTS techniques often use code instrumentation or very fine-grained dependency analysis, which may not be able to handle rapid changes. In this paper, we study in-depth the usage of RTS in CI environments for different open-source projects. We analyze 918 open-source projects using CI in GitHub to understand 1) under what conditions RTS is needed, and 2) how to balance the trade-offs between granularity levels to perform cost-effective RTS. The findings of this study can aid practitioners and researchers to develop more advanced RTS techniques for being adapted to CI environments.**

***Keywords*-regression testing; test case selection; continuous integration.**

I. Introduction

Continuous integration (CI) systems (e.g., Jenkins, Travis) are widely used in practice [8] to handle rapid software changes. When changes are made to projects, CI automatically runs regression tests to ensure that the changes did not break any working functionality. A survey [8] on 423 developers found that 70% of developers report that using CI helps them to catch bugs earlier and make them less worried about breaking the build. In the meantime, it is also critical to ensure CI testing is fast – it should run the "just-right" set of test cases to quickly detect bugs before the next change occurs.

Regression test selection (RTS) is a promising approach to speed up regression testing by selecting test cases that are important to execute [4], [17], [15], [12], [26]. While many techniques have been proposed to improve the cost-effectiveness of RTS, they are difficult to be adapted in CI environments. This is because traditional techniques tend to rely on code instrumentation and computation of fine-grained (e.g., statements, basic blocks, or methods) dependencies, which often require significant analysis time. However, in CI environments, testing requests arrive at frequent intervals, in which changes may have already occurred before the regression tests are selected and executed. For example, Hadoop 2.5.0 can experience up to 86 commits per day. Given the 16,837 methods and 44,552 tests, it is impossible to calculate dependencies and perform regression testing for each commit by tracking dependencies at the method-level.

On the other hand, the coarse-granularity technique is computationally efficient. For example, large organizations such as Google TAP often use very coarse-grained dependencies (e.g., module-level) to perform fast test selection [2]. However, coarse-grained dependencies can be imprecise such that a change in a module may result in selecting all tests. For example, Netflix project depends on an Inter Process Communication library – Ribbon. If Ribbon code changes, all tests for both Netflix and Ribbon will be run. While the coarse-granularity technique is computationally efficient, it may end up wasting time running a large number of irrelevant test cases for a small change. Therefore, the key trade-off between finer and coarser granularity is that finer granularity requires more time for tracking dependencies but can provide more precise results (i.e., selects a smaller number of more precise regression tests).

We believe that an explorative study on open source projects to quantitatively explore to what extent RTS is needed and how we balance the precision and cost of testing, can guide the design of, and improve techniques for addressing RTS in CI environments. A natural question to ask is whether sophisticated RTS is needed in CI. If change frequency is low in most the open source projects, then developers may use traditional RTS or coarse-grained dependency analysis techniques to select test cases because there will be enough time to perform analysis and execute a large number of test cases. On the other hand, if changes happen frequently, it is worth exploring cost-effective RTS techniques that can minimize the number of test cases being selected and reduce the number of analysis time.

In this paper, we perform an in-depth study and analyze 7,018,512 commits on 918 open-source projects using CI. We aim to uncover and quantify to which extend RTS is needed in open source projects. Specifically, we analyze the commit frequencies of all 918 projects to understand the speed of changes and how it can affect the efficiency of RTS. We also analyze the percentage of source files that are changed across commits to examine if analysis time should be spent a particular set of files. In addition, we use two static RTS techniques at different granularity levels to evaluate their cost-effectiveness compared to the ReTestAll (i.e., exercising all test cases).

2332-6549/18/$31.00 ©2018 IEEE
DOI 10.1109/ISSRE.2018.00024

The main findings of our study are as follows:

- A majority of commits (60%) happen in more than 10-minute time intervals. The results imply that if the total testing time is less than 10 minutes, RTS is not needed for 60% of commits.
- Code changes tend to concentrate on a small percentage (8.6%) of source files for the short commits whose intervals is less than 10 minutes. The results imply that in the presence of frequent changes (i.e., less than 10-minute time interval), RTS can focus on a small set of source files to reduce the cost of dependency analysis.
- A majority (62.2%) of Java projects whose average testing times do not exceed 25% of their average commit time intervals. The results imply that RTS may not be needed over the entire development cycle.
- On a majority (97.3%) of Java projects, static RTS at method-level performs much more poorly than ReTestAll. On the other hand, static RTS at the class-level saves more time than ReTestAll on more than half (56.8%) of Java projects. The results imply that method-level RTS, although selects fewer test cases, is not a time-efficient choice for project in CI environments.

We see this study as a way to share with researchers and practitioners the RTS issues that CI brings and potentially guide the development of practical RTS techniques in CI environments. The rest of the paper is organized as follows. We first present background and related work in Section II. We then describe our research questions in Section III and study setup in Section IV. Our results are demonstrated in Section V, followed by discussions and threats to validity in Section VI, and end with conclusions in Section VII.

II. Background and Related Work

A. Regression Testing

Let P be a program, let P' be a modified version of P, and let T be a test suite for P. Regression testing is concerned with validating P'. To facilitate this, engineers often begin by reusing T, but reusing all of T (the *retest-all approach*) can be inordinately expensive. Thus, a wide variety of approaches have been developed for rendering the reuse more cost-effective via regression test selection and test case prioritization ([24] provides a recent survey).

