Prioritizing Regression Tests for Desktop and Web-Applications based on the Execution Frequency of Modified Code

Matthias Hirzel Jonathan Immanuel Brachthäuser Herbert Klaeren
Wilhelm-Schickard-Institut
University of Tübingen, Germany
{hirzel,brachthaeuser,klaeren}@informatik.uni-tuebingen.de

ABSTRACT

Regression testing can be very time expensive when running all available test cases. Test prioritization seeks to find faults early by reordering tests. Existing techniques decide in which order tests should be run based on coverage data, knowledge of code changes, historical data of prior test execution or a combination of them. Others postpone tests if similar ones are already selected for early execution. However, these approaches do not take into account that tests which appear similar still might explore different parts of the application's state space and thus can result in different test outcome. Approaches based on structural coverage or on historical data might ignore small tests focusing on behavior that rarely changes. In this paper, we present a novel prioritization technique that is based on the frequencies with which modified code parts are executed by the tests. Our technique assumes that multiple executions of a modified code part (under different contexts) have a higher chance to reveal faults than a single execution of this code. For this purpose, we use both the output of regression test selection as well as test traces obtained during test development. We propose multiple variants of our technique, including a feedback mechanism to optimize the prioritization order dynamically, and compare them in an evaluation of Java-based applications to existing approaches using the standard APFD metric. The results show that our technique is highly competitive.

CCS Concepts

•General and reference → Metrics; Evaluation; •Software and its engineering → Software testing and debugging; *Automated static analysis; Dynamic analysis;*

Keywords

Test case prioritization; Regression testing; Fault detection effectiveness; Empirical study; Execution frequency

PPPJ '16, August 29-September 02, 2016, Lugano, Switzerland

ISBN 978-1-4503-4135-6/16/08...$15.00

DOI: http://dx.doi.org/10.1145/2972206.2972222

1. INTRODUCTION

When adding new features to a software system an important part of the development process is to test the new features. At the same time, it is equally important to repeatedly test that the behavior of existing parts of the system is left unchanged (regression testing). This strategy ensures that the extended software behaves as expected, and at the same time prevents that new functionality compromises existing parts of the program. Executing all tests regularly (retest-all approach) and observing no faults gives confidence in the system as a whole and in all features which are covered by the tests.

In large applications with big tests suites, regression testing can take several hours until results are available (e.g. [6, 28, 17]). Some authors even report on several days [32] or even weeks [30]. Thus, this kind of testing can be very time expensive which in turn hinders continuous integration [15] since test results only arrive with a massive delay. For this reason, developers proposed approaches to reduce the testing effort and thereby shorten the response time. They can be divided into the three main categories [35]: *Test Case Minimization*, *Test Case Selection* and *Test Case Prioritization*. A survey of existing approaches is given by e.g. Yoo et al. [35], Engström et al. [12] or Rothermel et al. [27]. In this paper, we will focus on the two latter approaches.

Test selection aims to execute only these tests of the original test suite that are affected by code changes and therefore are able to detect faults (fault-revealing tests [28]). Ideally, this part is significantly smaller than the original test suite. To achieve this goal, regression test selection (RTS) techniques try to detect changes made during the development of a program version in order to determine exactly these tests which are affected by the modifications. Some RTS techniques are safe [27], i.e., they select all the tests affected by a code change. Consequently, they achieve the same result as a retest-all approach would do.

In contrast, test case prioritization (TCP) runs all the tests but changes the order in which individual tests are executed. Possible prioritization criteria may be the test performance in previous test runs with respect to the fault detection effectiveness [21] or the rate of executed fault-prone functions [10] in the past. Besides, structural code coverage has often been used as criterion [8, 10, 11, 30, 29, 23]. Here, tests are ordered according to the number of covered functions, blocks or statements (code elements). Tests covering the most code elements have the highest priority (Greedy algorithm, e.g. [23]). In order to have an advantage over a retest-all

approach, TCP-techniques will usually stop as soon as a predefined criterion (e.g. time limit) is fulfilled. However, this leads to the risk that the test run is canceled before fault-revealing tests are executed.

Test prioritization and test selection are well-suited to each other, even though test prioritization does not aim at reducing the number of executed tests. In some situations, it may happen that RTS techniques select many or even all tests for re-execution. Possible reasons include a large number of changes in the code base or a single code change that has a big impact on the application. It has been shown that the latter often occurs in the context of end-to-end testing with web tests [17]. In these situations test suite reduction is much harder and in consequence, different techniques combining RTS and TCP have been proposed in the literature (e.g. [24, 26, 32]). Similar to RTS techniques, these approaches first try to detect changed code elements. Afterwards, tests are ordered according to a prioritization strategy, e.g. the number of changes covered by a test. The combination idea has also been applied to web services (e.g. [3, 5]) as well as to web applications (e.g. [31]).

In order to improve prioritization results, some strategies assume that similar tests are negligible. There are many attributes suitable to consider tests as similar. For some of the approaches mentioned before, the authors developed a mechanism to incorporate feedback from already executed test cases. For example, they iteratively reduce or – more generally – adapt the priority of tests that cover the same code elements as other tests which are already selected for execution. Rothermel et al. call this the "additional"-approach [30]. In the past, it has also been used in related approaches (e.g. [25, 37]). Authors of these approaches also refer to it as *feedback* mechanism (e.g. [8, 25]). Other strategies investigate the sequences of covered code elements and their (dis)similarity (e.g. [14, 22]).

The prioritization techniques described above exhibit different strengths and weaknesses. A high structural coverage (of blocks/methods) does not necessarily correlate with the ability of a test to detect a fault. Furthermore, approaches prioritizing tests according to the number of traversed code modifications have the risk that a unique prioritization order is impossible because all tests are affected by the same fundamental code changes. This may especially be true in the context of web applications. Finally, approaches based on the similarity of tests assume that tests reveal the same faults if a given similarity criterion matches. These approaches just pick a representative of a class of similar tests. However, tests that appear similar still might explore different parts of the application's state space and thus can result in different test outcome.

Existing approaches insufficiently consider recent code changes and possibly fault-revealing states. However, the application's state space is crucial for the ability to detect faults. Therefore, it is decisive to put the software in a fault-revealing state. We hypothesize that similar tests have been developed intentionally as they check different states of the application. We also hypothesize that code modifications often lead to a change of the application state. Based on these assumptions, we conclude that one should prioritize tests which run modified code multiple times under different context as these tests have the highest chance to detect a fault.

In this paper, we present several variants of a novel test prioritization approach. For all the code entities (functions, statements or even expressions) that are covered by a test, our technique determines the frequency of their execution. To this end, we statically compare an old program version with its successor, extract code changes and select all tests that are affected by these changes. In a second step, we prioritize those tests that execute the (most) code changes as often as possible (execution count of modified code). To the best of our knowledge, we are the first to base the execution order of tests on this decision criterion. We propose and evaluate three possible ways to prioritize tests based on the execution count and on the number of changes covered by a test. Additionally, we investigate and evaluate a dynamic feedback mechanism to adjust the order of tests at runtime. In summary, we make the following contributions:

- A discussion of weaknesses of existing approaches;
- Several static and dynamic variants of a new test case prioritization technique, considering the number of code changes and the execution frequency of these changes in a test;
- An exact formalization of the variants of our prioritization technique;
- An evaluation of our techniques on 3 Java desktop applications and two web applications based on Google Web Toolkit. To judge the efficacy, we use the APFD (Average of the Percentage of Faults Detected) [10, 30] measure;
- A comparison of our results with other approaches in the literature;
- An investigation of the effort and the efficiency when prioritizing test cases.

The evaluation shows that our technique produces highly performant test prioritization orders with a low deviation in the APFD values.

