Cost-Effective Regression Testing Using Bloom Filters in Continuous Integration Development Environments

Jung-Hyun Kwon, In-Young Ko
School of Computing, Korea Advanced Institute of Science and Technology
Daejeon, Republic of Korea
{junghyun.kwon, iko}@kaist.ac.kr

***Abstract*—Regression testing in continuous integration development environments must be cost-effective and should provide fast feedback on test suite failures to the developers. In order to provide faster feedback on failures to developers while using computing resources efficiently, two types of regression testing techniques have been developed: Regression Testing Selection (RTS) and Test Case Prioritization (TCP). One of the factors that reduces the effectiveness of the RTS and TCP techniques is the inclusion of test suites that fail only once over a period. We propose an approach based on Bloom filtering to exclude such test suites during the RTS process, and to assign such test suites with a lower priority during the TCP process. We experimentally evaluate our approach using a Google dataset, and demonstrate that cost-effectiveness of the proposed RTS and TCP techniques outperforms the state-of-the-art techniques.**

I. Introduction

Continuous integration (CI) is a development practice in which developers integrate their code into the mainline codebase often [1]. This practice involves maintaining a code repository, automating the build, committing to the code repository every day, etc. CI has many advantages such as early failure detection, and reduction of cost for the repeated integration process. Due to these advantages, large software companies such as Google, LinkedIn, and Netflix have applied CI [2], [3], [4] to their development processes.

One of the advantages of adopting CI is fast feedback on code failures. In a CI environment, a developer can integrate the software components early and often, and test whether the integrated components work correctly with other components. This integration process is often activated multiple times in a day (e.g., whenever the developer checks in to the mainline codebase). As a result, the developer can receive a faster feedback on test failures, which allows the developer to fix the failures early.

Fast feedback becomes even more important when testing large-scale software. There are usually a large number of test cases for large-scale software, and therefore, it takes a long time to test the entire software and receive feedback on failures. For instance, in Microsoft, the full regression test for a software product takes several days even though the test is performed by using a number of computers simultaneously [5]. In addition to the fast feedback, efficient utilization of computing resources is important as well. It is because when a large number of test cases are run, available computing resources might be run out [6].

In order to provide faster feedback on failures to developers while using computing resources efficiently, two types of regression testing techniques have been actively researched: Regression Testing Selection (RTS) and Test Case Prioritization (TCP) [7]. The RTS techniques provide a way of finding the subset of a test suite that conforms to certain selection criteria. One example of such a selection criteria is the selection of test cases that test the modified parts of a software. On the other hand, the TCP techniques involve the ordering of test cases in such a way as to maximize one or more objectives (e.g., to maximize the rate of fault detection).

The traditional RTS and TCP techniques require the use of code instrumentation, which involves embedding additional code to the target software in order to obtain a particular set of property values, such as code coverage. However, instrumenting code and obtaining property values are time-consuming tasks [8], and sometimes it is not possible to obtain the correct property values in CI environments, because the rate of code churn is usually fast in CI environments, and the instrumentation on the code becomes out-of-date, and needs to be modified according to the code changes [6]. Therefore, there have been some researches on developing new RTS and TCP techniques that do not require code instrumentation, and therefore, they can be applied to CI environments in an effective manner [6], [9], [10], [11].

In this paper, we propose cost-effective RTS and TCP techniques for CI environments. Our approach is motivated by a recent study that uses historical test-result data to solve the problems with RTS and TCP techniques [6]. The study is assumed that the test suites[1] that recently failed or did not run for a while are likely to fail again. Thus, those test suites are selected in RTS, and given higher priority in TCP. However, this basic assumption may not be true in some cases.

[1]In the study, the RTS and TCP techniques are applied at test-suite level, because the test data that the study used consists of test suites. Generally, a test suite is regarded as fail if at least one of the test cases in the test suite is failed, and otherwise, the test suite is regarded as pass. Because there is usually a large number of test suites available for large-scale software, applying the RTS and TCP techniques at the test-suite level might be effective to give faster feedback on failures to developers. In addition, the techniques can be also applied at test case level.

978-1-5386-3681-7/17 $31.00 © 2017 IEEE
DOI 10.1109/APSEC.2017.22

For instance, there can be a case in which a number of test cases that failed recently do not fail again. When such test suites are selected in RTS and given a higher priority in TCP, the feedback on failures revealed by other test suites can be delayed.

In this study, we first show that among the failed test suites in a sanitized Google dataset, there is a large number of test suites that failed only once [12]. Then, we propose new RTS and TCP methods that use the Bloom filter, which is a fast and space-efficient data structure, for avoiding or giving lower priority to such test suites. The RTS method can reduce the number of tests to be run, and therefore, the test time can be significantly reduced, compared to the methods that do not use the Bloom filter, and thus the RTS method can give more faster feedback on failure to a developer than the RTS method. The TCP method considers the results from the Bloom filter, and adds one more priority level to the previous TCP method, resulting in a more cost-effectiveness TCP method.

In the empirical study, the proposed RTS and TCP techniques are compared to the state-of-the-art approaches. In RTS, our approach achieves that the average number of failures detected by a single test suite is higher than the baseline approach (2.23 times higher on average). In addition, our TCP method detects test failures early, and therefore, it reduces the failure detection time by as much as 4.9 to 42.2 h.

The contributions of this work are as follows. This work identifies a characteristic of large-scale CI test datasets that can affect the cost-effectiveness of RTS and TCP techniques. In addition, this work suggests using Bloom filters in RTS and TCP, which enables fast and scalable operations in large-scale CI environments.

