Techniques for Improving Regression Testing in Continuous Integration Development Environments

Sebastian Elbaum[†], Gregg Rothermel[†], John Penix[‡]

[†]University of Nebraska - Lincoln
Lincoln, NE, USA
{elbaum, grother}@cse.unl.edu

[‡]Google, Inc.
Mountain View, CA, USA
jpenix@google.com

ABSTRACT

In continuous integration development environments, software engineers frequently integrate new or changed code with the mainline codebase. This can reduce the amount of code rework that is needed as systems evolve and speed up development time. While continuous integration processes traditionally require that extensive testing be performed following the actual submission of code to the codebase, it is also important to ensure that enough testing is performed prior to code submission to avoid breaking builds and delaying the fast feedback that makes continuous integration desirable. In this work, we present algorithms that make continuous integration processes more cost-effective. In an initial *pre-submit* phase of testing, developers specify modules to be tested, and we use regression test selection techniques to select a subset of the test suites for those modules that render that phase more cost-effective. In a subsequent *post-submit* phase of testing, where dependent modules as well as changed modules are tested, we use test case prioritization techniques to ensure that failures are reported more quickly. In both cases, the techniques we utilize are novel, involving algorithms that are relatively inexpensive and do not rely on code coverage information – two requirements for conducting testing cost-effectively in this context. To evaluate our approach, we conducted an empirical study on a large data set from Google that we make publicly available. The results of our study show that our selection and prioritization techniques can each lead to cost-effectiveness improvements in the continuous integration process.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools, Tracing*

General Terms

Reliability, Experimentation

Keywords

Continuous Integration, Regression Testing, Regression Test Selection, Test Case Prioritization

FSE '14, November 16–22, 2014, Hong Kong, China
Copyright 2014 ACM 978-1-4503-3056-5/14/11 ...$10.00.

1. INTRODUCTION

In continuous integration development environments, engineers merge code that is under development or maintenance with the mainline codebase at frequent time intervals [8, 13]. Merged code is then regression tested to help ensure that the codebase remains stable and that continuing engineering efforts can be performed more reliably. This approach is advantageous because it can reduce the amount of code rework that is needed in later phases of development, and speed up overall development time. As a result, increasingly, organizations that create software are using *continuous integration processes* to improve their product development, and tools for supporting these processes are increasingly common (e.g., [3, 17, 29]).

There are several challenges inherent in supporting continuous integration development environments. Developers must conform to the expectation that they will commit changes frequently, typically at a minimum of once per day, but usually more often. Version control systems must support atomic commits, in which sets of related changes are treated as a single commit operation, to prevent builds from being attempted on partial commits. Testing must be automated and test infrastructure must be robust enough to continue to function in the presence of significant levels of code churn.

The challenges facing continuous integration do not end there. In continuous integration development environments, testing must be conducted cost-effectively, and organizations can address this need in various ways. When code submitted to the codebase needs to be tested, organizations can restrict the test cases that must be executed to those that are most capable of providing useful information by, for example, using dependency analysis to determine test cases that are potentially affected by changes. By structuring builds in such a way that they can be performed concurrently on different portions of a system's code (e.g., modules or sub-systems), organizations can focus on testing changes related to specific portions of code and then conduct the testing of those portions in parallel. In this paper, we refer to this type of testing, performed following code submission, as "post-submit testing".

Post-submit testing can still be inordinately expensive, because it focuses on large amounts of dependent code. It can also involve relatively large numbers of other modules which, themselves, are undergoing code churn, and may fail to function for reasons other than those involving integration with modules of direct concern. Organizations interested in continuous integration have thus learned, and other researchers have noted [35], that it is essential that developers first test their code prior to submission, to detect as many integration errors as possible before they can enter the codebase, break builds, and delay the fast feedback that makes continuous integration desirable. Organizations thus require developers to conduct some initial testing of new or changed modules *prior* to submitting those modules to the codebase and subsequent post-submit testing [13]. In this paper, we refer to this pre-submission phase of testing as "pre-submit testing".

Despite efforts such as the foregoing, continuous integration development environments face further problems-of-scale as systems grow larger and engineering teams make changes more frequently. Even when organizations utilize farms of servers to run tests in parallel, or execute tests "in the cloud", test suites tend to expand to utilize all available resources, and then continue to expand beyond that. Developers, preferring not to break builds, may over-rely on pre-submit testing, causing that testing to consume excessive resources and become a bottleneck. At the same time, test suites requiring execution during more rigorous post-submit testing may expand, requiring substantial machine resources to perform testing quickly enough to be useful for the developers.

In response to these problems, we have been investigating strategies for applying regression testing in continuous integration development environments more cost-effectively. We have created an approach that relies on two well-researched tactics for improving the cost-effectiveness of regression testing – regression test selection (RTS) and test case prioritization (TCP). RTS techniques attempt to select test cases that are important to execute, and TCP techniques attempt to put test cases in useful orders (Section 2 provides details). In the continuous integration context, however, traditional RTS and TCP techniques are difficult to apply. Traditional techniques tend to rely on code instrumentation and be applicable only to discrete, complete sets of test cases. In continuous integration, however, testing requests arrive at frequent intervals, rendering techniques that require significant analysis time overly expensive, and rendering techniques that must be applied to complete sets of test cases overly constrained. Furthermore, the degree of code churn caused by continuous integration quickly renders data gathered by code instrumentation imprecise or even obsolete.

In this work, therefore, we utilize new RTS and TCP techniques. We have created RTS techniques that use time windows to track how recently test suites[1] have been executed and revealed failures, to select test suites to be applied during pre-submit testing. We then utilize TCP techniques based on such windows to prioritize test suites that must be performed during subsequent post-submit testing. The approaches we utilize can be applied to individual test suites as their execution is requested. They also utilize relatively lightweight analysis, and do not require code instrumentation, rendering them appropriate for use within the continuous integration process.

To evaluate our approach, we conducted an empirical study on a large data set obtained from Google – a data set that allows us to simulate and track the results of various regression testing approaches on a substantial codebase. The results of our study show that our RTS technique can greatly improve the cost-effectiveness of pre-submit testing; that is, it produced large reductions in test case execution costs, while deferring relatively few faults to post-submit testing. We also find that our TCP technique can substantially reduce the time required to provide feedback on failing test cases during post-submit testing, allowing developers to address problems more quickly. Thus, overall, our techniques contribute directly to the goals of the continuous integration process.

In summary, the contributions of this work are:

- A characterization of how regression testing is performed in a continuous integration development environment and the identification of key challenges faced by that practice that cannot be addressed with existing techniques.
- A definition and assessment of new regression testing techniques tailored to operate in continuous integration development environments. The insight underlying these techniques is that past test results can serve as lightweight and effective predictors of future test results.
- A sanitized dataset collected at Google, who supported and helped sponsor this effort, containing over 3.5M records of test suite executions. This dataset provides a glimpse into what fast and large scale software development environments are, and also lets us assess the proposed techniques while Google is investigating how to integrate them into their overall development workflow.

The remainder of this paper is organized as follows. Section 2 provides background and related work. Section 3 discusses the continuous integration process used at Google, and the Google dataset we use in the evaluation. Section 4 presents our RTS and TCP techniques. Section 5 presents the design and results of our study. Section 6 discusses the findings and Section 7 concludes.

2. BACKGROUND AND RELATED WORK

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 reuse more cost-effective via regression test selection (e.g., [4, 22, 24, 31]) and test case prioritization (e.g., [7, 10, 25, 27, 28, 34]). Yoo and Harman [33] provide a recent survey.