Regression test selection (RTS) techniques attempt to select, from test suite T, a subset T' that contains test cases that are important to re-run, and omit test cases that are not as important. Previous research has shown [16] that when certain conditions are met, RTS is *safe*; i.e., it cannot omit test cases which, if executed on P', would reveal faults in P' due to code modifications. Test case prioritization (TCP) techniques attempt to reorder the test cases in T to reach the testing objectives earlier and the potential objectives involved in revealing faults [18].

A key insight behind the use of regression testing techniques is that certain testing-related tasks such as collecting coverage information can be performed in the "preliminary period" of testing, before the changes to a new version are completed. This insight, however, only applies when there are the sufficiently long preliminary periods, and which is typically not the case in CI environments. In this work, we focus on regression test selection (RTS), which is orthogonal to TCP and can be combined for added savings.

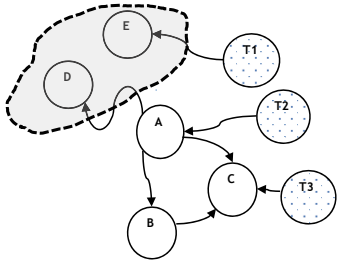

Fig. 1: Static regression test selection.

B. Regression Test Selection

Prior research on RTS can be broadly classified into dynamic and static techniques [11]. Dynamic RTS often requires instrumenting the program to collect the runtime information [6], [25], [5], which could limit its applicability in practice.

In contrast, static RTS techniques [1], [10], [19] rely on program analysis to infer dependencies among program entities (e.g., statements, methods, classes) to enable test selection. Specifically, when a change c occurs, all entities that transitively depend on c, denoted by D, are computed and tests associated with D are selected. The dependency analysis can be done at different granularity levels of program entities.

To illustrate, Figure 1 shows a method-level dependency graph, where nodes A–E represent methods and T represent test cases. The figure indicates that A, B, and C are dependent on each other and are all dependent on D. The nodes E and D belong to the same class file C1 and nodes A, B, and C exist in the same class file C2. When performing a method-level RTS, when E is changed, only $T1$ is selected. On the other hand, a class-level RTS would select all test cases because C2 depends on C1 based on their method dependencies. However, class-level dependency analysis is cheaper than method-level, although more test cases can be selected.

Recent research has proposed to make regression testing more cost-effective [6], [7] in modern software projects. For example, Gligoric et al. [7] propose a class-level dynamic RTS technique and their evaluation shows that it is more efficient than finer-level dynamic RTS. However, their work focuses on dynamic RTS, which is often impractical in CI environments due to runtime overhead.

Legunsen et al [11] show that class-level static RTS substantially outperforms method-level static RTS on 22 open-source Java projects regardless of their development environments. In contrast, we focus on software projects using CI and compare the costs of static RTS techniques at different granularity levels.

Elbaum et al. [5] create RTS techniques that use time windows to track how recently test suites have been executed

and revealed failures. Their technique is history-based, which may sacrifice both the safety (missing relevant tests) and the precision (selecting irrelevant tests). In addition, the evaluation focuses on specific software projects with frequent commits, in which RTS is needed. We intend to study static RTS techniques based on dependency analysis. In contrast, we study a wide range of open source projects to better understand the change frequency and its relationship to the use of RTS techniques.

C. Continuous Integration

There has been some research projects on studying software development activities in CI environments. Hilton et al. [9] study the usage of CI in open-source projects, such as to what extent CI is adopted in software development. Legunsen et al. [11] evaluate the performance benefits of static RTS techniques and their safety in modern software projects. Memon et al. [13] share their experience and results in RTS on Google projects. Vasilescu et al. [23] study the productivity of CI based on 246 GitHub projects. However, these studies focus on specific RTS techniques and do not evaluate the extent to which RTS is needed or quantify the factors affecting the cost and effectiveness of RTS in a wide range of open source projects.

III. Research Questions

CI encourages developers to break down their work into small tasks because smaller and frequent commits help them keep track of their progress and reduce debugging effort [3]. Miller [14] observed that on average, Microsoft developers made code commits every day, whereas while off-shore developers committed less frequently. Zhao et al. [27] report that after CI is introduced, the commit frequency is 78 commits every day.

In the presence of frequent commits in CI, testing must be automated and conducted cost-effectively to make sure each commit would not break the build. RTS is used to select test cases (often determined by dependency analysis) that are most likely to be affected by the commits. Existing research has shown that traditional RTS cannot be applied in CI environment because it requires significant analysis time, which cannot catch up with the speed of changes [5]. While commit frequencies can be used to measure the extent of CI usage, the degree to which RTS can be applied depends on the arrival rates of commits or commit intervals (the time between two consecutive commits). For example, if commits (i.e., changes) happen frequently in short intervals, then cost-effective RTS is needed. On the other hand, at a certain development period (i.e., when the project is stable), when commits happen in larger intervals, developers may just re-execute all test cases to simplify the testing. Therefore, we ask the following research question:

RQ1: *What are the commit intervals across different projects*?

A main concern about using fine-grained dependency analysis to select test cases is the expensive analysis time before every testing. As the frequent commits tend to be small, they usually occur to some specific project files (i.e., finish coding a Java class within a day by a few commits). In this case, file/class-level dependency analysis may not be needed for every single commit that attributes to the same file/class. Therefore, we would like to know whether the frequent changes tend to concentrate on certain classes/files or in project written in specific languages. The following research question is asked.