2. MOTIVATION

Test prioritization seeks to find faults as soon as possible [35]. To achieve this, the goal is to find an optimal ordering of tests revealing these faults. The example in Figure 1 shows tests and the faults they are able to reveal. Tests $t_A - t_E$ have been taken from Elbaum et al. [10]. To explain our motivation, we have added two more tests t_F and t_G. Elbaum et al. argue that an ordering starting with the execution of t_C followed by t_E (order $t_C t_E$) is superior because it detects all the faults the most rapidly.

		Fault revealed by test case									
		1	2	3	4	5	6	7	8	9	10
Test	A	×				×					
	B	×				×	×	×			
	C	×	×	×	×	×	×	×			
	D					×					
	E								×	×	×
	F		×	×	×						
	G					×					

Figure 1: Fault revealing tests, adapted from Elbaum et al. [10]

```
public void onClick(ClickEvent event) {
  searchValue = display.
    getSearchValue() .trim();
  eventBus.fireEvent(
    new RefreshMessagesEvent(searchValue));
}
```

Figure 2: Validator for user inputs with marked additions in Version P' compared to P.

As already described by Yoo et al. [35], the problem in practice is that we do not know in advance which test will detect a fault. Even when using a test selection technique, we only have knowledge about code modifications. We do not know yet which modification leads to a test failure. Thus, which decider recognizes that $t_C t_E$ is the best choice?

Furthermore, the order $t_C t_E$ might not be optimal when there are several possible solutions. Taking test t_F into account, order $t_B t_F t_E$ covers the same faults. When searching for a decider which of the two possible solutions should be applied, some approaches use the number of faults previously detected and/or the number of statements covered by the tests as pointed out in Section 1. However in practice, examples can be found for which this strategy is not optimal. As a simple example, Figure 2 shows a code snippet of a program in two different versions P and P', taken from a mail client "Hupa" [18] (see also Section 4.1). The code snippet validates user inputs in a mail search dialog. In the subsequent version P', valid user inputs have been restricted by adding a call to `trim()`, which might cause tests to fail.

Imagine that test t_G validates several possible search items that have been inserted into the search box. Another test t_D takes a search item in order to check whether the mail body will be displayed correctly when double clicking on a search result. Both test cases t_D and t_G cover the same code change. Test t_D covers more statements as it also checks the double click in the search results. Nevertheless, it is conceivable that t_G has a higher chance to fail and to reveal this fault because it checks more possible inputs. So it is more reasonable to prioritize test case t_G rather than t_D.

A similar problem occurs when a fundamental code modification affects (almost) all test cases. This might especially happen when regression testing web tests as we have pointed out in previous work [17]. In this situation, (almost) all tests are affected by the same single code change. So it does not suffice to consider the number of covered faults per test case. The number of executed statements might be misleading as described previously. Besides, we found that using historical data did not correlate with current code changes in one application. For this reason, we propose an approach considering the number of traversed code entities per test case.

3. APPROACH

Our approach builds on the assumption that the chance to detect a fault is greater when a code modification is executed in many different application states. At the same time, we assume that the more often a code modification is executed, the higher the chance that it is executed in different application states which might reveal faults. Hence, we propose to take the *execution frequency* of code modifications into account when prioritizing tests cases.

In general, there are many reasons why a code entity, such as a statement, is being executed multiple times. For instance, it might (a) be part of a loop which itself is executed many times or (b) be enclosed in a function that is called multiple times by either the application itself or by tests.

While a test executes a code entity several times, it might change parts of the application state or use different parameters. This can reveal faults that only show up in certain application states. In the same sense, we also assume that tests are small and that they check a concrete functionality or use case rather than large parts of the program. If a different test partially executes the same code modification as other tests, we assume that it does this intentionally and therefore also has potential to reveal a fault. Thus, in our approach, we favor tests that execute more code modifications than others.

In the remainder of this section, we explain our approach in detail. We present various variants that incorporate the number of code modifications and their execution frequency in different ways. For each of the variants, we propose an additional strategy, incorporating a dynamic feedback mechanism, that uses knowledge about test failures during test execution. This allows us to dynamically adjust the test prioritization and reevaluate previous decisions.

3.1 Considering Execution Frequency of Modified Code

To perform prioritization, our technique always considers two versions of a program P and P'. Here, P' is the program under test including a set of modifications, compared to P. For our purposes, we define a *code identifier* (CID) to be a unique identifier assigned to every source code entity. In Java, this can be for instance methods, statements or even expressions leading to a different granularity of the analysis.

Further, we define a *test trace* to be a multi set of CIDs that are traversed during the execution of the test. The order of the entities is not important for our approach, however, we count for each entity in the trace how often it has been traversed by a test.

The approach is divided into two steps. The first step performs regression test selection and thereby tries to reduce the set of tests we have to consider for prioritization. At the same time, it provides the necessary information required for step two: For every test, it provides its corresponding test trace. And it expounds which CIDs in the trace correspond to source code modifications. The second step then performs the actual prioritization.

Step 1 – Regression Test Selection

This first step is independent of the choice of an RTS technique. Many different techniques have been studied in the past. Graph-walk-based approaches, slicing-, and firewall-based approaches are only a few examples. More details can be found for instance in the survey of Yoo et al. [35]. To collect the information necessary for the second step and at the same time perform RTS, we apply a graph-walk based RTS technique, described in previous work [16, 17]. At every code entity, the application source code is instrumented to report the corresponding CID to a logging server that inserts the CIDs into a database. This enables us to create traces for a test, identifying all the code entities which are executed by the test. The used RTS technique is completely generic and for instance, can be applied to desktop and web applications.

Test	Execution Frequency						
	c_1^Δ	c_2^Δ	c_3^Δ	c_4^Δ	c_5^Δ	c_6	c_7
t_1	4	0	4	2	0	0	1
t_2	0	4	3	0	1	7	0
t_3	2	1	4	0	2	2	1
t_4	2	5	0	1	0	0	0

(a) Matrix f of execution frequencies per test and CID.

Test	Metrics			
	$\texttt{sum}^\Delta$	$\texttt{count}^\Delta$	$\texttt{max}^\Delta$	count
t_1	10	3	4	4
t_2	8	3	0	4
t_3	9	4	4	6
t_4	8	3	5	3

(b) Result of the test metrics.

Figure 3: Example matrix f and resulting metrics for tests covering the same CIDs with $n = 4, m = 5, p = 7$.

$$\texttt{sum}_i^\Delta = \sum_{1\le j\le m} f_{ij} \qquad \texttt{count}_i^\Delta = \sum_{1\le j\le m} \min(f_{ij}, 1)$$

$$\texttt{count}_i = \sum_{1\le j\le p} \min(f_{ij}, 1) \qquad \texttt{id}_i = i$$

$$\texttt{max}_i^\Delta = \max_{1\le j\le m} \left(f_{ij} * \left\lfloor \frac{f_{ij}}{\max(\max_{1\le k\le n}(f_{kj}), 1)} \right\rfloor \right)$$

Figure 4: Metrics assigning each test t_i an integer value as an estimate for its importance.

Figure 3a shows an output of the first phase as a matrix f which we will use as a running example. Every entry f_{ij} in the matrix represents how often code-entity c_j has been traversed while executing test t_i. We will refer to an entry f_{ij} as *execution frequency*. CIDs representing code changes are marked with Δ. Thus, c_6 and c_7 denote CIDs that do not represent a code change.

Step 2 – Prioritization

The second step of our approach computes a prioritization of the tests in form of a strict total ordering of the tests, using the matrix of execution frequencies as its sole input. After establishing necessary preliminaries, we propose multiple prioritization techniques and give their formal definitions.