Regression test selection (RTS) techniques select, from test suite T, a subset T' that contains test cases that are important to re-run. When certain conditions are met, RTS techniques can be *safe*; i.e., they will not omit test cases which, if executed on P', would reveal faults in P' due to code modifications [23].

Test case prioritization (TCP) techniques reorder the test cases in T such that testing objectives can be met more quickly. One potential objective involves revealing faults, and TCP techniques have been shown to be capable of revealing faults more quickly [11].

Because TCP techniques do not themselves discard test cases, they can avoid the drawbacks that can occur when regression test selection cannot achieve safety. Alternatively, in cases where discarding test cases is acceptable, test case prioritization can be used in conjunction with regression test selection to prioritize the test cases in the selected test suite. Further, test case prioritization can increase the likelihood that, if regression testing activities are unexpectedly terminated, testing time will have been spent more beneficially than if test cases were not prioritized.

A key insight behind most of the RTS and TCP techniques studied to date is that certain testing-related tasks (such as gathering code coverage data) can be performed in the "preliminary period" of testing, before changes to a new version are complete. The information derived from these tasks can then be used during the "critical period" of testing after changes are complete and when time is more limited. This insight, however, applies only in cases where sufficiently long preliminary periods are available, and this is not typically the case in continuous integration development environments.

[1] Traditionally, RTS and TCP techniques have been applied to test cases. In this work we apply them to test suites, primarily because the datasets we use to study our approach include test suites, and analysis at the test suite level is more efficient. The approaches could also, however, be performed at the level of test cases.

While most work on regression testing has focused on development processes that fit the foregoing description, there has been some work considering more incremental processes. Do et al. [6] study test case prioritization in the presence of time constraints such as those that arise when faster development-and-test cycles are used. Walcott et al. [30], Zhang et al. [36], and Alspaugh et al. [1] present prioritization techniques that operate in the presence of time constraints. This work, however, does not specifically consider continuous integration processes.

There has been some recent work on techniques for testing programs on large farms of test servers or in the cloud (e.g., [5, 20, 28]). This work, however, does not specifically consider continuous integration processes or regression testing.

A few pieces of work do address continuous integration and testing. Saff and Ernst [26] consider a form of continuous testing in which regression tests are run continuously as developers write code. This work, however, focuses on testing during the development effort itself, not at the testing periods typically utilized in continuous integration processes. Jiang et al. [18] consider continuous integration environments, and discuss the use of test case prioritization following code commits to help organizations reveal failures faster, but their work focuses on the ability to use the failures thus revealed in statistical fault localization techniques. Kim and Porter [19] use history data on test cases to prioritize them, as do we, but they do not consider the continuous integration process. In the work most closely related to ours, Marijan et al. [21] briefly present a prioritization algorithm for continuous regression testing. The algorithm also utilizes the time since the last test cases failed, but does not consider the notions of an execution or a prioritization window, and it assumes a limit on time allotted for test execution.

In the work most closely related to ours, Yoo et al. [35], also working with Google data sets, describe a search-based approach for using TCP techniques to help developers perform pre-submit testing more cost-effectively. Their study of the approach, while modest in scale, does suggest that it can function cost-effectively in the Google context. They do not, however, consider the use of RTS techniques, or consider the application of their technique to post-submit testing.

3. REGRESSION TESTING AT GOOGLE

Continuous integration development environments are employed by various organizations in various forms. Because we are familiar with the use of continuous integration at Google, and because we have access to a large collection of data relevant to the processes employed there, we describe the ways in which continuous integration is used there, in relation to other important aspects of testing. Note that our description, which is based on materials available at [15], is necessarily somewhat simplified, but is sufficient to support the presentation of our approach and of our empirical study. In Section 6, we provide further discussion of several complexities that arise when continuous integration is employed in practice.

Google's developers create test suites utilizing specialized XUnit-like frameworks (e.g., the Google C++ Testing Framework [14]), and extensive execution and monitoring support. This holds for test cases at the unit, integration, and system levels. In this context, test cases are composed of both inputs and oracles, and thus most test execution and oracle checking is automated. (There are exceptions such as tests that require human judgment, but they are relatively infrequent.) This enables the execution of test suites on a common test suite execution infrastructure that can leverage the massive computational power of server farms or the cloud to obtain test results faster.

While developers create the test suites used at Google, an independent Test Infrastructure Team manages the infrastructure used in testing, and takes steps necessary to support fully automated test execution. One element of this effort involves keeping track of dependencies between code modules;[2] this supports efforts to focus testing resources on changed and dependent modules during post-submit testing (as suggested in Section 1) [16]. Given the scale of testing requirements, Google's test execution infrastructure operates at the level of entire test suites.

Google's process includes both pre-submit and post-submit testing as described in Section 1. When a developer completes their coding activities on a module M, the developer presents M for pre-submit testing. In this phase, the developer provides a *change list* that indicates modules *directly relevant* to building or testing M. Pre-submit testing requests are queued for processing and the test infrastructure performs them as resources become available; this testing includes, to different extents, the execution of all test suites relevant to all of the modules listed in the change list. If failures are encountered, the developer is informed and must correct the faults responsible and repeat the process.

When pre-submit testing succeeds for M, a developer submits M to source code control, a process that causes it to be considered for post-submit testing. At this point, tools are used to determine, using module dependency graphs, modules that are *globally relevant* to M. This includes modules on which M depends as well as modules that depend on M. (The dependence analysis relied on in this step is coarse, and the dependencies calculated often form an over-approximation – this is necessary in order to ensure that the process is sufficiently fast [16].) All of the test suites relevant to these modules are queued for processing.

The foregoing processes also apply to new code. New modules are submitted for pre-submit testing with change lists along with new test suites. New modules are then submitted to source code control and subsequent post-submit testing, where their dependence information is calculated and used to conduct testing of dependent modules.

The Google Shared Dataset of Test Suite Results. In conducting this work, we have been able not only to use real industry data, but also to sanitize and package a large dataset of test suite execution results, that Google has willingly made available for use by the software engineering research community [12]. It is this data that we rely on in evaluating the techniques presented in this paper, and because this data also helps illustrate the continuous integration process, we present basic information on it here.

The Google Shared Dataset of Test Suite Results (GSDTSR) contains information on a sample of over 3.5 million test suite executions, gathered over a period of 30 days, applied to a sample of Google products. The dataset includes information such as anonymized test suite identifiers, change requests, phases (pre-submit and post-submit) at which test suites were executed, outcome statuses of test suite executions, and time required to execute test suites. (Note, however, that the dataset contains only a small sample of observations relative to to the entire pantheon of observations that could be accrued at Google.)

Table 1 provides data on the test suites in GSDTSR, focusing on the relative sizes of test suites utilized in the two phases, the percentages of these test suites that have been observed across each phase and size, and the percentages of test suite executions that

[2] We use the term "code modules" in this paper for simplicity, to refer to any element of the codebase that can be modified and passed to a build process, as well as larger code units for which the build processes involve multiple independently built modules.

