Redefining Prioritization: Continuous Prioritization for Continuous Integration

Jingjing Liang
University of Nebraska - Lincoln
jliang@cse.unl.edu

Sebastian Elbaum
University of Nebraska - Lincoln
elbaum@cse.unl.edu

Gregg Rothermel
University of Nebraska - Lincoln
grother@cse.unl.edu

ABSTRACT

Continuous integration (CI) development environments allow software engineers to frequently integrate and test their code. While CI environments provide advantages, they also utilize non-trivial amounts of time and resources. To address this issue, researchers have adapted techniques for test case prioritization (TCP) to CI environments. To date, however, the techniques considered have operated on test suites, and have not achieved substantial improvements. Moreover, they can be inappropriate to apply when system build costs are high. In this work we explore an alternative: prioritization of *commits*. We use a lightweight approach based on test suite failure and execution history that is highly efficient; our approach "continuously" prioritizes commits that are waiting for execution in response to the arrival of each new commit and the completion of each previously scheduled commit. We have evaluated our approach on three non-trivial CI data sets. Our results show that our approach can be more effective than prior techniques.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; **Empirical software validation**;

KEYWORDS

continuous integration, regression testing, large scale testing

ACM Reference Format:
Jingjing Liang, Sebastian Elbaum, and Gregg Rothermel. 2018. Redefining Prioritization: Continuous Prioritization for Continuous Integration. In *ICSE '18: ICSE '18: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3180155.3180213

1 INTRODUCTION

Continuous integration (CI) environments automate the process of building and testing software, allowing engineers to merge code with the mainline code base at frequent time intervals. Companies like Google [32], Facebook [15, 39], Microsoft [14], and Amazon [43] have adopted CI and its ability to better match the speed and scale of their development efforts. The usage of CI has also dramatically increased in open source projects [22], facilitated in part by the availability of rich CI frameworks (e.g., [3, 24, 44, 45]).

CI environments do, however, face challenges. System builds in these environments are stunningly frequent; Amazon engineers have been reported to conduct 136,000 system deployments per day [43], averaging one every 12 seconds [33]. Frequent system builds and testing runs can require non-trivial amounts of time and resources [7, 16, 22]. For example, it is reported that at Google, "developers must wait 45 minutes to 9 hours to receive testing results" [32]; and this occurs even though massive parallelism is available. For reasons such as these, researchers have begun to address issues relevant to the costs of CI, including costs of building systems in the CI context [7], costs of initializing and reconfiguring test machines [16], and costs of test execution [13, 32, 40, 52].

Regression Testing Challenges in CI. Where testing in CI development environments is concerned, researchers have investigated strategies for applying regression testing more cost-effectively. In particular, researchers [6, 13, 25, 30–32, 52] have considered techniques (created prior to the advent of CI) that utilize regression test selection (RTS) (e.g., [9, 17, 21, 29, 34, 35, 37, 41, 49, 50]) and test case prioritization (TCP) (e.g., [2, 10, 20, 30, 38, 51]). RTS techniques select test cases that are important to execute, and TCP techniques arrange test cases in orders that allow faults to be detected earlier in testing, providing faster feedback to developers.

In CI environments, traditional RTS and TCP techniques can be difficult to apply. A key insight behind most traditional techniques is that testing-related tasks such as gathering code coverage data and performing program analyses 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 when sufficiently long preliminary periods are available, and this is not typical in CI environments. Instead, in CI environments, test suites arrive continuously in streams as developers perform commits. Prioritizing individual test cases is not feasible in such cases due to the volume of information and the amount of analysis required. For this reason, RTS and TCP techniques for CI environments have typically avoided the use of program analysis and code instrumentation, and operated on test suites instead of test cases.

TCP in CI. In this paper we focus on the application of TCP techniques within CI environments. We do so because RTS techniques discard test cases, while TCP techniques do not, and often-times organizations we work with are loathe to omit test cases that might have exposed faults, had they been executed.

Prior research focusing on TCP in CI environments [13, 52] has resulted in techniques that either reorder test suites within a

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180213

commit (*intra-commit*) or across commits (*inter-commit*). Neither of these approaches, however, have proven to be successful.

Intra-commit prioritization schemes rarely produce meaningful increases in the rate at which faults are detected. As we shall show in Section 3, intra-commit techniques prioritize over a space of test suites that is too small in number and can be quickly executed, so reordering them typically cannot produce large reductions in feedback time. This approach also faces some difficulties. First, test suites within commits may have dependencies that make reordering them error-prone. Second, test scripts associated with specific commits often include semantics that adjust which test suites are executed based on the results of test suites executed earlier in the commit; these devices reduce testing costs, but may cease to function if the order in which test suites are executed changes.

Inter-commit techniques have the potential for larger gains, but are founded on unrealistic assumptions about CI environments. In these environments, developers use commits to submit code modules, and each commit is associated with multiple test suites. These test suites are queued up until a clean build of the system is available for their execution. Extending the execution period of a commit's test suites over time (across commits) increases the chance for test suites to execute over different computing resources, hence requiring additional system build. Given the cost of such builds (Hilton et al. [22] cite a mean cost of 500 seconds per commit, and for the Rails artifact in our study the mean cost ratio of building over testing was 41.2%), this may not be practical.

Key insight and proposal. We conjecture that in CI environments, prioritizing test suites (either within or between commits) is not the best way to proceed. Instead, *prioritization should be performed on commits*, a process we refer to as *inter-commit prioritization*. Inter-commit prioritization avoids the costs of performing multiple builds, and problems involving test suite dependencies and the disabling of cost-saving devices for reducing testing within commits. We further believe that inter-commit prioritization will substantially increase TCP's ability to detect faulty commits early and provide faster feedback to developers.

In this work, we investigate this conjecture by providing an algorithm that prioritizes commits. An additional key difference between our TCP approach and prior work, however, is that we do not wait for a set or "window" of test suites (or in our case, commits) to be available, and then prioritize across that set. Instead, we prioritize (and re-prioritize) all commits that are waiting for execution "continuously" as prompted by two events: (1) the arrival of a new commit, and (2) the completion of a previously scheduled commit. We do this using a lightweight approach based on test suite failure and execution history that has little effect on the speed of testing. Finally, our approach can be tuned dynamically (on-the-fly) to respond to changes in the relation of the incoming testing workload to available testing resources.

We have evaluated our new approach on three non-trivial data sets associated with projects that utilize CI. Our results show that our algorithm for prioritizing commits can be much more effective than prior approaches. Our results also reveal, however, several factors that influence our algorithm's effectiveness, with implications for the application of prioritization in CI environments in general.

2 BACKGROUND

Conceptually, in CI environments, each developer commits code to a version control repository. The CI server on the integration build machine monitors this repository to determine whether changes have occurred. On detecting a change, the CI server retrieves a copy of the changed code from the repository and executes the build and test processes related to it. When these processes are complete, the CI server generates a report about the result and informs the developer. The CI server continues to poll for changes in the repository, and repeats the previous steps.

There are several popular open source CI servers including Travis CI [45], GoCD [18], Jenkins [24], Buildbot [5], and Integrity [23]. Many software development companies are also developing their own [14, 32, 39, 43]. In this paper we utilize data gathered from a CI testing effort at Google, and a project managed under Travis CI, and the next two sections provide an overview of these CI processes. Section 5.1 provides a more quantitative description of the data analyzed under these processes.

