Assessing Transition-based Test Selection Algorithms at Google

Claire Leong, Abhayendra Singh
Google Inc.
Mountain View, CA, USA
{claireleong, abhayendra}@google.com

Mike Papadakis, Yves Le Traon
University of Luxembourg
Luxembourg
{michail.papadakis, yves.letraon}@uni.lu

John Micco
Los Gatos, CA, USA
john.micco@gmail.com

***Abstract*—Continuous Integration traditionally relies on testing every code commit with all impacted tests. This practice requires considerable computational resources, which at Google scale, results in delayed test results and high operational costs. To deal with this issue and provide fast feedback, test selection and prioritization methods aim to execute the tests which are most likely to reveal changes in test results as soon as possible. In this paper we present a simulation framework to support the study and evaluation, with real data, of such techniques. We propose a test selection algorithm evaluation method, and detail several practical requirements which are often ignored by related work, such as the detection of transitions, the collection and analysis of data, and the handling of flaky tests. Based on this framework, we design an experiment evaluating five potential regression test selection algorithms, based on simple heuristics and inspired by previous research, though the evaluation technique is applicable to any number of algorithms for future experiments. Our results show that algorithms based on the recent (transition) execution history do not perform as well as expected (given the previously reported results) and that the test selection problem remains largely open. We found that the best performing algorithms are based on the number of times a test has been triggered and the number of distinct authors committing code that triggers particular tests. More research is needed in order to close the gap between the current approaches and the optimal solution.**

***Index Terms*—Continuous Integration, Regression Testing**

I. Introduction

To achieve fast and large scale development, Continuous Integration (CI) requires the continuous (frequent) integration of code into a shared repository [1]. Code verification and validation is enforced through a set of automatic and manual checks, such as builds, tests and code reviews. This software development paradigm has been adopted by many large software vendors, such as Microsoft [2], Facebook [3], Google [4] and Netflix [5]. CI is also becoming popular in the Open-Source community [6], [7], and is supported by many frameworks and tools [8], [9].

The success of the CI process heavily relies on testing, and at large scale requires substantial computational resources in order to regression test every code change [1], [3], [4]. At Google for example, there is on average one code commit every second, triggering more than 150 million test executions every day using the current testing infrastructure. The vast computational demands of the Google CI Environment lead to high operating costs and long test execution times (ranging up to 9 hours) to integrate code changes [4].

Naturally, researchers are directing their efforts towards reducing the cost of CI related activities. For example, techniques aimed at reducing the cost related to project builds [10], test machines' configuration [11] and test executions [12], [13] have been proposed. Generally, there are two main ways of tackling the regression testing problem, known as *regression test selection* (RTS) and *regression test case prioritization* (TCP) [14]. RTS methods select relevant/important sets of test cases to execute, while TCP methods order test cases with the intention of detecting faults earlier. Most of these techniques leverage test information from previous releases, such as coverage [15] or input diversity [15], [16].

Applying regression testing in the Google CI environment is challenging, mainly due to its scale and code commit pace. The need for fast responses makes the collection of coverage information prohibitively expensive. Test infrastructure should consume the smallest possible amount of resources, while at the same time informing Google developers if a commit triggers any transitions as soon as possible. A transition is a change in state in the sequence of results across commits for a test, either from Pass to Fail or Fail to Pass. The requirement of detecting transitions is something ignored by previous research, which mainly focuses on detecting test failures. Evaluating RTS performance by test failures alone ignores the importance of identifying Fail to Pass transitions, and impacts results - at Google there are over 20 times more test failures than transitions in a month.

Another important problem in regression testing is the flaky test problem, i.e., tests with non-deterministic outcomes [17]. At Google 84% of transitions are caused by flaky tests, and 16% of tests involve some level of test flakiness [18]. This issue has serious implications for regression testing, particularly for RTS and TCP methods.

Existing lightweight CI RTS techniques select tests which failed in the recent execution history [12], [13]. Unfortunately, since flaky tests fail arbitrarily and often, such techniques tend to prioritize flaky instead of non-flaky tests. This calls into question whether these methods offer practical benefits.

In this paper we focus on applying different RTS algorithms in the Google CI environment. We also develop a mechanism for comparing the success rate of test selection methods with regards to transition detection at code commits which can be used to evaluate any RTS algorithm.

This evaluation framework uses the historical sequence of test results to simulate the performance of test selection methods on real data. It accounts for test flakiness by filtering flaky test executions, and evaluates methods by analyzing their ability to schedule test transitions when skipping execution for some percentage of tests at each commit. The framework aims to identify the relative trade-offs between skipping tests (lower cost) and discovering transitions (higher effectiveness) achieved by the techniques studied.

Due to the design of the evaluation, it also assesses the methods' abilities for use in TCP, and as such they could be considered as both RTS and TCP techniques. Because Google's intended use is for RTS (since tests are run in parallel, there is reduced need for TCP) we focus on this aspect of the algorithms and results.

The ultimate goal of our work is to identify signals with strong results which could lead to future research into test selection schemes with relatively accurate trade-offs, and to develop an inexpensive (relative to implementing the algorithms at scale) way to compare algorithm performance before a full implementation.

We evaluate five algorithms using simple heuristics; two based on the recent commits impacting a test (number of affecting commits and number of authors committing affecting code), two based on previous test results (pre-submit outcomes - testing prior to commit - and number of historical transitions), and one based on test characteristics (number of directories shared with modified files).

Our results demonstrate that flaky tests have a significant impact on the evaluation of regression testing techniques, and should be discarded. We show that algorithms based on the recent affecting code commits of a test performed best, outperforming a random method for over 30% of transition commits (commits where a transition occurs) in some cases. Algorithms based on prior testing results (pre-submit history and transition count) show worse performance but still a significant improvement over random selection. Unfortunately, the algorithm based on directory overlap performs similarly to random. Interestingly, the gap between the algorithms' performance and the optimal case is large (up to 90%), indicating a lot of room for improvement.