TABLE I: Types of configurations

# Project	C	C++	Java	JavaScript	Python	PHP	Ruby	Scala
918	60	55	77	217	183	124	180	22

RQ2: How many project files are changed at different time intervals and how are they related to different programming languages?

When determining whether RTS is useful, it is important to know the time cost for test execution. If a majority of commit intervals are much shorter than the time of executing all test cases, RTS may be necessary to reduce the cost of regression testing and provide fast feedback for developers. Otherwise, there may be no need to perform RTS for particular projects or at certain development periods. Therefore, we are interested in the following research question:

RQ3: To what extent is RTS needed compared to ReTestAll method?

When RTS is needed, to determine which test cases to select, one approach is to use dependency analysis to identify test cases associated with the changed elements and other elements affected by the changes. Different analysis techniques differ in precision and overhead. Techniques that collect finer-granularity dependencies may be more precise, selecting fewer tests to be run, but can incur higher analysis overhead; in contrast, techniques that collect coarser-granularity test dependencies may be less precise but can have lower analysis overhead [11]. Therefore, we would like to investigate the cost-benefits of RTS techniques at different levels of granularity. We ask the following research question:

RQ4: What are the cost-benefits of dependency analysis with different levels of granularity in RTS?

IV. Study Setup

A. Collecting Data Sets

The projects were from the datasets provided by Vasilescu et al. [23], in which they used 924 GitHub projects to study the productivity of CI. We cloned 918 projects of the list from the official repository to our own GitHub repository for conducting our experiments, because some projects were not publicly available. We then sorted the projects by their programming languages, including JavaScript, Python, Ruby, PHP, C, Java, and Scala. For each project, we used GitHub APIs to collect the following history information: project name, the primary programming language, the reference number of all the commits, the changed files at each commit, and the time stamp of each commit. To gain a deep understanding about the historical information of these projects, the data collected is

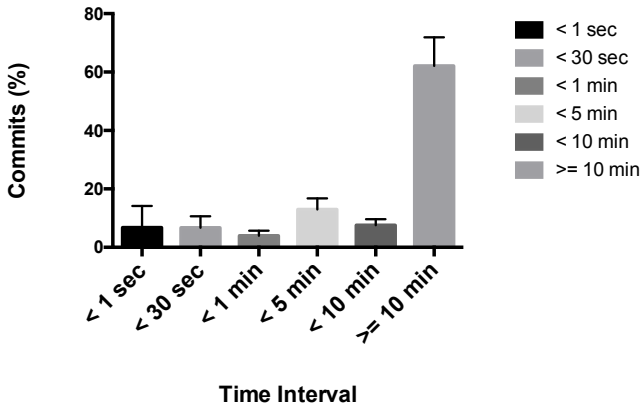

Fig. 2: Percentage of commits at different time intervals

further processed by Python scripts. Table I lists the number of projects under different programming languages.

To answer RQ3 and RQ4, we need to execute the programs and perform program dependency analysis. We use Understand [22], a commercial static analysis tool for Java to analyze dependencies among methods and classes. Therefore, we need to find projects that can 1) successfully compile under our environment settings; 2) work with the program dependency analysis tool. Therefore, 37 Java projects were selected. Next, we need to simulate the real scenarios of developers' commits. We use "git log" to get the reference IDs of all commits of the project history and time stamps of the commits. We selected a sequence of 20 consecutive versions for each project. In total, there are 7,018,512 commits. We then wrote a shell script to automatically push the selected revisions to the repository based on the commit IDs following their time order.

We use Jenkins CI [21], which is one of the most popular CI servers, to setup our CI environment. To enable the repo-driven trigger, we setup the webhook between GitHub and Jenkins. Both can be found under the settings of each project repository. Once each revision is committed to the repository, the test case selection is triggered as one part plugin of the Jenkins CI process. Once the test selection is being finished, Jenkins will start the building and testing for the revisions. Since we wish to study how in general code changes affect RTS, we consider single changes. It would be worth exploring pushes, as the reviewer suggest. The conclusions and implications would be similar because push intervals are larger than commit intervals.

V. Results

We now present our results for each of the four research questions [1].

A. RQ1: Frequency of Commits

The first research question pertains to the productivity of the project in order to help us understand to what extent RTS is needed in CI environments. As discussed in Section I, CI runs the integration of compile, build and test at each commit of the software life cycle. If several authors are very active,

[1]Artifacts and experimental data are available at https://github.com/Ting007/masterProject

TABLE II: Correlation coefficient between the number of total commits and the percentage of commits

Interval	<1 sec	< 30 sec	< 1 min	< 5 min	< 10 min	> 10 min
Cor. Coe.	0.58	0.21	0.13	-0.14	0.18	-0.61

the commits often happen in a rushed fast wave, running a complete testing of the whole project could be too expensive and time consuming. It might be more practical and optimal to select the most necessary test cases to run. On the other hand, RTS may not be needed under the following situations: 1) the project has only a few developers who are unlikely to make changes concurrently; 2) the project is stable over certain time periods in which fewer changes are made; 3) the project has a small number of test cases. To better understand to what extent RTS should be used in CI, we first study the frequency of commits (the number of commits happening within some certain time intervals) for different projects.

Due to the large number of commits, the results are represented by the percentage of commits instead of the actually numeric numbers. Figure 2 shows the percentage of commits over all projects distributed across different time intervals. The results indicate that of all 7,018,512 commits, 6% of them happened in less than one second, 30% happened in less than 5 minutes, and 60% happened in greater than 10 minutes.

