2018 IEEE International Conference on Software Maintenance and Evolution

Test Re-prioritization in Continuous Testing Environments

Yuecai Zhu
Department of Computer Science and Software Engineering
Concordia University
Montreal, QC, Canada
Email: zhuyuecai@gmail.com

Emad Shihab
Data-driven Analysis of Software Lab (DAS)
Department of Computer Science and Software Engineering
Concordia University
Montreal, QC, Canada
Email: emad.shihab@concordia.ca

Peter C. Rigby
Department of Computer Science and Software Engineering
Concordia University
Montreal, QC, Canada
Email: peter.rigby@concordia.ca

***Abstract*—New changes are constantly and concurrently being made to large software systems. In modern continuous integration and deployment environments, each change requires a set of tests to be run. This volume of tests leads to multiple test requests being made simultaneously, which warrant prioritization of such requests. Previous work on test prioritization schedules queued tests at set time intervals. However, after a test has been scheduled it will never be reprioritized even if new higher risk tests arrive. Furthermore, as each test finishes, new information is available which could be used to reprioritize tests. In this work, we use the conditional failure probability among tests to reprioritize tests after each test run. This means that tests can be reprioritized hundreds of times as they wait to be run. Our approach is scalable because we do not depend on static analysis or coverage measures and simply prioritize tests based on their co-failure probability distributions. We named this approach CODYNAQ and in particular, we propose three prioritization variants called CODYNAQSINGLE, CODYNAQDOUBLE and CODYNAQFLEXI. We evaluate our approach on two data sets, CHROME and Google testing data. We find that our co-failure dynamic re-prioritization approach, CODYNAQ, outperforms the default order, FIFOBASELINE, finding the first failure and all failures for a change request by 31% and 62% faster, respectively. CODYNAQ also outperforms GOOGLETCP by finding the first failure 27% faster and all failures 62% faster.**

***Keywords*-Regression Testing; Test Minimization; Dynamic Test Prioritization; Test Dependency; Continuous Testing; Continuous Integration**

I. INTRODUCTION

The practice of releasing often and releasing early - also known as *rapid release* - has seen wide adoption in the software engineering field [1]. A key technology to enable rapid release is the use of a continuous integration (CI) process, where feedback on changes to the software are given quickly [5]. CI requires continuous testing of every change which can be very expensive.

For large projects, the burden of continuous testing can be significant and lead to many thousands of tests that are triggered daily. Prior studies have shown that for industrial projects with as little as 100 changes, such as the video conference system analyzed in Marijan *et al.*'s work [14], the testing time may surpass 2 days. This long testing cycle can become a bottleneck delaying the development and release of code. To address this issue, prior work, which we refer to as GOOGLETCP, has proposed the reordering of tests after a certain time period has passed [7].

Although prior work has been able to improve the speed of testing, most studies are based on the assumption that tests are independent. In practice with GOOGLETCP this means, that tests are prioritized once when the request is made. We argue that the tests (or their failures) are not independent and that prior test outcomes should be used to prioritize the continuous testing efforts. In particular, we propose the test re-prioritization approach, CODYNAQ, which uses the co-failure distributions of tests to dynamically reprioritize queued tests. For example, if test A and test B co-fail in the past 75% of the time, then if we observe a failure of test A in the current run, we may be able to speed up the execution of test B since there would be a high probability that it will fail as well. In fact, as we show later, more than 58% of the tests in CHROME co-fail with at least another test.

Our approach introduces two novel concepts: first, prioritization based on the co-failure distribution of tests, and second, re-prioritization after each test run. We implement this approach as the single queue re-prioritization, CODYNAQSINGLE. Since we continuously reprioritize tests, we find that tests with low failure probabilities can be pushed back in the queue (*i.e.* "starved") and delay final test result for a change. To deal with this problem, we introduce a double re-prioritization queue, CODYNAQDOUBLE, and a flexible re-prioritization queue, CODYNAQFLEXI.

To evaluate CODYNAQ and compare it with the state-of-the-art, GOOGLETCP, we use two data sets. The first is the internal Google testing data that Elbaum *et al.* [7] released when developing GOOGLETCP. The second is scraped data from the CHROME project which runs as many as 149 tests per minute. The CHROME data set includes over 4.4 million test runs.

Our goal is to speed-up the detection of test failures. We use the actual test request order as a baseline and compare the speedup/slowdown we attain in finding the *first failure* and the time to find *all failures* for each request. On both data sets

2576-3148/18/$31.00 ©2018 IEEE
DOI 10.1109/ICSME.2018.00016

Table I: The median and maximum number of test requests per minute

	Median	Maximum
Google	27	2,461
CHROME	26	149

CODYNAQ outperforms the state-of-the-art, GOOGLETCP. CODYNAQ is most effective when there are many co-failures and the speedup compared to GOOGLETCP can be as much as 26.65% and 61.8% faster in finding FIRSTFAIL and ALLFAIL respectively. In general, we are able to find the first failure and all failures approximately 11.33% and 61.84% faster than the default order (FIFOBASELINE) in CHROME and find the first failure and all failures 31.01% and 39.60% faster in the Google data set.

This paper is organized as follows. We provide background on continuous testing in Section II. Section III summarizes the related work. Section IV sets up our case study and introduces our proposed re-prioritization algorithms and evaluation criteria. Section V, presents our preliminary results. Section VI presents the results of our case study. Section VII discusses how data set characteristics can impact which re-prioritization algorithms should be used. Sections VIII and IX discuss the threats to validity, future work, and conclude the paper.

II. BACKGROUND

The continuous integration process and the sheer volume of changes at large companies, such as Google and Microsoft, derive a need for efficient test prioritization techniques. In most cases, the status quo is to prioritize tests based on their request order. We suggest that co-failure conditional probabilities based on the past failure distributions may lead to finding failures earlier.