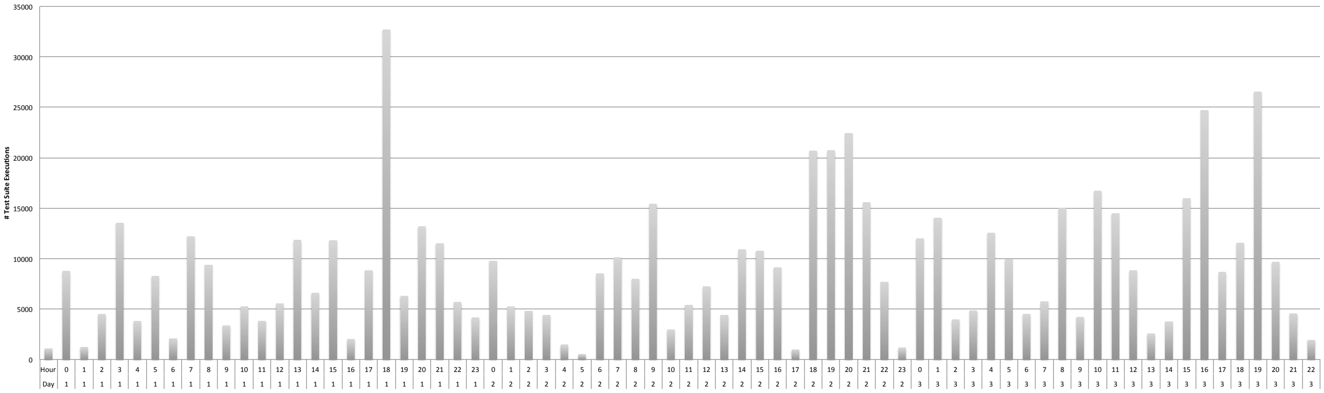

Figure 1: Flow of Incoming Test Suites for Pre-Submit Testing over Three Days.

failed. Size classifications of test suites are meant to serve as qualitative indicators of their scope and of the resources required to execute them [32]. In general, "Small" test suites are those that focus on a class, "Medium" test suites are those that target a class and its neighboring classes, and "Large" test suites are those that most closely represent field scenarios. Additionally, with each size there are also rules that enforce duration and resource usage limits.

Table 1: Google Dataset: Distribution of Test Suite Executions per Testing Phase and Test Suite Size

Phase	Size	% Total	% Failing
pre-submit	Small	26	0.02
pre-submit	Medium	10	0.13
pre-submit	Large	6	0.27
post-submit	Small	35	0.01
post-submit	Medium	13	0.32
post-submit	Large	10	0.65

Table 2 provides a different view of the test suites in GSDTSR, showing the percentages of test suites of different sizes that exist for each of the major programming languages utilized. This table illustrates the diversity of languages that a large continuous building and testing process must accommodate, and how the language choice varies depending on the testing scope.

Table 2: Google Dataset: Distribution of Test Suite Executions per Language across Test Suite Sizes

Language	% Total	Small	Medium	Large
Config	0.1	0	0.3	0.1
C	39.2	33.7	66.8	20.9
GO	0.4	0.2	0.9	0
GWT	0	0	0	0.2
Java	7.2	6	7.1	11.5
Python	7.7	7.8	9.8	4.7
Shell	41.9	52.1	14.1	42.9
WEB	3.6	0.1	1.1	19.6

To further illustrate the type of information present in GSDTSR, as well as to further illustrate the effects that continuous integration has on test suite execution at Google, Figure 1 presents data on test suites executed during pre-submit testing over a three day period. The horizontal axis depicts hourly time periods (labels correspond GMT); the vertical axis depicts the number of test suite executions occurring. Note that requests for test executions arrive continuously – this is because engineers are working in various locations around the globe. The heaviest periods of test execution, however, occur in the time periods corresponding to lunch time in Mountain View, California, where engineers tend to submit modules for testing prior to taking their breaks.

4. APPROACH

The foregoing process, employed in a continuous integration context, has many advantages. Frequent system builds and regression testing of new and changed modules ensure faster feedback on potential problems. Post-submit testing reduces the number of problems that slip through into the codebase and affect future builds, while pre-submit testing helps prevent an excess of problems during post-submit testing.

The process also faces other challenges. As noted in Section 1, these include developers listing too many or too few modules on change lists, resulting in bottlenecks in testing, or excessive faults being left undetected until post-submit testing. Furthermore, even when executing test suites on larger server farms or in the cloud, the rapid pace at which code is changed and submitted for testing in either phase, and the sheer numbers of changes being made by developers, can lead to bottlenecks in either testing phase.

We now describe two approaches that we use to address these challenges, involving regression test selection (RTS) and test case prioritization (TCP) techniques.

4.1 Continuous Regression Test Selection

A natural question that arises in the context of pre-submit testing is whether RTS techniques can be used to select a subset of the test suites related to the modules listed in a change list, rendering that testing more cost-effective without unduly harming the overall (pre-submit together with post-submit) testing process.

As noted in Section 2, a wide range of RTS techniques have been developed and studied, and could potentially be applied in this context. In practice, however, existing techniques will not suffice. Google's codebase undergoes tens of changes per minute [16]. Most existing RTS techniques utilize instrumentation to track the code executed by test cases, and then analyze code changes and relate them to these test executions. The rate of code churn in the Google codebase, however, is quite large, and this can cause code instrumentation results to quickly become inaccurate. In such situations, keeping coverage data up to date is not feasible [9].

We therefore sought an alternative approach. It has long been suggested, in the testing literature, that some test cases (or test suites) are inherently better than others at revealing failures [19]. In an evolving system, test suites that have failed in a recent version are in some ways "proxies" for code change – they target code that is churning. We conjectured that an RTS approach that selects test suites based on some "failure window" might be cost-effective in pre-submit testing.

The foregoing approach also has the effect of ignoring test suites that have not, in recent builds, revealed faults. We conjecture that doing this should have relatively minor deleterious effects initially, but in time, as test suites are repeatedly ignored, the effectiveness of regression testing might be reduced. Thus, a second issue to consider when selecting test suites for execution in pre-submit testing involves the use of some "execution window", where test suites not executed within that window are also selected.

A third issue in regression testing relates not to existing test suites, but to new test suites created to exercise new or modified system code or functionality. Such test suites are clear candidates for execution in pre-submit testing on a modified system version.

The foregoing issues provide motivation for Algorithm 1, SelectPRETests. To help explain the algorithm we provide Figure 2. In the figure, each rectangle represents a set of test suites that have been submitted for execution. The grey rectangle in the figure represents a set of test suites at the point at which they are about to be considered for execution. Rectangles to the right of the grey rectangle represent sets of test suites that have been executed in the past, from most recently executed to least recently moving from left to right. Rectangles to the left of the grey rectangle represent sets of test suites that have been queued for future execution, from most recently queued to least recently moving from left to right. (Test suites queued for future execution are not considered in Algorithm 1, we include them in the figure because we consider them in our second algorithm, discussed in Section 4.2.) Let W_f be a failure window and let W_e be an execution window, as discussed above. In practice, W_f and W_e can indicate lengths of time, or numbers of test suites executed – in this work we rely on the former. The figure depicts W_f and W_e projected over the sets of test suites that have been executed in the past.

Let T be the set of test suites for modules submitted by a developer or required by a developer, at the point at which they are about to be considered (again, test suites represented by the grey rectangle in Figure 2). SelectPRETests selects from T, for execution, all test suites that (1) have been observed to fail within window W_f, (2) have not been executed within window W_e, or (3) are new. Note that in the case of Algorithm 1, failure and execution windows for test suites are assessed relative to both prior pre-submit and prior post-submit testing results.