Let T be a test suite or – more formally – a set of test cases, let n be the number of test cases in T, p be the total number of CIDs and m to be the number of CIDs that correspond to changes. Without loss of generality, we assume CIDs c_j with $1 \le j \le m$ to represent code changes and CIDs with $j > m$ and $j \le p$ to represent unchanged code entities. For each test t_i and each CID c_j, we record the frequency of t_i executing c_j in the matrix f and reference the execution frequency f_{ij} by indexing.

Each of our prioritization techniques is based on a series of *test metrics*. A test metric M assigns each test t_i a corresponding integer value as an estimate of the chance of t_i to reveal a fault. Figure 4 gives five such metrics ($\texttt{sum}_i^\Delta$, $\texttt{count}_i$, $\texttt{count}_i^\Delta$, $\texttt{max}_i^\Delta$, and $\texttt{id}_i$) that will be explained in more detail when used to define our prioritization techniques.

Definition 1 (Lifting of Metrics)**.** Given a metric M we define a corresponding order $t_i \lesssim_M t_l$ on tests t_i and t_l by

$$t_i \lesssim_M t_l \quad \text{iff} \quad M_i \le M_l$$

■

That is, to compare two tests we pointwise compare the corresponding integer values yielded by the metric. The resulting relation $\lesssim_M$ is a total preorder[1] and induces an equivalence relation for tests according to metric M:

$$t_i \sim_M t_l \quad \text{iff} \quad t_i \lesssim_M t_l \ \wedge \ t_i \gtrsim_M t_l$$

We also refer to the total preorder on tests $\lesssim_M$ induced by a metric M as *prioritization criterion* (PC) and use the name of the metric also for the prioritization criterion (that is, the preorder) when it is clear from the context. For instance, we use $\texttt{sum}^\Delta$ to refer to the preorder $(T, \lesssim_{\texttt{sum}^\Delta})$ and to refer to the metric for a test t_i as $\texttt{sum}_i^\Delta$.

In general, a metric M might assign the same integer values to two or more distinct tests resulting in no strict order for the tests in the corresponding equivalence class $\sim_M$. To allow a more fine grained ordering of tests we introduce the hierarchical composition for two total preorders PC_1 and PC_2.

Definition 2 (Hierarchical Composition)**.** For every two tests t_i and t_l the hierarchically composed order $PC_{1\triangleright 2} = PC_1 \triangleright PC_2$ is defined by:

$$t_i \lesssim_{PC_{1\triangleright 2}} t_l \quad \text{iff} \quad (t_i \lesssim_{PC_1} t_l) \vee (t_i \sim_{PC_1} t_l) \wedge (t_i \lesssim_{PC_2} t_l)$$

■

That is, two tests can either strictly be ordered by PC_1 or if they are equivalent with regard to PC_1 they are (not strictly) ordered by PC_2. The hierarchical composition of two total preorders is again a total preorder and hence hierarchical composition can be applied recursively. In addition, the hierarchical composition is associative.

Finally, given the matrix of execution frequencies, a *prioritization technique* uses a series of PCs to create a suggested ordering of execution of the tests in the test suite T. A strict total order is achieved by terminating the sequence of hierarchical compositions with the metrics $\texttt{id}$. Thus the scheme for defining some X-frequency-based prioritization technique using k prioritization criteria is:

$$X\text{FP} = PC_1 \triangleright PC_2 \triangleright \cdots \triangleright PC_k \triangleright \texttt{id}$$

3.2 Global Frequency-based Prioritization Technique (GFP)

The first prioritization technique of *global frequency-based prioritization* (GFP) builds on the assumption that tests

[1]Since a metric might assign the same value to two distinct tests, $\lesssim_M$ is not antisymmetric.

which execute the most code changes are the most likely to reveal a fault. To this end, GFP applies $\texttt{sum}^{\Delta}$ which simply accumulates the execution frequencies of all code changes covered by a test t_i (PC_1). The definition of the corresponding metric can be found along the other metrics in Figure 4. Tests with a higher accumulation value (global frequency) have a higher priority than tests with a lower global frequency.

To find an ordering of tests with the same global frequency $\texttt{count}^{\Delta}$ is applied. Here, the frequency is ignored and it is only counted how many code changes have been executed by a test t_i, hence modeling a coverage of changes (PC_2). A test covering more changes has higher priority. Please note again that PC_2 only affects tests whose order is not ambiguous, yet. The same applies to all subsequent prioritization criteria.

If there are still ambiguities, `count` is applied to also include CIDs that do not correspond to code changes, hence modeling a general, classic code coverage metric (PC_3). Tests achieving a higher code coverage are executed earlier.

Finally, as for all of our prioritization techniques, if no ordering can be decided after applying the first three prioritization criteria, the last criterion `id` arranges the tests in the order they appear in the matrix of frequencies (PC_4). The prioritization criteria that GFP uses are defined in Table 1.

Example. In the example of Figure 3a test t_1 covers three changes c_1^{Δ}, c_3^{Δ} and c_4^{Δ}. Summing up the corresponding execution frequencies gives $\texttt{sum}_1^{\Delta} = 10$. All other tests in this example yield lower values as can be seen in Figure 3b and thus t_1 has the highest priority. Applying $\texttt{sum}^{\Delta}$ (PC_1) gives the following equivalence classes from high to low priority: $\{t_1\}, \{t_3\}, \{t_2, t_4\}$. We can notice that t_2 and t_4 cannot be distinguished according to PC_1, so following our definition of hierarchical composition, $\texttt{count}^{\Delta} \rhd \texttt{count} \rhd \texttt{id}$ will be applied to find an ordering for this equivalence class. The criterion $\texttt{count}^{\Delta}$ (PC_2) gives 3 for both t_2 and t_4, so again $\texttt{count} \rhd \texttt{id}$ needs to be applied. Finally, PC_3 gives $\texttt{count}_2 = 4$ and $\texttt{count}_4 = 3$ leading to the strict ordering $\{t_1\}, \{t_2\}, \{t_4\}, \{t_3\}$.

If in the example t_2 and t_4 would have been equivalent according to PC_3 then `id` (PC_4) would give $\texttt{id}_2 = 2$ and $\texttt{id}_4 = 4$. In general, `id` always results in a stable and strict total ordering since every test has a unique and stable row index.

3.3 Local Frequency-based Prioritization Technique (LFP)

The *local frequency-based prioritization technique* (LFP) builds on the assumption that for every code change, there is an optimal test: The test that executes this code change the most often. To this end, for each code change c_j, this technique first selects the test t_i with the greatest execution frequency f_{ij} (local maximum, $\texttt{max}^{\Delta}$, PC_1). This selection mechanism is encoded numerically in the equation for $\texttt{max}_i^{\Delta}$ in Figure 4 by first dividing by the maximum of a column j followed by rounding down to the next integer[2]. Tests are then prioritized in descending order of their corresponding maximum.

Tests that are equivalent according to $\texttt{max}^{\Delta}$ are further ordered by $\texttt{count}^{\Delta}$ (PC_2), `count` (PC_3) and `id` (PC_4).