Multi-Request Environment: Both, Marijan *et al.* [14] and Elbaum *et al.* [7] reported heavy work loads in the product testing queues. We find that CHROME has a similarly heavy load. In the CHROME project, when a new change is submitted for review, the developer or reviewer requests a set of tests to determine the quality of a change. Usually the set of tests requested are different from those requested for a different change. We will refer to the set of tests requested for a new code change as a test request. In some cases, a large number of tests can be requested for each change leading to slow test results and delaying the merging of changes. Long wait times for test results can be a serious issue when developers need to pass quality checks before moving onto other work.

To solve this problem, multi-request test environments facilitate the handling of multiple test requests at the same time. Figure 1 shows the distribution of test requests and Table I shows the median and maximum number of test requests per minute. As the table shows, in CHROME, a median of 26 tests are requested per minute, and in extreme cases, that number can be as high as 149 test requests. In the Google data, the median is 27 test requests per minute, and in extreme cases, it can be as high as 2,508 test requests per minute.

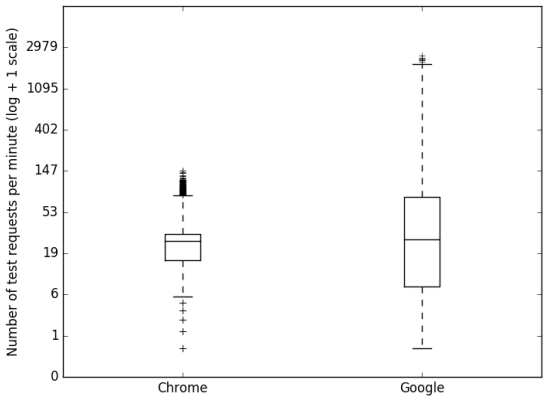

Figure 1: Number of changes submitted to the test queue per minute for Google and CHROME

Concurrent Test Request Execution: In a multi-request environment, multiple test machines are used and tests are run in parallel. On both CHROME and at Google the default approach is to schedule tests based on their request time. This common strategy is used as our FIFOBASELINE approach. Elbaum *et al.* proposed to schedule tests only based on their assigned priority in the dispatch queue [7]. Since tests are scheduled solely based on their priority without considering when they are requested, test requests are executed *concurrently*.

We illustrate this idea in figure 2, where test requests $R1, R2, R3$ are submitted in order, but the priority of their tests $T1, T2, T3$ is not based on a FIFO scheme. Rather, the test execution order is based on some other priority scheme. We adopt this idea in CODYNAQ, i.e., we also reprioritize tests based on their failure likelihood and not their arrival order.

III. RELATED WORK

Program analysis based test selection and prioritization techniques have been extensively researched in the regression testing literature [13][18][10][21][17][15][16]. Kim and Porter [11] were the first to use historical test failure distributions. Their approach does not consider multiple test requests or re-prioritization. Elbaum's *et al.* work was the first to acknowledge the problems of multiple test requests and to employ historical failures. Elbaum *et al.* [7] proposed the GOOGLETCP algorithm and evaluated it by the time required to provide feedback on failing tests. In their approach, they

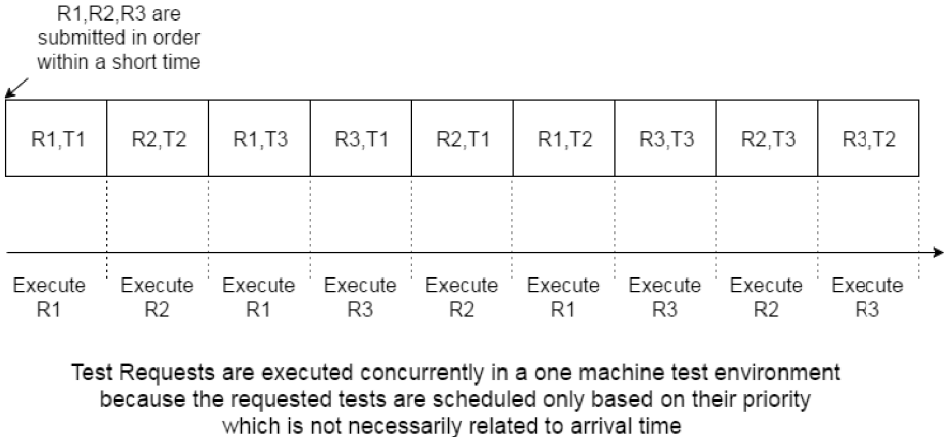

Figure 2: Example of concurrent test execution

prioritize tests in the scope of the whole waiting queue instead of per request. When the prioritize window checking condition is met, a certain number of tests in the head of the waiting queue will be pulled into a prioritized queue and are scheduled to be run. This prioritized queue is referred to as the dispatch queue. They prioritize test suites in the dispatch queue based on how often a test has failed in the past without considering when the test is requested. Hence, test requests are executed concurrently and developers obtain the test results faster than in the traditional approaches. Elbaum *et al.*'s approach is limited because once a test is prioritized in the dispatch queue, it will not be reprioritized before it is run missing co-failure information.

A further problem commonly seen in real time tasks scheduling is starvation of low priority tasks. To solve this problem, Elbaum *et al.* [7] introduced an algorithm called prioritize window checking. We further develop this approach with double queues and flexible double queues prioritization models.

Marijian *et al.* [14] also improved test case prioritization in continuous regression testing and used test execution time as one of the metrics to evaluate their algorithm. Their approach assumes a limit on time allotted for test execution and does not make use of runtime results. Furthermore they only prioritize tests in the scope of each change request. Compared to our approach, they did not reprioritize tests and thus prioritization becomes non-optimal after a few tests are run.