Algorithm 1 SelectPRETests

```
Parameters:
    Test Suites $T$,
    Failure window $W_f$,
    Execution window $W_e$
for all $T_i \in T$ do
    if $TimeSinceLastFailure(T_i) \leq W_f$ or
            $TimeSinceLastExecution(T_i) > W_e$ or
            $T_i$ is new then
        $T' \leftarrow T' \cup T_i$
    end if
end for
return $T'$
```

4.2 Continuous Test Suite Prioritization

Algorithm SelectPRETests is meant to reduce testing effort in pre-submit testing, while preserving much of its effectiveness, and the extent to which it may do so needs to be studied empirically. However, SelectPRETests can have a secondary effect: it can shift the execution of failing test suites from pre-submit testing to post-submit testing. Skipping passing test suites during pre-submit testing does not cause an extra load during post-submit testing as they will be executed then anyway. Test suites that would have failed during pre-submit testing are skipped, however, can cause delays in failure finding that may slow down development.

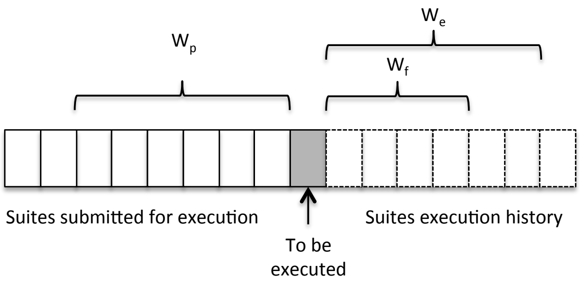

Figure 2: Test Suite Execution Windows

Thus, we wished to find an approach by which to reduce such delays in post-submit testing. To do this, we looked to test prioritization. Just as test suites selected for pre-submit testing can be selected based on failure and execution windows, so also the test suites that must be utilized in post-submit testing can be prioritized based on those windows. We conjecture that by using this approach, test suites that reveal failures can be executed earlier than they otherwise would. Engineers can then use information on failures to determine whether to continue a system build, or to return to the drawing board.

In the context of prioritization, however, there is one additional issue to consider. Prioritization techniques have traditionally operated on entire sets of test cases or test suites. In continuous integration development environments, test suites arrive for post-submit testing continuously, in relatively small batches or in bursts related to code submits, and these test suites are placed in a dispatch queue. (In Figure 2, this queue is represented by all of the rectangles to the left of, and including, the grey rectangle). As resources become available, test infrastructure removes test suites from this queue and launches them.

Assigning priorities to test suites or small batches of suites as they reach the dispatch queue may cause lower-priority test suites to remain unexecuted for longer than desirable periods of time. It may also cause test suites that have relatively similar behavior to all be assigned high priorities and run earlier than other test suites with dissimilar behavior, lowering the effectiveness of prioritization. Finally, prioritizing test suites as they arrive is likely to focus on test suites related to single submissions. We conjecture that a vast majority of the test suites related to specific submissions do *not* fail (since they were already validated by pre-submit testing), and grouping all of the test suites related to a single submission ahead of those related to later submissions may have the effect of queuing a number of likely-to-pass test suites ahead of more-likely-to-fail test suites.

To address these issues, as well as to gather data that might help us evaluate whether the foregoing effects can be significant in practice, we rely on an additional form of window, referred to as a *prioritization window* (W_p), and illustrated in Figure 2. Like W_e and W_f, W_p can either be a number of test suites, or a time – in this work we utilize time. As test suites arrive they are placed in the dispatch queue, and when W_p has been exceeded, we prioritize the test suites in the queue that have not yet been prioritized. In this approach, setting W_p to 1 (if W_p is a number of test suites) or to a

to pick up everything
at least once, cf.
other paper

time window less than that required by any test suite (if W_p is an amount of time) is equivalent to not prioritizing.

Algorithm 2, PrioritizePOSTTests, performs the foregoing actions. PrioritizePOSTTests is triggered when prioritization window W_p has been met, and operates on the set of test suites that has been submitted for post-submit testing, denoted here by $POSTQueue$. (Recall that these test suites include test suites for modules submitted by developers to source code control, along with test suites for all modules assessed to depend on, or be depended on by, the submitted modules.) Again, let W_f be a failure window and let W_e be an execution window. Let P_p be a pointer to the last test suite in the queue that was prioritized during the prior prioritization window.

Algorithm 2 PrioritizePOSTTests

```
Parameters:
    POSTQueue,
    Failure window $W_f$,
    Execution window $W_e$,
    Starting point $P_p$ in POSTQueue
for all $T_i \in$ POSTQueue after $P_p$ to lastEntry.POSTQueue do
    if $TimeSinceLastFailure(T_i) \leq W_f$ or
            $TimeSinceLastExecution(T_i) > W_e$ or
            $T_i$ is new then
        $T_i$.Priority $\leftarrow 1$
    else
        $T_i$.Priority $\leftarrow 2$
    end if

end for
SORTBYPRIORITY(POSTQueue, $P_p$, lastEntry.POSTQueue)
$P_p$ = lastEntry.POSTQueue
```

PrioritizePOSTTests begins at location P_p in POSTQueue. The algorithm considers each test suite T_i in the queue, from the suite immediately after P_p to the end of the queue, and assigns a priority to each of these test suites. Priority assignment involves applying a function that seeks to capture the relationship between the test suite and W_e and W_f. There are various functions that might be utilized, in our current implementation, we use a simple and easy-to-compute function that assigns one of two priority levels, 1 (high) or 2 (low), to test suites, as follows. For each test suite T that (1) has been observed to fail within window W_f, (2) has not been executed within window W_e, or (3) is new, T is assigned priority level 1. All other test suites are assigned priority level 2. Note that in the case of PrioritizePOSTTests, we consider failure and execution data relative only to prior executions in post-submit testing, for reasons discussed in Section 6. After all test suites have been assigned priority levels, the algorithm sorts them such that higher priority test suites appear first in the queue (beginning at point P_p).[3] Finally, P_p is updated to the last prioritized test suite in the queue.

Note that, as defined, PrioritizePOSTTests assigns test suites to priority levels in the order in which it extracts them from POSTQueue. In theory, a secondary prioritization criterion might be applied to the suites assigned to the two queues, at additional costs. For example, traditional coverage-based prioritization techniques often randomly sort test cases that are judged to have equivalent priorities; other approaches apply secondary characteristics of tests. In this work, we employ no such secondary criterion; thus, the orders in which test cases are executed, within each priority level, are fully deterministic.

There are several factors that may affect the cost-effectiveness of the foregoing approach. We have already discussed a few issues related to the choice of W_p that suggest the use of a window size promoting the prioritization of "batches" of test suites, rather than of single test suites. Given that choice, however, other issues arise. First, the rate at which test requests arrive, and the expense of executing test suites, play a role, because together they determine the rate at which test suites accumulate and can be optimally prioritized. Second, the resources available for test execution (i.e., machines on which to run test suites) also play a role. For example, if the number of machines available for testing exceeds W_p, then partitioning W_p test suites into high or low priority suites is not likely to have any effect on testing outcomes, since all test suites may then be executed at once. Third, the relationship between W_p, W_f and W_e may play a role: if W_p is too close to W_f or W_e, then prioritization may miss important information. We explore several of these issues further in our empirical study; the other factors merit further study as well.