Example. For each code change in Figure 3a we first determine the test with the highest frequency for that code change as: t_1 for c_1^{Δ} with $f_{11} = 4$, t_4 for c_2^{Δ} with $f_{42} = 5$, t_1 and t_3 for c_3^{Δ} with $f_{13} = f_{33} = 4$, t_1 for c_4^{Δ} with $f_{14} = 2$ and t_3 for c_5^{Δ} with $f_{35} = 2$. Selecting the maximum of these highest frequencies for each test results in the metric $\texttt{max}^{\Delta}$ in Figure 3b. In particular, test t_2 never has the highest frequency for any of the code changes and is assigned $\texttt{max}_2^{\Delta} = 0$ in turn. Applying $\texttt{max}^{\Delta}$ gives the following equivalence classes $\{t_4\}, \{t_1, t_3\}, \{t_2\}$. To further compare t_1 and t_3 PC_2 gives $\texttt{count}_1^{\Delta} = 3$ and $\texttt{count}_3^{\Delta} = 4$ which leads to the strict ordering $\{t_4\}, \{t_3\}, \{t_1\}, \{t_2\}$. It is not necessary to apply PC_3 or PC_4 in this example.

[2]To avoid division by zero the denominator needs to be at least 1, whence $\max(..., 1)$.

GFP	=	$\texttt{sum}^{\Delta}$	$\rhd$	$\texttt{count}^{\Delta}$	$\rhd$	`count`	$\rhd$	`id`
LFP	=	$\texttt{max}^{\Delta}$	$\rhd$	$\texttt{count}^{\Delta}$	$\rhd$	`count`	$\rhd$	`id`
CFP	=	$\texttt{count}^{\Delta}$	$\rhd$	$\texttt{sum}^{\Delta}$	$\rhd$	`count`	$\rhd$	`id`
		PC_1		PC_2		PC_3		PC_4

Table 1: Definitions of our prioritization techniques.

3.4 Change Frequency-based Prioritization Technique (CFP)

The *change frequency-based prioritization technique* (CFP) prioritizes tests that execute the most changes. If two tests execute the same amount of changes it prefers the test that has a higher global frequency. It thus works in exactly the same way as GFP but swaps PC_1 and PC_2 in order to investigate whether global frequency or number of code changes tend to have a higher relevance in practice.

Example. To prioritize the tests in Figure 3a with CFP we first apply $\texttt{count}^{\Delta}$ and obtain $\{t_3\}, \{t_1, t_2, t_4\}$. Applying $\texttt{sum}^{\Delta}$ (PC_2) yields $\{t_3\}, \{t_1\}, \{t_2, t_4\}$ and `count` (PC_3) finally gives $\{t_3\}, \{t_1\}, \{t_2\}, \{t_4\}$.

3.5 Discussing Frequency-based Prioritization Techniques

In general, we do not use the total amount of code entities as first prioritization criterion. Consequently, we do not necessarily run large tests first. It always depends on the code changes and their execution frequencies. As mentioned before, we assume tests to be small and to focus on a specific use case or functionality.

As opposed to CFP, the techniques GFP and LFP imply that we might execute tests first that do not cover the maximum number of modifications. In Figure 3, for example, t_3 covers 4 code changes (c_1^{Δ}, c_2^{Δ}, c_3^{Δ}, c_5^{Δ}). All other tests cover fewer changes. Despite this fact, t_3 will not be executed first in GFP and LFP. However, this does not contradict the goal of test prioritization. As t_3 executes specific changes less often, it might fail to reveal a fault. In comparison, t_1 (in the case of GFP) or t_4 (in the case of LFP) respectively might set the application under test in more states and thus might be able to expose a fault. Besides, a test covering fewer code modifications might finish faster. So, there might be more time to run other tests.

Another implication of our GFP- and CFP-technique is that several highly prioritized tests might cover the same code changes. If a test has already failed, another test covering the same faulty modification runs in vain. For this reason, we propose the following dynamic feedback mechanism.

3.6 Dynamic Feedback for Frequency-based Prioritization

In previous evaluations (e.g. [10, 25, 29]), it became apparent that proposed prioritization techniques often performed better in conjunction with a feedback mechanism [25], also known as "additional" approach [10, 29]. These kind of techniques use knowledge gained during the current prioritization of tests and recursively include this information to re-prioritize the remaining tests. Thus, the final test execution order is not known before all tests have been executed.

Employing a similar strategy, we propose for the (static) X-frequency-based prioritization technique (XFP; see 3.1, step 2) a *dynamic* counterpart. It aims at lowering the priority of not yet executed test cases that will traverse one or several CIDs due to which previously executed tests have already failed. Thus, the dynamic X-frequency-based prioritization technique (DXFP) continuously adapts the test execution ordering at runtime by re-calculating prioritization criteria using information about test failures as input.

Initially, the dynamic technique employs static XFP to specify an ordering of the tests in the test suite T. After each test case execution, it checks the result of the test t that has just finished. If test t has succeeded, the next test in the prioritization order will be executed as usual. If test t has failed, the prioritization is re-evaluated. For this purpose, it checks which code changes where involved in a failing test run and thus might be fault-revealing. Their execution frequencies will be excluded from further re-prioritization. If DXFP cannot determine the exact code changes that have led to a test failure, it ignores all CIDs that have been traversed by a failed test. Afterwards, XFP is applied to the remaining test cases resulting in a test suite T' with a new test prioritization order. The test system repeats this process until all tests have been executed or until another predefined stop criterion is fulfilled.

The dynamic feedback variant can be applied to any static (X)-frequency-based prioritization technique. Considering the above-introduced strategies, we refer to the resulting variants as *Dynamic Global Frequency-based Prioritization Technique* (DGFP), *Dynamic Local Frequency-based Prioritization Technique* (DLFP), and *Dynamic Change Frequency-based Prioritization Technique* (DCFP), correspondingly.

Example. Let us illustrate the dynamic variant for GFP, DGFP by revisiting our example in Figure 3. This technique accumulates solely the execution frequencies of those code changes that have not been shown to be fault-revealing in any previously executed tests. Whenever a test fails during test execution, the accumulation has to be re-calculated for all tests not yet executed.

DGFP starts by prioritizing tests using GFP. Let us assume that t_1 fails due to c_3^Δ. This triggers a new re-prioritization of the remaining test cases. During the recalculation, c_3^Δs execution frequencies f_{i3} $(1 \leq i \leq 4)$ will be ignored. Thus, we obtain new results of the test metrics, which can be found in Figure 5.

Figure 5a shows the revised matrix. As c_3 is excluded from further calculation of metrics, t_4 gives $\texttt{sum}_4^\Delta = 8$. All other remaining tests yield lower values (see Figure 5b). Applying $\texttt{sum}^\Delta$ (PC_1) gives the following equivalence classes: $\{t_4\}$, $\{t_2, t_3\}$. The criterion $\texttt{count}^\Delta$ (PC_2) gives 3 for t_3 and 2 for t_2. This already leads to the strict re-ordering $\{t_4\}, \{t_3\}, \{t_2\}$.

	Execution Frequency						
Test	c_1^Δ	c_2^Δ	c_3^Δ	c_4^Δ	c_5^Δ	c_6	c_7
t_2	0	4	3	0	1	7	0
t_3	2	1	4	0	2	2	1
t_4	2	5	0	1	0	0	0

(a) Revised matrix f of execution frequencies per test and CID. c_3^Δ is ignored when calculating metrics, see Figure 5b.

	Metrics			
Test	$\texttt{sum}^\Delta$	$\texttt{count}^\Delta$	$\texttt{max}^\Delta$	count
t_2	5	2	0	3
t_3	5	3	2	5
t_4	8	3	5	3

(b) Revised result of the test metrics incorporating the failure of t_1 due to c_1^Δ.

Figure 5: Revised example matrix f and resulting metrics for DGFP after t_1 has been executed.

At some point, it might happen that there are only tests left that do not cover any code changes or whose execution frequencies are excluded. So there are no execution frequency data left. Thus, the dynamic approach cannot further improve the execution order with the aid of PC_1 or PC_2. Structural code coverage (PC_3) would be used to order the remaining tests which defeats our effort to prioritize tests according to their execution frequency. For this reason, we cancel any further re-ordering and the last known prioritization order will be used to run the remaining tests. This way, a strict ordering is guaranteed.