Next we inspect the percentage of commits at each time interval for projects with different total commits. Because we would like to know if the total number of commits would introduce some influence to the commit frequency. For projects in the list, the total number of commits is not evenly distributed. To eliminate the bias from the population of projects, each column data is calculated based on 9-12 % of the total projects. The bin sizes are chosen based on the number of projects. Specifically, To avoid the bins being skewed, they contain approximately the same number of projects. The results are plotted in Figure 3. For example, the percentage of commits happening within less than one second is in an increasing trend (from 3.6% to 8.9 %), particularly for those projects with a larger number of total commits(above 3500 commits). This indicates that projects tend to be more active with short-time intervals (<1 sec). However, the trend is not obvious for larger intervals (e.g., <30 sec, <1 min, < 5 min). In fact, when the time interval goes greater than 10 minutes, the percentage of longer commits (>= 10 min) is decreased from an average of 66.4% to 55.4% as the number of total commits are increased from 0 to 17500.

Table II shows the correlation coefficients between the number of total commits and its percentage under different time intervals. We use Pearson's correlation coefficient [20]. The coefficient value of > 0.50 or <-0.50 are considered to be strong, while values between -0.49 and -0.30 or between 0.30 and 0.49 are considered to be moderate and values between -0.30 and 0.30 are considered to be weak. The results shows that the correlation coefficients for < 1-second time interval and >= 10-minute interval are strong, but in an opposite way. This suggests that, for larger commit intervals, the percentage of commits is decreased. It implies that RTS may not be

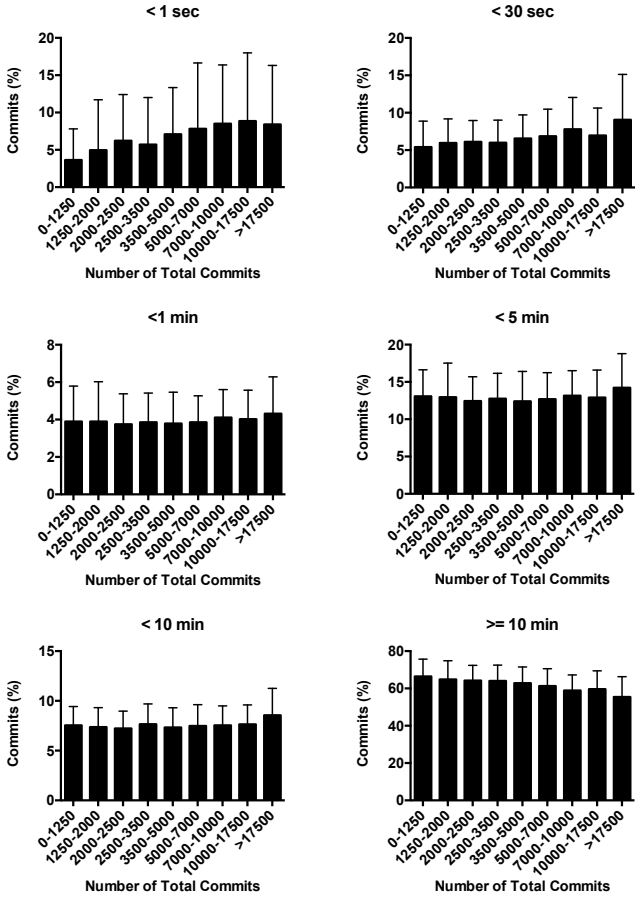

Fig. 3: Percentage of commits at different time intervals for projects with different numbers of total commits

needed at larger commit intervals. As the number of total commits grows with the project development (more than 3500 commits), there will be a higher rate of short time intervals ($<$ 1 sec).

> **RQ1:** A majority of commits (60%) happen in more than 10-minute time intervals. In other words, RTS is useful for a majority of projects if the test execution times exceed 10 minutes.

B. RQ2: File Changes

We would like to know the number of source files changed at each commit. This information can help developers better decide whether or not to skip a testing or to make a dependency analysis. For example, if some consecutive commits concentrate on a particular set of class files, there is no need to perform dependency analysis for each of these commits because the analysis would always output the same set of affected files.

To eliminate the influence from different programming languages, we first sorted all projects by the category of different programming languages (C+, C, Java, JavaScript,