Overall, we make the following contributions:

- We demonstrate that flaky test transitions have a significant effect on RTS method evaluations in the CI context.
- We identify the requirements for CI-based test evaluations (non-flaky test transition data).
- We developed a fast, lightweight off-line test scheduling evaluation framework.
- We perform a preliminary evaluation of RTS algorithms using the proposed framework.

II. Google's Continuous Integration Environment

The scheduling and execution of tests in Google's CI environment is managed by their Test Automation Platform (TAP). A detailed explanation of TAP can be found in Memon et al.'s paper on Google testing (Section II) [4]. We briefly outline the salient points here.

At every code commit, TAP is responsible for identifying the set of affected tests - a subset of tests containing all tests possibly impacted by the commit. TAP produces this subset by evaluating whether the code commit modified files in the transitive closure of a test's dependencies. A test in Google's CI context is called a "test target", which is a code unit that can contain several test classes and methods.

Due to the rapid rate of code churn and large size of the test pool, affected targets will not be run at the time of commit, but rather delayed until the next "milestone" is cut. A milestone is a point in time at which all targets affected since the previous milestone will be executed. This allows TAP to skip multiple executions of frequently affected targets between milestones. Milestones are cut at Google as often as allowed by the available resources.

TAP utilizes two stages of testing: pre-submit and post-submit. Pre-submit testing occurs prior to commit, and is typically limited to targets in the modified code's set of projects. Post-submit testing occurs after commit and considers affected targets across all projects in the shared code repository.

When a test failure is detected in post-submit, TAP performs culprit finding in order to identify the exact commit which caused the failure. As a result, all pass to fail transitions are accurately recorded by TAP.

To mitigate test flakiness at Google, targets returning a fail result will be rerun several times TAP. In addition, Google has developed a flakiness oracle which uses sophisticated analysis of target results and reruns to further identify flaky targets and executions.

This paper evaluates the use of RTS algorithms at Google, to be considered in post-submit testing for commits taking place between milestones. Running a small set of likely transitioning targets at commit time, instead of at the next milestone, would reduce delays in executing tests, and allow Google to further spread out milestones to save resources while still providing signals to developers with low latency.

III. Influence of Flaky Tests on Regression Testing Evaluations

Flaky tests are a significant source of transitions at Google: in our data over 80% of observed transitions were caused by confirmed flaky results. These flaky transitions create noise in our data as they don't represent a genuine code breakage or fix. Previous work on Google data [12], [13], did not consider test flakiness, with the unfortunate effect of including both genuine and flaky transitions in their evaluations.

To analyze the effects flakiness has on our evaluations, we ran our experiment using both raw and deflaked Google data (using the procedure outlined in Section IV). Figure 1 shows the results of evaluating one of our algorithms (outlined in Section VI-B), with and without flaky executions in the test data. This graph records the percentage of commits where the algorithm skipped targets transitioning at the commit (on the y-axis) for every percentage of tests skipped (on the x-axis).

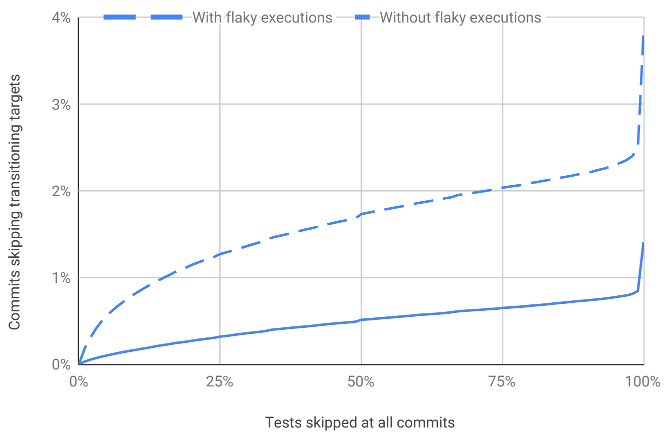

Fig. 1. Performance of the transition count algorithm (Section VI-B), on data with and without flaky executions.

This means that the higher the values we observe, the worse our results are as we want to minimize selections which miss a transition.

From these results we can see that flakiness leads to misleading results, in this case causing the algorithm performance to be significantly underestimated. We observe significantly more commits with transitions for the flaky executions line: the total number (the maximum value in the graph) is 3.8%, in contrast to 1.4% without flaky executions. We can also observe that the two lines follow different trends; selecting with flaky executions has a logarithmic curve, while the non-flaky case is more linear. Thus, any real RTS algorithm would need to have some mechanism for filtering flaky executions to be effective and evaluated successfully.

IV. Data Collection

Our experiment uses TAP's pre and post-submit testing records, spanning one month. This dataset included more than 500K commits across over 5.5 million distinct targets, resulting in more than 4 billion target executions. As mentioned, TAP performs culprit finding on Pass to Fail transitions, which is reflected in our dataset - there are more accurate records for this type of transition.

To clean our data of flaky results we used several techniques. We first removed from our data any flaky executions identified by TAP (using reruns). We then removed results identified as flaky by Google's flakiness oracle. Both passes discarded less than 0.01% of the total pool of results.

To further clean the data, we additionally removed obviously flaky behavior from our sample by discarding targets displaying rapid and frequent transitions, which were highly unlikely to be related to real break / fix cycles. After analyzing target transition histories against Google flakiness records, we removed targets showing over 14 transitions in the data. This pass removed 51% of our remaining transitions (96% of which were caused by targets with known previous flaky executions), 0.1% of all targets, and 0.08% of results.

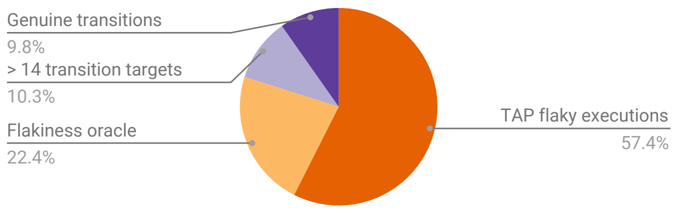

Fig. 2. The sources of transitions in raw test data.