4. EVALUATION

In order to assess our solutions to provide an efficient selective regression testing technique for desktop and web applications, we have implemented our technique as a library. It is available on Github[3]. We discuss our approach in terms of the following three research questions:

RQ1 How does our technique perform compared to standard coverage-based approaches [10, 8], bayesian network-approaches [25, 24] and similarity-based approaches [14]?

RQ2 Is a dynamically adapted execution order able to outperform a static prioritization order?

RQ3 Is our technique suitable to enrich an RTS technique in a cost-effective way? How effective is the prioritization in terms of reducing test execution time?

4.1 Software under Evaluation

Previous test prioritization approaches often use software provided in the "Software-artifact Infrastructure Repository (SIR)" [7]. In order to be able to compare the results of our prioritization techniques with these approaches, we use three Java desktop applications as benchmarks that have been frequently used in the literature: JMeter, JTopas, and XML-Security. Additionally, we enhance our study

[3]https://github.com/MH42/fprio

by two web applications created with Googe Web Toolkit (GWT), namely Hupa [18] and Meisterplan [19].

JMeter [34] is a load and performance tool to test web services and web applications. According to SIR, the most recent version contains 43.400 LOC in 389 classes. JTopas [4] is able to tokenize and parse text files or streams. It is the smallest tool and contains 5400 LOC in 50 classes. XML-Security [2] encompasses libraries supporting XML-signature syntax as well as XML-encryption syntax. It contains 16800 LOC in 143 classes.

Hupa [18] is a mid-sized open source GWT-based mail client. The latest revision in the public repository (revision number 1684702) consists of approximately 40.000 non-empty lines of code (NLOC) in 979 classes and interfaces. In order to assess our approach thoroughly, we have chosen an industrial application as an additional experimental object. Meisterplan [19] is a highly dynamic and interactive resource and project portfolio planning software for executives. The source code consists of approximately 170.000 NLOC (without imports) in roughly 2300 classes and interfaces. It is the largest application in our evaluation.

4.2 Variables and Measures

During our evaluation, we have used different variable settings. We provide an overview below.

4.2.1 Independent Variables

Our evaluation depends on two independent variables: regression test selection technique that offers many different settings to analyze the software, and prioritization technique.

Regression test selection technique.

As described in Section 3.1, our prioritization technique uses the results of a previous regression test selection step as input. Our regression test selection technique [17] offers several parameters. The user can decide to analyze the software statically on a predefined analysis level or to use a heuristics which decides dynamically which analysis granularity of the code is the best one. Additionally, the user can define a lookahead value that enables a more in depth-analysis finding more code modifications.

For the evaluation, we have used a parameter setting which is the result of our findings in [17]. It takes advantage of a heuristics that analyses modified code on the static analysis level "Expression Star" (= statement level that also considers e.g. conditional expressions) with a lookahead value 5. Unmodified code will be analyzed on body declaration level with the same lookahead value 5.

Prioritization technique.

We investigate three different versions of our prioritization technique and additionally analyze for each of them the impact of a feedback mechanism. So in total, we consider six test prioritization techniques.

4.2.2 Dependent Variables

We evaluate the results of our prioritization technique on the basis of a metric, called Average of the Percentage of Faults Detected (APFD) [10] which is the standard metric in the literature. It measures the ability of an approach to detect faults as soon as possible. The codomain attains values in the interval $I = [0; 100]$, where higher values indicate a better fault detection. The metric requires as input the number of tests n in a test suite T, the number of code modifications resulting in a failure of at least one of the test cases and an ordered test suite T'. The index of the first test in T' that detects fault number i is referred to as TF_i. Then, the APFD metric is the result of the equation:

$$APFD = 1 - \frac{TF_1 + TF_2 + \cdots + TF_m}{nm} + \frac{1}{2n}$$

Using this metric, an objective comparison with previously published approaches is possible. This is even true when considering our dynamic feedback techniques. The APFD metric is applied only when after all tests have been finished, the prioritization order does not change anymore.

4.3 Experimental Setup

For each of the applications described in Section 4.1, several versions are available. The desktop applications contain seeded errors which are independent. Thus, we know for each code modification whether it is a fault. The web applications represent regular revisions taken from a repository with real errors (if there are any). As real code modifications depend on each other, it is not possible to check due to which code changes a test has failed. So, in case of a test failure, we consider all changes covered by the failed test as faults. This way, we are able to compare our results.

Figure 6 provides an overview of the overall number of versions available per software, the aggregate amount of faults in all versions of a software, and the number of tests in the latest version. The evaluation has been performed on a Intel Core i5 2.4 GHz with 8 GB RAM.

	JMeter	JTopas	XML-Security
versions	6	4	4
faults	9	6	5
tests	78	128	83

	Hupa	Meisterplan
versions	4	6
changes considered as faults[4]	8[4]	136[4]
tests	32	106

Figure 6: Errors in software

4.4 Threats to Validity

The results of our evaluation and our conclusions might be threatened by different factors. We will discuss these issues and how we have tried to minimize their effects.

4.4.1 External Threats to Validity:

In general, the results might depend on the size of the applications used for the evaluation. Besides, the kind of application (desktop application vs web application) might influence the ability to generalize the results. To limit these factors, we studied both desktop applications and web applications of different size. As we also investigate an industrial application, we can benefit from real faults and real tests cases. However, in order to be able to compare our results with previous results in the literature, the range of possible applications have been predetermined to some degree by the selection of former state-of-the-art publications. Concerning the applications in the SIR repository, errors have

[4]More details on considering code changes as faults follow in section 4.4.2.

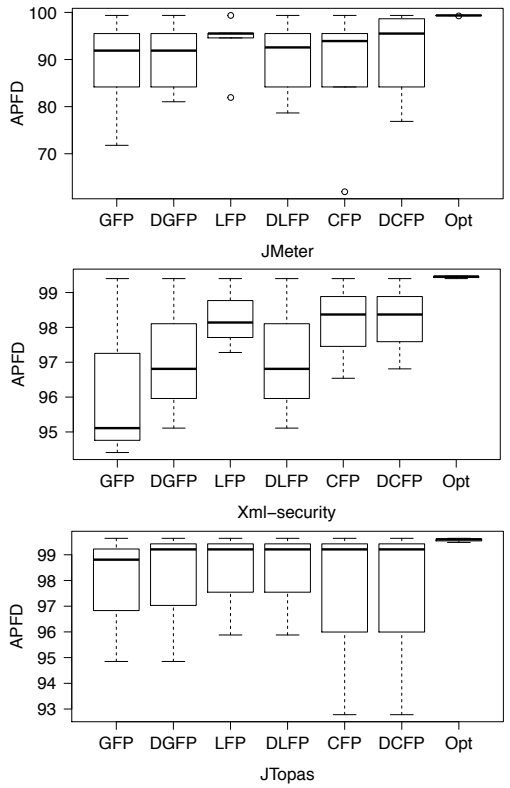

Figure 7: APFD values for our techniques applied to standard Java applications

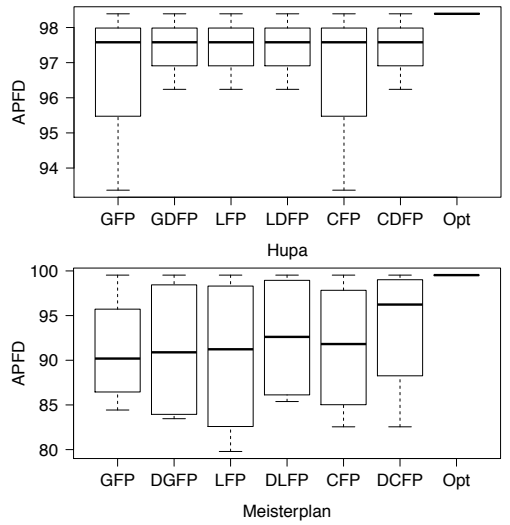

Figure 8: APFD values for our techniques applied to transcompiled GWT applications

been seeded manually, so these faults might differ from those emerging in industrial environments. Besides, we rely on the original software obtained from the SIR repository to preserve comparability without additionally seeding errors. However, some other authors additionally seeded errors.