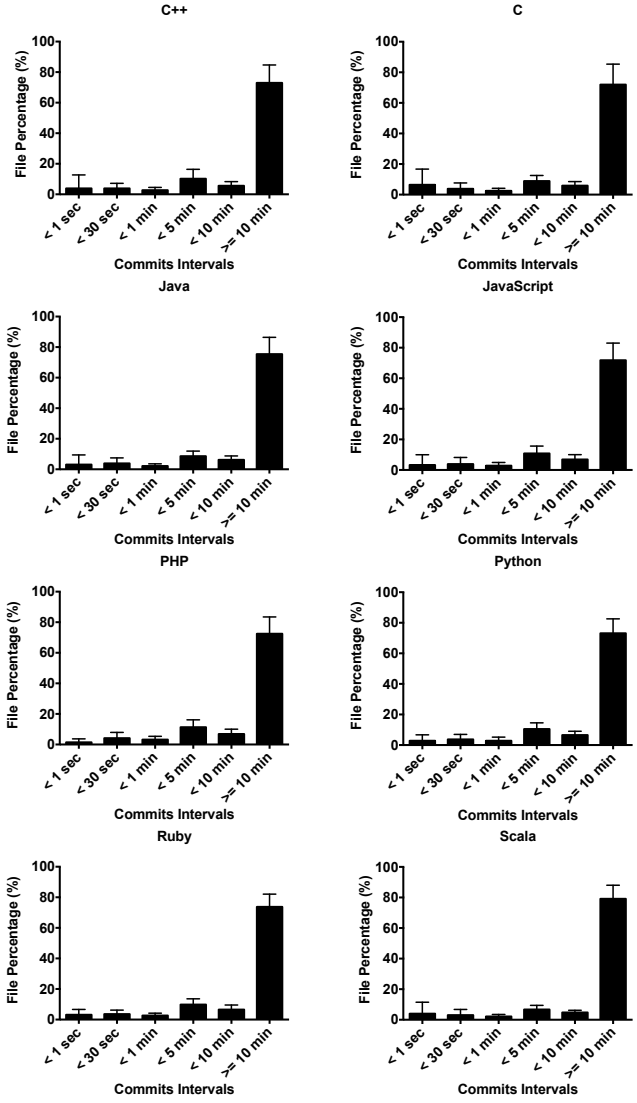

Fig. 4: Percentage of changed files per commit across different time intervals

PHP, Python, Ruby, Scala). Figure 4 and Figure 5 represent the data results of all projects classified by their programming languages. Figure 4 focuses on the percentage of changed files per commit at different time intervals. Figure 5 focuses on the percentage of different types of changed files per commit.

Figure 4 indicates that the percentage of the changed files increases with the time interval, especially for those $>$ 5 min. This trend is consistent across different languages. For example, for the projects mainly written in PHP, there are only 1% changed files on average per commits for 1-second interval, whereas up to 8% files are changed for 10-minute interval. Comparing the number of changed files at the 1-second interval to those at the 5-minute interval, the percentage of changed files is almost 3 times for programs written in Java, JS, Scala and C. This percentage is increased by a factor of 5 for programs written in PHP, Python and Ruby. The results

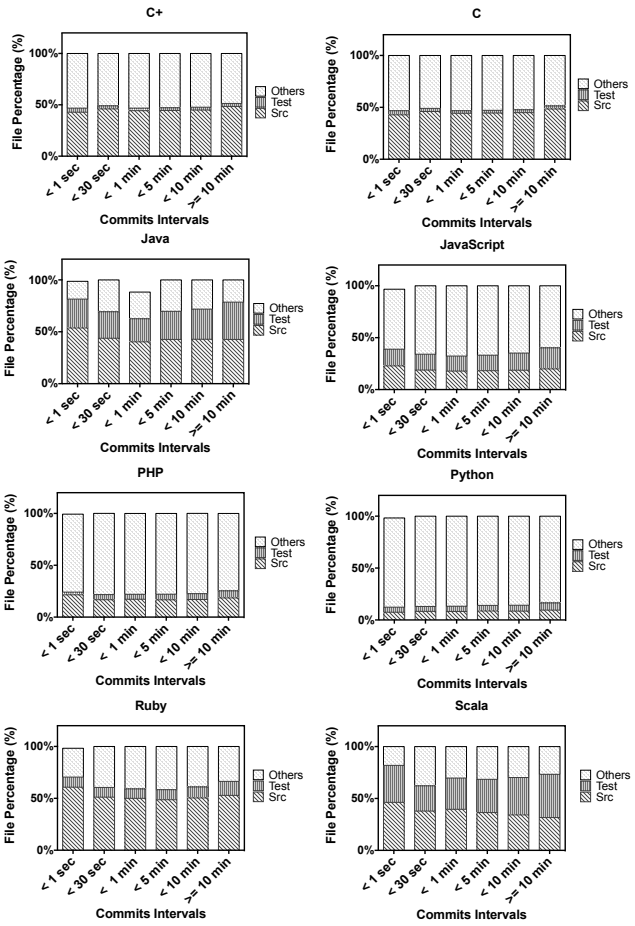

Fig. 5: Percentage of changed files in different types across different time intervals

are not surprising as the number of changes is expected to increase if more time had been available between commits.

However, the percentage of changed files may not reflect changes in actual source files that affect RTS. For example, many changes happen in text files only involving readme or configuration files. To further understand our results, we calculated the percentage of changed files in terms of their file types. Specifically, we classify programs files into source files, test files, and other files. The changed test files/classes are also supposed to execute for a new commit of source file.

Figure 5 shows the percentage of changed files in different types. The average percentage of changed project source files at each time interval is about 50% for programs written in C+, C, Java, Ruby and Scala; 10–20% for programs written in PHP and Python. The percentage of test files changed at each time interval is about 20–40% for programs written in Java, JavaScript, Scala. However, for projects written in C+, C, PHP, Python, and Ruby, the percentage of changed test files is small, only 10%.

Figure 5 also shows that the percentage of changed files in different types does not vary much at different time intervals. For programs written in the same language, the percentage of each file type differs less than 10% across different time intervals. Therefore, longer time intervals may not imply more effort will be spent on the actual project coding.

The results also suggest that the percentage of file types differs a lot among different languages. We conjecture the reason is because many of the programming languages are used for different purposes. For example, Ruby is a dynamic, reflective, and object-oriented languages. JavaScript is a language that does not include any I/O, such as networking, storage or graphics. The primary application of PHP is the implementation of web pages of dynamic content. Thereafter, some projects written in programming languages, like Python, only has 10–20% of files per commits are directly related to the code and tests of the program. For some programs written in Java or Scala, 80% of the files per commits are related to the code and tests of the software.