Saff *et al.* [20] presented a form of continuous testing where regression tests are run continuously as developers write code. This work, however, focuses on the protocol of continuous testing in CI instead of the regression testing optimization techniques. Jiang *et al.* [9] considered the use of test case prioritization following code commits to help organizations reveal failures faster in continuous integration environments. However their work focus on improving fault localization with test case prioritization techniques and hence solves a different problem.

Our work is novel in the use of co-failure distributions and re-prioritization after each test run. These simple advances allows our approach to outperform the state-of-the-art.

IV. Case Study Setup

Our paper presents an approach to prioritize testing in a multi-request environment. In this section, we detail the methodology, highlighting the data, the prioritization algorithms, and the performance evaluation criteria used to compare the proposed algorithms.

A. Case Study Data

To perform our case study, we require test request and execution data. We obtained data from two different projects - namely, Google's internal test data provided by Elbaum *et al.* [7] and CHROME data, which we mined. We detail each data set below.

Internal Google data: Elbaum *et al.* [7] made the test data used in their study publicly available. The data set provides tests from two phases, the pre-submission and post-submission (of commits) testing. Like Elbaum *et al.* we use the post-submission testing data. The data set contains 11,457 change requests that result in 847,057 test executions, which are executed over 17 days. We also obtain the time cost and the result of each test execution. We separate the data set into two periods since we need two folds, a training data set to obtain the co-failure distributions and a testing data set to evaluate the re-prioritization algorithms.

CHROME: CHROME has two sub-projects: the Chromium web browser and the operation system Chromium OS. Every source code change in CHROME is reviewed and tested after submission. Once a change is submitted, reviewers can select the tests that they think need to run against the change.[1] All

[1]In most cases, these tests can be considered as test suites since they are composed of multiple tests

of the test execution information is recorded in their code review tool. We mine this data to obtain 1) the test requests, 2) the submission time of a test request and 3) the outcome of each test execution. CHROME does not store the running time for each test. As a result, we randomly assigned a running time between 1 and 60 minutes. While this affects the validity of our approach for use on by CHROME developers, it has no impact on our work from a theoretical research perspective of evaluating our test prioritization strategies. Our CHROME data set contains 235,917 change requests that result in 4,487,008 test executions. For experimental purposes, we divided the data into 5 six month periods, which also corresponds to CHROME's six month release cycle.

B. Prioritization Algorithms

To cope with the large number of test requests, prioritization algorithms are often used to facilitate effective test executions. In this section, we first discuss the current state-of-the-art algorithm used to prioritize test execution, FIFOBASELINE [7], [19], [11] and then we present three novel algorithms that we propose to effectively prioritize test executions.

FIFOBASELINE: The most basic algorithm is to provide no prioritization of tests at all. In such a case, tests are prioritized based on a First-In-First-Out criteria. This approach is very common and in our discussions with companies is used by Ericsson, Microsoft and Google by default. When multiple tests are requested at the same time, they are put into a dispatch queue. In our experiments, we set the dispatch queue to a size of 60 per machine. The order of the tests in the dispatch queue is arbitrarily determined, and in most cases is based on the request time. We refer to this as the FIFOBASELINE. Previous work showed that, prioritizing tests randomly is as cost-effective as advanced program analysis based techniques [11].

However, prior work by Kim *et al.* suggested that historical data from test cases can be leveraged to prioritize tests so that they fail earlier [11]. Hence, we are motivated by the aforementioned work and the idea that test failures are not completely independent [22], and devise three algorithms that leverage this insight. Specifically, our algorithms take into consideration the past test co-failure distributions to dynamically prioritize test execution. We adopt a dynamic scheduling protocol in real time task scheduling to perform the test re-prioritization. Full details of the original dynamic scheduling protocol can be found in Chetto *et al.*'s task scheduling work [2]. Essentially, the algorithm updates the test order in the dispatch queue based on the result (i.e., pass/fail) of the recently completed tests in the same request. The computation of the modified priority is calculated based on the following scoring function.

Given a pair of tests t_1, t_2 in the same request, the scoring function is given as follows:

$$new\_sc = previous\_sc + (P(t_2 = fail | t_1 = fail) - 0.5) \quad (1)$$

If t_1 is passed, then:

$$new\_sc = previous\_sc + (P(t_2 = fail | t_1 = pass) - 0.5) \quad (2)$$

If t_2 is a newly added test, then we can either set its conditional failure rate to 1 to give it the maximum priority, or 0.5 to maintain its original priority. However, since we are only interested in the performance of re-prioritization based on past co-failure, we did not reprioritize new tests in our simulation. Equation 1 predicts the failure rate of a test by its co-failure history, while Equation 2 is another way to estimate its failure rate. Algorithm 1 explains the procedure on how to reprioritize a test based on the two aforementioned equations. In Algorithm 1, line 2 checks whether the input test t is a recently added test, namely no co-fail history. If t is not a new test then reprioritize t by Equation 1 at line 4 if t_{finish} fails or by Equation 2 at line 6 otherwise. Algorithm 1 takes no action if t is a new test. The time complexity of $rePrioritize()$ is $O(1)$.

Algorithm 1: $rePrioritize(t, t_{finish})$

```
   input  : t_finish : the test that just finished its execution
            t : the test that needs to be reprioritized
   Result: reprioritize t based on the result of t_finish
1  initialization;
2  if t is not a new added test then
3  |  /* Apply Equation 1 to update the
   |     priority of t if t_finish fails.   */
4  |  if t_finish failed then t.setPriority(t.getPriority
   |   +(P(t fails|t_finish fails) − 0.5)) ;
5  |  /* Apply Equation 2 to update the
   |     priority of t if t_finish passes.
   |     */
6  |  else t.setPriority(t.getPriority
   |   +(P(t fails|t_finish passes) − 0.5)) ;
7  |  /* Only reprioritize tests with
   |     past co-fails. No priority
   |     update to new added tests          */
8  end
```