2.1 Continuous Integration at Google

The Google dataset we rely on in this work was assembled by Elbaum et al. [13] and used in a study of RTS and TCP techniques; the dataset is publicly available [12]. Elbaum et al. describe the process by which Google had been performing CI, and under which the dataset had been created. We summarize relevant parts of that process here; for further details see Reference [32].

Google utilizes both *pre-commit* and *post-commit* testing phases. When a developer completes his or her coding activities on a module M, the developer presents M for pre-commit testing. In this phase, the developer provides a *change list* that indicates modules that they believe are *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, using all test suites relevant to all of the modules listed in the change list. The commit testing outcome is then communicated to the developer.

Typically, when pre-commit testing succeeds for M, a developer submits M to source code control; this causes M to be considered for post-commit testing. At this point, algorithms are used to determine the modules that are *globally relevant* to M, using a coarse but sufficiently fast process. This includes modules on which M depends as well as modules that depend on M. All of the test suites relevant to these modules are queued for processing.

2.2 Continuous Integration in Travis CI

Travis CI is a platform for building and testing software projects hosted at GitHub. When Travis CI is connected with a GitHub repository, whenever a new commit is pushed to that repository, Travis CI is notified by GitHub. Using a specialized configuration file developers can cause builds to be triggered and test suites to be executed for every change that is made to the code. When the process is complete, Travis sends notifications of the results to the developer(s) by email or by posting a message on an IRC channel. In the case of pull requests,[1] each pull request is annotated with the outcome of the build and test efforts and a link to the build log.

[1] The fork & pull collaborative development model used in Travis CI allows people to fork an existing repository and push commits to their own fork. Changes can be merged into the repository by the project maintainer. This model reduces friction for new contributors and it allows independent work without up-front coordination.

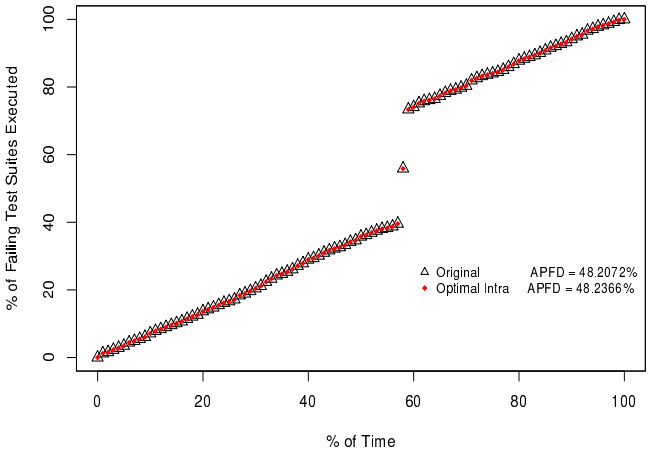

Figure 1: Intra-commit prioritization on Google post-commit.

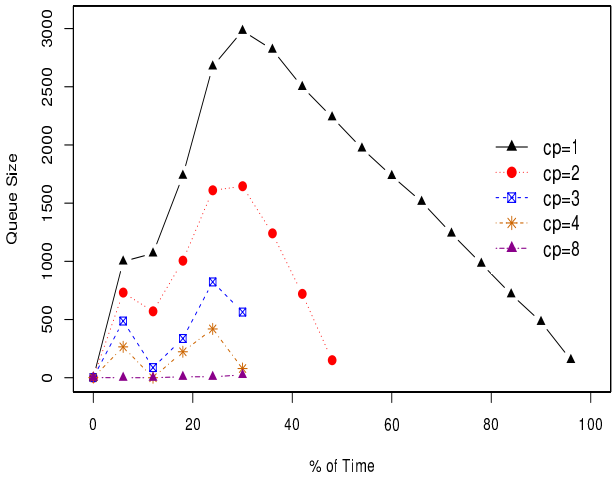

Figure 2: Google post-commit arrival queue size over time for five levels of computing resources

3 MOTIVATION

There are two key motivations for this work. First, existing prioritization techniques, that prioritize at the level of test suites yield little improvement in the rate of fault detection. Second, as commits queue up to be tested, they can be prioritized effectively.

Figure 1 plots Google post-commit data supporting the first of these claims. The horizontal axis represents the passage of time, in terms of the percentage of the total testing time needed to execute a stream of almost 1.5 million test suites. The vertical axis denotes the percentage of test suites that have failed thus far relative to the number of test suites that fail over the entire testing session.

The figure plots two lines. One line, denoted by triangles, represents the "Original" test suite order. This depicts the rate at which failing test suites are executed over time, when they (and the commits that contain them) are executed in the original order in which they arrived for testing in the time-line captured by the Google dataset, with no attempt made to prioritize them.[2] The second line, denoted by diamonds that are smaller than the triangles, represents an "Optimal Intra-commit" order. In this order, commits continue to be executed in the order in which they originally arrived, but within each commit, test suites are placed in an order that causes failing test suites to all be executed first. (Such an optimal order cannot be achieved by TCP techniques, because such techniques do not know *a priori* which test suites fail, but given test suites for which failure information is known it can be produced *a posteriori* to illustrate the best case scenario in comparisons such as this.)

In graphs such as that depicted in Figure 1, a test suite order that detects faults faster would be represented by a line with a greater slope than others. In the figure, however, the two lines are nearly identical. The gains in rate of fault detection that could be achieved by prioritizing test suites within commits in this case are negligible. The actual overall rates of fault detection using the $APFD_C$ metric for assessing such rates (discussed in Section 5.2.2), also shown in the figure, are 48.2072% for the original order, versus 48.2366% for the optimal order; this too indicates negligible benefit.

[2]The "gaps" between points at around the 58% time are caused by a pair of commits that contained large numbers of test suites that failed.

Next, consider the notion of prioritizing commits rather than individual test suites. Figure 2 shows the sizes of several queues of commits over time, for the Google post-commit dataset, assuming that commits are executed one at a time (not the case at Google but useful to illustrate trends), and are queued when they arrive if no computer processors (cp) are available for their execution. Because the execution of test suites for commits depends on the number of computing resources across which they can be parallelized, the figure shows the queue sizes that result under five such numbers: 1, 2, 3, 4 and 8. With one computing resource, commits queue up quickly; then they are gradually processed until all have been handled. As the number of resources increases up to four, different peaks and valleys occur. Increasing the number to eight causes minimal queuing of commits.

What Figure 2 shows is that if sufficient resources are not available, commits do queue up, and this renders the process of prioritizing commits potentially useful. Clearly, additional computing resources could be added to reduce the queuing of commits, and for companies like Google and services like Travis, farms of machines are available. The cost of duplicating resources, however, does become prohibitive at some point. In the case of Travis, for example, the price for resources increases by 87% when moving from one to two concurrent jobs. And even for companies like Google this is an ongoing challenge [32].