> **RQ2:** Changes tend to concentrate on a small percentage (8.6%) of files for short-time commit intervals. This implies that dependency analysis may not be needed at each commit in order to save the analysis time of RTS. Moreover, the results imply that RTS should take program languages into consideration — programs written in C+, C, Java, JavaScript, Ruby, and Scala have a higher probability of containing changed source code than those written in Python, PHP and JavaScript.

C. RQ3: Testing Times

RQ1 and RQ2 analyze the commit intervals and changed files. In this research question, we would like to know if the frequency of commits leaves enough time for the testing or not. If the time between commits is enough for executing all test cases, then no RTS is needed.

We collected runtime statistics for the studied projects, including 1) compile time: the time spent on compiling the project; 2) testing time: the time spent on exercising all tests; 3) rapid revisions: the commits whose time intervals are shorter than the testing time required for exercising all tests.

Figure 6 shows the compilation times and testing times from the 37 Java projects. Our results indicate that the total build time (compilation and testing) ranged from 8.2 seconds to 527 seconds, which is consistent with the findings by Hilton et al. [9]. The average testing times range from 2.1 seconds to 2129 seconds across all 37 projects. However, on 30 projects, the average testing times are less than 500 seconds. Recall that RQ1 indicates that 60% of commits happen in time intervals greater than 600 seconds. These results suggest that RTS may not be needed in a majority of commits for most of the projects.

We next counted the number of rapid revisions. Figure 7 shows the percentage of rapid revisions for the 37 Java projects. The results indicate that 23 out (62.2%) of 37 projects contain less than 25% rapid revisions. In other words, RTS is needed for the 25% of the commits on these projects. For the

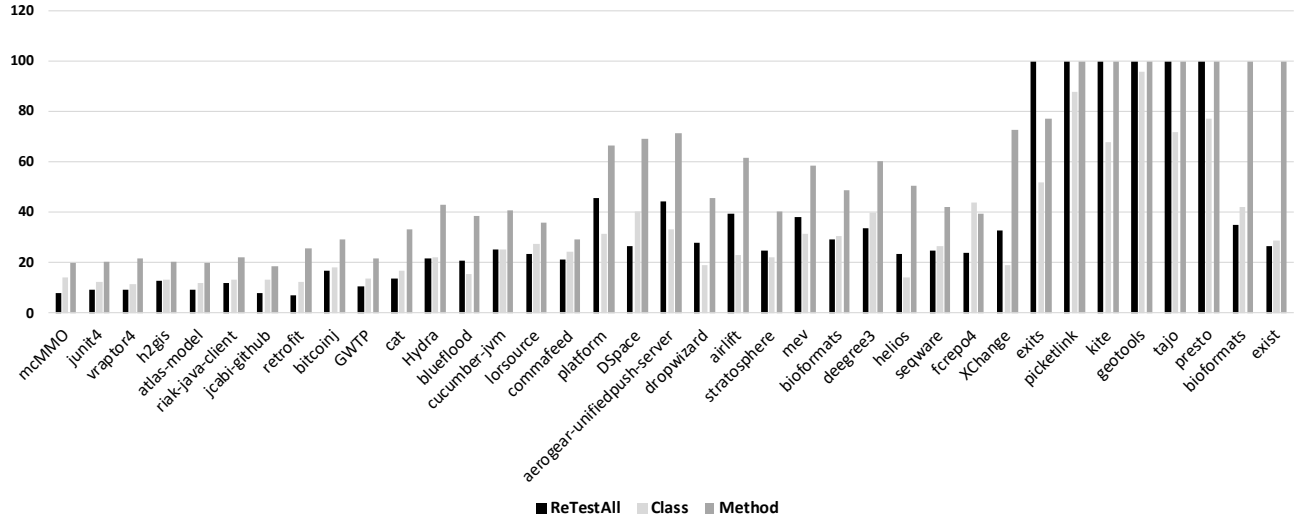

Fig. 6: Time of compilation and testing

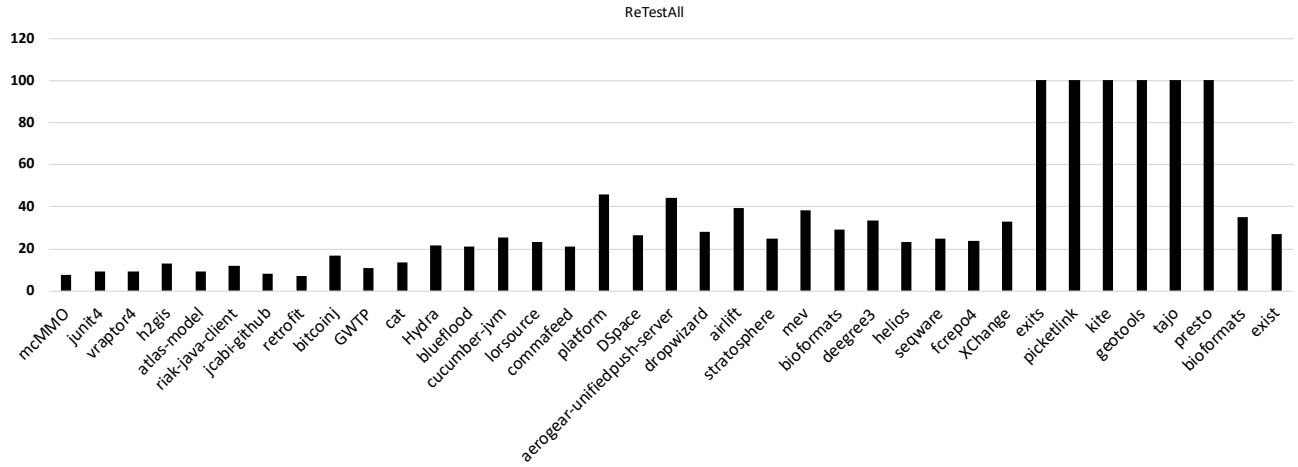

Fig. 7: Percentage of rapid revisions when using ReTestAll

seven projects whose testing time is over 10 minutes, almost 100% of commits are rapid revisions.