The rest of the paper is organized as follows. Section II explains some background information related to our approach. In Section III, we explain the result of a post-hoc analysis on the Google CI dataset, explain the motivation for our study, and describe our approach of selecting and prioritizing test suites using Bloom filters. In Section IV, we describe the experiment and analyze its result. The summary and a description about our future work is provided in Section V.

II. Background and Related Work

A. Regression test case selection and test case prioritization techniques

When a version v_1 for a program P is updated to a version v_2, the test cases passed in v_1 may fail in v_2. Regression testing is an activity that tests the existing functionalities of P by running existing test cases when P is updated. As it can take a long time to run all the test cases every time when a version is changed, and require a lot of computing resources for such recurrent tests, Regression Test case Selection (RTS) and Test Case Prioritization (TCP) techniques are being studied to solve these problems.

The RTS techniques reduce the cost of regression testing by identifying the test cases for the code that is changed after version changes and running only a subset of the entire test suite. Using these techniques, the test cases that execute the newly added code, changed cases, and deleted code can be identified and selected [13]. One of the most popular RTS techniques is to generate graphs such as control flow graphs or data flow graphs from the static and dynamic analyses of code, and identify the changed parts in the graphs by using a graph traversal method. These methods mainly require the execution trace or coverage information of each test case, which can be obtained after the code instrumentation [14], [15].

The TCP techniques achieve one or more goals such as maximizing the fault detection rate or achieving a target test coverage as soon as possible by scheduling the sequence of test cases [16], [17]. Priority strategies vary according to the purpose, and the test cases are sorted according to the strategy. TCP techniques make it possible to find many failures within a given time when the test time is limited. The existing TCP techniques mainly adopt a strategy of scheduling test cases based on the coverage information of each test case [18], [19].

The existing RTS and TCP methods require code instrumentation, and an execution trace or coverage information for each test case. However, recent study shows that test coverage is not highly related to fault detection effectiveness of the test suites [20]. In addition, existing studies assume that there is sufficient time to obtain these information about the newer version of code. However, for a large-scale software, there is a large number of test cases, and it can take a long time to obtain such information. Furthermore, in a CI environment, the time between different software versions is shorter than that in traditional software development environments because multiple developers may check in their changes in code into the mainline codebase frequently. Therefore, it is difficult to use the existing RTS and TCP methods for CI environments.

B. Regression testing techniques in CI environments

The RTS and TCP methods proposed in recent studies use time windows that track the execution history and the test request queue where test suites submitted for execution are queued [6]. Unlike traditional methods, these methods do not require code instrumentation or code coverage information, and therefore, they enable developers to keep up with rapid and frequent code changes and ease the software integration in CI environments. In our study, we also use the time windows — the failure window (W_f), execution window (W_e), and prioritization window (W_p). W_f and W_e keep track of the time range of the execution history used in both RTS and TCP techniques. W_f tracks the last failure of each test suite in the range, and W_e tracks the last execution of each test suite in the range. W_p is used only in the TCP technique, and represents the time range over the queue of test suites. These windows are depicted in Figure 1. In the RTS method, a subset of the test suites based on the last failure time (LFT) and the last execution time (LET) of each test suite is chosen. If the LFT of a test suite is less than W_f, the LET of the test suite is more than W_e, or the test suite is new, the test suite is selected. In the TCP method, the scheduling of test suites starts when the test request time interval between the first test suite and the last test suite in the queue exceeds W_p. The

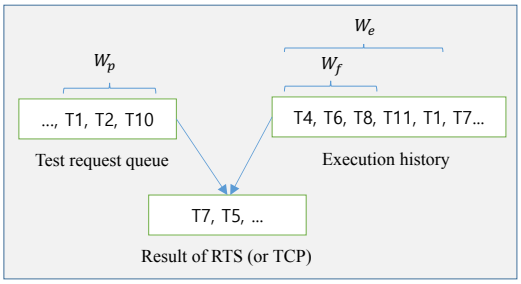

Fig. 1. Failure window (W_f) and execution window (W_e) over the execution history, and prioritization window W_p over the test request queue. New test request is accumulated to the left side of the queue, and the test suites are executed from the most right side of the queue. RTS and TCP techniques are applied before test suites in the queue are executed. The most recently executed test suite is appended to the left of the test history.

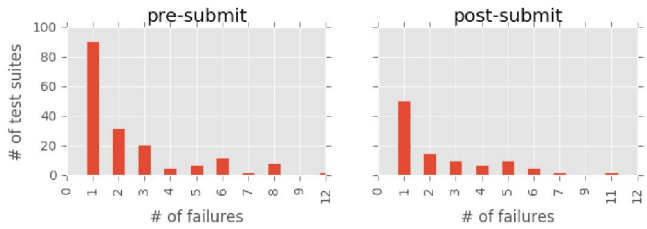

Fig. 2. Number of failures that are detected by test suites in the pre-submit and post-submit phases

priority of each test suite is determined based on W_f and W_e. If the LFT of a test suite is less than W_f, the LET of the test suite is bigger than W_e, or the test suite is new, the priority of the test suite is set to 1; otherwise, it is set to 2. Then, the test suites are ordered in an ascending order according to the assigned priority. For more information on the RTS and TCP methods, please refer to the paper [6].