5. EMPIRICAL STUDY

We wish to evaluate the cost-effectiveness of our approach, and also to assess the effects that result from the use of different window sizes. To do this we consider the two components of the approach (RTS and TCP) independently; this is reasonable because each of these components is used in a separate phase of testing, and in a practical setting, an organization might choose to employ either or both, depending on cost-effectiveness. As baseline approaches we consider the case in which RTS and TCP techniques are not applied. This yields the following research questions.

RQ1: How cost-effective is the RTS technique during pre-submit testing, and how does its cost-effectiveness vary with different settings of W_f and W_e?

RQ2: How cost-effective is the TCP technique during post-submit testing, and how does its cost-effectiveness vary with different settings of W_p?

5.1 Objects of Analysis

As objects of analysis, we use the Google Shared Dataset of Test Suite Results (GSDTSR) described in Section 3. This dataset contains a sample of the results of test suites executed by Google. The dataset is provided as comma delimited file. The dataset includes a summary of the data and a per field description [12].

5.2 Variables and Measures

5.2.1 Independent Variables

Our independent variables involve the techniques and windows used. For RQ1 we employ three techniques: the technique presented in Section 4.1, which selects a subset of the test suites in the change list for a module M, a baseline approach in which all test suites in the change list are utilized, and a random RTS approach that selects a percentage of test suites matching that of the proposed approach to enable their comparison (averaged over 5 runs). We utilize three execution window sizes, $W_e = \{1, 24, 48\}$, and nine failure window sizes, $W_f = \{0.25, 0.5, 1, 2, 4, 12, 24, 48, 96\}$, each representing different numbers of hours.

For RQ2, we employ two techniques: the technique presented in Section 4.2 and a baseline approach that does not prioritize test suite execution. We arbitrarily fix $W_f = 12$ and $W_e = 24$, the median values for those windows, and we explore seven values for the prioritization window, representing different numbers of hours: $W_p = \{0.1, 0.5, 1, 2, 4, 8, 12\}$.

[3] To simplify the presentation, we assume that access to the POSTQueue or queue section to be sorted is atomic.

5.2.2 Dependent Variables

As dependent variables, for RQ1, we measure the percentages of test suites that are selected, the percentage of execution time required, and the percentages of failures detected by our technique, relative to the percentages required or detected by the baseline technique. We do this for each combination of W_e and W_f.

For RQ2, we measure the time it takes for the test suites to exhibit a failure. Note that this measure differs from the APFD measure typically used in studies of test case prioritization. APFD considers cumulative fault detection over time, which is reasonable in a situation in which batches of test cases are being employed, but not as applicable in continuous integration environments where the focus is on obtaining feedback on individual test suites.

5.3 Study Operation

To study the proposed techniques we implemented the algorithms described in Section 4, using approximately 300 lines of Python. We used the GSDTSR dataset to simulate a continuous testing environment. The simulation performs a walk through the data file, assuming that the test suites are executed by the computing infrastructure at the time, for the duration, and with the result reported in the dataset.

The SelectPRETests implementation utilizes the GSDTSR data, a failure window size, and an execution window size, and reports the number of test suites selected, the time required to execute those suites, and the number of failures they detected. It does this by reading each line in GSDTSR, determining whether the test suite in the line would be executed given the failure and execution windows, and updating the the latest failure and execution information for the test suite. If the test suite is to be executed, the implementation updates the test suite counter, the test suite execution time accumulator (with the time recorded in the line for that test suite), and (if the test suite resulted in a failure) the failure accumulator.

PrioritizePOSTTests takes the same inputs plus a prioritization window, and produces a list of test suites prioritized within each specified window size. Our implementation of the technique operates similar to our implementation of SelectPRETests, except for the manipulation of a moving prioritization window.

Note that we apply regression test selection only to test suite executions related to pre-submit testing and prioritization only to data related to test suite executions related to post-submit testing (see Table 1 for information on the sizes of those data sets).

5.4 Threats to Validity

Where external validity is concerned, we have applied our techniques to an extensive dataset, but that dataset represents testing processes conducted only in a small section of one industrial setting. We have compared our techniques to baseline approaches in which no RTS or TCP techniques are used, and compared our RTS technique to an approach using random test selection, but we have not considered other alternative RTS or TCP techniques from the literature (although most would need significant changes to work under the continuous integration settings we are intending to operate). We have utilized various window sizes, but have necessarily limited our choices to a finite set of possible sizes. We have not considered factors related to the availability of computing infrastructure, such as variance in numbers of platforms available for use in testing. These threats must be addressed through further study.

Where internal validity is concerned, faults in the tools used to simulate our techniques on the GSDTSR objects could cause problems in our results. To guard against these, we carefully tested our tools against small portions of the dataset, on which results could be verified. Further, we have not considered possible variations in testing results that may occur when test results are inconsistent (see the discussion of "Flaky Test Suites" in Section 6); such variations, if present in large numbers, could potentially alter our results.

Where construct validity is concerned, our measures include costs in terms of numbers of test suites executed and testing time, and effectiveness in terms of numbers of failing test suites detected. Other factors, such as whether the failure is new, costs in engineer time, and costs of delaying fault detection are not considered, and may be relevant.

5.5 Results and Analysis

We now analyze the results of our study relative to each of our research questions. Section 6 provides further discussion.

5.5.1 RQ1: Regression Test Selection

Results of applying the continuous RTS technique for the three W_e sizes are depicted in Figures 3, 4, and 5. Each figure shows, for each W_f (on the x-axis), the percentage of test suites selected, their execution time, and the failures detected (which corresponds to the set of failing test suites selected) compared with the baseline case in which all test suites are run.

Across all three figures we observe similar trends. The numbers of failing suites increase rapidly initially as W_f increases, and then begin to level off at around $W_f = 12$. At that point the percentage of failing test suites selected is at least three times greater than the percentage of test suites selected for execution and the percentage of test suite execution time. The gains, as measured by the detection of failing test suites, reach between 70% and 80% when $W_f = 96$. The other 20% to 30% of failing test suites were not selected for execution by our algorithm as they were executed recently but did not have a recent failure history. Random test suite selection with the same number of test suites selected for execution performed approximately six times worse in terms of failure detection. This illustrates the ability of continuous RTS to substantially reduce the testing load in pre-submit testing, at the cost of delaying the execution of a certain number (but much smaller percentage, relative to the costs saved) of failing test suites to post-submit testing.

As expected, using larger values of W_e led to more aggressive test suite selection. For example, for a failure window of $W_f = 96$ hours, the percentage of selected test suites decreases by almost 20% from $W_e = 1$ in Figure 3 to $W_e = 48$ in Figure 5. This is because as W_e increases, the number of non-failing test suites that fall within that window (and thus, that are *not* selected), increases.

Note that for lower values of W_f, the percentage of test suites executed is greater than the percentage of time used for test execution, but this changes as W_f increases. For example, for $W_e = 1$, for values of W_f less than 12, the percentage of test suites executed is smaller than the percentage of time used, but above 12 the percentage of time is greater. The primary reason for this is that larger test suites tend to have greater failure detection rates than smaller test suites (see Table 1), and as W_f increases, greater numbers of larger test suites are selected.

Overall, skipping tests during pre-submit testing ends up imposing only a small additional load on post-submit testing. Assuming that test suite execution failures rates are 0.5% (as in our dataset), selecting 33% of the test suites (the maximum in our simulation, for $W_e = 1$ and $W_f = 96$) implies that the test executions added to post-submit testing form less than 0.17% of the pre-submit testing load, and even a smaller percentage of the post-submit testing load. However, delaying the execution of test suites that will fail will cause an increase in the feedback latency to developers. This is the challenge we address with prioritization within post-submit testing, in the next section.