The reasons for considering inter-commit prioritization can be illustrated further by an example. (We use a theoretical example here for simplicity.) Table 1 shows a set of five commits (Rows 1–5), each containing up to ten test suites (Columns with headers 1–13), with "F" indicating instances in which test suites fail, "P" indicating instances in which test suites pass, and "-" indicating instances in which test suites are not used. Suppose the five commits depicted in Table 1 all arrive in a short period of time and are all queued up for execution, in order from top ("Commit 1") to bottom ("Commit 5"). Figure 3 plots the fault detection behavior of the test suites associated with the commits under three prioritization scenarios, denoted by three different line types. The *solid line* depicts the results when commits are executed in the order in which they are queued up (from first to last), and the test suites associated

Table 1: Commits, Test Suites, Failures Detected

Commit	Test Suite (P: pass, F: fail, -: not executed)												
	1	2	3	4	5	6	7	8	9	10	11	12	13
1	P	-	F	P	P	P	P	-	F	P	-	P	P
2	P	P	-	P	P	P	-	P	P	-	P	P	P
3	F	-	P	F	F	P	P	F	-	F	-	P	P
4	P	P	-	P	P	P	-	P	P	P	P	-	P
5	P	P	P	-	F	P	F	-	F	P	P	P	-

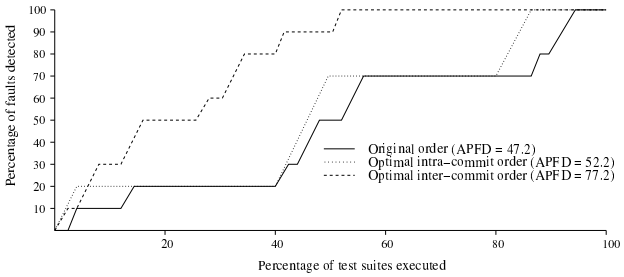

Figure 3: Effectiveness of commit prioritization.

with each commit are executed in their original order (from left to right). As the test suites are executed, they gradually expose faults until, when 100% of the test suites have been executed, 100% of the faults have been detected. The *dotted line* depicts the results when test suites within each commit are placed in optimal order (where fault detection is concerned), but commits are kept in their original, intra-commit order. (In other words, in Commit 1, test suites 3 and 9 are executed first, then the others follow.) The *dashed line* depicts the results when test suites within each commit retain their original order, but commits are ordered optimally (inter-commit) where fault detection is concerned. (In other words, commits are scheduled in order 3–5–1–2–4.)

The lines in Figure 3 illustrate why an inter-commit order can outperform an intra-commit order. The fault detection rate increases gained by an intra-commit order are limited to those that relate to reordering the comparatively small set of test suites (relative to all test suites executed under CI) contained within the commits, whereas the increases gained by an inter-commit order can potentially shift all fault detection to the first few commits. This theoretical example, however, involves optimal orders. To understand whether this example corresponds to what we may see in practice we need to study actual datasets, and orders that can be obtained by TCP heuristics.

4 PRIORITIZATION FOR CI

To prioritize testing efforts for CI we have created a new prioritization technique, CCBP, (Continuous, Commit-Based Prioritization), which has four distinctive characteristics:

- *Commit-focused*: the scale at which CI operates makes prioritization at the test case or test suite level irrelevant for accelerating faulty commit discovery.
- *Fast*: the speed at which CI operates requires prioritization schemes to be lightweight yet effective.
- *Continuous*: streaming commits result in streaming results which offer the opportunity to re-prioritize commits that are already queued for execution.
- *Resource-aware*: CI computing resources vary over time and prioritization must accommodate that variation.

CCBP is driven by two events associated with commits: the arrival of a commit for testing and the completion of the execution of the test suites associated with a commit. When a commit arrives, it is added to a commit queue and, if computing resources are available, the queue is prioritized and the highest priority commit begins executing. When a commit's processing is complete, a computing resource and new testing-related information from that commit become available, so any queued commits are re-prioritized and the highest ranked commit is scheduled for execution.

Note that by this approach, prioritization is likely to occur multiple times on the same queued commits, as new commits arrive or new information about the results of a commit become available. For this to be possible, prioritization needs to be fast enough so that the gains of choosing the right commit are greater than the time required to execute the prioritization algorithm. As discussed previously, techniques that require code instrumentation or detailed change analysis do not meet this criterion when applied continuously, and when used sporadically they tend to provide data that is no longer relevant. Instead, as in other work [13, 52], we rely on failure and execution history data to create a prioritization technique that can be applied continuously.

4.1 Detailed Description of CCBP

Algorithm 1 provides a more detailed description of CCBP. The two driving commit events (commit arrival and commit completion) invoke procedures *onCommitArrival* and *onCommitTestEnding*, respectively. Procedure *prioritize* performs the actual prioritization task, and procedure *updateCommitInformation* performs bookkeeping related to commits.

CCBP relies on three data structures. The first data structure concerns commits themselves. We assume that each commit has an arrival time and a set of test suites associated with it. We add a set of attributes including the commit's expected failure ratio (the probability that the commit's test suites will fail based on their failure history) and execution ratio (the probability that the commit's test suites have not been executed recently), which are used for prioritization. Second, we keep a single queue of commits that are pending execution (commitQ). Arriving commits are added to commitQ and commits that are placed in execution are removed from commitQ, and whenever resources become available commitQ is prioritized. The third data structure concerns computing resources. In the algorithm, we abstract these so that when a commit is allocated, the number of resources is reduced, and when a commit finishes, the resources are released. There are parameters corresponding to the size of the failure window (failWindowSize) and execution window (exeWindowSize), measured in terms of numbers of commits, that are important to the prioritization scheme. In this work we set these parameters to specific constant values, but in practice they could be adjusted based on changing conditions (e.g., when releasing a new version for which failures are more likely). We assume there is also a set of resources available.

Both *onCommitArrival* (Lines 5–9) and *onCommitTestEnding* (Lines 10–14), invoke *prioritize* (Lines 15–21). Procedure *prioritize* updates information about the commits in the queue (Lines 16-18),

Algorithm 1: CCBP: Prioritizing Commits

```
1  parameter failWindowSize
2  parameter exeWindowSize
3  resources
4  queue commitQ

5  Procedure onCommitArrival(commit)
6      commitQ.add(commit)
7      if resources.available() then
8          commitQ.prioritize()
9      end

10 Procedure onCommitTestEnding()
11     resources.release()
12     if commitQ.notEmpty() then
13         commitQ.prioritize()
14     end

15 Procedure commitQ.prioritize()
16     for all commit_i in commitQ do
17         commit_i.updateCommitInformation()
18     end
19     commitQ.sortBy(failRatio, exeRatio)
20     commit = commitQ.remove()
21     resources.allocate(commit)

22 Procedure commit.updateCommitInformation(commit)
23     failCounter = exeCounter = numTests = 0
24     for all test_i in commit do
25         numTests.increment();
26         if commitsSinceLastFailure(test_i) ≤ failWindowSize then
27             failCounter.increment()
28         end
29         if commitsSinceLastExecution(test_i) > exeWindowSize then
30             exeCounter.increment()
31         end
32     end
33     commit.failRatio = failCounter / numTests
34     commit.exeRatio = exeCounter / numTests
```

and then sorts them (Line 19). This sort function can be instantiated in many ways. In our implementation, we sort commits in terms of decreasing order of failRatio values and break ties with exeRatio values, but other alternatives are possible and we expect this to be an area of active research. The commit with the highest score is removed from the queue (Line 20) and launched for execution on the allocated resource (Line 21). Procedure *updateCommitInformation* (Lines 22–34) updates a commit's failRatio and exeRatio. It does this by analyzing the history of each test suite in the commit. If a test suite has failed within the last failWindowSize commits, its failure counter is incremented (Lines 26-27). If a test suite has not been executed within the last exeWindowSize, its execution counter is incremented (Lines 29-30). These numbers are normalized by the numbers of test suites in the commits to generate new ratios (Lines 33-34). Intuitively, CCBP favors commits containing a larger percentage of test suites that have failed recently, and in the absence of failures, it favors commits with test suites that have not been recently executed.