CODYNAQSINGLE: Our first enhancement is the *single queue test re-prioritization scheme, called* CODYNAQSINGLE, which executes tests concurrently and reprioritizes tests based on their past co-failure distribution. Initially, tests that are added to the queue are simply based on their request time. Then the tests are reprioritized according to the newly generated information and their co-failure history. The pseudo code of CODYNAQSINGLE is shown in Algorithm 2. CODYNAQSINGLE operates on the dispatchQueue which is a priority queue of tests that are waiting to run. We assume

Table II: Summary of Re-prioritization Algorithm Features

Technique	Prioritization Scheme	Concurrent Request Execution	Number of Queues	Re-prioritization	Starvation Control
FIFOBASELINE	Arrival time FIFO	No	1	No	No
GOOGLETCP	recent failed test distribution	Yes	2	No	Yes, when the elapsed time matches the prioritize window
CODYNAQSINGLE	Past co-failure distribution	Yes	1	Yes, after each test run	No
CODYNAQDOUBLE	Past co-failure distribution	Yes	2	Yes, after each test run	Yes, after the the dispatch queue is empty
CODYNAQFLEXI	Past co-failure distribution	Yes	2	Yes, after each test run	Yes, after the size of the dispatch queue is below a threshold

that the dispatchQueue is smaller than the waitingQueue, which is the most likely practical scenario. We also assume that two important functions are already implemented in the test infrastructure: $getFinishTests()$ listen on the test executors and returns an array of recently finished tests; $getOtherTests(t_{finish}, dispatchQueue)$: a function takes a finished test as its argument and returns the other tests that are in the same test request and waiting for execution. Given m the total of test executors and n the largest number of tests that would be included in a test request, $getFinishTests()$ is $O(m)$ and $getOtherTests()$ is $O(n)$. Hence CODYNAQSINGLE is $O(mn)$.

Algorithm 2: CODYNAQSINGLE ()

```
Result: reprioritize the relative tests in the dispatch queue
1  initialization;
2  while TRUE do
3      finishTests ← getFinishTests();
4      for t_finish ∈ finishTest do
5          otherTests ← getOtherTests(t_finish, dispatchQueue);
6          for t ∈ otherTests do
7              rePrioritize(t, t_finish);
8          end
9      end
10 end
```

During the simulation on the CHROME and Google data, we found that in some cases tests face *the starvation problem* in CODYNAQSINGLE. Some tests are highly unlikely to fail, these tests are assigned a low priority and since new tests request constantly arrive, these low failure probability tests will never be run. In essence, tests that have a higher priority are always jumping the line and being executed, while tests that have a low priority are being 'starved' and being constantly pushed to the back of the queue. This would not be a problem if these low priority tests never fail. However, in cases where they do actually fail, then this behaviour is undesirable since the developer requesting the test would not have any feedback for a long time. Therefore we develop our second algorithm, the CODYNAQDOUBLE.

CODYNAQDOUBLE: To address the aforementioned starvation problem, we propose another prioritization algorithm called CODYNAQDOUBLE. In CODYNAQDOUBLE, we limit the capacity of the dispatch queue and add another FIFO waiting queue to temporally store the newly arriving test requests.

When new tests are requested, they are stored in the waiting queue. After the dispatch queue is cleared, we fill it with tests in the head of the waiting queue. Then we execute test request concurrently and reprioritize tests in the dispatch queue the same way as in CODYNAQSINGLE. The pseudo code of CODYNAQDOUBLE is shown in Algorithm 3. In contrast with CODYNAQSINGLE, the $dispatchQueue$ is a priority queue with predefined capacity of tests that are prioritized and waiting to run. CODYNAQDOUBLE uses the $waitingQueue$, a FIFO queue of tests, to hold the requested tests before they are added to the $dispatchQueue$ for prioritization. In line 3 to 6, when CODYNAQDOUBLE finds that the $dispatchQueue$ is cleared, it pulls tests from the $waitingQueue$ and adds them to the $dispatchQueue$. Once a test is finished, the reprioritize algorithm is the same as CODYNAQSINGLE. Hence CODYNAQDOUBLE is $O(mn)$ as well.

The approach that controls the size of the prioritized dispatch queue is very similar to the prioritization window checking approach in Elbaum *et al.*'s GOOGLETCP [7]. The main difference however, is that in our approach when the dispatch queue is empty, only a few waiting tests will be added to the dispatch queue based on the dispatch queue capacity, while in GOOGLETCP, all waiting tests will be added to the dispatch queue and prioritized. Although our proposed algorithm helps reduced the starved tests in our simulation, the improvement is not substantial. Therefore we develop our third model that we describe next.

CODYNAQFLEXI: While CODYNAQDOUBLE reduces test starvation, we found that many tests still experience substantial delays. In order to minimize the number of delayed test ex-

Algorithm 3: CODYNAQDOUBLE ()

Result: reprioritize the relative tests in the $dispatchQueue$; Fill the $dispatchQueue$ with tests from the $waitingQueue$ when the $dispatchQueue$ is cleared

```
1  initialization;
2  while TRUE do
3      if dispatchQueue is empty then
4          while dispatchQueue is not full ∧
             waitingQueue is not empty do
5              dispatchQueue.insert(waitingQueue.poll())
6          end
7      end
8      finishTests ← getFinishTests();
9      for t_finish ∈ finishTest do
10         otherTests ← getOtherTests();
11         for t ∈ otherTests do
12             rePrioritize(t, t_finish);
13         end
14     end
15 end
```