Figure 2 shows the breakdown of transition sources based on these passes. Notably, only 9.8% of our transitions were genuine and remained after we performed the filtering.

V. Experimental Setup

A. Simulation Framework

Our system uses historical data to simulate the behavior of RTS methods on post-submit testing in Google's CI environment. The system simulates the behavior of the studied methods based on the set of affected targets (as identified by TAP) for each commit. At every commit, the implemented methods return an ordering of the affected target set, which is then used to determine which targets would be skipped when choosing to skip varying percentages of targets (the "skip rate"). The skipped tests (or "skip selection") are then analyzed to find whether transitions were missed at the commit.

In this way we evaluate the methods for use as both RTS and TCP techniques; knowing the performance at a skip rate tells us both how well it would perform when skipping that amount of tests for RTS, as well as the performance of detecting transitions earlier in the remaining un-skipped (i.e. prioritized to run) tests for TCP. The skip rate also allows for a fair comparison between algorithms, as they select the same number of targets at every commit for a given rate.

B. Metrics and measurements

To measure how well the studied methods detect transitions, we define the safety of a target, which is whether skipping the target at an affecting commit would have missed a transition. If skipping the target would not miss a transition we say that skipping the target is *safe* (for this commit). If skipping the target at the commit would definitely miss a transition it is *unsafe* to skip. To determine the safety of skipping targets our framework analyzes the chronological sequence of results for every target at each commit.

Due to Google's milestone mechanism, not all targets have a result at each commit. We assume that there is no transition for a target (so safe to skip) at a commit with unknown results if there is no transition between the preceding and subsequent known results. If the commits with known results immediately before and after indicate a transition, we cannot say at what point the transition took place, as it could have been caused by any of the intervening commits. In this case we set the target at all of these commits to a third state: *maybe unsafe*. Maybe unsafe means that skipping the target at the commit could potentially miss a transition.

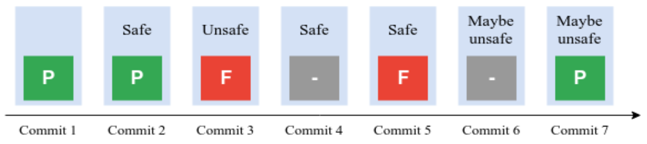

Fig. 3. Sequence of target results and safety across commits

To illustrate these safety states, consider the example sequence of target results in Figure 3. In this example, we have a test target across 7 affecting commits, ordered chronologically. At commit 1, we cannot determine safety without knowing the prior result, so the target at this commit would be removed from our evaluation results. At commit 2, we observe the same result as the previous commit, so consider the target safe to skip. At commit 3, the result differs from the previous commit, so we consider the target unsafe to skip. At commit 4 the target is affected but not run, so we look to the next result at commit 5 and the prior result at commit 3. As they have the same result (Fail) we say that the target is safe to skip at both commit 4 and commit 5. At commit 6, we again have an affected but not run result, but the next commit with a result, 7, has a different result than the previous result at commit 5. Since the transition could have occurred at commit 6 or 7, we say that the target is maybe unsafe to skip for both.

Following on from this, the metric we use to evaluate a skip selection is the safety of the skipped targets. We have two methods for determining the safety of a skip selection, based on the different use cases for RTS algorithms.

If we are interested in identifying all transitioning targets, a commit is safe if it only skips safe to skip targets (and no unsafe or maybe unsafe targets), and thus schedules all transitions. Accordingly, a selection is maybe unsafe if it skipped any maybe unsafe but no unsafe targets, and unsafe if any unsafe targets were skipped. We call this the **All** formulation of safety - where safe means we found *all* transitioning targets.

This analysis is not necessarily suitable for all practical purposes, as in some cases developers are only concerned with the detection of a transition commit, which can then be rolled back, and are not interested in finding all transitioning targets. To deal with this issue, we propose an alternate formulation of safety where a selection is considered safe so long as it doesn't skip all unsafe (or maybe unsafe) targets, and would thus identify a transition at the commit. We call this the **Any** formulation of safety - where safe means we found *any* transitioning target.

C. Dependent and Independent variables

The common independent variable across all results of the studied algorithms is the skip rates considered (from 0-100%). The dependent variable is the performance of the algorithms, measured by the percentage of commits for which the algorithm made a selection of some safety value.

To explore the impact of other parameters, we examine additional independent variables in each set of results. We consider in our first results the safety value being aggregated: safe, unsafe or maybe unsafe. Next, we examine different techniques for determining the safety of a selection - the Any and All formulations. Finally, we consider the algorithm window size, i.e. the length of time before the commit under evaluation the algorithm will consider when looking through the test execution history. All algorithms which use window sizes were run with windows of length $\{1, 2, 4, 8, 12, 24, 48, 96\}$ hours.

D. Technical Implementation

The simulation framework uses a highly parallelized pipeline, implemented using Google's FlumeJava library [19], which is separated into 2 sub-pipelines.

The first pipeline generates safety data by reading Google historical testing data. The first stage groups testing results by test target, resulting in a mapping from target name to all results for the target. In the second stage, each entry in this table is operated on to determine the safety of skipping each target result, using the technique outlined in Section V-B. Finally, the resulting table is then regrouped by commit and written. The result of this first pipeline is thus a table of commit to each affected target, with the safety of the target at the commit. We run this pipeline separately to allow the second pipeline to operate on this data, rather than needing to re-determine target safety values each time it is run.

The second pipeline simulates an RTS algorithm on this safety table. For each entry in the table, it passes the commit and set of affected targets to an algorithm, which returns the targets sorted according to skip ordering based on its implementation. We then operate on this output at every integer skip rate from 0-100%, taking a sublist of the ordered targets to determine which targets the algorithm "skipped" at each percentage. This sublist is analyzed to determine safety against the safety table, and the skipped target selection safety at that commit and skip rate is written. This output thus has the safety of using the RTS algorithm to skip targets for all commits in our records at all skip rates, which can then be aggregated and analyzed.