As presented, for simplicity, CCBP assumes that commits are independent and need not be executed in specified orders. The empirical studies presented in this paper also operate under this assumption, as we have discovered no such dependencies in the systems that we study. Dependencies among commits could, however, be accommodated by allowing developers to specify them,

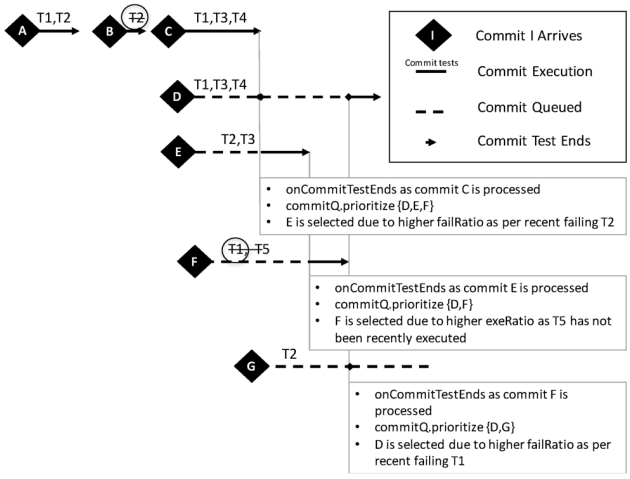

Figure 4: CCBP example.

and then requiring CCBP to treat dependent commits as singletons in which commit orders cannot be altered. We leave investigation of such approaches for future work.

Another issue is the potential for a commit to "starve", as might happen if it has never been observed to fail and if the pace at which new commits arrive causes the queue to remain full. This possibility is reduced by the fact that a commit's execution counter continues to be incremented, increasing the chance that it will eventually be scheduled; we return to this issue later in this paper.

For clarity, our presentation of CCBP simplifies some implementation details. For example, we do not prioritize unless we have multiple items in the commitQ and we keep a separate data structure for test suites to avoid recomputing test data across queued commits. Furthermore, to support the simulation of various settings we have included mechanisms by which to manipulate certain variables, such as the number of resources available and the frequency of commit arrivals. We discuss these aspects of the approach more extensively in Section 5.3.

4.2 Example

Figure 4 provides an example to illustrate Algorithm 1. For simplicity, we set parameters failWindowSize and exeWindowSize to two, and the number of computing resources to one. Commits are designated by uppercase letters and depicted by diamonds; they arrive at times indicated by left-to-right placement. When a commit is queued it is pushed down a line. Bulleted text in rectangles indicates steps in CCBP that occur at specific points of time.

The example begins with the arrival of commit A, with test suites $T1$ and $T2$. A is executed immediately because the resource is available. After A has been processed, commit B arrives, with test suite $T2$, and $T2$ fails. After B completes, commit C arrives, with test suites $T1$, $T3$, and $T4$. While C is being processed, commits D, E, and F arrive but since resources are not available they are added to commitQ. After C completes, since commitQ contains multiple commits, it is prioritized. In this case, commit E is placed first because its failRatio is higher than those for D and F (E is associated with test suite $T2$, which failed within the last two commits). After

E has been processed, commitQ is prioritized again, and F is selected for execution because its test suite, $T5$, has not been executed within the execution window. While F is executing, its test suite $T1$ fails; also, commit G arrives and is added to commitQ. When F finishes, commitQ is prioritized again. This time, D is placed first for execution because of the recent failure of its test suite, $T1$.

Two aspects of this example are worth noting. First, queued commits are prioritized continuously based on the most recent information about test suite execution results. Second, although test failure history is the main driver for prioritization, in practice, failing commits are less common than passing commits; this renders the execution window relevant, and useful for reducing the chance that a commit will remain in the queue for an unduly long time.

5 EMPIRICAL STUDY

We conducted a study with the goal of investigating the following research question: *Does CCBP improve the rate of fault detection when applied within a continuous integration setting?*

5.1 Objects of Analysis

As objects of analysis we utilize the CI data sets mentioned in Section 2, two from the Google Dataset and one from Rails, a project managed under Travis CI. [3]

5.1.1 The Google Data Set. 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 (commits), outcome statuses of test suite executions, launch times, and times required to execute test suites. The data pertains to both pre-commit and post-commit testing phases, and we refer to the two data subsets and phases as "GooglePre" and "GooglePost", respectively. We used the first 15 days of data because we found discontinuities in the later days. At Google, test suites with extremely large execution times are marked by developers for parallel execution. When executed in parallel, each process is called a shard. For test suites that had the same test suite name and launch times but different shard numbers, we merged the shards into a single test suite execution. After this adjustment, there were 2,506,926 test suite execution records. More information about this dataset can be found in the Google Dataset archive [12].

5.1.2 The Rails Dataset. Rails is a prominent open source project written in Ruby [36].. As the time in which we harvested its data, Rails had undergone more than 35,000 builds on Travis CI. Rails consists of eight main components with their own build scripts and test suites. Rails has a global Travis build script that is executed when a new commit is submitted to any branch or pull request. (Pull requests are not part of the source code until they are successfully merged into a branch, but Travis still needs to test them.) For each commit, the eight Rails components are tested under different Ruby Version Manager (rvm) implementations. Each pair of components and rvms is executed in a different job.

[3]As noted in Section 2.1, the Google Dataset is already available [12]; the Rails test data managed in Travis is now available at https://github.com/elbaum/CI-Datasets.git.

Table 2: Relevant Data on Objects of Analysis

	Dataset		
	Google Pre	Google Post	Rails
# of Total Commits	1,638	4,421	2,804
# of Failing Commits	267	1,022	574
Commit Arrival Rate (# / hour)	5	13	1
Avg Commit Duration (secs)	1,159.00	948.38	1,505.17
Avg Commit Queue Size	401.3	1,522.1	0.4
# of Distinct Test Suites	5,555	5,536	2,072
# of Distinct Failing Test Suites	199	154	203
Avg # of Test Suites per Commit	638	331	1280
# of Total Test Suite Executions	1,045,623	1,461,303	3,588,324
# of Failing Test Suite Executions	1,579	4,926	2,259
Test Suite Execution Time (secs)	1,898,445	4,192,794	4,220,482

When collecting data for Rails, we sought to gather a number of test suite executions similar to those found in GSDTSR. Because each Rails' commit executes around 1200 test suites on average, we collected 3000 consecutive commits occurring over a period of five months (from March 2016 to August 2016). From that pool of commits, we removed 196 that were canceled before the test suites were executed. We ended up with a sample of 3,588,324 test suite executions, gathered from 2,804 builds of Rails on Travis CI.