The Google dataset used in this study consists of the data collected from the pre-submit and post-submit testing phases. Pre-submit testing refers to the initial testing conducted before a new or changed module is checked into the mainline codebase. The modules to be tested are the changed modules and the modules that rely directly on the changed modules. One main purpose of having pre-submit testing phase is to give a developer faster feedback on failure before moving to the post-submit phase. If this pre-submit test is successful, the changed modules are committed to the codebase, and testing for the modules is conducted more thoroughly. Post-submit testing is performed to test the modules and the modules associated with the changed modules, both directly and indirectly. The dataset has more than three million test records spanning 30 days. Each record includes the details of each test record such as the name of the test suite, testing result, and execution time. Elbaum et al. applied the RTS technique to the pre-sumbit testing phase, and the TCP technique to the post-submit testing phase [6]; we also do the same in this research.

III. Motivation and Approach

The existing window-based method does not consider test suites that failed once but do not fail again. If many such test suites are selected by RTS, and assigned with the same priority as the test suites that failed more than once by TCP, the cost-effectiveness of the method can decrease significantly. We observe that such test suites are prevalent in the Google dataset. Figure 2 shows the number of failures detected by each test suite. About 44% of the failed test suites (205) failed only once in the pre-submit phase, and about 33% of the failed test suites (154) failed only once in the post-submit phase.

This situation is analogous to that of the disk cache for web objects. Maggs et al. found that about three quarters of the cached web objects in their content delivery network (CDN) system are accessed only once [21]. The authors referred to such objects as "one-hit-wonders"; we borrow this term to indicate the test suites that fail only once. They used Bloom filters to prevent the one-hit-wonders from being stored in the cache. Bloom filters are fast and space-efficient data structures that support the addition of an element into a set and the evaluation of whether an element is in the set [22], [23].

Bloom filter is a probabilistic data structure with k hash functions and an m-bit array. The bit array is initialized with zero when it is first created. Each hash function takes any type of data as its input and returns the position of one element in the bit array as the output. Therefore, k hash functions produce k element positions. The Bloom filter sets the corresponding values of the k element positions in the bit array to one. Since k element positions are almost always different for different input data, it is possible to identify which data has been added to the Bloom filter. The advantage of using the Bloom filter is that we can reduce the false positives by using multiple hash functions. That is, it can reduce the probability of cases in which none of the input data is added to the Bloom filter, but is judged to be present. Another advantage is that there is no false negative. That is, there is no possibility of cases in which data is added to the Bloom filter, but is judged to be absent. Finally, the hashing algorithm has a time complexity of $O(1)$, and therefore, the Bloom filter can quickly tell whether data has been added to the Bloom filter. The probability of a false positive p, the number of data to be added n, and m can be adjusted based on the following formula.

$$p \simeq 2^{-\frac{m \ln 2}{n}}$$

The cost-effectiveness of each method can be increased by using Bloom filters for existing window-based RTS and TCP methods. First, in the case of RTS, the testing time can be reduced as much as the execution time of the removed test suites, which are the ones that are not included in the Bloom filter from among the selected test suites. There is possibility of excluding test suites that may detect failures. However, such test suites are executed in the post-submit phase, therefore developers are able to recognize the failures. By using the RTS method in combination with the Bloom filter during the

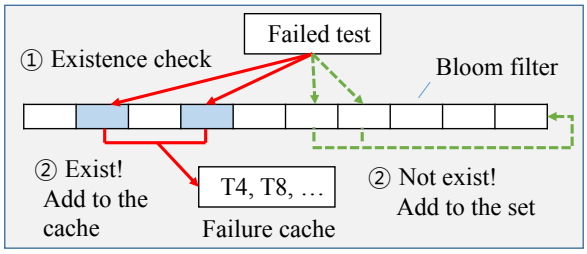

Fig. 3. Using a Bloom filter to filter out test suites that failed only once

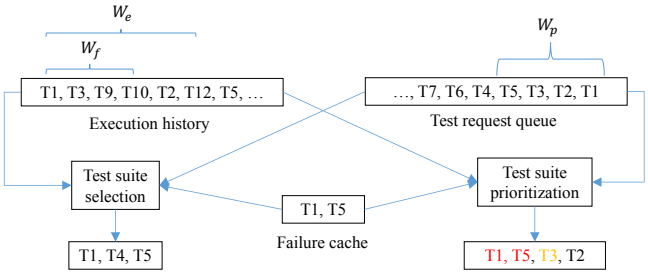

Fig. 4. Example of using the proposed RTS and TCP methods

pre-submit testing phase, developers run smaller number of test suites in the phase, and they can receive failure feedback faster than by using the RTS method alone.

For TCP, by giving the test suites in the Bloom filter a higher priority, the test suites that have failed more than once can be run before other test suites that have failed only once. Therefore, if there are many one-hit-wonders in the test suites, using the TCP method in combination with the Bloom filter during the post-submit testing step can give the developers faster feedback on the failures compared to the existing method.

We incorporated a Bloom filter to the window-based RTS and TCP methods to filter out one-hit-wonders. We also introduced a cache called the failure cache, to store the test suites that failed more than once. Figure 3 illustrates the mechanism by which the Bloom filter avoids storing one-hit-wonders in the failure cache. When a test suite fails, the Bloom filter is checked for its presence. If the test suite does not exist, it means that the test has failed for the first time. Then, the test suite is added to the Bloom filter; however, it is not cached. If the test suite already exists in the Bloom filter, it is added to the failure cache.