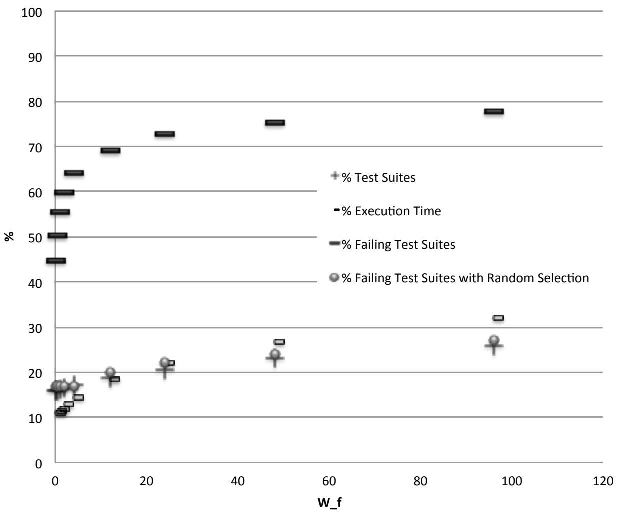

Figure 3: Test Suite Selection: $W_e = 1$

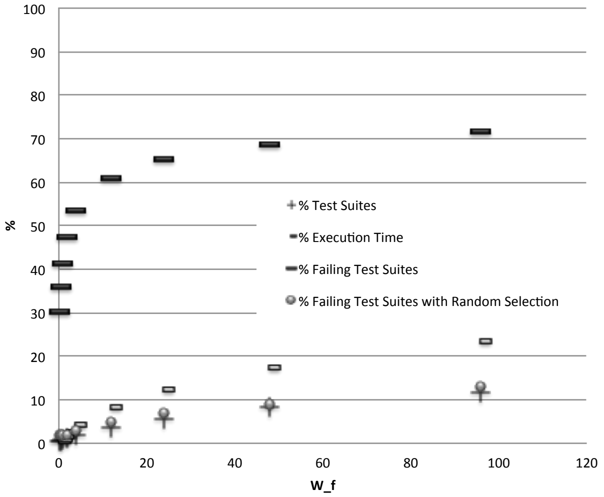

Figure 4: Test Suite Selection: $W_e = 24$

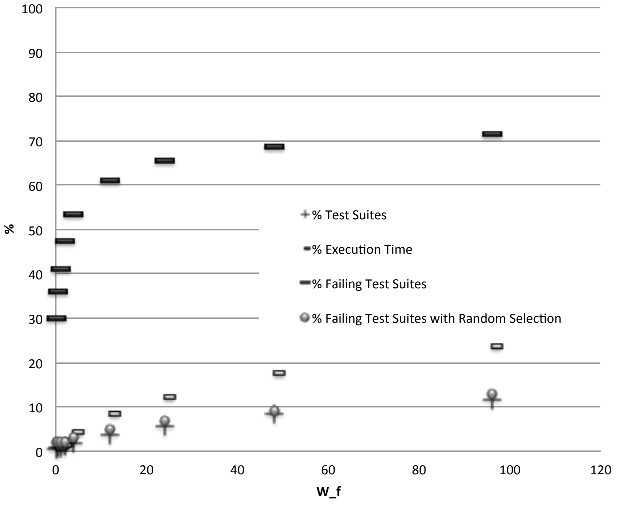

Figure 5: Test Suite Selection: $W_e = 48$

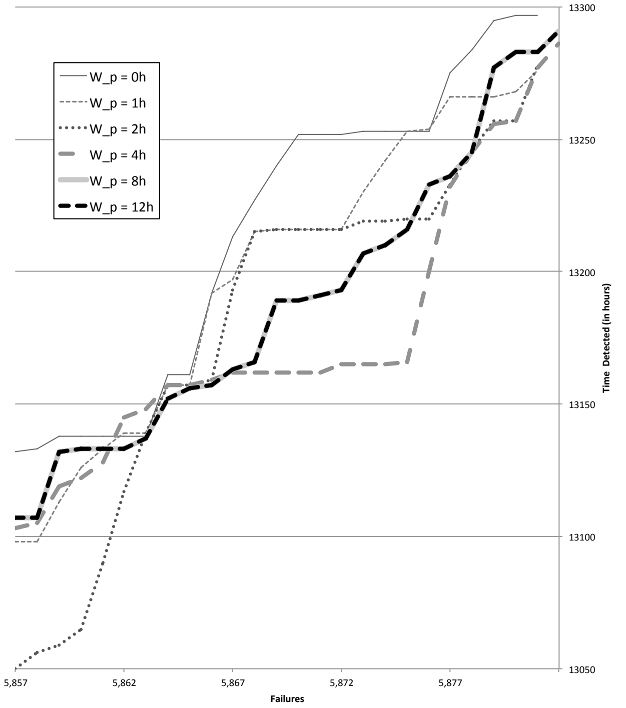

Figure 6: Sample of prioritization results for 25 failures

5.5.2 RQ2: Test Case Prioritization

Figure 6 presents a small sample of the results of applying test suite prioritization with windows $W_f = 12$, $W_e = 24$, and $W_p = \{0.1, 0.5, 1, 2, 4, 8, 12\}$. The x-axis corresponds to 25 of the more than 5000 failing test suites, and the y-axis denotes the times at which these test suites were executed. Thus, for example, the first failure shown, number 5,857, is detected at hour 13,050 of testing when test suites are prioritized with $W_p = 2$, and at hour 13,130 of testing when test suites are not prioritized ($W_p = 0$). Although all instances of prioritization techniques appear to perform better than no-prioritization, this sample illustrates a fact evident across the entire data set, namely, that there is large variance in prioritization technique performance across window sizes. For some failures, this difference can be measured in tens of hours.

To understand the trends in the data given this high level of variation, for each test suite execution resulting in a failure, we computed the differences in detection time between each instantiation of the prioritization technique and no-prioritization. This is equivalent to the vertical distance, in Figure 6, between the value of each technique and $W_p = 0$, at each failure point (for example, for failure 5,857, the difference in detection time when $W_p = 2$ and $W_p = 0$ is 80 hours). The results of this analysis are shown in Figure 7. We use a box-plot to show the distribution of the differences. In the figure, a positive difference indicates that prioritization outperformed the no-prioritization baseline. The x-axis corresponds to the different values of W_p, and for each of these we show the corresponding median (thick black line within a box, with a number next to it), the first and third quartiles (box bounds), the min and max (whiskers), and the outliers (black circles).