To retrieve data from Rails on Travis CI, we wrote two Ruby scripts using methods provided by the Travis CI API [46]: one downloads raw data from Travis CI and the other transforms the data into a required format. This last step required the parsing of the test suite execution reports for Rails by reverse engineering their format. The resulting dataset includes information such as test suite identifiers, test suite execution time, job and build identifiers, start times, finish times and outcome statuses (fail or pass).

5.1.3 Relevant Data on the Datasets. Table 2 characterizes the datasets we used in this study, and helps provide context and explain our findings. The table includes information on commits, test suites, and test suite executions. For commits, we include data on the total number of commits, the total number of failing commits (a commit is considered to fail when at least one of its associated test suites fails), the commit arrival rate measured in commits per second, the average commit duration measured from the time the commit begins to be tested until its testing is completed, and the average commit queue size computed by accumulating the commit queue size every time a new commit arrives and dividing it by the total number of commits. Within the test suite information, we provide the number of distinct test suites that were executed at least once under a commit, the number of distinct test suites that failed at least once, and the average number of test suites triggered by a commit. Regarding test suite executions, we include the total number of test suite executions, the total number of failing test suite executions, and the total time spent executing test suites measured in seconds.

5.2 Variables and Measures

5.2.1 Independent Variables. We consider one primary independent variable: prioritization technique. As prioritization techniques we utilize CCBP, as presented in Section 4, and a baseline technique in which test suites are executed in the original order in which they arrive. We also include data on an optimal order – an order that can be calculated a-posteriori when we know which test suites fail, and that provides a theoretical upper-bound on results. In our extended analysis, we also explore some secondary variables such as

available computing resources, the use of continuous prioritization, and failure window size.

5.2.2 Dependent Variables. CCBP attempts to improve the rate of fault detection as commits are submitted. Rate of fault detection has traditionally been measured using a metric known as *APFD* [11]. The original *APFD* metric, however, considers all test suites to have the same cost (the same running time). Our test suites and commits differ greatly in terms of running time, and using *APFD* on them will misrepresent results. Thus, we turn to a version of a metric known as "cost-cognizant *APFD*" (or $APFD_C$). $APFD_C$, originally presented by Elbaum et al. [10], assesses rate of fault detection in the case in which test case costs do differ.[4]

The original $APFD_C$ metric is defined with respect to test suites and the test cases they contain. Removing the portion of the original formula that accounts for fault severities, the formula is as follows. Let T be a test suite containing n test cases with costs $t_1, t_2, \ldots, t_n$. Let F be a set of m failures revealed by T. Let TF_i be the first test case in an ordering T' of T that reveals failure i. The (cost-cognizant) weighted average percentage of failures detected during the execution of test suite T' is given by:

$$APFD_C = \frac{\sum_{i=1}^{m}(\sum_{j=TF_i}^{n} t_j - \frac{1}{2}t_{TF_i})}{\sum_{j=1}^{n} t_j \times m} \quad (1)$$

In this work, we do not measure failure detection at the level of test cases; rather, we measure it at the level of commits. In terms of the foregoing formula, this means that T is a commit, and each t_i is a test suite associated with that commit. At an "intuitive" level, $APFD_C$ represents the area under the curve in a graph plotting failure detection against testing time, such as shown in Figure 3.

5.3 Study Operation

On the GSDTSR and Rails datasets, we simulated a CI environment. We implemented CCBP as described in Section 4, using approximately 600 lines of Java. Our simulation walks through the datasets, prioritizes commits, simulates their execution, and records when failures would be detected, providing data for computing $APFD_C$.

CCBP utilizes failure and execution window sizes (parameters failWindowSize and exeWindowSize in Algorithm 1); here we refer to these as W_f and W_e, respectively. For this study, we chose the value 500 for both W_f and W_e, because in preliminary work we found that differing values (we tried 10, 50, 100, 200, 500 and 1000 for each) did not make substantial differences.

Our simulation requires us to identify commits and their duration in the two datasets. In the Google dataset, each distinct "Change Request" is a commit id; thus, all test suite execution records with the same Change Request are considered to be in the same commit. In the Rails dataset, each test suite execution record has its own "Build Number"; thus, all test suite execution records with the same Build Number are considered to be in the same commit. In the Google dataset, each test suite execution record has a "Launch Time" and an "Execution Time", from which we can calculate the test suite's "End Time". We then order the test suites in a commit in terms of Launch Times, end-to-end, using their End Times as Launch Times for subsequent test suites. In the Rails dataset, each test suite execution record has a "Build Start Time", a "Build Finish Time", and a "Build Duration Time". Where possible, as the duration of each commit, we used Build Duration Time. In cases in which a build was stopped and restarted at a later point, we calculated the commit duration as Build Finish Time minus Build Start Time.

We usually assume that a single computing resource is available. An exception occurs in the second subsection of Section 6, where we explicitly explore tradeoffs that occur when the number of computing resources increases.

As a final note, in practice, we expect that CCBP could be granted a "warmup" period in which it monitors commit failure and execution information, allowing it to make more effective predictions when it begins operating. In this work, we did not utilize a warmup period. As a result, when prioritizing commits in the early stages of processing datasets, CCBP may not do as well as it would in the presence of warmup data. Our results may thus understate the potential effectiveness of CCBP in practice.

5.4 Threats to Validity

Where external validity is concerned, we have applied CCBP to three extensive datasets; two of these are drawn from one large industrial setting (Google) and the third from an open-source setting (Rails). The datasets have large amounts and high rates of code churn, and reflect extensive testing processes, so our findings may not extend to systems that evolve more slowly. We have compared CCBP to a baseline approach in which no TCP technique is used, but we have not considered other alternative TCP techniques (primarily because these would require substantial changes to work under the CI settings we are operating in). While we have provided an initial view of the effect that variance in the numbers of computing resources available for use in testing can have, most of our results are based on a simulation involving a single computing resource. These threats must be addressed through further studies.

Where internal validity is concerned, faults in the tools used to simulate CCBP on the datasets could cause problems in our results. To guard against these, we carefully tested our tools against small portions of the datasets, on which results could be verified. Further, we have not considered possible variations in testing results that may occur when test results are inconsistent (as might happen in the presence of "flaky test suites"); such variations, if present in large numbers, could potentially alter our results.

Where construct validity is concerned, we have measured effectiveness in terms of the rate of fault detection of test suites, using $APFD_C$. 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. In addition, $APFD_C$ itself has some limitations in this context, because it is designed to measure rate of fault detection over a fixed interval of time (e.g., the time taken to regression test a system release); in that context, ultimately, any test case ordering detects all faults that could be detected by the test suite that is being utilized. In the CI context, testing does not occur in a fixed interval; rather, it continues on potentially "forever", and the notion of all test case orderings eventually converging on detection of 100% of the faults that could be detected at a certain time does not apply. Finally, our cost numbers (test execution times) are gathered on only one specific machine.

[4] $APFD_C$ also accounts for cases in which fault severities differ, but in this work we ignore this component of the metric.