The RTS method that we extended with the Bloom filter selects a subset of the test suites that are chosen by the window-based RTS method. The subset includes new test suites and test suites in the failure cache. Figure 4 shows an example of the proposed approach. The left side of the figure shows the RTS part. Among the test suites that are in the test request queue, the test suites selected by the window-based RTS method are T_1, T_3, T_4, and T_5. In the execution history, the LFT of T_1 and T_3 is in W_f, T_4 is a new test suite, and the LET of T_5 is outside W_e. T_1, T_4, and T_5 are finally selected, because they are in the failure cache.

Our TCP method gives a higher priority to the test suites that are in the failure cache, among the test suites that satisfy the W_f or W_e criteria. In the existing method, the test suites satisfying the W_f or W_e criteria, or new test suites are given a priority of one, and the test suites that do not satisfy any of the criteria are given a priority of two. Our TCP method assigns zero as a priority to the test suites that are in the failure cache among test suites that satisfy the W_f or W_e criteria. The test suites in the queue are sorted in an ascending order according to the given priority. The test suites with the same priority will be tested according to the order in which they are entered into the queue. The right side of Figure 4 shows an example of using our TCP method. The test suites whose queued time is within W_p are scheduled. As T_1 and T_5 are in the failure cache, they are given a priority of zero. Since T_3 satisfies the W_f criteria, its priority is set to one, and T_2 is assigned a priority of two, because it does not satisfy any of the criteria.

IV. Evaluation

In this empirical study, we compared the proposed RTS and TCP methods against the state-of-the-art approach developed by Elbaum et al. [6], and demonstrated how the Bloom filter-based approach improves the cost effectiveness of regression testing in CI environments. To do this, we answered the following research questions:

RQ 1: How does excluding one-hit wonders by using the Bloom filter-based RTS technique improve the cost effectiveness of the baseline RTS technique?

RQ 2: How does giving more priority to test suites that have failed more than once by using the Bloom filter-based TCP technique improve the cost effectiveness of the baseline TCP technique?

We follow an experiment design similar to the study of Elbaum et al. [6] to compare our approach with their approach. For this experiment, we used the Google CI dataset [12], from which we could retrieve the following: a list of test suites that are used for CI development, the time when the test suites began running and finished, and the test results. Therefore, LFT and LET required for the proposed RTS and TCP technique could be calculated from the dataset information. The RTS and TCP techniques were applied to the data generated from the pre- and post-submit phases, respectively.

A. Variable and Measures

1) Independent Variables: Two independent variables were used in this experiment. One independent variable was an RTS or TCP technique and the other involved windows (W_e, W_f, and W_p). The techniques included the proposed and baseline approaches. Windows $W_e, W_f, and W_p$ were assigned various values. Specifically, in the case of RTS, we used the following window values: W_e and W_f values: $W_e \in \{1, 24, 48\}$, $W_f \in \{0.25, 0.5, 1, 2, 4, 12, 24, 48, and 96\}$. For TCP, we used the following window values: W_f was 12, W_e was 24, and W_p had one of a value from $\{0.1, 0.5, 1, 2, 4, 8, 12\}$. Because

we experimented with all possible window combinations, we conducted the experiments 54 times to evaluate the RTS techniques ($2 \times 3 \times 9$), and 14 (2×7) times to evaluate the TCP techniques.

2) Dependent Variables: To measure the cost effectiveness of the RTS method, we use the failure-detection efficiency defined in [24]:

$$Eff_{det} = \frac{\#\ faults\ found}{\#\ test\ suites\ executed}$$

Since the execution time of each test suite is available in the dataset, we define another measure to quantify the cost-effectiveness of the RTS method. We call the measure as the failure-detection efficiency regarding execution time, and define as follows:

$$Eff_{time} = \frac{\#\ faults\ found}{time\ for\ test\ suites\ executed}$$

The Eff_{det} value indicates the number of faults found on average by running a test suite, whereas the Eff_{time} value indicates the number of faults found on average during a unit time (e.g., hour).

To evaluate the cost effectiveness of the TCP methods, we measured the time to detect each failure by using the proposed and baseline approaches that are similar to the evaluation described in the study of Elbaum et al. [6]. In addition, we classified each failure-detection result into one of the following categories: 1) the proposed method detects the failure earlier than does the baseline approach (positive case), 2) the proposed method detects the failure later than does the baseline approach (negative case), and 3) the proposed method detects the failure simultaneous to that detected in the baseline approach (zero case). More positive than negative cases means that the proposed approach was faster than the baseline approach at finding failures. In addition, the mean, maximum, and minimum values of the difference between the failure-detection times of the proposed and baseline techniques for each failure were measured. These measures indicate how quickly developers can receive feedback regarding failures.

B. Study Operation

To evaluate the RTS method, we perform the following steps:

1) Set W_e and W_f (27 for each technique)
2) Read a line (a test suite) from the pre-submit data
3) Decide whether to choose the test suite for each technique
4) If selected, increment a counter variable by 1 and add the test suite's execution time to a cumulated time variable.
5) Repeat the Steps 2-4
6) Measure Eff_{det} and Eff_{time}

To evaluate the TCP method, we performed the following steps:

1) Set W_p (7 per technique)
2) Read a line (a test suite) from the post-submit data
3) Assign a priority to the test suite according to each technique
4) For each failure in prioritization, when each technique is applied, record the cumulated time at which the failure is found
5) For each failure, compare the failure detection time between the proposed and baseline approaches, and classify the comparison result into the three categories (positive, negative, and zero)

C. Threats to Validity

External: One-hit wonders may be a phenomenon found specifically in Google CI data or in a specific time period of CI development. However, the Google dataset covers many languages (C, Java, Python, etc.), many modules (5555 modules at the pre-submit phase; 5536 modules at the post-submit phase), and frequent updates. We investigated and determined that other software systems such as Netflix and Samsung Tizen also have these features. Therefore, we believe that the one-hit-wonder phenomenon can also be found in other datasets. We plan to apply the proposed RTS and TCP methods to other systems to prove the general effectiveness of the approach. Another concern is that when a developer uses the Bloom filter-based approach, they must set the appropriate array size. This is because the probability of generating false positives increases as the number of test suites in the test system increases. To solve this problem, we can apply a method of dynamically expanding the size of the array [25] or a technique in which information stored in the Bloom filter is deleted after a certain period [26].