Algorithm 4: CODYNAQFLEXI ()

input : p: the prioritize window ranges between 0 to 1

Result: reprioritize the relative tests in the dispatch queue; Fill the $dispatchQueue$ with tests from the $waitingQueue$ when the size of $dispatchQueue$ is smaller than $p * c$

```
1  initialization;
2  while TRUE do
3      if dispatchQueue.size ≤
         p * dispatchQueue.capacity then
4          while dispatchQueue is not full ∧
             waitingQueue is not empty do
5              dispatchQueue.insert(waitingQueue.poll())
6          end
7      end
8      finishTests ← getFinishTests();
9      for t_finish ∈ finishTest do
10         otherTests ← getOtherTests();
11         for t ∈ otherTests do
12             rePrioritize(t, t_finish);
13         end
14     end
15 end
```

ecutions, we add a prioritization window to CODYNAQDOUBLE and develop CODYNAQFLEXI. In CODYNAQFLEXI, if the size of the dispatch queue is less than the prioritize window which is the product of a predefined threshold times the dispatch queue capacity in our implementation, the tests in the head of the waiting queue will be prioritized randomly and added to the dispatch queue until the dispatch queue is full. The prioritize window is the size at which we add waiting tests to the dispatch queue. This allows us to add tests to the dispatch queue and achieve a tradeoff between starving (low priority) tests and getting higher gain for highly risky tests, with high priority. Once a test is finished, all other tests that are in the same request and waiting in the waiting queue or dispatch queue, are reprioritized in the same way as in the previous two algorithms. Algorithm 4 displays the steps of CODYNAQFLEXI. The only difference between CODYNAQFLEXI and CODYNAQDOUBLE is the condition on which to trigger the filling of the $dispatchQueue$. In line 3 of Algorithm 4, when the number of prioritized tests in the dispatch queue is less or equal to the product of the prioritize window p and dispatch queue capacity c, CODYNAQFLEXI fills the $dispatchQueue$ with tests from the $waitingQueue$. The time complexity of CODYNAQFLEXI is the same as that of CODYNAQDOUBLE.

Figures 3, 4, and 5 illustrate the main differences between the three algorithms. Table II summarizes the different features of the prioritization algorithms.

C. Performance Evaluation Measures

To measure the performance of our prioritization algorithms, we use three measures 1) time to the first failure (FIRSTFAIL), 2) time to detect all failures in a test suite (ALLFAIL) and the percentage of delayed failure tests. We detail each measure below:

Speedup in the First Failure Detection (FIRSTFAIL). When developers submit a change to be tested the first feedback they can respond to is a failing test or, if there are no failures, the time to pass all the tests. We refer to this measure as the First Failure Detection Time (FIRSTFAIL). We measure FIRSTFAIL in testing time saved. This measure is very similar to the measure Response Time in system programming which is used to evaluate how fast a shared system reacts to the user [12].

For a given test request, *Gain* in FIRSTFAIL is calculated as taking the difference between the FIRSTFAIL of the baseline algorithm and that of the algorithm under evaluation. Then *Speedup* in FIRSTFAIL is calculated by Equation 3:

$$speedup = \frac{median\_gain}{median\_waiting\_time} \tag{3}$$

If speedup in FIRSTFAIL is positive, then the evaluated algorithm provides result for a single change earlier than the FIFOBASELINE algorithm and vice versa.

Speedup in All Failure Detection (ALLFAIL). For a given failing suite (which includes many tests), ALLFAIL is the waiting time for a test failure to be reported for the entire

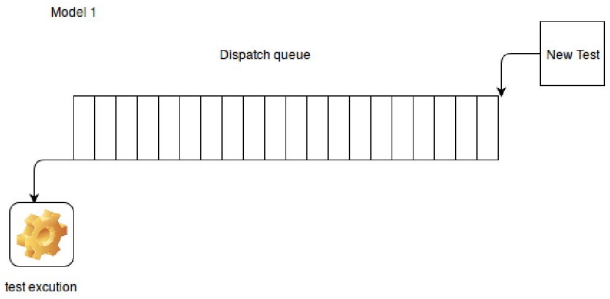

Figure 3: CODYNAQSINGLE

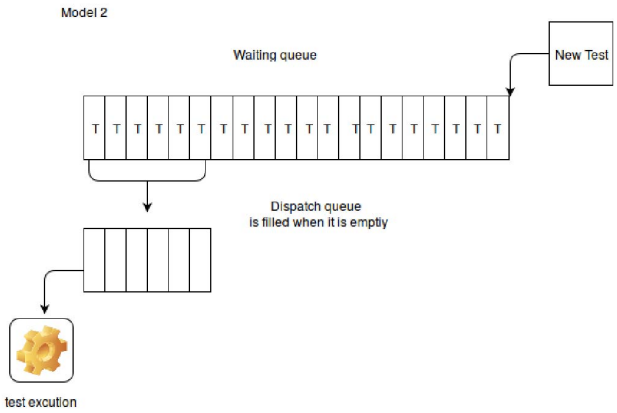

Figure 4: CODYNAQDOUBLE

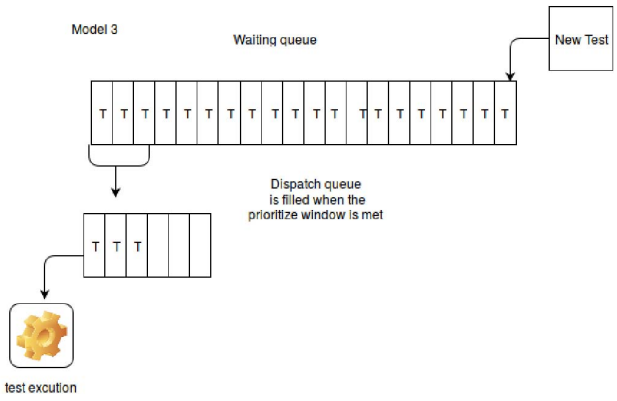

Figure 5: CODYNAQFLEXI

test suite. In our case, we use the period as an indicator of a test suite. Using ALLFAIL gives us a different perspective from FIRSTFAIL, since FIRSTFAIL is about how quickly we get results for a single change, whereas ALLFAIL gives us an indication of how fast we get feedback for *all* test failures. In many ways, these two measures complement each other. FIRSTFAIL tells us how quickly the prioritization is able to provide feedback for a change, whereas ALLFAIL tells us how well the prioritization can handle the starvation problem, since it considers all test failures.