4.4.2 *Internal Threats to Validity:*

Our instrumentation, our logging tool, and our tool to determine the prioritization could contain a fault. To exclude this threat as far as possible, we have executed our tools on a set of small programs and manually inspected the results.

Another threat to the validity of our conclusions might arise out of the number of faults. Some applications contain seeded faults, some contain faults emerging during further development. Nevertheless, the number of faults has been rather small except three versions in Meisterplan. We could imagine that a higher number of faults lead to additional or other insights. In case of the Meisterplan versions, the company could not tell us anymore which code modifications are actually faults. For this reason, we have considered all code changes covered by a failed test as faults, which might be wrong.

Finally, we have noticed that some of the test suites belonging to SIR-projects seem to have evolved slightly. Comparing the statistics of previous papers, the number of test-methods sometimes slightly differed (e.g. [8, 36]). For this reason, our results might be not completely comparable with results obtained from other techniques. Nevertheless, the differences are small, so we still get a good indication on how our technique performs.

4.4.3 *Threats to construct validity:*

We use the APFD metric that measures the weighted average percentage of detected faults. Elbaum et al. [9] introduce a cost-cognizant version of the APFD metric that considers the severity of faults as well as costs for test cases. Especially the test costs (i.e. time) play an important role in our case, too. However, as all necessary data (mainly code instrumentation and the creation of test execution logs) can be obtained from the previous regression test selection, the ordering algorithm takes only a few seconds. Besides, as we do not have data on the severity (e.g. time to correct faults or damages caused by a fault) and - again - in order to be able to compare our results with approaches in the literature, we use the APFD metric.

4.5 Results

In the following subsections, we will discuss the results of our evaluation in terms of our research questions.

4.5.1 *RQ1: Comparing Performance of our Technique with existing Approaches*

Figure 7 and Figure 8 show the APFD values obtained during our evaluation. We use box plots to report on the results of each software under test. The horizontal line within the box plots represents the median. The boxes below/above the median contain the lower/upper 25% of the APFD values. The vertical lines (whiskers) at both ends of the box show the other values ignoring possible outliers. The horizontal axis of the figures represents the proposed techniques as well as the optimal technique that orders the tests according to their ability to detect faults as fast as possible. Of course, this is only possible because we have knowledge which tests fail.

Our techniques achieve very high APFD-values which are sometimes close or even equals to the optimum. Furthermore, we note that the value distribution of our techniques is often very small. When comparing the median values of all six prioritization variants, the difference always lies in the interval $I = [0; 6, 2]$. So none of our techniques falls extremely behind the others. Having a closer look at the

Objects	M	Test-method level							
	SD	T1	T2	T3	T4	T5	T6	T7	T8
JMeter	M	48	60	34	74	34	77	42	55
	SD	37	19	38	24	38	18	35	35
XML-S.	M	48	71	96	96	97	87	96	96
	SD	34	17	3	3	3	12	3	3
JTopas	M	35	61	68	97	68	97	77	75
	SD	21	12	50	2	51	2	19	19

(a) Mean value and standard deviation of APFD values in Do et al.'s former assessment [8]

	S1	S2	S3	S4
JMeter	[65; 75]	[62; 72]	[70; 75]	[62; 72]
XML-Security	[90; 95]	[90; 95]	[90; 95]	[90; 95]

(b) Median APFD values of a technique replicated from Mirarab et al. [25]

	T1	T2	T3	T4
JTopas	[60; 65]	[55: 60]	[50; 55]	[50; 55]
XML-Security	[80; 85]	[80: 85]	[65; 70]	[65; 70]
	T5	T6	T7	T8
JTopas	[55; 60]	[45; 50]	[30; 35]	[55; 60]
XML-Security	[80; 85]	[80; 85]	[80; 85]	[75; 80]

(c) Median APFD values of a technique replicated from Fang et al. [14]

Table 2: APFD values of alternative techniques

results, LFP and (D)CFP outperforms GFP in most of the cases. Considering XML-Security, (D)CFP shows slightly better results than all other techniques. However, when considering JMeter, LFP is better.

In order to judge the performance of our technique better, we compare our results with those obtained by other state-of-the-art techniques. Our comparison includes structural coverage techniques investigated by Do et al. [8]. Table 2a shows the results of their study. It includes the following techniques: original ordering (T1), random ordering (T2), and orderings according to some variants of block/method coverage with(out) considering change information (T3+T4/T5-T8). In detail, T3 prioritizes tests according to the total number of executed blocks. T4 additionally incorporates feedback. T5 prioritizes tests according to the total number of executed methods. Similar to T4, T6 adds the feedback-mechanism to T5. T7 runs tests first that show the highest coverage of methods that have been modified. Do et al. obtain change information by using the *diff function* provided by Unix. Finally, T8 extends T7 by a feedback mechanism.

Table 2b and Table 2c summarize the median values we have gathered from boxplots presented by Mirarab et al. [25] and Fang et al. [14]. As it is difficult to determine the exact values from the boxplots, we use intervals $I = [a; b]$. Fang et al. considered similarity to prioritize test cases. They present four own approaches (T5-T8) and compare them to other similarity-based approaches. T5 and T6 are based on an algorithm called "Farthest-first Ordered Sequence". T7 and T8 use an algorithm called "Greed-aided-clustering Ordered Sequence". Mirarab et al. used bayesian networks to prioritize test cases. They have evaluated their approach with four different settings S1-S4. For each of these settings, they obtain several APFD values. We unify the APFD values of every setting in the corresponding interval in Table 2b.

We rely in the same way as Do et al. on the test suites obtained from SIR without extensions. In contrast Mirarab et al. and Fang et al. both use individually developed mutation faults. This hampers a direct comparison, but as they use exactly the same software as Do et al. and we do, we still get an overview.

Compared with the approaches in Table 2, our techniques tend to achieve higher APFD-values, which especially outperform the results of Do et al. (compare e.g. JMeter in Table 2a with the corresponding values in Figure 7). Furthermore, we observe that our standard deviation is usually smaller. Regarding XML-security, the overall results of Fang et al. and Do et al. are more similar to ours, but especially Fang et al. show lower values than we do.

In summary, the evaluation of our test-prioritization technique shows that considering the execution frequency as main factor is a good criterion to calculate a test prioritization order which is close to the optimum.

4.5.2 RQ2: Dynamically Adapted Execution Order vs. Static Execution Order

As a first result, we can see that the dynamically adapted execution orders created with DGFP and DCFP are never worse than their statically calculated counterpart. Solely DLFP repeatedly falls behind the static variant. In some cases (e.g. XML-security's DGFP, JMeter's DCFP and Meisterplan's DCFP), the dynamic variant performs better due to the knowledge about failed tests. Of course, this benefit involves an increasing runtime as the prioritization order has to be recalculated after each failed test. Basically, this is no problem as long as the following conditions are met: a) the runtime of the prioritization technique is low, and b) the dynamically adapted order prioritizes fault revealing tests.