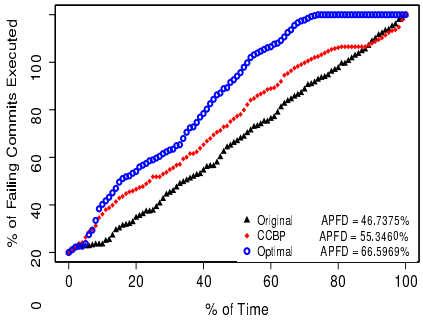

Figure 5: $APFD_C$ on GooglePre

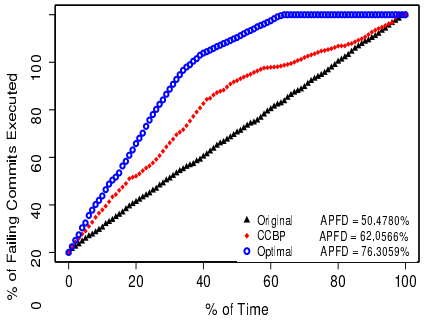

Figure 6: $APFD_C$ on GooglePost

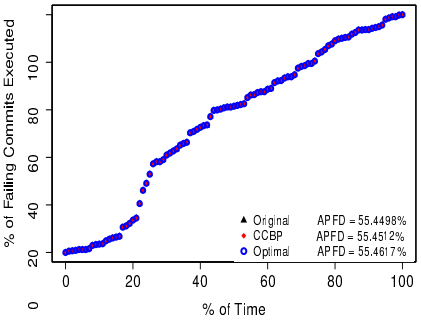

Figure 7: $APFD_C$ on Rails

6 RESULTS AND ANALYSIS

Figures 5, 6, and 7 summarize the results of applying CCBP to each dataset. In each figure, the x-axis represents the percentage of testing time, the y-axis represents the percentage of failing commits detected, and the three plotted lines correspond to the original commit ordering, the inter-commit ordering produced by CCBP, and the optimal commit ordering, respectively.

The figures reveal two distinct patterns. On the one hand, for GooglePre and GooglePost, the space available for optimizing the commit order to improve the rate of failure detection is clearly noticeable. The differences between the original and optimal orders are 20% and 25% for GooglePre and GooglePost, respectively. In both cases CCBP, although still far from optimal, is able to provide gains on that space over the original ordering, of 9% and 12%, respectively.

The space available for optimization in Rails, on the other hand, is almost non-existent. The optimal and original orders overlap and their $APFD_C$ values do not differ until the second decimal place. We conjecture that in this case the commit arrival rate is low enough (Table 2, row 3), for the resources available, that commits do not queue in large enough numbers to benefit from prioritization.

We explored this conjecture further by compressing the five months of Rails data into one month (by changing all the months in all the date-type fields to March) to cause artificial queuing of commits, and then applying CCBP. Figure 8 shows the results (hereafter referred to as "Rails Compressed" data), and they support our conjecture. Having compressed commit arrivals, there is now space for improving the rate of failure detection and CCBP does provide gains of 6% over the original order. This figure also illustrates an interesting situation: within the first 25% of the testing time, the inter-commit order created by CCBP underperforms the original order. This occurs because early in the process, CCBP does not have enough failure history data to make informed prioritization decisions. This points to the previously mentioned need for a "warm-up" period to collect enough history before applying CCBP.

The "continuous" in CCBP matters. To better understand CCBP's effectiveness gains we performed a follow up experiment. We designed an inter-commit prioritization technique that uses the same heuristics as CCBP but prioritizes queued commits only once. The technique employs two queues: one for arriving commits and one for prioritized commits. When a resource becomes available, the highest priority commit from the prioritized queue is chosen or, if

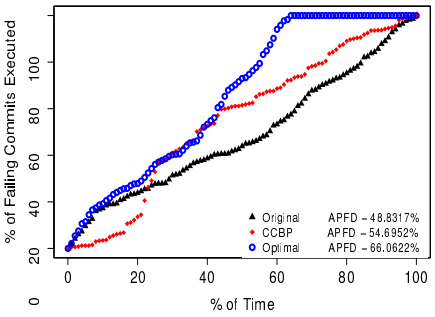

Figure 8: $APFD_C$ on Rails-Compressed

that queue is empty, the arriving queue is prioritized and moved to the prioritized queue.

Table 3 compares the $APFD_C$ results of this one-time per commit prioritization scheme to CCBP. Across the three datasets on which CCBP produced improvements in the rate of fault detection (GooglePre, GooglePost, Rails Compressed), continuous prioritization provided at least half of the increases in $APFD_C$. For example, for Rails-Compressed, the $APDF_C$ under original was 48.84%, when prioritized only one time it increased to 51.22%, and when prioritized with CCBP it increased to 54.7%. We also note that CCBP prioritization was triggered 3265 times for GooglePre, 8840 for GooglePost, 3703 for Rails, and 5599 for Rails Compressed, an average of 1.8 times higher than with the "One Time" technique. Prioritizing only once means that we miss the latest failure information generated while the commits are waiting in the queue – information that is key for effective prioritization. This is an important discovery, because *all* previous prioritization approaches that we are aware of prioritize queued items just once. Clearly, as new failure information emerges there are opportunities to benefit from operating continuously to better match CI environments.

Table 3: $APFD_C$ for Continuous vs. One-Time Prioritization

		CCBP		
	Original	One Time	Continuous	Optimal
GooglePre	46.74	47.97	55.35	66.60
GooglePost	50.48	55.27	62.06	76.31
Rails	55.45	55.45	55.45	55.46
Rails Comp.	48.83	51.22	54.70	66.06

Table 4: $APFD_C$ on GooglePost across Computing Resources

CP	Original	CCBP	Optimal
1	50.4780	62.0566	76.3059
2	75.2413	79.8735	84.3087
4	84.4399	85.0662	85.4634
8	85.4918	85.4993	85.5071
16	85.5074	85.5075	85.5079

Trading computing resources and prioritization. If a sufficient number of computing resources are available, commits do not queue up, rendering prioritization unnecessary. As discussed previously, however, computing resources are costly and not always readily available. We briefly explore this relationship by simulating what would occur with the $APFD_C$ values of GooglePost if the existing computing resources were repeatedly doubled. Table 4 summarizes the results. As expected, increasing the computing resources increases $APFD_C$ values because more commits can be processed in parallel. For the dataset we consider, the gains saturate around $APFD_C$ = 85.5, when the resources are multiplied by eight. Also as expected, the opportunities for prioritization to be most effective are greater when computing resources are most scarce (the most noticeable gains achieved by CCBP compared to the original orders occur when there are just one or two resources).

In this context, however, it is most interesting to focus on the tradeoffs across these dimensions. For example, if we could prioritize optimally, we could obtain larger $APFD_C$ gains compared to the original ordering with a single computing resource than if we had duplicated the resources without using any prioritization scheme. Similarly, although not as appealing, CCBP can provide almost half of the gains (12%) that would be achievable by duplicating the computing resources from one to two on the original ordering (25%). Last, since CCBP allows for the incorporation of resources on the fly, one could let CCBP request additional computing power when the size of the commit queue reaches a threshold, and release resources when prioritization suffices.

On the speed of prioritization. We have argued that for prioritization to operate in CI environments it needs to be fast. Quantifying what fast means depends on how fast the CI environment operates. For our datasets, as shown in Table 2, the average commit duration (measured from the time the commit's first test suite begins executing until the last test suite completes execution) is 1159 seconds for GooglePre, 948 seconds for GooglePost, and 1505 seconds for Rails.