> **RQ3:** A majority (62.2%) of Java projects whose average testing times do not exceed 25% of their average commit intervals. The results imply that RTS is only needed in a small percentage of time intervals instead of every commit.

D. RQ4: Dependency Analysis

We analyze the cost-effectiveness of RTS techniques at two granularity levels: class-level and method-level. We first apply the class-level dependency analysis for each revision/-commit. Among all 7,018,512 revisions, the selection ratio of test classes varies between 0% and 56% of RetestAll (i.e., executing all test cases). The time varies between 8% and 265% (where over 100% is slowdown) of RetestAll. We next apply the method-level dependency analysis for each revision. Among all 7,018,512 revisions, the selection ratio of test classes varies between 0% and 38% of RetestAll (i.e., executing all test cases). The time varies between 65% and 492% of RetestAll.

Figure 8 shows the percentage of rapid revisions for RetestAll and the two RTS techniques. Among all 37 Java projects, when applying the class-level RTS, the number of rapid revisions is reduced in 16 projects, ranging from 2% to 15%. On the other hand, when applying the method-level RTS, the number of rapid revisions is reduced in only one project and on the rest of the 36 projects, the the number rapid revisions is increased from 0% to 298%.

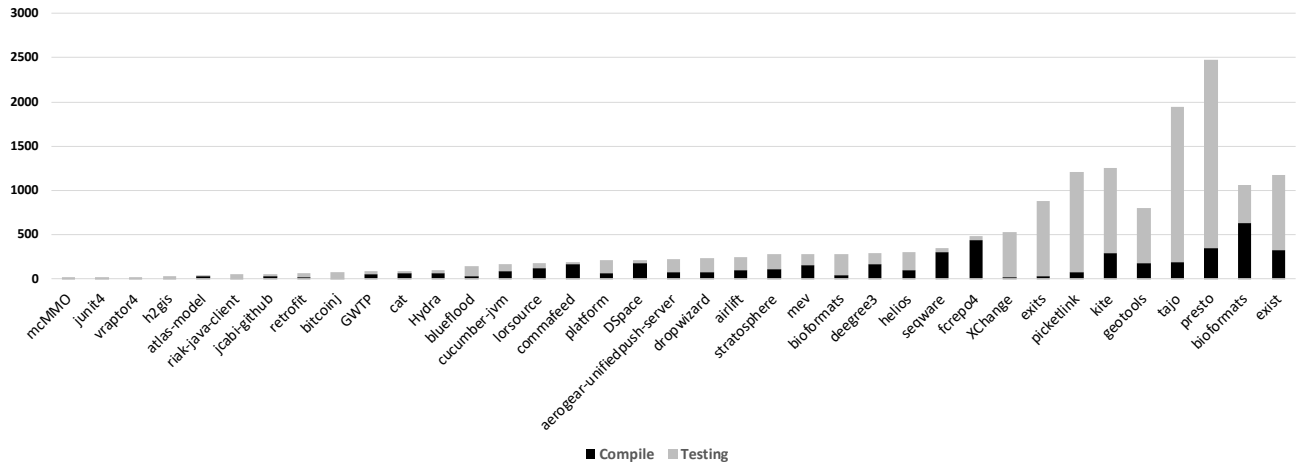

Fig. 8: Percentage of rapid revisions when using ReTestAll, class-level dependency analysis, and method-level dependency analysis

RQ4: Class-level dependency analysis performs much better than method-level dependency analysis and can reduce the cost of RTS in 43.2% of the projects. On the remaining 56.8% of the projects, the two static RTS techniques do not show benefits in reducing the cost.

VI. Discussion

A. Implications

CI is motivated as an integration technique to automate the process of compilation, building and testing. One benefit of CI testing is to guarantee the quality of code at every single commit. To adapt regression testing in CI environments, the testing times must catch up with the frequency of commits. In our RQ1, however, we found that a majority (60%) of commits are over 10 minutes. The findings can help developers make testing decisions during software development. For example, if the total test execution exceeds 10 minutes, RTS techniques should be applied.

Our study assumes a sequential execution of the tests. While it is not uncommon to execute CI pipeline runs (i.e., executing tests in parallel), since one goal in our study is to compare RTS and ReTestAll, we wish to give RTS more opportunities because the longer the tests execute the more RTS is needed. As the results suggest, even in the case of sequential test execution, RTS is not needed in a majority of projects and commits. In addition, machine resources for parallel execution are difficult to control in the experiment and for others to replicate.

Second, programming language is measured as a factor influencing the changes of different file types, as shown in RQ2. The results imply that projects written in C/C+, Java, Ruby are in a higher demand of using RTS than projects written in other languages.

