PRIORITIZATION AND PARALLEL EXECUTION OF TEST CASES FOR CERTIFICATION TESTING

A Thesis

by

Şahin Dirim

Submitted to the
Graduate School of Sciences and Engineering
In Partial Fulfillment of the Requirements for
the Degree of

Master of Science

in the
Department of Computer Science

Özyeğin University
June 2021

PRIORITIZATION AND PARALLEL EXECUTION OF TEST CASES FOR CERTIFICATION TESTING

Approved by:

Assoc. Prof. Hasan Sözer, Advisor
Department of Computer Science
Özyeğin University

Assoc. Prof. O. Örsan Özener
Industrial Engineering Department
Özyeğin University

Assoc. Prof. Geylani Kardaş
International Computer Institute
Ege University

Date Approved: 8 June 2021

To my family ...

ABSTRACT

Systems that are developed in the consumer electronics domain are subject to testing for certification regularly. For instance, each Smart TV product must go through a certification process that is regulated by application vendors like Netflix, YouTube and Amazon. There exist a separate test suite to be used as part of the certification process pertaining to each application. Each test suite includes hundreds of test cases. Many of these test cases have to be executed manually and it might take several days to complete the execution of certification test suites. There also exists a high variation among the execution times of test cases and severities of faults detected by these test cases. Therefore, prioritization of test cases and parallel test execution can significantly impact the duration and effectiveness of the process. We measure test effectiveness by calculating the average of the percentage of faults detected. In this thesis, we introduce variants of existing metrics to take parallel test execution, varying test execution times, and fault severity levels in ordinal scale into account. We also propose an approach for prioritizing such test cases that can be executed on multiple test stations in parallel. We employ integer linear programming for optimizing the allocation of test execution tasks on a set of available test stations. The goal of this allocation is to minimize the overall test execution time. Test cases that are allocated to a particular stations are executed in increasing order of their execution times. We conducted an industrial case study in the context of certification testing for Smart TV software. We used certification test suites of 3 Smart TV applications applied on 3 TV software projects as real experimental objects. We repeated our measurements for two test cycles for each project, adding up to 6 tests performed for each of the 3 test suites. So, our evaluation involves 18 test sessions. We evaluated 3 scenarios for

each of these sessions, where the number of available test stations are 1, 3 and 5. We compared the effectiveness of our approach with respect to a greedy approach as the baseline. We observed that the overall test execution time can be reduced by up to 16% even when only 3 test stations are available. Test effectiveness is also improved as a result of optimal scheduling of test cases.

ÖZETÇE

Tüketici elektroniği alanında geliştirilen sistemler düzenli olarak sertifikasyon testine tabidir. Örneğin her Smart TV ürünü Netflix, YouTube ve Amazon gibi uygulama satıcıları tarafından düzenlenen bir sertifika sürecinden geçmelidir. Her bir uygulama ile ilgili sertifikasyon sürecinin bir parçası olarak kullanılacak ayrı bir test grubu vardır. Her test grubu yüzlerce test senaryosu içermektedir. Bu test senaryolarının çoğunun manuel olarak yürütülmesi gerekmektedir. Bu nedenle sertifika test gruplarının koşumunun tamamlanması birkaç gün sürebilmektedir. Ayrıca test senaryolarının yürütme süreleri ve test senaryolarının bulduğu hataların önem dereceleri arasında da büyük farklılıklar vardır. Bu nedenle test senaryolarının önceliklendirilmesi ve paralel olarak yürütülmesi, test yürütme sürecinin süresini ve etkinliğini önemli ölçüde etkileyebilmektedir. Tespit edilen hataların yüzdesinin ortalaması hesaplanarak test etkinliği ölçülmektedir. Bu tezde, paralel test yürütmeyi, değişen test yürütme sürelerini ve hata önem dereceleri sıralı ölçekte hesaba katacak şekilde mevcut metriklerin varyantları sunulmuştur. Ayrıca birden çok test istasyonunda paralel olarak yürütülebilecek bu tür test senaryolarının sıralanması için bir yaklaşım önerilmiştir. Bir dizi mevcut test istasyonunda test yürütme planlarının paylaşımını eniyilemek için tam sayı doğrusal programlama kullanılmıştır. Bu paylaşımın amacı, genel test yürütme süresini en aza indirmektir. Belirli bir test istasyonuna paylaştırılan test senaryolarının yürütme sürelerinin artan sırasına göre koşumu tamamlanmıştır. Smart TV yazılımı için sertifika testi bağlamında endüstriyel bir vaka çalışması gerçek-leştirilmiştir. Sertifika test grupları, 3 TV yazılım projesinde kullanılan 3 farklı Smart TV uygulaması üzerinde gerçek deneysel nesneler olarak kullanılmıştır. Her proje için iki test döngüsü olarak ölçümlerimiz tekrarlanmıştır

ve 3 test grubunun her biri için 6 test koşumu gerçekleştirilmiştir. Sonuç olarak değerlendirmemiz 18 test oturumu içermektedir. Bu oturumların her biri, mevcut test istasyonlarının sayısının 1, 3 ve 5 olduğu 3 senaryo ile değerlendirilmiştir. Yaklaşımımızın etkinliği temel olarak açgözlü yaklaşımla karşılaştırılmıştır. Yalnızca 3 test kurulumu mevcut olduğunda bile genel test yürütme süresinin %16'ya kadar azaltılabileceği gözlemlenmiştir. Test senaryolarının test istasyonlarına paylaşımının eniyilemesi sonucunda test etkinliği de iyileşmiştir.