The gain in ALLFAIL is calculated as the difference between the time it takes for all failures to be found in the tests suite using the baseline algorithm and the evaluated algorithm. And the speedup in ALLFAIL is given by Equation 3. If speedup in ALLFAIL is positive, then the evaluated algorithm fails the given tests earlier than the FIFOBASELINE algorithm and vice versa.

Percentage of Delayed Failure Detection. Another view of how well an algorithm copes with starvation is to measure the difference in delayed test failures compared to the FIFOBASELINE. The main difference between percentage of delayed failure detection and ALLFAIL lies in the fact that ALLFAIL measures the relative amount of time gain, while percentage of delayed failure detection does not consider the amount of delay but just the percentage of delayed failures. For example, if many failures were delayed by a negligible amount ALLFAIL would still report good results, whereas percentage of delayed failure detection would not. Since this measure is related to delayed failure detection, the smaller the value the better.

V. CO-FAILING BEHAVIOUR OF CHROME TESTS

Prior to delving into our results, we wanted to examine the co-failing behaviour of tests in CHROME. Generally, test prioritization work assumes that tests are independent from each other [6]. However, we believe that tests are not independent, in particular for system level testing [22][3]. Some tests may cover different functionality located in the same file collection, while other tests may cover different files where a high degree of file dependency is found. These tests tend to fail together and result in co-failing behavior. In both cases, the basis of the co-failing behavior comes from the inherent interrelation inside the file system, which is not likely to change. Hence, we believe that the co-failing behavior is a long term phenomenon.

We used the machine learning library, Apache Spark, to capture the frequently co-failing test suites in the CHROME's test data and generate Table III. The column Test Pair is the pair of tests under investigation, the column Fail Together is the number of times they all failed in the same request, the column Run Together is the number of times the given test pair were executed in the same request, the column Period is the investigated period. For performance reasons, we used the FP-Growth algorithm which is described by Han *et al.* [8].

Table III shows that some groups of test suites failed together frequently in both periods 1 and 2. For example, the test pair `mac_rel` and `win_rel`, were run together 36,344 times and failed together 6,133 times in period 1 while they were run together 31,501 times and failed together 4,375 times in period 2. The conditional probability that, given `mac_rel fails`, `win_rel` also fails is 74% in

period 1 and 78% in period 2. This observation illustrates our proposition about the co-failing behaviour of tests.

This co-failing behaviour justifies our motivation that, re-prioritizing tests can be beneficial since it can delay tests that are more likely to co-fail and give higher priority to tests that have low co-failure probability (i.e., have tests that are most different from the failing tests run first). This prioritization calculation is performed after (and based on the outcome of) each test run, so that we can always update how likely the remaining tests will fail and keep the ordering of tests as accurate as possible.

VI. Case Study Results

In this section, we perform simulations to evaluate the performance of each algorithm. We perform our case study using the GoogleTCP algorithm and our three proposed algorithms namely - CoDynaQSingle, CoDynaQDouble and CoDynaQFlexi. We compare the algorithms using both data sets, Chrome and internal Google data. In particular, we compare the algorithms in terms of the speedup for FirstFail, AllFail and the percentage of delayed test failures. Table IV shows the speedup for FirstFail and AllFail and the percentage of delay test failures.

FirstFail: Table IV shows the median speedup for each algorithm over the FifoBaseline. The best performing algorithm is highlighted in bold. In terms of FirstFail we see that CoDynaQSingle provides the greatest speedup for Chrome of 11.33%, and CoDynaQSingle also provides the greatest speedup for Google of 31.01%. In contrast, the GoogleTCP a speedup for Chrome is only 3.34% over the baseline and the speedup for Google of 4.36%.

AllFail: From Table IV when a developer wants to find all failures, AllFail, CoDynaQSingle performs best for the Chrome data set, achieving a speedup of 61.84%, for the Google data set, CoDynaQSingle also performs best, achieving a speedup of 39.6%. The speedup for GoogleTCP is 0.04% for Chrome and 4.61% for Google, making CoDynaQSingle is 61.8% and 33.19% faster than GoogleTCP.

Percentage of Delayed Test Failures: In terms of test starvation, *i.e.* delay, Table IV shows that the shortest delay for Chrome is achieved by CoDynaQFlexi, with less than 1% of the test failures being delayed. For Google, CoDynaQFlexi again achieves the shortest delay with 6.20% of test being delayed compared to the baseline. While GoogleTCP delays 31.78% of test failures in Chrome and 8.22% in Google.

Best overall algorithm: All evaluated algorithm provides positive speedup in FirstFail and AllFail. CoDynaQSingle drastically outperforms the other algorithms in terms of speedup. Especially, it achieves a very significant speedup in AllFail for Chrome. However, there is starvation revealed by the percentage of delayed failures. With CoDynaQDouble we control starvation by setting a longest waiting time for the tests in the dispatch queue. We only add new tests after all tests in the dispatch queue have been run. The longest waiting time for a test already in the dispatch queue would be the total execution time of all the other tests in the dispatch queue. CoDynaQDouble does only slightly reduces the portion of delayed test failures. Therefore, we proposed CoDynaQFlexi, The problem with CoDynaQDouble is that even a test in the waiting queue with a high priority through reprioritization will still have to wait for the tests that are already added to the dispatch queue. While in CoDynaQFlexi, high priority tests have the the chance to jump in the line without waiting for all the tests that are added in the dispatch queue earlier. CoDynaQFlexi offers the lowest percentage of delayed test failures.