VI. RTS Algorithms

The goal of our study is to compare several RTS algorithms, based on their ability to schedule the execution of transitioning targets at the code commit causing the transition. We want simple algorithms that are cheap to evaluate at scale and require no expensive code analysis. Our main goal is to develop and utilize a comparison framework and methodology that can ultimately be used to support the development of more complex RTS techniques at Google.

The studied methods implement a function to score a target at a commit, which is then used to sort the affected target list (targets with the same score will be ordered randomly). This list represents the order in which targets will be skipped (low to high scoring targets). The simulator then uses this ordering to determine what is skipped at every skip rate.

For example, at a 50% skip rate it sets the first half of the sorted affected target list as skipped. When we say a target is skipped "after" another target, it means it follows the target in this skip ordering, and thus would not be skipped at lower skip rates that might skip the earlier target.

A. Baseline algorithms

The simplest way to select targets to skip is at random. Although trivial, a random approach gives the average results, giving a natural baseline for experiments.

To identify the boundaries of the examined techniques we consider the optimal and pessimal behavior. The optimal behavior models the best case results (achieved with knowledge of target safety) by skipping all safe to skip targets first, then all maybe unsafe targets and finally all unsafe targets. The pessimal behavior models the worst case performance using the reverse of the optimal order.

B. Transition count

The transition count algorithm is based on the idea that recently transitioning tests are more likely to transition again in the near future. Thus, this approach scores highly targets that transitioned most in their recent execution history. This approach is based on the algorithm introduced by Elbaum et al. [12], which schedules targets which either fail or have no results in prior time windows. We consider this algorithm as it forms the current state-of-the-art, and we believe that previous results from the algorithm were influenced by test flakiness. Intuitively, scheduling frequently failing targets promotes running flaky targets over non-flaky ones, which could significantly skew any algorithm results on raw data. Our analysis leverages the idea of Elbaum et al. and adjusts it to our framework.

Elbaum et al. recorded the results of the targets scheduled by their algorithm at each commit, and used these results when looking for failures or executions at previous commits (in a set failure and execution window). Due to the parallel nature of the framework, we cannot determine algorithm behavior at previous commits. To solve this issue, we model our algorithm's transition detection behavior on Google's milestone mechanism. Before using the algorithm, we simulate our own "milestones" from test result data, set at time intervals separated by a "milestone window" of our choosing. At each milestone, we use the historical record to find the most recent result for all targets prior to the milestone time, and set this as each target's result. If we observe different results to those at the previous milestone for a target, we identify a transition as having occurred at milestone time. The milestone window thus approximates the behavior of an execution window: it allows us to control the sparsity of our transition data as well as setting a guaranteed maximum time between each execution of a target. Since, in our model, milestones would still be part of the new system, we no longer need to force infrequently executed targets to run (the main purpose of an execution window).

Another significant difference from the Elbaum et al. evaluation is that we consider transitions, where Elbaum et al. considered failures. This is because Google developers want to be informed of transitions (i.e. when a target has been broken or fixed by a particular code commit). Therefore we order test targets according to the number of transitions observed at milestones in a set "transition window" before the current affecting commit under consideration. Targets with a higher number of transitions will be skipped after targets with lower.

Algorithm 1 Target scoring: count algorithms

```
1: Parameters: Affected targets $T$, Window $W$, Events $E$
2: ScoreTarget($T_i$)
3: $count = 0$
4: for ($E_i \in E(T_i)$) do
5:     if ($TimeSince(E_i) < W$) then
6:         $count++$
7:     end if
8: end for
9: return $count$
10: }
```

C. Affected count

The affected count algorithm uses the count of times a target has been affected in a prior "affected window" to score targets. At a commit, affected targets with a low count of recent times affected will be skipped before targets with higher counts.

The idea behind this algorithm is that recent frequently affected targets test code that is under active, vigorous development. These tests should be more likely to transition than those which are affected infrequently.

In a previous survey of Google testing, Memon et al. [4] found that files which are modified frequently are more likely to cause test transitions at their commits. In order to keep our algorithm lightweight and free of analysis of code changes we do not consider how frequently the code files are modified. However, at Google changing the same file at different commits will trigger the same tests. As a result, targets which test frequently modified code are frequently affected.

Frequently affected targets are tests that exercise a much larger portion of code than non-frequently affected ones. This is because these tests have a higher chance of being triggered by arbitrary changes than tests exercising lower portions of code. Selecting relevant tests exercising much larger portions of code are of value, as shown by previous research [15], [20].

D. Author count

The author count algorithm is a refinement on the affected count algorithm, which considers the count of distinct authors which have committed a change affecting the target recently.

Memon et al. [4] found that modifying files which have been changed by a larger number of distinct developers are more likely to cause breakages. Similarly to the affected count algorithm, looking at the authors causing a target to become affected should approximate the same behavior.

E. Pre-submit history

Logically, any non-flaky target that was run in pre-submit immediately prior to submission should have the same result in post-submit, as the same code instance is tested in both cases. This means we should be able to skip the execution of targets which were already run in pre-submit.

Using this idea, we use the pre-submit testing results to score affected targets in post-submit. When ordering the set of post-submit affected targets, we choose to first skip any targets which were skipped in pre-submit, as they presumably are still safe to skip in post-submit. Tests in TAP pre-submit may be skipped for a number of reasons and will usually be skipped in post-submit for the same reason.

The next set of targets to be skipped are those run in pre-submit, which had no transition when compared to the last known post-submit result for the target. Third, any targets which were not set as affected in pre-submit due to being outside the commit's projects will be skipped. Finally, those targets run in pre-submit which showed a transition when compared to the previous post-submit result are skipped.

Not all commits in post-submit have test results in pre-submit for a number of reasons. In this case, since we have no signals with which to score the targets, the ordering will be random. This occurred for 39% of our commits.