Internal: Because no published source code was available for the baseline RTS and TCP approaches, we implemented the approaches. As a result, the performance trend based on changes to W_e, W_f, and W_p were similar to the results shown in [6]. However, detailed performance measures of the baseline methods were somewhat better than the results reported in this study. This may be because some parts of the methods were implemented differently than in the original implementation. However, we confirmed that the baseline RTS and TCP methods and the proposed approach worked as intended after inspecting some samples of the dataset. In addition, for the Bloom filter, we used an open source implementation from the Python software repository (PyPI)[2].

Construct: One of the cost effectiveness metrics that we used to evaluate the RTS methods was adopted from the metric defined in [24], and the other metric was defined in this work by considering the time efficiency of detecting failures. For TCP, we counted the number of positive, negative, and zero cases to evaluate the TCP methods against the state-of-the-art approach. The reason for employing these measures is to consider the benefits of using the Bloom filters for RTS and TCP in a specific manner. In this study, similar to [6], we regarded the "cost" as the number of test suites that must be performed, as well as the execution time of the test. We did

[2]https://pypi.python.org/pypi/pybloom_live/2.2.0

not address other aspects of cost such as the importance of failures.

D. Result and Analysis

1) RQ1: Regression Test Selection: Figure 5 shows Eff_{det} and Eff_{time} of the proposed RTS method for various sizes of W_f and W_e. The X-axis represents a pair of W_f and W_e separated by a comma. The blue bars indicate the proposed approach ('proposed'), and the green bars indicate the baseline approach ('baseline'). For all W_f and W_e configurations, the cost effectiveness of the proposed approach outperformed those of the baseline approach. The means of Eff_{det} values for the proposed and baseline approach are about 0.05 and 0.02, respectively (2.23 times higher). The means of Eff_{time} values for the proposed and baseline approach are about 6.28 and 3.92, respectively (1.6 times higher). The range of differences between the proposed and baseline approaches were from 0.01 to 0.05 for Eff_{det}, and from 0.8 to 4.6 for Eff_{time}.

One interesting characteristic is that the two graphs show similar trends. However, the ratios of Eff_{det} of the baseline to that of the proposed approach were smaller than those for Eff_{time}. It is possible that one-hit wonders tend to have shorter execution time on average than that of the test suites selected by the proposed approach.

Another interesting characteristic is that Eff_{det} and Eff_{time} of both approaches with $W_e = 1$ tended to have different patterns than approaches with $W_e = 24$ and $W_e = 48$. When W_e was 1, Eff_{det} and Eff_{time} were much lower than those when W_e was 24 or 48. When W_e was 1, many test suites were selected because the value of W_e was small (1). In other words, LET of a test suite was more likely to be greater than 1 than was 24 or 48. Therefore, many test suites were selected, but many of them passed only once or failed. Thus, the performance of the baseline was low. However, our approach filters out one-hit wonders and test suites that have never failed despite meeting the W_e criteria. Therefore, our approach showed much better performance than did the baseline approach when W_e was 1.

To show that the performance differences between the baseline and proposed approaches were statistically significant, we conducted a significance test. Because the distributions of differences in Eff_{det} and Eff_{time} are not known, we used the Wilcoxon signed-rank test [27] instead of a student t-test. We set the null hypothesis (NH) as follows:

NH1: Eff_{det} of the baseline and proposed approaches are the same

NH2: Eff_{time} of the baseline and proposed approaches are the same

Table I shows the results of the significance test. The p-values of both Eff_{det} and Eff_{time} were less than 0.001. Therefore, we rejected both NH1 and NH2. In addition, we calculated the effect size by using Cohen's d to determine whether the differences were effective [28]. The effect sizes for cost effectiveness and cost efficiency were 2.22 and 0.94, respectively, which indicates the effect sizes were huge, namely, between large and very large, respectively.

TABLE I
WILCOXON SIGNED-RANK TEST FOR Eff_{det} AND Eff_{time} OF THE PROPOSED AND BASELINE APPROACHES

	p value	effect size
Eff_{det}	<0.001	2.22 (Huge)
Eff_{time}	<0.001	0.94 (B/W Large and very Large)

TABLE II
PRECISION ANALYSIS OF THE BASELINE AND PROPOSED RTS TECHNIQUES

W_f, W_e	method	Selected & failed (TP)	Selected & passed (FP)	prec.	TP diff	FP diff
12.00,1	baseline	1823	381231	0.00	-179	-349711
	proposed	1644	31520	0.05		
12.00,24	baseline	1517	38561	0.04	-56	-16133
	proposed	1461	22428	0.06		
12.00,48	baseline	1515	33826	0.04	-56	-11402
	proposed	1459	22424	0.06		

To determine why the proposed approach was superior to the baseline approach, we further analyzed the precision of each RTS approach. For the test suite selected by each approach, we classified the test suite into truth positive (TP), which was selected for an RTS technique and actually failed, or false positive (FP), which was selected by an RTS technique but passed in the pre-submit testing. Among 27 pairs of W_f and W_e, we chose three pairs, as described in Table II. These were chosen because the TF and FP for the other pairs were similar to those for the three pairs.