Overall, we see that GoogleTCP does not perform well for all three performance measures on the Chrome data. The ordering provided by GoogleTCP did not produce a substantial speed up over the simple FifoBaseline. However, the performance of GoogleTCP for all performance measures is close to our best approach for the Google data sets. Hence, next we examine the characteristics of the different data sets in order to better understand the variance in performance.

VII. Discussion

As discussed in section VI, GoogleTCP only performs well on the Google data set. While CoDynaQSingle always offers the best speedup and CoDynaQFlexi provides the lowest percentage of delay test failures with an accurate ordering of tests for both, the Google and Chrome, data sets. In this section, we analyze the distribution of the test failures and present measures, related to continuous testing, to better understand the performance of our findings. In addition to better understanding our results, our goal is to examine the underlying test failure distribution to influence our decision on the best prioritization strategy for a particular continuous testing environment.

We use three measures to examine the data: test execution failure rates, kurtosis of failure distributions and co-failures.

Execution Failure Rate measures the overall failure rate of tests in the data set. Particularly, we calculate the Failure Rate as the fraction of the number of failed test executions over the total number of test executions.

Kurtosis of Failure Distributions examines which tests tend to fail. In particular, kurtosis measures how skewed the test failures are. If test failures are distributed evenly throughout the set of tests, then the failure distribution will have a low KURTOSIS. If on the other hand, the test failures are concentrated on a small set of tests, then the failure distribution will have a high KURTOSIS. We use the kurtosis coefficient to measure the kurtosis of the test distribution [4].

Proportion of Test Co-Failures (or Cohesion): We use two measures of cohesion: (1) the portion of tests that the co-fail in a period of time and (2) the co-failure rate of the tests

Table III: Frequent Co-failing Test Suites at Periods 1 and 2

Test Pair	Fail together	Run Together	Period
mac_rel,win_rel	6,133	36,344	1
mac_rel,win_rel	4,375	31,501	2
linux_CHROMEos,win_rel	6,113	35,599	1
linux_CHROMEos, win_rel	4,918	30,605	2
linux_rel,win_rel	5,902	36,366	1
linux_rel,win_rel	4,087	31,392	2
linux_aura,linux_CHROMEos	5,259	34,942	1
linux_aura,linux_CHROMEos	4,121	25,608	2
linux_rel,mac_rel	5,237	37,010	1
linux_rel,mac_rel	3,515	31,492	2
linux_rel,linux_CHROMEos	5,209	35,639	1
linux_rel,linux_CHROMEos	3,648	30,585	2
linux_aura,win_rel	5,082	34,794	1
linux_aura,win_rel	3,874	25,549	2

Table IV: Median of Gain of FIRSTFAIL, ALLFAIL and Percentage of Delayed Failures Compared to the FIFOBASELINE

CHROME	CODYNAQSINGLE	CODYNAQDOUBLE	CODYNAQFLEXI	GOOGLETCP
FIRSTFAIL	**11.33%**	0.84%	5.01%	3.34%
ALLFAIL	**61.84%**	0.05%	0.08%	0.04%
Delayed Failures	19.11%	17.34%	**0.51%**	31.78%
Google				
FIRSTFAIL	**31.01%**	0.04%	0.18%	4.36%
ALLFAIL	**39.60%**	0.09%	0.20%	4.61%
Delayed Failures	31.14%	27.26%	**6.20%**	8.22%

Table V: Failure Rate of Data sets

Data set	Failure Rate	Kurtosis
CHROME	12.53 %	22.90
Google	0.29%	3228.51

Table VI: Proportion of Test Co-Failures

	Over All Tests	Over Failed Tests
CHROME	58.21%	64.86%
Google	0.90%	43.10%

in the next period. The first measures the short-term cohesion of the tests. The second, measures the degree to which the current period can be used to predict the subsequent period.

A. Test Failure Rates

We measure the failure rate of tests in the two data sets and present them in Table V. We observe that the Google data set has a very low failure rate. In fact, upon closer analysis, four tests were responsible for 51.3% of the failures in the data set, and the one test is responsible for 28.4% of the failures. In effect we can these four tests tests to find 50% of all failures in Elbaum *et al.* [7] Google data set. In contrast, the CHROME data set has more evenly distributed failure rate, where 89.73% of the tests have failed at least once. Google's highly skewed data set artificially inflates the effectiveness of GOOGLETCP and we find that on the CHROME data set it performs much less well than CODYNAQ.

B. Kurtosis of Failure Distributions

In addition to examining the failure rate, we also investigate the kurtosis of the test failure distributions. Table V shows the kurtosis values of the two datasets. We observe that the Google data set has a very large kurtosis value, indicating that the majority of the failures are concentrated in a few tests. Since the GOOGLETCP algorithm is designed to prioritize tests that recently fail over tests that never fail or have not failed in a long time, it performs well on such highly-concentrated data sets. That said, we see that CODYNAQ performs well in both data sets, even in the Google data set, which is promising.

C. Proportion of Test Co-Failures (a.k.a Cohesion)

Since CODYNAQ mainly relies on co-failure history to predict the failure rate, we suspect that the cohesion of test co-failures would have an important impact on its performance. Hence, we examine the test cohesion in order to better understand when to select the most appropriate algorithm. Table VI shows the co-failure proportions for all tests and failed tests. We see that in the Google data set, the proportion of co-failures (all tests) is low. Since only 2.78% of tests in the Google data set fail, when we examine co-failure among test that have failed at least once, we see a high proportion of test co-failures, 43.1% . On the other hand, CHROME has a high proportion of test co-failures in both, all tests and failed tests.

Triangulating our findings from Table VI and Table IV we can see that when the proportion of test co-failures is low over all tests (e.g., Google), either GOOGLETCP,

CODYNAQSINGLE or CODYNAQFLEXI are good choices for prioritization algorithms. However, if the tests that fail are evenly distributed or the co-failures rate is high (e.g., CHROME), then the CODYNAQSINGLE or CODYNAQFLEXI is superior.