Algorithm 2 Target scoring: pre-submit history algorithm

```
1: Parameters: Affected targets $T$, Pre-submit results $P$
2: ScoreTarget($T_i$)
3: if ($P(T_i)$ == SKIPPED) then
4:     return 1
5: end if
6: if ($P(T_i)$ == NO TRANSITION) then
7:     return 2
8: end if
9: if ($P(T_i)$ == NOT AFFECTED) then
10:    return 3
11: end if
12: if ($P(T_i)$ == TRANSITION) then
13:    return 4
14: end if
15: }
```

F. Shared directories

To score targets, the shared directory algorithm uses the number of directories in the prefix common to both the affected target's name and the modified file paths from the commit. The precise score we used was the number of shared directories over the total number of directories in the target + modified file path. For this metric, the lowest score is 0, where there are no common directories, and the highest is 0.5, when all directories are common. For a target, we took the highest score when compared with all modified file paths, i.e. the score using the modified file the target should be closest to in the directory structure. Targets with a low shared directory count will be skipped before those with a higher score.

The intuition for this algorithm is that targets closer in the repository should be more relevant to the modified code, and thus would be more likely to be testing code changes and more likely to transition.

It is also more likely that tests closer in the directory structure will be closer to modified files in the dependency graph. Dependency distance has already been shown to be an indicator of test failure at Google [4]. However, calculating dependency distance is too computationally expensive for consideration in this experiment.

Algorithm 3 Target scoring: shared directories algorithm

```
1: Parameters: Affected targets $T$, Modified files $M$
2: ScoreTarget($T_i$)
3: bestScore = 0
4: for ($M_i \in M$) do
5:     $score = CommonFilePrefixDirLength(M_i, T_i) / (NumDirs(T_i) + NumDirs(M_i))$
6:     if ($score > bestScore$) then
7:         $bestScore = score$
8:     end if
9: end for
10: return $bestScore$
11: }
```

VII. Research Questions

Our first aim is to investigate the relative performance of RTS algorithms in the Google CI environment. Therefore, a natural question to ask is:

RQ1: *What is the relative performance of the examined algorithms when evaluated for safety (safe, maybe unsafe, unsafe)?*

In this RQ we evaluate the extent to which the algorithms schedule all non-flaky test transitions. This formulation demonstrates the conservative ability of the methods with regards to all triggered targets.

However, in practice it is possible that commits cause multiple targets to transition. In many cases engineers only need the information that their commit caused a transition and not the information about all transitions caused.

We can thus alternatively consider a selection as safe if we schedule any of the transitioning targets for a particular commit (thus identifying the commit as a transition commit). This leads us to our second RQ:

RQ2: *What is the performance in relation to safety metric formulation (any vs all)?*

Having evaluated the relative performance of the approaches we turn to investigating their sensitivity with respect to the amount of data (i.e. length of time windows) we use as input. Therefore we ask:

RQ3: *What is the performance of the approaches in relation to window size?*

VIII. Results

A. RQ1 relative performance with respect to safety

1) Safe results: Figure 4 depicts the results when aggregating the transition commits (commits which could find a transition by executing affected targets, i.e., with unsafe or maybe unsafe affected targets) with safe skip selections. The skip rate on the x-axis represents the percentage of tests skipped at every commit, which impacts the percentage of transition commits where a safe skip selection occurred on the y-axis. A higher value indicates a better result (more commits with a safe skip selection). All algorithm parameters (window sizes) have been set to produce the optimal observed algorithm performance.

Transition commits make up only 8.3% of all commits in the data, but since safety can be varied only at these commits we solely focus on them. Accordingly, the remaining 91.7% of our commits have all safe targets which don't transition at the commit - implying that we could skip testing at these commits completely and never miss a transition.

From these results we can observe that the random algorithm has nonlinear behavior. This can be explained by the probability distribution of making safe selections. Consider, for example, selecting k targets at a commit with N affected targets, among which n are safe to skip. The number of ways to make a safe selection is the number of ways to select k targets out of the n safe targets - n choose k. Similarly, the number of possible ways to make any selection of size k is N choose k. Thus, the probability P of making a safe selection can be calculated as following:

$$P(\text{select k safely}) = \frac{\binom{n}{k}}{\binom{N}{k}}$$

This binomial probability distribution results in random performing in the non-linear curve shown in the results.

The shared directory algorithm shows poor performance, indistinguishable from random. This is likely due to the fact that it selects tests used during pre-submit testing (which are less likely to transition in post-submit). As tests run in pre-submit are in the same projects as the modified code files, they are likely to be close in the directory structure, and thus selected by the shared directory algorithm. This result is in accordance with the observations made by Memon et al. [4] who found that dependency distances of 5-10 are most likely to transition, indicating that the closest tests are in fact not likely to transition.

pre-submit history and transition count show a small improvement over random, but with similar overall shapes. The pre-submit history is better than transition count at lower rates, while transition count performs better at skip rates above 82%. This indicates that pre-submit is better at skipping safe targets first, while transition count is better at skipping unsafe targets last. One explanation for the disappointing result for the pre-submit history and transition count algorithms is that they have very sparse input data to use when scoring targets. Only 61% of commits had pre-submit results, and only 12.6% of commits

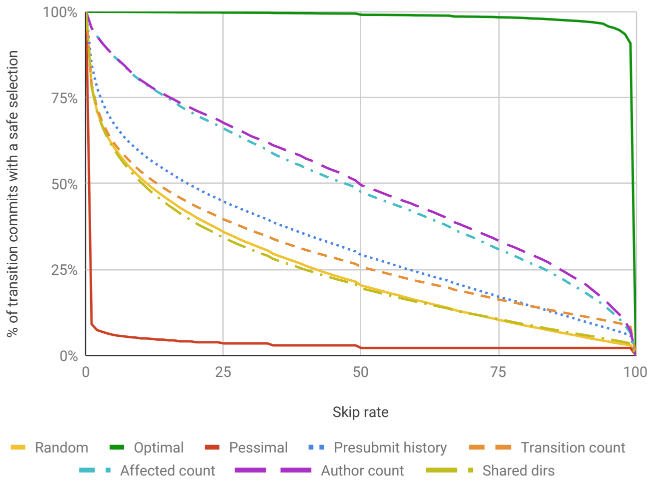

Fig. 4. Algorithm performance in making safe skip selections at commits with transitions.