The number of detected failures of the proposed approach was smaller than those of the baseline approach, as shown in the table. The differences between TP values of the two approaches ranged from -179 to -56 ("TP diff" column in the table). However, the proposed approach reduced FP to a greater extent than did the baseline approach, ranging from -349711 to -11402 ("FP diff" column in the table). By reducing the many FP, the proposed approach enables a developer to receive faster feedback regarding failures. In addition, the failures that were detected when using the baseline approach but missed when using the proposed approach can be detected during the post-submit testing period because all the queued test suites including the missed failure-inducing test suites are run in the post-submit period.

We measured the precision of each technique for each pair of W_f and W_e. Precision was calculated as follows:

$$Precision = \frac{TP}{TP + FP}$$

The precision values of the proposed approach ranged from 0.04 to 0.08. However, for the baseline approach, the precision values ranged from 0.004 to 0.05. Because many FP were generated by the baseline approach, the precision of the proposed approach was better than that of the baseline approach.

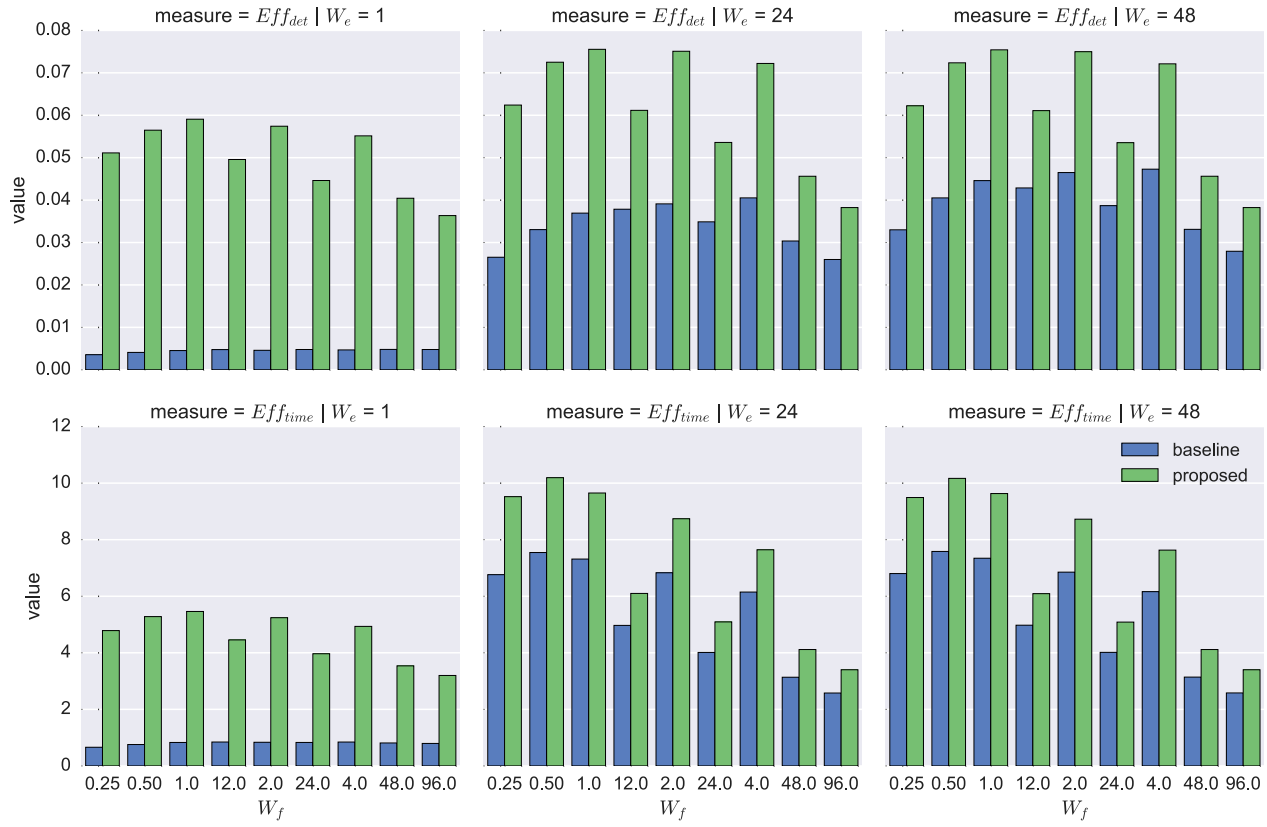

Fig. 5. Cost effectiveness of the RTS techniques for various sizes of the window

We also compared the baseline and proposed RTS approaches against the random selection approach. The comparison result is not shown in Figure 5 because of the space limitation. The random selection approach randomly selects test suites from the RTS dataset as much as the number of test suites that are selected by the baseline approach. Then, Eff_{det} and Eff_{time} values are calculated based on the random selection. The random selection is repeated five times, and the Eff_{det} and Eff_{time} values from the iterations are averaged, respectively. The mean value of Eff_{det} of random selection over all the window configurations was 0.001, which is 94% less than the baseline approach, and 97% less than the proposed approach. The mean value of Eff_{time} of random selection over all the window configurations was 0.29, which is 93% less than the baseline approach, and 95% less than the proposed approach.