When run on a Macbook Pro, to provide a prioritized order of commits, CCBP's execution time averaged 0.04 seconds for GooglePost, 0.01 seconds for GooglePre, and 0.0005 seconds for Rails. The differences in prioritization times were due primarily to commit queue sizes – longer queues required more time to update and prioritize (via the *commit.updateCommitInformation* procedure in Algorithm 1). Even if we run the prioritization algorithm twice per commit (this is the worse case: once for each commit arrival and once for each commit completion), the overhead per commit is less than 0.008% for all datasets. With such performance, we estimate that CCBP could easily be incorporated into the workflow of CI.

On the effect of W_f selection. The failure window is a key parameter in Algorithm 1. Previous studies have explored the impact of different window sizes defined in terms of "time since the last observed failure" [13] and reasoned that if the selected W_f is too small, prioritization may not be sensitive enough to relevant failures. Previous work also showed that if this window is too large, too many test cases would have failed within the window, diluting the value of the most recent failures and reducing the opportunities for effective prioritization.

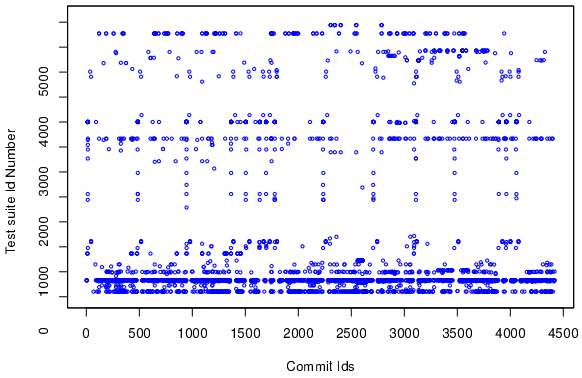

Figure 9: Test suite failures over commits for GooglePost

Given this potential range of effects and the fact that we are operating at the commit level, we decided to again explore a range of failure window sizes in terms of the number of commits. To our surprise, we found that at the commit level the approach was more robust to the choice of W_f. More precisely, failure windows between 10 and 500 commits achieved similar results.

Despite this finding, the reason for choosing large enough windows still holds. We observe that when test suites fail, they tend to do so in spurts. Windows of commits large enough to contain those spurts are effective. Figure 9 illustrates this for GooglePost. In the figure, the x-axis corresponds to commits ids, the y-axis corresponds to test suite ids, and the circles indicate when a test suite failed. As noted earlier, a small percentage of test suites fail, but in the figure we can also see that, for the original commit ordering, failures on a given test suite usually occur in small clusters (sequences of points for a given test suite across commits). As long as W_f is as large as most of those clusters ($W_f \geq 10$), then the prioritization scheme functions effectively. Less intuitive is the notion that having much larger windows (of up to 500 commits) does not negate the benefits of the approach. We believe this is due to the small portion of test suites that exhibit failures (10% to 20% across the datasets). With such a failure rate, larger windows do not result in a major influx of failure information. This means that, at least for this level of failure rates, the approach based on test suite failure history information may be more resilient than anticipated to dilution effects.

Prioritizing test suites within commits has limited impact. Traditionally, prioritization effectiveness has been measured in terms of how soon *test cases* or *test suites* detect faults. In this work, because we focus on how soon a *failing commit* is detected, we measure cost-effectiveness in terms of how soon failing commits are detected. In this context, just one failing test suite was required to consider a commit to have failed. Under that reference measure, CCBP does not attempt to improve test suite ordering within a commit, instead it simply modifies the order of queued commits.

If we were to focus again on how soon *test suites* detect failures, then CCBP might be enhanced by further prioritizing test suites

within each commit. We explored such an enhancement with both the GooglePre and the GooglePost datasets, measuring $APFD_C$ over the percentage of *test suites* executed. We prioritized with three techniques: (1) CCBP as defined, (2) CCBP with intra-commit prioritization of test suites, and (3) the original order with intra-commit prioritization of test suites. For the GooglePre dataset, we found that neither technique 2 nor technique 3 provided much improvement over CCBP in terms of effectiveness. We attribute this to the large proportion of distinct failing test suites in this dataset, which makes failure prediction at the test suite level ineffective (even though when failures are aggregated at the commit level they provide noticeable gains). For the GooglePost dataset the $APFD_C$ for CCBP was 17% greater than for technique 3, and technique 2 provided only a 0.5% gain, rendering the contribution of intra-commit prioritization marginal.

Delays in detecting failing commits. Although the $APFD_C$ metric measures the rate of fault detection, it can be useful and more intuitive to compare techniques' cost-effectiveness in terms of the reduction in delays to detect failing commits under the different orderings. It is these reductions in delays that allow prioritization to provide potential advantages to developers. To capture the notion of delays, we accumulated failing commit time differences between CCBP and the original commit order. More specifically, we compute $\sum_{f=1}^{n}(commit_f.startTime + commit_f.duration/2)$ (this last term assumes an average time for a commit to fail) over all the failing commits f under CCBP and the original order, and normalize that by the number of failing commits in each artifact (267, 1022, and 574 failing commits for GooglePre, GooglePost, and Rails respectively as per Table 2, row 2). The findings are consistent with those reported earlier. On average, CCBP reduces the delays in detecting failing commits by 46 hours for GooglePre, 135 hours for GooglePost, and 69 hours for Rails Compressed, while achieving no reductions for Rails. These delay reductions, although significant, are in computing hours, and as such, can be reduced by using multiple computing resources as described previously. Still, as we argued earlier, computing resources come with a cost and CCBP reduces feedback time without incurring additional costs. Since our dataset does not provide information on resource availability we leave further assessment of the impact of delays on developer's time for future work.

7 RELATED WORK

Do et al. [8], Walcott et al. [47], Zhang et al. [53], and Alspaugh et al. [1] study test case prioritization in the presence of time constraints such as those that arise when faster development-and-test cycles are used. This work, however, does not consider test history information or CI environments. Other work [2, 26, 48] has used test history information and information on past failures to prioritize test cases, as do we, but without considering CI environments.

There has been considerable research on predicting fault-prone modules in software systems. Some of this work has considered dynamic prediction, as we do, most notably work by Hassan et al. [19] and Kim et al. [27]. This work, however, does not consider CI environments, or attempt to use fault proneness information in the service of regression testing techniques.

There has been some recent work on techniques for testing programs on large farms of test servers or in the cloud (e.g., [4, 28, 42]). This work, however, does not specifically consider CI processes or regression testing.

Hilton et al.[22] report results of a large-scale survey of developers to understand how and why they use or do not use CI environments. One of the implications they derive is that CI requires non-trivial time and resources, and thus, the research community should find ways to improve CI build and testing processes.

Memon et al. [32], working in the context of Google's CI processes, investigate approaches for avoiding executing test cases that are unlikely to fail, and for helping developers avoid actions leading to test case failures. Like our work, this work attempts to reduce delays in providing feedback; however, the work relies on test selection, whereas we focus on prioritization.