VIII. THREAT TO VALIDITY

Internal Validity: Internal validity examines the factors that could have influenced our results. To obtain the test execution information for CHROME, we mine publicly available test run data. Although we obtained over 4.4 million executions, some results are only viewable by internal to Chrome developers.

We fix the size of the dispatch queue to 60 per machine when performing our experiments. Although varying the size of the dispatch queue may impact the results, since our goal is the comparison of the different prioritization techniques, the key point is to ensure that the queue sizes are the same when evaluating the different prioritization techniques. Hence, we believe that as long as the sizes are the same for the different experiments, there is no impact on our findings.

Construct Validity: Threats to constructed validity concern the relationship between theory and observation. We use three metrics, FIRSTFAIL, ALLFAIL and delayed failures to evaluate the performance of our approach and compare it to the GOOGLETCP and FIFOBASELINE approaches. We chose these metrics because they have been used in the evaluation of prior works. However, these are not the only measures that can be used to evaluate test re-prioritization algorithms and different measures may yield to different results.

Test are considered to be co-failing if they fail together. However, we do not necessarily determine if these tests co-failed due to the same failure. That said, we believe that tests that co-fail together, even if due to different failures, are interesting since this co-failure may point to a hidden or indirect interaction between the tests, which clearly leads to a co-failure occurring.

External Validity: Threats to external validity concern the generalization of our findings. We study the internal testing results from Google and the open source project Chrome. Chrome is lead by Google and many of its developers are Google employees. Our results may not generalize to other companies or projects. As discussed earlier, the test failure rate distributions play a major role in the performance of the different re-prioritization algorithms, hence, our results may not generalize to other projects that have different test re-prioritization failure distributions.

IX. CONCLUSION AND FUTURE WORK

As continuous integration becomes more popular, the need for continuous testing is also growing. However, given the large number of software changes being continuously integrated everyday, delays may occur. Effective test prioritization is a requirement to reducing the continuous integration cycle time. In this paper, we present test re-prioritization algorithms that take into account test co-failures, namely CODYNAQ. We compare the proposed algorithms to the FIFOBASELINE approach and the state-of-the-art approach, GOOGLETCP. We evaluate our approach using three metrics, FIRSTFAIL, ALLFAIL and the percentage of delayed failures. We show that our approach significantly outperforms both, the FIFOBASELINE and the GOOGLETCP approaches for CHROME by over 61%. The GOOGLETCP approach is ineffective on CHROME data. GOOGLETCP was designed for the Google data set, however, CODYNAQ outperforms GOOGLETCP by 33.19% on this data set. We note that the Google data set from [7] is highly skewed with only 2.78% of the test ever failing, which artificially increasing the effectiveness of GOOGLETCP which simply runs recently failed tests first.

In the future, we plan to evaluate the approach on other projects and to continue to examine the underlying failure distributions to further improve test prioritization. Moreover, we plan to explore the use of other prioritization techniques, such as test coverage-based prioritization.

REFERENCES

[1] K. Beck. *Extreme programming explained: embrace change*. addison-wesley professional, 2000.

[2] H. Chetto, M. Silly, and T. Bouchentouf. Dynamic scheduling of real-time tasks under precedence constraints. *Real-Time Systems*, 2(3):181–194, 1990.

[3] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn. Constructing test suites for interaction testing. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 38–48. IEEE, 2003.

[4] L. T. DeCarlo. On the meaning and use of kurtosis. *Psychological methods*, 2(3):292, 1997.

[5] P. M. Duvall, S. Matyas, and A. Glover. *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

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

[7] S. Elbaum, G. Rothermel, and J. Penix. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 235–245, New York, NY, USA, 2014. ACM.

[8] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *ACM SIGMOD Record*, volume 29, pages 1–12. ACM, 2000.

[9] B. Jiang, Z. Zhang, T. Tse, and T. Y. Chen. How well do test case prioritization techniques support statistical fault localization. In *2009 33rd Annual IEEE International Computer Software and Applications Conference*, volume 1, pages 99–106. IEEE, 2009.

[10] R. Just, G. M. Kapfhammer, and F. Schweiggert. Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis. In *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*, pages 11–20. IEEE, 2012.

[11] J.-M. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, pages 119–129. IEEE, 2002.

[12] L. Kleinrock. *Queueing systems, volume 2: Computer applications*, volume 66. wiley New York, 1976.

[13] N. Kukreja, W. G. Halfond, and M. Tambe. Randomizing regression tests using game theory. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 616–621. IEEE, 2013.

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

[15] C. Nguyen, P. Tonella, T. Vos, N. Condori, B. Mendelson, D. Citron, and O. Shehory. Test prioritization based on change sensitivity: an industrial case study, 2014.

[16] C. D. Nguyen, A. Marchetto, and P. Tonella. Change sensitivity based prioritization for audit testing of webservice compositions. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 357–365. IEEE, 2011.

[17] A. Panichella, R. Oliveto, M. Di Penta, and A. De Lucia. Improving multi-objective test case selection by injecting diversity in genetic algorithms. *IEEE Transactions on Software Engineering*, 41(4):358–383, 2015.

[18] X. Qu, M. B. Cohen, and G. Rothermel. Configuration-aware regression testing: an empirical study of sampling and prioritization. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 75–86. ACM, 2008.

[19] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Test case prioritization: An empirical study. In *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*, pages 179–188. IEEE, 1999.

[20] D. Saff and M. D. Ernst. An experimental evaluation of continuous testing during development. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 76–85. ACM, 2004.

[21] R. K. Saha, L. Zhang, S. Khurshid, and D. E. Perry. An information retrieval approach for regression test prioritization based on program changes. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 1, pages 268–279. IEEE, 2015.

[22] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin. Empirically revisiting the test independence assumption. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 385–396, New York, NY, USA, 2014. ACM.