Overall, we observe that all instantiations of the prioritization technique perform better than no-prioritization. Increasing W_p led to a greater number of requested test suite executions marked as high priority (ranging from 5% to 8%) because larger windows

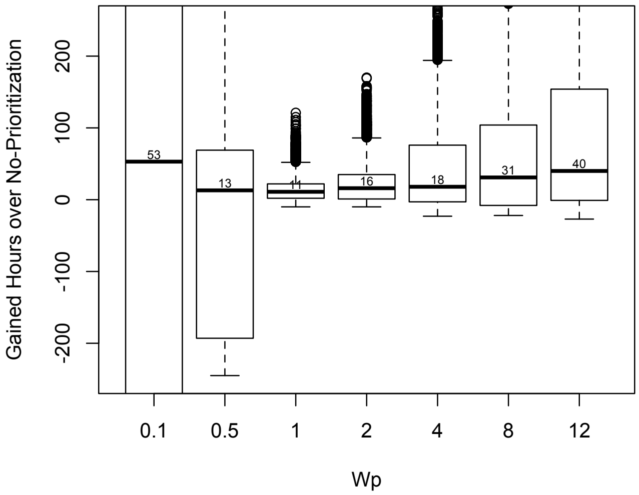

Figure 7: Boxplots comparing prioritization techniques with $W_p = \{0.1, 0.5, 1, 2, 4, 8, 12\}$ against no-prioritization

meant more stale data. The techniques with the best median performance, however, were also those that exhibited the greatest variation. For example, for the smallest window, $W_p = 0.1$, the median was 53 hours better than the median for no-prioritization, but the second quartile begins at -1644 hours. The largest window of $W_p = 12$ hours had a median of 40 hours but the third quartile ended at 151 hours. The prioritization windows in between $W_p = 1$ and $W_p = 12$ result in smaller medians and variances. For $W_p = 1$, the median is the lowest at 11 hours and the second and third quartiles end at 2 and 22, respectively. We conjecture that given an overly small prioritization window, the prioritization algorithm operates with only very recent data. In some cases, as when there are many instances of the same test suite awaiting execution in the pending queue, this may be detrimental as the prediction signal may be overly strong. On the other hand, given too large a prioritization window, the prioritization algorithm may not have enough recent test execution data to make good predictions about which tests suites should be given higher priority.

6. DISCUSSION

In this section we summarize our findings, and discuss several additional issues related to RTS and TCP techniques that must be considered when operating in large and continuous integration development environments.

Summary and Implications. Our results show that our continuous RTS technique can lower the testing load in pre-submit testing. In doing so it can delay the detection of faults, but it does so at a far lower rate than its rate of savings in terms of testing execution cost. Since delayed test suites are eventually executed during post-submit testing, delayed faults are still detected later; thus, this process offers potential cost-benefit tradeoffs that can be beneficial when pre-submit testing is acting as a bottleneck. Our results also show that our continuous TCP technique can reduce delays in fault detection during post-submit testing. Utilizing both techniques together during continuous integration, then, can improve the cost-effectiveness of the continuous integration process overall.

Our results also show that selections of window sizes W_e, W_f and W_p affect the effectiveness of our algorithms. Organizations will need to investigate the effects of choices of these parameters in their particular process settings in order to determine what values to use. We also believe, however, that the parameters can be dynamically tuned based on data gathered while testing is ongoing – a process that fits well with the notion of continuous integration. Further research is needed, however, to determine how such dynamic adjustments could be performed.

Integration of approaches at Google. Google, who supported and partially sponsored this effort, is investigating how to integrate the proposed techniques into their overall development workflow. One challenge in that context is that, in spite of using the same testing infrastructure, different teams have different tolerances for delaying failure detection in pre-submit testing, or for changing the test suite execution order in post-submit testing, due to their specific testing and release practices. The proposed approaches simplify the problem by assuming that a single set of parameters is sufficient, but clearly more sophisticated levels of configuration are necessary.

Reducing Test Execution Time vs. Delaying Failure Detection. One of the impacts of utilizing continuous RTS in pre-submit testing is the potential delay in detecting failing test suites until post-submit testing. In practice, this chance of delay may be compensated for by faster pre-submit test execution; however, organizations may wish to assess the tradeoffs more quantitatively.

To obtain an initial notion about what this delay may involve, we further analyzed our data. Note that the cost of the delay depends in part on the latency between pre-submit and post-submit testing. For passing pre-submit test suites, the average latency between pre-submit and post-submit testing in the best case (comparing only the last test suite execution in pre-submit testing to the first in post-submit testing) is 1.2 hours, and in the worst case (comparing only the last test suite execution in pre-submit testing to the last in post-submit testing) is 2.5 hours. Clearly, this is just a coarse approximation of the costs of delays in fault detection, and one that accounts only for time. Future studies could evaluate this cost more carefully, perhaps by employing RTS for only some of the systems under test, while collecting not just failure result data but also failure detection latency.

Specifying Windows using Time versus Numbers of Events. As noted in Section 4, our RTS and TCP techniques can be used with window sizes specified in terms of time or numbers of events, where an event can be, for example, a test suite execution. Our experience indicates that specifying windows in terms of time is more intuitive for managers of test execution pipelines, and this is one reason we chose to use time. However, using numbers of events to define window sizes would allow our techniques to adapt more readily to fluctuations in the inflow of test execution requests.

We conjecture that such adaptability may be desirable in some instances but not others. For example, setting an event-based window may be beneficial if test suites fail Friday afternoon when the inflow of test suites slows down, such that the failing test suites should be considered high priority on Monday morning even though there is a large window of time in between. On the other hand, when the inflow of test suites is high, an event-based window may cause test suites utilized just minutes (but many hundreds of test suites) ago to be discarded. We leave the comparison of different specification mechanisms as future work.

Adding Pre-Submit Test Data to Post-Submit Prioritization. In our study we used failure and execution data from previous post-submit test suite executions to drive current post-submit test suite prioritization. One could also consider including failure and execution data from pre-submit test executions to prioritize post-submit testing. We investigated this possibility on our data set, and discovered that the addition of pre-submit test data did not improve prioritization. For example, when prioritizing with $W_f = 12$, $W_e = 24$, and $W_p = 1$ hours, the median gain in fault detection time for prioritization with respect to no-prioritization was 11 with a variance of 17 whether pre-submit test data was included or not. For the same setting but with $W_p = 12$, the median gain in fault detection time when using just post-submit test data was 40 and the variance was 120, and when adding pre-submit test data the median was 34 and the variance was 118. We conjecture that the reason for this lack of additional effectiveness may involve the different roles that pre-submit test data plays in development. For example, it is relatively common for developers to use pre-submit testing to explore the robustness of a certain piece of code. Failure and execution data collected through such explorations can mislead the prioritization technique during post-submit testing, resulting in mixed effectiveness gains.

Running on a Large Computing Infrastructure. When applied at large scales, continuous integration processes often rely on large computing infrastructure to increase execution efficiency, reducing the amount of time required to provide feedback on test execution to developers. The RTS and TCP techniques we propose, however, are not cognizant of the capabilities of or the variability that can be introduced by the underlying computing infrastructure. This means that, for example, for a fixed W_f, a test suite that has previously failed may or may not be selected for execution depending on how quickly the infrastructure can schedule it to run.

The parameters provided by our techniques in terms of failure and executions windows can help to compensate for some of the variation in the computing infrastructure. Still, more resource-cognizant techniques could identify ways to cluster and allocate test suites more effectively. There is a clear a tension between designing techniques that are general enough to be applicable to many settings, and specialized enough to leverage the available computational resources. Development organizations operating with such infrastructure must deal with such decisions frequently.

Evolving Test Suites. Test suites are not stagnant, they evolve with code. Our algorithms give new test suites higher value, but they are not cognizant of whether a test suite is obsolete or has changed. Obsolete failing test suites are eventually removed or deprecated by developers so we estimate that their influence in the overall test suite selection or prioritization performance is going to be steady but rather small. Obsolete passing suites will also have a steady effect on the algorithms depending primarily on the value of W_e, but they do not seem to provide a clear and safe signal for our approach to discard them automatically. Changed test suites, however, may offer a refinement opportunity for our approach. Considering that changes in test suites may be meant to validate new and potentially faulty program behavior, and changes in suites may be faulty or at least inconsistent with the software under test, it seems that the act of changing in itself may be a valuable signal for triggering test suite execution. Hence, a natural step in the evolution of the proposed algorithms will be to consider the notion of a "test suite change window".