	Software whose tests will be prioritized				
	JMeter	XML-s.	JTopas	Hupa	Meister.
GFP	1,77	1,81	1,94	2,55	38
DGFP	1,93	1,76	2,05	2,98	91
LFP	1,46	1,94	1,97	2,71	34
DLFP	2,31	1,77	1,85	3,43	75
CFP	1,74	1,82	1,67	2,65	34
DCFP	1,89	1,97	2,24	3,24	91

Figure 9: Runtime (sec) of prioritization techniques

Table 9 shows median values for the runtime of all techniques (in seconds). The runtime for static variants of the desktop applications has been shown to be always less than two seconds. When ignoring the time required for querying the database, the time for running the algorithms has usually been in the range of milliseconds. Regarding the larger applications Hupa and Meisterplan, the static variants still required less than one minute to calculate the test execution order.

Concerning the dynamic variants, the runtime increases. The main reason is the criterion PC_3 (highest structural coverage) as all the CIDs affected by a test have to be summed up. In large applications with several millions of CIDs, this becomes easily a time factor especially when the

prioritization order has to be updated often due to many tests failures. This is, in particular, true for Meisterplan.

In total, dynamic techniques only provide a rather small benefit compared to the static ones. This is somewhat different to the findings of some researchers who observed a significant improvements when using a feedback mechanism (e.g. [8]), but confirms the findings of other authors (e.g. [25]) who also observed small improvements and an even worse runtime overhead. Thus, the results are twofold as a dynamic approach improves the APFD value, but impairs the runtime. Is there a high chance for many failures, the static technique should be preferred.

4.5.3 RQ3: Prioritization as Supplement to RTS Techniques

RTS techniques always have an overhead for analyzing the code and selecting the tests affected by code changes. When comparing RTS techniques with a retest-all approach, they have to reduce the number of tests in order to be cost-effective. Sometimes, however, a change affects all tests so the RTS technique is unable to determine a subset of the original test suite that is still safe [27]. Besides, the test execution order is unclear. To avoid running all tests requires knowledge about the probability that a test will reveal a fault. The extra overhead should be minimal. Our prioritization technique is very well suited as it is able to define such an order very quickly. All required data (i.e. code modifications, execution frequency, test traces) are already available from the test selection. This is the phase, which causes overhead for instrumenting code and for running tests on instrumented code. (In case of Meisterplan, the extra overhead for running an instrumented test and for logging the corresponding CIDs has been 51 sec on average. More details can be found in [17].) In contrast, the effort for prioritizing tests is completely negligible. Regarding the desktop applications, the techniques have always finished in less than two seconds. Even prioritizing tests for the industrial application has been very fast.

When applying dynamic techniques, we have to take into account how often the order will be recalculated. This implies that we have to estimate, how error-prone the code is. If the source base is almost stable, a dynamic technique would be a good choice. Otherwise, a static technique should be preferred. In any case, our techniques are able to define a test prioritization order that usually accomplishes in running fault revealing tests soon. The extra time-effort is minimal. This is very important as developers can decide to run only a subset of the original test suite. The risk to miss a fault due to a test that has not been executed is rather small.

5. DISCUSSION

Test Suite Granularity.

Do et al. [8] have considered the effects of different test suite granularities and investigated tests at class and at method level. Executing tests at a class level implies executing all of the test methods within that class, whereas, at a method level only a sinlge test method is executed.

While it would be possible to also consider tests at class level in our approach, we believe that this would have negative impact on the precision of the prioritization. Imagine a test class C consisting of many test methods. Let us assume that the test selection (see step 1) selects many test classes for re-execution, including C, but only a single test method in C covers a code change. As a consequence, all test methods in C would be executed, even though it is obvious from the test selection step that only one test method has the chance to reveal a fault. This is contradictory to our target to run only a subset of the original test suite in order to be cost-effective. Furthermore, we have analyzed the applications on statement level and considered even expressions (e.g. conditional expression) to keep the set of selected tests as small as possible. For this reason, we consider in our approach every test method to be a test.

Depending on Old Frequency Data.

Our approach highly relies on the frequency data gather by prior test runs as a heuristics when assigning priorities to tests in the current test run. The frequencies for a given test have either been gathered in the preceding test run or, if the test has previously been excluded by test-selection, from an even older test run. We assume test selection to be a conservative approximation and thus it should not make a difference from which test run the frequencies origin, as long as all the selected tests are all eventually executed.

In an interactive development cycle, however, one can imagine that test runs are frequently aborted to only test the newest program version. Such behavior could cause frequency data to be outdated and thus violate assumptions of our approach potentially causing a loss of precision. More research is necessary to determine the impact of relying on such outdated frequency data and how our system can be used in interactive development with repeated preliminary abortion of the full test-run.

Dynamic Prioritization Variant.

As already discussed in Section 3.5, our main motivation for the dynamic feedback mechanism has been to avoid running tests in vain. This could easily happen when a test has already failed and another one covering similar code with other parameters is about to start. It has turned out that a dynamic approach, in fact, reduces the APFD value, but only to a small extent. This is due to the relatively high APFD values we already obtain in our static variants. In case of parameterized tests, it might even happen that the dynamic variant provides no benefit at all. This is for example true for JTopas. The test suite contains several parameterized tests. All of them have a high priority.

Running Tests in Parallel.

When searching for opportunities to reduce the runtime overhead, companies try to run tests in parallel. Our approach of combining a RTS technique with a test prioritization approach meets this demand. For our static variants, the only requirement is that tests run independently. Regarding our dynamic variants, further work has to be done. These approaches benefit from the knowledge of failed tests. Even if all processes that run in parallel have this knowledge, it has to be ensured that similar tests with differing parameters do not run in parallel on separate machines. Otherwise, this could impact the runtime benefit gained by parallel execution. To solve this problem, test groups are adequate. This concept is already known from unit testing tools like TestNG [33]. Even without using a test-framework that supports groups, it is straightforward to provide meta-data on which tests belong together. These tests should not run in parallel.

6. RELATED WORK

Many approaches use structural coverage as decider on how to prioritize test cases. Elbaum et al. [10, 11] propose several techniques considering the coverage of statements or functions ("total"-approach). In addition, Do et al. consider in [8] coverage of blocks. In refined versions, both papers additionally take into account whether statements have already been covered by a test ("additional"-approach/feedback). Moreover, Elbaum et al. present techniques estimating the probability that a test exposes faults or that any fault exists. We also use coverage, but we consider coverage in terms of execution frequency of code entities. Concerning feedback, our mechanism is completely dynamic as it modifies the prioritization order on the fly at runtime.

There are several papers which use code change information for test prioritization as we do. Aggrawal et al. [1] apply a version specific test selection to obtain information about changed lines of code. The test prioritization prefers tests that cover the most changed lines of code. Mirarab et al. [24] also use code change information and consider additionally coverage metrics and software quality metrics to create a bayesian network. This model serves as a basis to decide upon a prioritization order. Besides, they describe in [25] an "additional"-approach. Zhao et al. [37] present a hybrid form that additionally takes a code-coverage based clustering approach into account to detect similarities in test cases in order to reduce their priority.

Panigrahi et al. [26] use an extended system dependency graph-based RTS technique to determine code changes. They start with the test covering the most changes. For prioritizing the remaining tests, they assign weights to changes. The weights are reduced by a fixed value if a test selected for execution covers the changes. A very similar approach has also been presented by Srivastava et al. [32] before. They use a binary matching tool called BMAT to identify changed blocks. They also rely on weights but do no longer consider blocks as modified if they have been executed once. Jeffrey et al. [20] rely on statement/branch coverage and additionally apply relevant slicing to get information about code that (potentially) affects the test result.