2) RQ2: Test Case Prioritization: Figure 6 and Table III show improvements when using the proposed approach as compared to the baseline approach as the value of W_p changed from 0.1 to 12 h. The cumulative testing time for each of the 6878 failures was calculated.

As the size of W_p increased, fewer prioritizations were performed ("counts" column in the table) because more test suites were scheduled when the window size was larger.

Our approach proved to be more cost-effective than the baseline approach because more positive cases existed ranging from 2290 to 5758 ("pos" column in the table) than negative cases ranging from 59 to 473 ("neg" column). According to the difference mean values (as shown in the "mean (h)" column), all values were positive, indicating that the proposed approach was more cost-effective than the baseline approach on average. In other words, a developer can receive faster feedback ranging from 0.03 to 3.04 h on average. In the best case, the developer can receive failure feedback approximately 42 hours earlier than with the baseline approach. In the worst case, the developer receives failure feedback approximately eight hours later than with the baseline approach.

E. Discussion: Additional Cost Analysis When Using Bloom Filter

When the Bloom filter is used together with the existing window-based method, the test case selection in the RTS process may be delayed, and the time taken to prioritize test cases in the TCP process may become longer due to the overhead of using the Bloom filter. We measure the running time of the Bloom-filter-based RTS and TCP processes for the Google dataset, and compare the time against the time

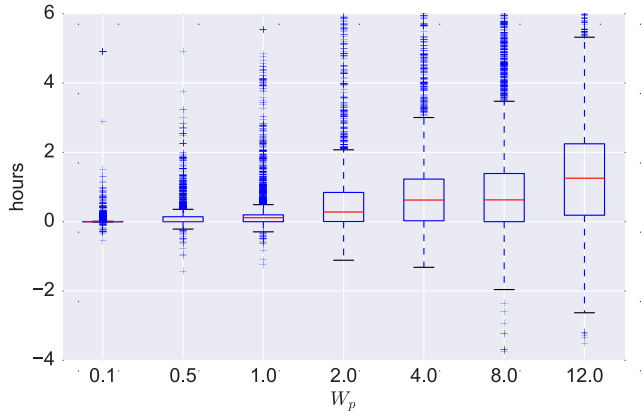

Fig. 6. The feedback time differences between the baseline and proposed TCP techniques for each failure and W_p. For all W_p, the positive differences are more than the negative differences, which mean cost-effectiveness of the proposed TCP technique is higher than the baseline approach

TABLE III
RESULT OF TCP TECHNIQUES

W_p	counts	pos	zero	neg	mean (h)	max (h)	min (h)
0.1	1191	2290	4497	91	0.03	4.91	-0.55
0.5	423	3855	2911	112	0.20	26.75	-1.42
1.0	252	4859	1609	410	0.34	26.75	-1.24
2.0	145	5412	1407	59	0.71	35.50	-1.11
4.0	81	5546	1241	91	1.26	35.50	-1.32
8.0	45	5241	1164	473	1.98	39.06	-8.07
12.0	32	5758	921	199	3.04	42.27	-8.20

taken to run the baseline approaches. The runtime analysis is repeated over all the values of W_f, W_e and W_p. The experiment environment is as follows: the CPU is Intel(R) Core(TM) i7-5930K CPU @ 3.50GHz and the memory is 32004 MB.

Table IV shows the result of the runtime analysis of the baseline and proposed RTS and TCP approaches. The columns represent the proposed approach, the baseline approach, and difference of running time between the proposed and baseline approaches for the RTS and TCP processes. The rows in the table show the statistics of the running time generated by experimenting with all window configurations.

In RTS, the proposed approach takes 214 seconds more on average for all 27 configurations than the baseline approach to process all the test suites in the pre-submit dataset (1431765). This means that using the Bloom filter requires approximately additional 0.15 milliseconds to select a test suite. The difference in execution time of the selected test suites between the proposed and baseline approaches is about 650 hour on average. Therefore, 214 seconds is relatively insignificant time overhead, and we can say that the proposed RTS method is more cost-effective than the baseline method.

Comparing to the baseline approach, it took 13 more seconds on average to schedule 2,037,183 test suites by using the proposed TCP technique with the seven window

TABLE IV
RUNTIME ANALYSIS FOR THE BASELINE AND PROPOSED RTS AND TCP METHODS (TIME UNIT: S)

	RTS			TCP		
	proposed	baseline	diff	proposed	baseline	diff
count	27.0	27.0	27.0	7.0	7.0	7.0
mean	7952.9	7738.7	214.2	5895.1	5882.1	13.0
std	71.4	126.5	83.9	17.1	30.4	29.9
min	7767.0	7483.0	73.0	5870.0	5834.0	-34.0
25%	7926.0	7647.0	156.0	5882.5	5863.5	-5.5
50%	7972.0	7782.0	190.0	5902.0	5886.0	20.0
75%	7999.0	7831.0	255.0	5905.5	5904.5	34.0
max	8048.0	7923.0	423.0	5918.0	5919.0	48.0

configurations. In other words, on average, additional 6.38 μs per test suite is taken when using the proposed TCP approach. With considering the average 1.08 hours of faster feedback on failures that can be resulted in by using the proposed approach for each configuration, 13 seconds is a ignorable time overhead. Therefore, we can say that the proposed TCP method is more cost-effective than the baseline approach.