had any targets with non-zero transition counts (with a 48 hour transition window). The commits with inputs had, on average only 63% of targets with pre-submit results, and 5.6% with non-zero transition count targets. As a result, there is a lot of randomness in the selection order for these algorithms, caused by the high number of target score collisions at commits with small or no input. Additionally, the median number of transitions for a transitioning target is 2, so a majority of these targets only transition once or twice. The transition count algorithm would not be very effective at scheduling these low transition count targets. Nevertheless, both pre-submit and transition count provide clear signals that they improve performance over random when scheduling tests.

Author count is the most effective algorithm, with a slight improvement over affected count. Both do significantly better than random, with a much more linear curve. This is consistent with the previously mentioned findings motivating the algorithms. Additionally, tests are affected very frequently at Google, so for these algorithms there are non-zero inputs for all targets, and scores are more or less evenly distributed over the range of values, resulting in a small degree of randomness in selections.

2) Unsafe Results: Figure 5 depicts the results when aggregating the unsafe commits (commits with unsafe targets) which have unsafe skip selections at different skip rates. Here a lower value indicates a better result (less commits with unsafe selections). Interestingly, we observe different behavior than in the safe results. This can be partially explained by the small amount of data that belongs to this category, as commits with unsafe affected targets only make up 1.4% of all commits.

Transition count is the clear winner here indicating that targets which have transitioned in the recent past are more likely to have future unsafe transitions than the other algorithms. It is unclear why the transition count algorithm works so much better for unsafe results, and requires further investigation.

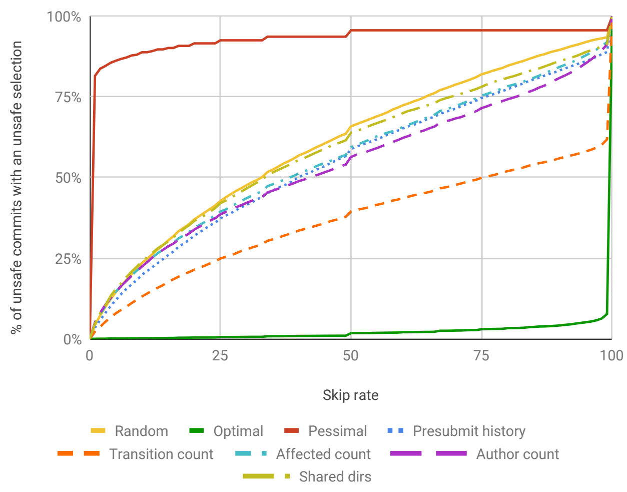

Fig. 5. Algorithm performance in making unsafe skip selections at commits with unsafe targets.

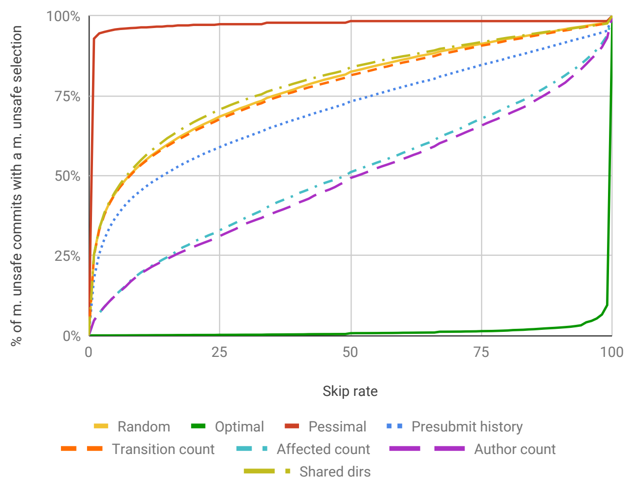

Fig. 6. Algorithm performance in making maybe unsafe skip selections at commits with maybe unsafe targets.

Our other algorithms are much closer in performance here than in the safe results , but show a similar ordering of performances.

3) Maybe Unsafe Results: Figure 6 depicts the results when aggregating the maybe unsafe commits (commits with maybe unsafe but no unsafe targets) which have maybe unsafe skip selections at different skip rates. These maybe unsafe commits make up 6.9% of all commits, substantially more commits than the unsafe commits (1.4%) previously discussed.

Unfortunately, these results are an overestimation, as only one in each group of maybe unsafe commits at a transition is unsafe. In our data, the median number of maybe unsafe commits at a transition was 11, and the mean value 34.55.

Nevertheless, what we observe is that author and affected count algorithms are the most effective. The intuitive explanation is that because these algorithms prioritize running targets with many recently affected commits, they are likely to have some number or affected but not run commits between any transitions in the targets they run. Being frequently affected also means that for any maybe unsafe transitions they run, there will be a large number of maybe unsafe commits for the targets in between the commits with results that will be run.

The pre-submit history and shared directory performance is consistent with their ordering in safe and unsafe results, however transition count performs significantly worse than in the unsafe results.

4) General Outcomes: The results for the shared directory algorithm are uniformly indistinguishable to random, and give no benefit. All other algorithms outperform random but with varying performances. The pre-submit history algorithm is clearly an indicator of transitions, but is either similar to or worse than both the affected count and author count algorithms in all safety cases. Our author count algorithm, followed closely by affected count, performed by far the best in safe and maybe unsafe cases.