Prioritization in CI has emerged as a large issue but until this work it focused exclusively on test cases or suites. Jiang et al. [25] consider CI environments, and mention that prioritization could be used 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. Busjaeger and Xie [6] present a prioritization algorithm that uses machine learning to integrate various sources of information about test cases. They argue that such algorithms are needed in CI environments, and they analyze the needs for such algorithms in one such environment, but their focus remains on prioritizing individual test cases. Marijan et al. [31] present prioritization algorithms that also utilize prior test failure information to perform prioritization but focus on individual test cases. Yoo et al. [52], also working with data from Google, describe a search-based approach for using TCP techniques in CI environments for test suites within commits. We also describe a prioritization technique for use in CI environments considering information on past test failures and elapsed time since prior test executions [13]. Their approach, however, applies to individual test suites without considering commit boundaries. They also perform prioritization over windows of test suites and not continuously.

8 CONCLUSIONS

We have presented a novel algorithm, CCBP, for increasing the rate of fault detection in CI environments via prioritization. Unlike prior algorithms, CCBP prioritizes at the level of commits, not test cases, and it does so continuously as commits arrive or complete. CCBP is lightweight and operates quickly, allowing it to be sufficiently responsive in CI environments. In future work we intend to further explore the effects of factors such as the rate of change, commit dependencies, and available resources, on the cost-effectiveness of CCBP. Given that results of the algorithm can vary with different workloads, we would like to be able to dynamically adapt it to be more cost-effective as workloads change. This could apply, for example, when the computing resources available for testing increase or decrease, or when arrival rates or sizes (in terms of associated test suites) of commits change.

9 ACKNOWLEDGMENTS

This work has been supported in part by by National Science Foundation Award #1526652 . We thank the developers at Google and Travis who helped us understand the ongoing challenges in CI.

REFERENCES

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

[2] J. Anderson, S. Salem, and H. Do. 2014. Improving the Effectiveness of Test Suite Through Mining Historical Data. In *Proceedings of the Working Conference on Mining Software Repositories*. 142–151.

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

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

[5] Buildbot 2018. Buildbot. https://buildbot.net. (2018).

[6] B. Busjaeger and T. Xie. 2016. Learning for test prioritization: An industrial case study. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 975–980.

[7] A. Celik, A. Knaust, A. Milicevic, and M. Gligoric. 2016. Build system with lazy retrieval for Java projects. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 643–654.

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

[9] E. D. Ekelund and E. Engstrom. 2015. Efficient regression testing based on test history: An industrial evaluation. In *International Conference on Software Maintenance and Evolution*. 449–457.

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

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

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

[13] S. Elbaum, G. Rothermel, and J. Penix. 2014. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the ACM Symposium on Foundations of Software Engineering*. 235–245.

[14] H. Esfahani, J. Fietz, A. Ke, Q.and Kolomiets, E. Lan, E.and Mavrinac, W. Schulte, N. Sanches, and S. Kandula. 2016. CloudBuild: Microsoft's distributed and caching build service. In *Proceedings of the International Conference on Software Engineering Companion*. 11–20.

[15] Facebook 2017. Facebook. https://code.facebook.com/posts/270314900139291/rapid-release-at-massive-scale/. (2017).

[16] A. Gambi, Z. Rostyslav, and S. Dustdar. 2016. Poster: Improving cloud-based continuous integration environments. In *Proceedings of the International Conference on Software Engineering*. 797–798.

[17] M. Gligoric, L. Eloussi, and D. Marinov. 2015. Practical regression test selection with dynamic file dependencies. In *Proceedings of the International Symposium on Software Testing and Analysis*. 211–222.

[18] GoCD 2018. GoCD. https://www.gocd.io. (2018).

[19] A. E. Hasan and R. C. Holt. 2005. The top ten list: Dynamic fault prediction. In *Proceedings of the International Conference on Software Maintenance*.

[20] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y. Le Traon. 2016. Comparing white-box and black-box test prioritization. In *Proceedings of the International Conference on Software Engineering*. 523–534.

[21] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy. 2015. The Art of Testing Less Without Sacrificing Quality. In *Proceedings of the International Conference on Software Engineering*. 483–493.

[22] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig. 2016. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the International Conference on Automated Software Engineering*. 426–437.

[23] Integrity 2018. Integrity. https://integrity.github.io. (2018).

[24] Jenkins 2018. Jenkins. https://wiki.jenkins-ci.org/display/JENKINS/Home. (2018).

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

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

[27] S. Kim, T. Zimmerman, E. J. Whitehead, and A. Zeller. 2007. Predicting faults from cached history. In *Proceedings of the International Conference on Software Engineering*. 489–498.

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

[29] O. Legunsen, F. Hariri, A. Shi, Y. Lu, L. Zhang, and D. Marinov. 2016. An extensive study of static regression test selection in modern software evolution. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 583–594.

[30] D. Marijan. 2015. Multi-perspective Regression Test Prioritization for Time-Constrained Environments. In *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security*. 157–162.

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

[32] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco. 2017. Taming Google-scale continuous testing. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice Track*. 233–242.

[33] Newrelic 2016. Newrelic. https://blog.newrelic.com/2016/02/04/data-culture-survey-results-faster-deployment/. (2016).

[34] J. Öqvist, G. Hedin, and B. Magnusson. 2016. Extraction-based regression test selection. In *Proceedings of the International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. 5:1–5:10.

[35] A. Orso, N. Shi, and M. J. Harrold. 2004. Scaling regression testing to large software systems. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 241–251.

[36] Rails 2018. Rails. https://travis-ci.org/rails/rails. (2018).

[37] G. Rothermel and M. J. Harrold. 1997. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology* 6, 2 (1997), 173–210.

[38] G. Rothermel, R. Untch, C. Chu, and M. J. Harrold. 2001. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering* 27, 10 (2001), 929–948.

[39] T. Savor, M. Douglas, M. Gentili, L. Williams, K. Beck, and M. Stumm. 2016. Continuous Deployment at Facebook and OANDA. In *Proceedings of the International Conference on Software Engineering Companion*. 21–30.

[40] A. Shi, S. Thummalapenta, S. K. Lahiri, N. Bjorner, and J. Czerwonka. 2017. Optimizing test placement for module-level regression testing. In *Proceedings of the International Conference on Software Engineering*. 689–699.

[41] A. Shi, T. Yung, A. Gyori, and D. Marinov. 2015. Comparing and combining test-suite reduction and regression test selection. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 237–247.

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

[43] Techbeacon 2016. Techbeacon. https://techbeacon.com/going-big-devops-how-scale-continuous-delivery-success. (2016).

[44] ThoughtWorks. 2014. Go: Continous delivery. www.thoughtworks.com/products/go-continuous-delivery. (2014).

[45] Travis 2018. Travis CI. http://travis-ci.org. (2018).

[46] Travis API 2018. Travis CI API. https://travis-ci.com/api. (2018).

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

[48] X. Wang and H. Zeng. 2016. History-based dynamic test case prioritization for requirement properties in regression testing. In *Proceedings of the International Workshop on Continuous Software Evolution and Delivery*. 41–47.

[49] L. White and B. Robinson. 2004. Industrial real-time regression testing and analysis using firewalls. In *Proceedings of the International Conference on Software Maintenance*. 18–27.

[50] G. Wikstrand, R. Feldt, J. K. Gorantla, W. Zhe, and C. White. 2009. Dynamic Regression Test Selection Based on a File Cache An Industrial Evaluation. In *International Conference on Software Testing Verification and Validation*. 299–302.

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

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

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