Dealing with Non-Deterministic Test Suites. Another interesting side-effect of continuously running large numbers of test suites is that there may be large numbers of test suites for which results are inconsistent; these are often also referred to as "flaky test suites". Flaky test suites may fail intermittently without any variations occurring in the module under test or in the test suite itself. In some cases, the cause for this lies in the test suite design, due to failures to check assumptions or control environmental factors. In other cases, these variations may be so rare and difficult to control that it may not be cost-effective for developers to address them. For our RTS and TCP techniques to be effective, they must distinguish between true failures and flaky failures, or risk the chance of actually magnifying the effect of flaky test suites by promoting their execution. One common practice is to "deflake" failing test suites by rerunning them (e.g., [2]) and computing a "flakiness index" for each test suite in the process. We anticipate that this index could be used as a threshold to determine whether a failing test suite should be considered as such in a W_f.

7. CONCLUSION

Just as software integration has become a continuous process, so has the testing performed during that process. Integration and testing are increasingly intertwined as software moves closer to deployment. This is why, to be cost-effective, regression testing techniques must operate effectively within continuous integration development environments. In this work we described the challenges associated with performing regression testing under continuous integration, introduced two new regression testing techniques that use readily available test suite execution history data to determine what tests are worth executing and executing with higher priority, and we shared a sanitized industry dataset with millions of test suite executions that we used to assess the techniques presented.

We anticipate that follow up work will include the refinement of RTS techniques under continuous integration and testing, particularly through the incorporation of other light-weight sources of data. For example, we are exploring whether the transitions from a pass to a failure during test suite execution may indicate a new breakage (instead of ongoing broken modules) as this differentiation may trigger different follow-up processes and toolsets. Similarly, we would like to explore mechanisms for incorporating a test suite "changed" window and also for adjusting window sizes dynamically so that we can consider more of the potential performance factors we have identified. We will also extend our simulation to use multiple test execution resources that better match the "cloud" setting used to obtain the dataset. Last, we wish to extend our studies to other datasets and contexts to better understand the limitations of RTS techniques as the scale of code and change increases.

8. ACKNOWLEDGMENTS

This work has been supported in part by the Air Force Office of Scientific Research through award FA9550-10-1-0406, and by National Science Foundation Award #1218265. Sebastian Elbaum was a visiting scientist at Google when part of this work was performed. We are thankful to the developers at Google who support the continuous testing infrastructure, and helped us better understand the challenges in continuous integration and testing.

9. REFERENCES

[1] S. Alspaugh, K. Walcott, M. Belanich, G. Kapfhammer, and M. Soffa. Efficient time-aware prioritization with knapsack solvers. In *Proceedings of the ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies*, pages 17–31, November 2007.

[2] Android api reference. http://developer.android.com/reference/android/test /FlakyTest.html, 2014.

[3] Atlassian. Atlassian software systems: Bamboo. https://www.atlassian.com/software/bamboo, 2014.

[4] D. Binkley. Semantics guided regression test cost reduction. *IEEE Transactions on Software Engineering*, 23(8), Aug. 1997.

[5] S. Bucur, V. Ureche, C. Zamfir, and G. Candea. Parallel symbolic execution for automated real-world software testing. In *Proceedings of the Sixth Conference on Computer Systems*, pages 183–198, 2011.

[6] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel. The effects of time constraints on test case prioritization: A series of empirical studies. *IEEE Transactions on Software Engineering*, 36(5), Sept/Oct 2010.

[7] H. Do and G. Rothermel. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Transactions on Software Engineering*, 32(9), Sept. 2006.

[8] P. M. Duvall, S. Matyas, and A. Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Pearson Education, 2007.

[9] S. Elbaum, D. Gable, and G. Rothermel. The impact of software evolution on code coverage. In *Proceedings of the International Conference on Software Maintenance*, pages 169–179, Nov. 2001.

[10] S. Elbaum, A. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proceedings of the International Conference on Software Engineering*, pages 329–338, May 2001.

[11] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2), Feb. 2002.

[12] S. Elbaum, J. Penix, and A. McLaughlin. Google shared dataset of test suite results. https://code.google.com/p/google-shared-dataset-of-test-suite-results/, 2014.

[13] M. Fowler. Continuous integration. martinflowler.com/articles/continuousIntegration.html, 2014.

[14] Google c++ testing framework. https://code.google.com/p/googletest/, 2014.

[15] Google engineering tools. http://googletesting.blogspot.com, 2014.

[16] P. Gupta, M. Ivey, and J. Penix. Testing at the speed and scale of google. http://googletesting.blogspot.com/2014/01/the-google-test-and-development_21.html, 2014.

[17] Jenkins. Jenkins: An extendable open source continous integration server. jenkins-ci.org, 2014.

[18] B. Jiang, Z. Zhang, T. H. Tse, and T. Y. Chen. How well do test case prioritization techniques support statistical fault localization. In *Proceedings of the Computer Software and Applications Conference*, pages 99–106, July 2009.

[19] J.-M. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource-constrained environments. In *Proceedings of the International Conference on Software Engineering*, pages 119–129, May 2002.

[20] Y. Kim, M. Kim, and G. Rothermel. A scalable distributed concolic testing approach: An empirical evaluation. In *Proceedings of the International Conference on Software Testing*, pages 340–349, Apr. 2012.

[21] D. Marijan, A. Gotlieb, and S. Sen. Test case prioritization for continuous regression testing: An industrial case study. In *Proceedings of the International Conference on Software Maintenance*, pages 540–543, Sept. 2013.

[22] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2004.

[23] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8), Aug. 1996.

[24] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2), Apr. 1997.

[25] G. Rothermel, R. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10), Oct. 2001.

[26] D. Saff and M. Ernst. An experimental evaluation of continuous testing during development. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 76–85, July 2004.

[27] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2002.

[28] M. Staats, P. Loyola, and G. Rothermel. Oracle-centric test case prioritization. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 311 – 320, Nov. 2012.

[29] ThoughtWorks. Go: Continous delivery. www.thoughtworks.com/products/go-continuous-delivery, 2014.

[30] A. Walcott, M. Soffa, G. Kapfhammer, and R. Roos. Time-aware test suite prioritization. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 1–12, July 2006.

[31] L. White and B. Robinson. Industrial real-time regression testing and analysis using firewalls. In *Proceedings of the International Conference on Software Maintenance*, Sept. 2004.

[32] N. Whittaker. How google tests software - part five. http://googletesting.blogspot.com/2011/03/how-google-tests-software-part-five.html.

[33] S. Yoo and M. Harman. Regression testing minimisation, selection and prioritisation: A survey. *Software Testing, Verification and Reliability*, 22(2), 2012.

[34] S. Yoo, M. Harman, P. Tonella, and A. Susi. Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, pages 201–212, 2009.

[35] S. Yoo, R. Nilsson, and M. Harman. Faster fault finding at Google using multi objective regression test optimisation. In *Proceedings of the International Symposium on Foundations of Software Engineering, Industry Track*, Sept. 2011.

[36] L. Zhang, S.-S. Hou, C. Guo, T. Xie, and H. Mei. Time-aware test-case prioritization using integer linear programming. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 213–224, July 2009.