Interestingly, the transition count algorithm performs the same as random for maybe unsafe, but much better than other algorithms when considering unsafe commits. Why this inconsistency in results occurs is unclear, but it does indicate which algorithms would perform better in each type of CI environment. For an environment where we know exact transitions and have results at every commit, we would expect results similar to the unsafe case which only considers commits where we know the exact transition triggering commit. For this environment we would expect the transition count algorithm to perform best. For an environment like Google, where we do not schedule all targets and there is thus some ambiguity in which commit precisely caused a transition, we would value running the target at any of the intervening commits and thus expect performance similar to a combination of maybe unsafe and unsafe, for which we would expect the author and affected counts to win (as there are substantially more maybe unsafe than unsafe commits).

B. RQ2 relative performance with any transition

Our analysis up to this point has considered safety with respect to all transitions caused by a commit - the All formulation of safety, where safe means scheduling all unsafe targets. As discussed, this analysis is not appropriate for purposes where we want to find transitioning commits rather than targets, resulting in the development of the any formulation of safety, where safe means scheduling any unsafe target.

Figure 7 presents the safe Any formulation results. The results are consistent with those we obtained for the all formulation case, i.e., 4, with the author and affected count algorithms the most effective, and the shared directory the least effective. There is a shift of the curves towards the optimal behavior (as expected), which makes the differences between the methods smaller, but this does not seem to change any of the already observed trends.

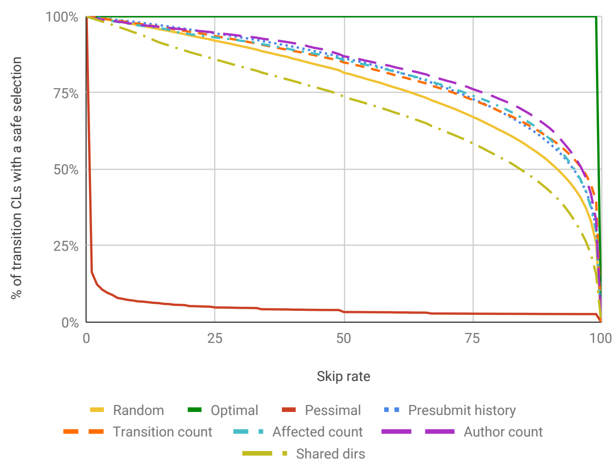

Fig. 7. Algorithm performance in making safe skip selections at transition commits, when using the “Any” formulation to evaluate safety.

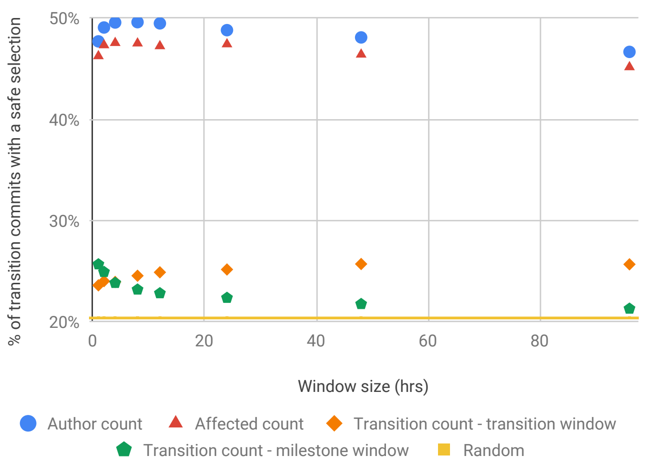

Fig. 8. Algorithm performance in making safe skip selections at a 50% skip rate, varying the size of the algorithm windows.

C. *RQ3 sensitivity to time window*

To investigate the sensitivity of the algorithms to the time windows we used, we experimented with different window sizes. Figure 8 shows our results with respect to safe skip selections at a fixed 50% skip rate.

We observe that the affected and author count algorithms perform best with a 4 hour window, indicating after this period the frequently affected targets begin to change and the old data becomes stale, worsening performance.

For the transition count algorithm, performance peaks at a 48 hour transition window. The milestone window leads to worse performance with each increase from 1 hour. This behavior is to be expected, as at higher transition windows we have more transition data, and with more frequent milestones we find more transitions. However smaller milestone windows require more test executions and have a higher compute cost.

Of course different windows involve different amounts of data to be stored. Nevertheless, at the optimal identified cases the amount of data to be stored should not cause any problems.

IX. Threats to Validity

One issue with our data is that, due to milestone scheduling, TAP doesn’t run tests at every affecting commit. As a result, a large number of our commits have targets with an unknown status - in only 23% of cases affected targets had a definitive (Pass / Fail) result. Thus, we make simplifying assumptions to evaluate the target’s safety at commits with no result, as outlined in section V-B.

There is a possibility that the data is impacted by target flakiness despite our filtering techniques. There is also a risk that we have removed genuine transitions by removing targets with more than 14 transitions. We developed these flakiness filtering techniques to minimize the risk of removing genuine transitions while allowing us to remove highly flaky targets. We only remove a very small fraction of targets during this process (0.1%).

It possible that these results will not be generalizable to other testing pools and CI environments. Google’s TAP has several distinctive features such as milestone scheduling and using pre-submit and post-submit testing, and Google’s repository has a uniquely huge test pool and extremely rapid rate of code change. This could lead to changes in the results when compared to more traditional, simple CI environments and test automation methods.

Finally, there is a potential threat that undetected errors in data collection, evaluation system, or algorithm implementations we developed could influence the results.

X. Related Work

There is a large body of research investigating all aspects of regression testing [14]. Most work in this area explicitly targets the regression test selection [21], [22] and prioritization [15], [20] problems for traditional release-based software development. Here we briefly discuss work related to CI-based regression testing.

Elbaum et al. [12] introduced a CI-based prioritization method at Google, using test results in recent execution history as a signal. As previously discussed, Elbaum et al. prioritizes targets that either failed or have no execution in a dynamic execution window. We examined this method, adapted to suit our framework (the transition count algorithm).

Liang et al. [13], [23] showed that prioritizing commits instead of individual targets provides opportunities for significant execution improvements. Unfortunately, performing such an analysis in Google is complicated and difficult, as it requires redesigning the CI environment and dealing with code dependencies. Nevertheless, such a scheme provides promising avenues for future research.