In contrast to our work, none of the approaches considers the execution frequency of a code change to cover as much application states as possible, which increases the possibility to detect a fault.

To the best of our knowledge, execution frequency has almost never been considered as coverage measure. Only the approach of Fang et al. [14] is close to ours as they also apply the execution frequency of code entities. However, they start with the test that covers the most code elements. They use the execution count to order sequences of program entities. Based on these sequences, they apply an edit distance function to calculate the similarity of remaining tests. In the end, most dissimilar tests are executed first. Similarly, Leon et al. [22] also use dissimilarity metrics to maximize the diversity of highly prioritized tests. As opposed to our approach, they do not use an RTS technique to identify code modifications and do not consider the execution frequency of changed code as decider on which tests should run with high priority. Moreover, we regard similar tests as an indicator of fault-prone code and do not lower their priority.

Epitropakis et al. [13] present a solution that incorporates three different prioritizing criteria to achieve multi-objective regression test case prioritization. This is somewhat similar to our approach as we also consider many different criteria for prioritizing test cases. However, we do not use Pareto fronts but use the accumulated execution frequencies of traversed code modifications.

In the area of web applications, Sampath et al. prioritize tests - among others - according to the test length, the frequency a sequence of web pages is accessed in a test, and randomly. Again, they do not consider changes and the frequency with which these changes are executed by a web test.

7. CONCLUSION AND FUTURE WORK

Test case prioritization is a possibility to improve the execution order of a test suite when other techniques are not able to find an unambiguous one. We are the first who use the execution frequency of code modifications as a coverage measure to prioritize tests. We have presented three different static variants of our technique plus three dynamic counterparts with a feedback mechanism. To assess the performance of our technique, we have evaluated the different variants on three Java desktop and two web applications of different size and used the standard APFD metric to compare our results with findings in former papers.

Our results show similar to previously published papers that a feedback mechanism is able to improve the ability to detect faults early. However, the improvements are rather small and imply a higher runtime. As the APFD values of our static variants are already high and the deviation is rather small, it has to be decided individually whether one of our dynamic variants is able to outperform the static counterpart. Especially when looking at the industrial application, the additional runtime overhead of the dynamic techniques has been too large to be cost efficient.

We have used one large industrial application to evaluate our technique. The other ones have been of mid- or low scale size. In future work, we should confirm our findings on more large scaled applications.

8. REFERENCES

[1] K. K. Aggrawal, Y. Singh, and A. Kaur. Code coverage based technique for prioritizing test cases for regression testing. *SIGSOFT Softw. Eng. Notes*, 29(5):1–4, Sep. 2004.

[2] Apache Santuario. XML Security. http://santuario.apache.org/, 2016.

[3] B. Athira and P. Samuel. Web services regression test case prioritization. In *Computer Information Systems and Industrial Management Applications (CISIM), 2010 Internat. Conference on*, pages 438–443, Oct 2010.

[4] H. Blau. JTopas. jtopas.sourceforge.net/jtopas/, Nov. 2004.

[5] L. Chen, Z. Wang, L. Xu, H. Lu, and B. Xu. Test case prioritization for web service regression testing. In *Service Oriented System Engineering (SOSE), 2010 Fifth IEEE International Symposium on*, pages 173–178, June 2010.

[6] Y.-F. Chen, D. S. Rosenblum, and K.-P. Vo. TestTube: A System for Selective Regression Testing. In *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, pages 211–220, 1994.

[7] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical*

Software Engineering: An International Journal, 10(4):405–435, 2005.

[8] H. Do, G. Rothermel, and A. Kinneer. Prioritizing junit test cases: An empirical assessment and cost-benefits analysis. *Empirical Software Engineering*, 11(1):33–70, 2006.

[9] S. Elbaum, A. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 329–338, 2001.

[10] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '00, pages 102–112, 2000.

[11] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Trans. Softw. Eng.*, 28(2):159–182, Feb. 2002.

[12] E. Engström, P. Runeson, and M. Skoglund. A Systematic Review on Regression Test Selection Techniques. *Inf. Softw. Technol.*, 52(1):14–30, Jan. 2010.

[13] M. G. Epitropakis, S. Yoo, M. Harman, and E. K. Burke. Empirical evaluation of pareto efficient multi-objective regression test case prioritisation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 234–245, 2015.

[14] C. Fang, Z. Chen, K. Wu, and Z. Zhao. Similarity-based test case prioritization using ordered sequences of program entities. *Software Quality Journal*, 22(2):335–361, 2013.

[15] M. Fowler. Continuous Integration. http://martinfowler.com/articles/continuousIntegration.html, May 2006.

[16] M. Hirzel. Selective Regression Testing for Web Applications Created with Google Web Toolkit. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '14, pages 110–121, 2014.

[17] M. Hirzel and H. Klaeren. Graph-Walk-based Selective Regression Testing of Web Applications Created with Google Web Toolkit. In *Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering 2016 (SE 2016), Wien, 23.-26. Februar 2016*, pages 55–69, 2016.

[18] Hupa. Overview. http://james.apache.org/hupa/index.html, June 2012.

[19] itdesign. Take Your Project Portfolio Management to a New Level. https://meisterplan.com/en/features/, 2015.

[20] D. Jeffrey and R. Gupta. Test case prioritization using relevant slices. In *Computer Software and Applications Conference, 2006. COMPSAC '06. 30th Annual International*, volume 1, pages 411–420, Sept 2006.

[21] J.-M. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 119–129, 2002.

[22] D. Leon and A. Podgurski. A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases. In *Proceedings of the 14th International Symposium on Software Reliability Engineering*, ISSRE '03, pages 442–, 2003.

[23] Z. Li, M. Harman, and R. M. Hierons. Search algorithms for regression test case prioritization. *IEEE Transactions on Softw. Eng.*, 33(4):225–237, 2007.

[24] S. Mirarab and L. Tahvildari. A prioritization approach for software test cases based on bayesian networks. In *Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering*, FASE'07, pages 276–290, 2007.

[25] S. Mirarab and L. Tahvildari. An empirical study on bayesian network-based approach for test case prioritization. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 278–287, April 2008.

[26] C. Panigrahi and R. Mall. A heuristic-based regression test case prioritization approach for object-oriented programs. *Innovations in Systems and Software Engineering*, 10(3):155–163, 2014.

[27] G. Rothermel and M. J. Harrold. Analyzing Regression Test Selection Techniques. *IEEE Transactions on Software Engineering*, 22, 1996.

[28] G. Rothermel and M. J. Harrold. A Safe, Efficient Regression Test Selection Technique. *ACM Trans. Softw. Eng. Methodol.*, 6(2):173–210, Apr. 1997.

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

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

[31] S. Sampath, R. C. Bryce, G. Viswanath, V. Kandimalla, and A. G. Koru. Prioritizing user-session-based test cases for web applications testing. In *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, ICST '08, pages 141–150, 2008.

[32] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, pages 97–106, 2002.

[33] TestNG. Documentation - TestNG Test-Groups. http://testng.org/doc/documentation-main.html#test-groups, 2016.

[34] The Apache Software Foundation. Apache JMeter. http://jmeter.apache.org/, 2016.

[35] S. Yoo and M. Harman. Regression Testing Minimization, Selection and Prioritization: A Survey. *Softw. Test. Verif. Reliab.*, 22(2):67–120, Mar. 2012.

[36] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei. Bridging the gap between the total and additional test-case prioritization strategies. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 192–201, 2013.

[37] X. Zhao, Z. Wang, X. Fan, and Z. Wang. A clustering-bayesian network based approach for test case prioritization. In *Computer Software and Applications Conference (COMPSAC), 2015 IEEE 39th Annual*, volume 3, pages 542–547, July 2015.