Third, in RQ3, we found that testing times exceed the commit intervals for less in less than 25% of commits for a majority of Java projects. The results imply RTS is often needed over a small portion of the software development cycle. Therefore, adaptive RTS techniques might be developed to selectively execute RTS for certain commits. For example, when a new project feature is introduced in which more code commits are expected, RTS can be applied to reduce the cost of testing; otherwise, it would be more safe to execute more test cases.

Fourth, RQ4 indicates that class-level RTS substantially outperforms method-level RTS. However, the class-level RTS still performs poorly in projects with short testing times (i.e., less than 500 seconds). The results imply that RTS techniques based on dependency analysis may not be applicable to projects with short testing times. Other techniques, such as history-based RTS (e.g., selecting test cases based on the number of faults revealed in the past) might be appropriate, but subject to further studies.

B. Threats to Validity

The primary threat to external validity for this study involves the representativeness of our subject programs. The data we gathered comes from 918 projects deployed in Travis CI environments. Other subjects and CI server may exhibit different behaviors. However, we do reduce this threat to some extent by using several varieties of well studied open source code subjects for our study.

The primary threats to internal validity for this study involve possible faults in the implementation of dependency analysis in RTS. We controlled for this threat by extensively testing our tools and verifying their results against a smaller program for which we can manually determine the correct results.

The primary threat to construct validity involves the dataset and metrics used in the study. To mitigate this threat, we used open source projects from GitHub, which are publicly available and well studied by existing research [23]. We have also used well known metrics in our data analysis such as the number of commits and correlation analysis, which are

straightforward to compute.

VII. Conclusions

We have performed an empirical study employing regression test selection (RTS) in continuous integration (CI) environments. We have analyzed 7,018,512 commits on 918 open-source projects from GitHub. Our study investigates several research questions related to RTS, including frequency of commits, how project files are changed across commits, to what extent testing times exceed commit intervals, and the influence of dependency analysis at different levels of granularity. The study provides guidance for future research on RTS in CI environments. In the future, we will extend our study on more subject programs and propose techniques to improve the cost-effectiveness of RTS.

Acknowledgments

This research is supported in part by the NSF grant CCF-1652149.

References

[1] Linda Badri, Mourad Badri, and Daniel St-Yves. Supporting predictive change impact analysis: a control call graph based technique. In *12th Asia-Pacific Software Engineering Conference*, pages 9–pp, 2005.

[2] Google Testing Blog. Testing at the speed and scale of google. Website, 2011. https://testing.googleblog.com/2011/06/testing-at-speed-and-scale-of-google.html.

[3] Paul M Duvall, Steve Matyas, and Andrew Glover. *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[4] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159–182, 2002.

[5] Sebastian Elbaum, Gregg Rothermel, and John Penix. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 235–245, 2014.

[6] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. Ekstazi: Lightweight test selection. In *Proceedings of the 37th International Conference on Software Engineering*, pages 713–716, 2015.

[7] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. Practical regression test selection with dynamic file dependencies. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 211–222, 2015.

[8] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *IEEE/ACM International Conference on Automated Software Engineering*, pages 426–437, 2016.

[9] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, 2016.

[10] David Chenho Kung, Jerry Gao, Pei Hsia, Jeremy Lin, and Yasufumi Toyoshima. Class firewall, test order, and regression testing of object-oriented programs. *JOOP*, 8(2):51–65, 1995.

[11] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. An extensive study of static regression test selection in modern software evolution. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 583–594, 2016.

[12] Hong Mei, Dan Hao, Lingming Zhang, Lu Zhang, Ji Zhou, and Gregg Rothermel. A static approach to prioritizing junit test cases. *IEEE Transactions on Software Engineering*, 38(6):1258–1275, 2012.

[13] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. Taming google-scale continuous testing. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP),*, pages 233–242, 2017.

[14] Ade Miller. A hundred days of continuous integration. In *Agile Conference*, pages 289–293, 2008.

[15] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. Scaling regression testing to large software systems. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 241–251, 2004.

[16] Gregg Rothermel and Mary Jean Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, 1996.

[17] Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, 1997.

[18] Gregg Rothermel, Roland J. Untch, and Chengyun Chu. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):102–112, 2001.

[19] Barbara G Ryder and Frank Tip. Change impact analysis for object-oriented programs. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 46–53, 2001.

[20] Philip Sedgwick. Pearson's correlation coefficient. *BMJ: British Medical Journal (Online)*, 345, 2012.

[21] John Ferguson Smart. *Jenkins: The Definitive Guide*. O'Reilly Media, Inc., 2011.

[22] SciTools.com, 2018. https://scitools.com.

[23] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. Quality and productivity outcomes relating to continuous integration in github. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 805–816, 2015.

[24] Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: A survey. *Journal of Software Testing, Verification, and Reliability*, 22(2):67–120, 2012.

[25] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. Localizing failure-inducing program edits based on spectrum information. In *27th IEEE International Conference on Software Maintenance*, pages 23–32, 2011.

[26] Lingming Zhang, Ji Zhou, Dan Hao, Lu Zhang, and Hong Mei. Prioritizing junit test cases in absence of coverage information. In *IEEE International Conference on Software Maintenance*, pages 19–28, 2009.

[27] Yangyang Zhao, Alexander Serebrenik, Yuming Zhou, Vladimir Filkov, and Bogdan Vasilescu. The impact of continuous integration on other software development practices: a large-scale empirical study. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 60–71, 2017.