Busjaeger and Xie [24] and Marijan et al. [25], respectively, suggested prioritizing tests based on several characteristics of the affected tests (coverage, execution similarity, content similarity, failure history, test age) and based on test failures encountered in the recent execution history. Again, our transition count algorithm examines the same principles as failure history. Unfortunately more complex metrics, such as test coverage, execution similarity, etc. were too expensive and time-intensive to collect for the scope of our experiment.

Memon et al. [4], studied test characteristics linked to test failure at Google. Their ultimate goal was to improve understanding of test selection and test failures. We have designed our algorithms to leverage the findings of this work. We go a step further by defining and evaluating selection schemes.

Zhu et al. [26], proposed re-prioritizing tests after each test run using a conditional failure probability among tests (probability of a test failure given the test execution result of another). Their results showed an improvement over the approach of Elbaum et al. [12]. Unfortunately, their evaluation was based on test failures and old Google data, making it incompatible with our analysis and results. However, as this approach provides an alternative way of treating historical data, we will consider it in our future schemes.

XI. Conclusion

In this paper we identified the practical requirements - data collection and analytical techniques, methods for handling flaky tests and a transition-based evaluation method - needed for analysis when designing and evaluating regression testing techniques. Based on these, we simulate five RTS algorithms, using real operational data from Google. We found that three algorithms - the number of times a test has been triggered, the number of authors committing code that triggers a test, and the recent execution history - provide the strongest signals towards test selection, although the problem remains open for future research.

In order for practitioners to benefit from analyzing RTS techniques it is vital that they record test results and have robust mechanisms for excluding flakiness. Identifying transitions as soon as possible after commit is the main goal of CI related activities; CI should inform developers as soon as possible when committed changes break and fix builds.

Our simulation framework forms the first step towards large-scale regression test selection evaluation in CI. We anticipate that future research will lead to new RTS techniques, and will dig deeper into the causal relationship between algorithms and results.

XII. Acknowledgments

The authors would like to thank Prof. Sebastian Elbaum and John Penix for their comments during the earlier versions of the paper. Prof. Le Traon and Dr. Papadakis are supported by the Luxembourg National Research Funds (FNR) INTER/MOBILITY/16/11457617/Serval@Berkeley and C17/IS/11686509/CODEMATES.

References

[1] M. Fowler, "Continuous integration," https://martinfowler.com/articles/continuousIntegration.html, online; accessed 10 August 2018.

[2] 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*, 2016, pp. 11–20.

[3] M. Harman and P. O'Hearn, "From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis," in *SCAM*, 2018.

[4] A. M. Memon, Z. Gao, B. N. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, "Taming google-scale continuous testing," in *ICSE-SEIP*, 2017, pp. 233–242.

[5] "Neflix ci," https://medium.com/netflix-techblog/towards-true-continuous-integration-distributed-repositories-and-dependencies-2a2e3108c051, online; accessed 10 August 2018.

[6] 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*, 2017, pp. 821–830.

[7] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, and D. Dig, "Trade-offs in continuous integration: assurance, security, and flexibility," in *ESEC/FSE*, 2017, pp. 197–207.

[8] "Travis ci," http://travis-ci.org, online; accessed 10 August 2018.

[9] "Jenkins," https://wiki.Jenkins-ci.org/display/JEKINS/Home/, online; accessed 10 August 2018.

[10] A. Çelik, A. Knaust, A. Milicevic, and M. Gligoric, "Build system with lazy retrieval for java projects," in *FSE*, 2016, pp. 643–654.

[11] A. Gambi, R. Zabolotnyi, and S. Dustdar, "Poster: Improving cloud-based continuous integration environments," in *ICSE*, 2015, pp. 797–798.

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

[13] J. Liang, S. G. Elbaum, and G. Rothermel, "Redefining prioritization: continuous prioritization for continuous integration," in *ICSE*, 2018, pp. 688–698.

[14] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Softw. Test., Verif. Reliab.*, vol. 22, no. 2, pp. 67–120, 2012.

[15] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y. L. Traon, "Comparing white-box and black-box test prioritization," in *ICSE*, 2016, pp. 523–534.

[16] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. L. Traon, "Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines," *IEEE Trans. Software Eng.*, vol. 40, no. 7, pp. 650–670, 2014.

[17] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *FSE*, 2014, pp. 643–653.

[18] J. Micco and A. Memon, "Gtac 2016, test flakiness google," https://docs.google.com/presentation/d/1iVf-TogkdoIcvs8OpRMMWx76s9Zk4_f0JJ-e1sZIxog/edit#slide=id.p659, online; accessed 10 September 2018.

[19] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. Henry, R. Bradshaw, and Nathan, "Flumejava: Easy, efficient data-parallel pipelines," in *PLDI*, 2010, pp. 363–375.

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

[21] T. L. Graves, M. J. Harrold, J. Kim, A. A. Porter, and G. Rothermel, "An empirical study of regression test selection techiques," *ACM Trans. Softw. Eng. Methodol.*, vol. 10, no. 2, pp. 184–208, 2001.

[22] L. Zhang, "Hybrid regression test selection," in *ICSE*, 2018, pp. 199–209.

[23] J. Liang, "Cost-effective techniques for continuous integration testing," Master's thesis, University of Nebraska-Lincoln, Lincoln, Nebraska, United States, 2018.

[24] B. Busjaeger and T. Xie, "Learning for test prioritization: an industrial case study," in *FSE*, 2016, pp. 975–980.

[25] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige, "Reinforcement learning for automatic test case prioritization and selection in continuous integration," in *Software Engineering und Software Management 2018, Fachtagung des GI-Fachbereichs Softwaretechnik, SE 2018, 5.-9. März 2018, Ulm, Germany.*, 2018, pp. 75–76.

[26] Y. Zhu, E. Shihab, and P. C. Rigby, "Test re-prioritization in continuous testing environments," in *ICSME'18*, 2018.