V. CONCLUSION

In this paper, we proposed a method to improve the cost effectiveness of Regression Test Selection (RTS) and Test Case Prioritization (TCP) for regression testing in CI environments, especially considering the fact that, in our study, many test suites proved to have failed only once over a certain period. A Bloom filter was used to identify such test suites fast while consuming computing resources in an efficient manner even though the number of test suites to manage becomes large. The RTS process can be more efficient by excluding the test suites that are filtered out by the Bloom filter. During the TCP process, such test suites will have a lower priority in the failure cache. The experiment conducted for TRS demonstrates that the proposed Bloom-filter-based technique improves the failure detection effectiveness 2.23 times higher on average comparing to the state-of-the-art approach. We also showed that the proposed Bloom-filter-based TCP technique is more cost effective than the state-of-the-art approach by providing feedback on failures to the developers faster by 1.08 hours on average.

We plan to extend our current approach as follows. Firstly, to reduce the false-positive cases, we will add a timeout property to the test suites that are managed by the Bloom filter, so that the test suites that have not been updated for a while can be removed from the Bloom filter. Next, we will use machine learning techniques by considering other features in the CI dataset such as a module path and a language, and devise a method to update the machine learning model incrementally as test results, which can be the input of the model, are continuously recorded in the test history.

ACKNOWLEDGMENT

This work was supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT and Future Planning (2016R1A2B4007585)

REFERENCES

[1] P. M. Duvall, *Continuous Integration*. Pearson Education India, 2007.

[2] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, "Taming google-scale continuous testing," in *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*. IEEE Press, 2017, pp. 233–242.

[3] "The software revolution behind linkedin's gushing profits," Apr 2013. [Online]. Available: https://www.wired.com/2013/04/linkedin-software-revolution/

[4] "Deploying the netflix api – netflix techblog – medium," Aug 2013. [Online]. Available: https://medium.com/netflix-techblog/deploying-the-netflix-api-79b6176cc3f0

[5] J. Anderson, S. Salem, and H. Do, "Improving the effectiveness of test suite through mining historical data," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 142–151.

[6] 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*. ACM, 2014, pp. 235–245.

[7] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[8] 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*, vol. 1. IEEE, 2015, pp. 268–279.

[9] 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*. IEEE, 2002, pp. 119–129.

[10] 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*, vol. 1. IEEE, 2009, pp. 99–106.

[11] D. Marijan, A. Gotlieb, and S. Sen, "Test case prioritization for continuous regression testing: An industrial case study," in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. IEEE, 2013, pp. 540–543.

[12] S. Elbaum, A. Mclaughlin, and J. Penix, "The google dataset of testing results," 2014. [Online]. Available: \url{https://code.google.com/p/google-shared-dataset-of-test-suite-results}

[13] G. Rothermel and M. J. Harrold, "Analyzing regression test selection techniques," *IEEE Transactions on software engineering*, vol. 22, no. 8, pp. 529–551, 1996.

[14] ——, "Empirical studies of a safe regression test selection technique," *IEEE Transactions on Software Engineering*, vol. 24, no. 6, pp. 401–419, 1998.

[15] R. Gupta, M. J. Harrold, and M. L. Soffa, "An approach to regression testing using slicing," in *Software Maintenance, 1992. Proceerdings., Conference on*. IEEE, 1992, pp. 299–308.

[16] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE transactions on software engineering*, vol. 28, no. 2, pp. 159–182, 2002.

[17] M. Harman, "Making the case for morto: Multi objective regression test optimization," in *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*. IEEE, 2011, pp. 111–114.

[18] D. Hao, L. Zhang, L. Zhang, G. Rothermel, and H. Mei, "A unified test case prioritization approach," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 2, p. 10, 2014.

[19] L. Mei, W. K. Chan, T. Tse, B. Jiang, and K. Zhai, "Preemptive regression testingof workflow-based web services," *IEEE Transactions on Services Computing*, vol. 8, no. 5, pp. 740–754, 2015.

[20] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 435–445.

[21] B. M. Maggs and R. K. Sitaraman, "Algorithmic nuggets in content delivery," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 3, pp. 52–66, 2015.

[22] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[23] Wikipedia, "Bloom filter — wikipedia, the free encyclopedia," 2016, [Online; accessed 6-December-2016]. [Online]. Available: \url{https://en.wikipedia.org/w/index.php?title=Bloom_filter&oldid=753303724}

[24] E. Engström, P. Runeson, and G. Wikstrand, "An empirical evaluation of regression testing based on fix-cache recommendations," in *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*. IEEE, 2010, pp. 75–78.

[25] P. S. Almeida, C. Baquero, N. Preguiça, and D. Hutchison, "Scalable bloom filters," *Information Processing Letters*, vol. 101, no. 6, pp. 255–261, 2007.

[26] I. Grigorik, "Flow analysis and time-based bloom filters," Jan 2010. [Online]. Available: https://www.igvita.com/2010/01/06/flow-analysis-time-based-bloom-filters/

[27] Wikipedia, "Wilcoxon signed-rank test — wikipedia, the free encyclopedia," 2017, [Online; accessed 7-July-2017]. [Online]. Available: \url{https://en.wikipedia.org/w/index.php?title=Wilcoxon_signed-rank_test&oldid=782283572}

[28] ——, "Effect size — wikipedia, the free encyclopedia," 2017, [Online; accessed 7-July-2017]. [Online]. Available: \url{https://en.wikipedia.org/w/index.php?title=Effect_size&oldid=783356653}