ACKNOWLEDGEMENTS

We would like to thank software developers and software test engineers at Vestel Electronics for sharing their artefacts and resources with us to support our case study.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Safety-critical software systems can be subject to catastrophic failures. Therefore, certification tests are commonly applied to these systems. These tests are regulated by standards. For instance, DO-178B [1] is adopted for testing avionics software. Systems that are developed in the consumer electronics domain are not subject to catastrophic failures. However, testing for certification is also common in this domain, where guidelines and rules for testing are defined by application vendors. For example, a Smart TV product must be certified to be accepted as a Netflix-supported device [2]. Netflix provides a test suite for this purpose. There are several other vendors like YouTube and Amazon. They provide separate test suites to be used as part of the certification process pertaining to their own applications. These test suites have to be repeatedly executed for each product. Each test suite includes hundreds of test cases. Many of these test cases have to be executed manually. For instance, some of them require watching video content for several minutes. As a result, it might take several days to complete the execution of all the certification test suites. Test suites are also repeatedly executed in the context of regression testing [3–6]. Repeated test executions ensure that modifications and extensions on software do not introduce bugs or side effects. However, these test executions can turn out to be too costly for large-scale systems. There have been 3 basic approaches [4] applied to address this issue: *i)* test case selection, *ii)* test suite minimization, and *iii)* test case prioritization. Test case selection [7] is applied to select a subset of test cases within a test suite. This subset is associated with the changed parts of the software that are assumed to be error-prone. Test suite minimization [8] aims at identifying

obsolete or redundant test cases by considering multiple objectives and constraints. These test cases are then eliminated from the test suite. A subset of the test cases can be eliminated permanently [6]. In this case, the applied approach is also called test suite reduction [9–12]. Test case minimization, selection, and reduction approaches are not applicable for certification testing. One cannot arbitrarily select a subset of test cases or discard any test case from a certification test suite. None of the test cases can be omitted or deemed redundant. However, test case prioritization can be applied to schedule the execution order of test cases and as such improve efficiency and testing effectiveness.

Test case prioritization aims at finding an optimal execution order of test cases in a test suite [13]. The optimization can be performed based on various measures regarding test cases such as their cost [14], code coverage [15], the priority of the verified requirements [16], and historical data regarding their previous ordering and fault detection effectiveness [17]. The desired outcome of this optimization is mainly to detect faults as early as possible [6, 18, 19]. Existing prioritization methods and techniques mainly assume that test cases have to be executed sequentially on a single test station. Only a few studies [20] considered a scenario, where parallel execution of test cases on multiple test stations is possible. In this thesis, we present an approach for prioritizing test cases and that can be executed on multiple test stations in parallel. As a particularity in this context, there exists a high variation among the execution times of test cases in this context. Hereby, the cost is measured in terms of the time it takes to execute a test case. Therefore, scheduling test execution tasks can significantly impact the duration and effectiveness of the process. There is also a high variation regarding fault severities. Some of the faults are highly critical and they have to be fixed immediately. Some other faults might not be even bothering. More often than not, consumer electronics products are shipped with known software faults [21]. Fault severity is mainly measured in ordinal scale. There have been

studies [22] that take such particularities into account. However, we observed that they still fall short to satisfy the needs of our application context. In particular, existing metrics for assessing the rate of fault detection fall short due to the way that fault severities are measured and due to the lack of consideration of a parallel test execution scenario. We discuss such defficiencies and the room for improvement based on our case study with the certification test suites of 3 Smart TV applications as real experimental objects.

We propose two variants of existing metrics for evaluating the effectiveness of test prioritization. The first one, named as LAPFD (Lexicographic ordering of Average of the Percentage of Faults Detected), calculates the average of the percentage of faults detected over time, during the execution of test cases. Hence, test execution time is taken into account as the cost of test cases. In addition, the calculation is performed separately per severity class. Then, a lexicographic ordering is performed based on these classes. The second metric, named as APFD_{pc}, is a variant of the Average of the Percentage of Faults Detected (APFD) [18] metric. It takes costs of test cases into account by calculating the average of the percentage of faults detected over time, like LAPFD. However, the calculation of completion times of test cases are performed by taking their parallel execution into account.

We propose a novel approach for prioritizing test cases that can be executed on multiple test stations in parallel. We employ Integer Linear Programming (ILP) for optimizing the allocation of test execution tasks on a set of available test stations. The goal of this allocation is to minimize the overall test execution time, that is, the completion time of the last test, which is known as the makespan [23]. Test cases allocated to a particular test station are executed in increasing order of their execution times to prioritize tests with the shortest execution times first.Test cases allocated to a particular test station are executed in increasing order of their execution times to prioritize tests with the shortest execution times first.

We use certification test suites of 3 Smart TV applications applied on 3 TV software projects as real experimental objects like the previous experiment. We repeat our measurements for two test cycles for each project, adding up to 6 tests performed for each of the 3 test suites. As a result, our evaluation involves 18 test sessions in total. We evaluate 3 scenarios for each of these sessions, where the number of available test stations are 1, 3 and 5.

We compare a random ordering of test cases with respect to a greedy approach [20] as the baseline for the sequential test execution case where there is only one test station. These alternative orderings are evaluated by using the LAPFD metric. We observe that greedy approach ordering of test cases consistently outperformed random ordering. We also observe that there is a large room for improvement regarding the effectiveness of test case prioritization in this application domain. We perform another evaluation by using the APFD_{pc} metric for scenarios where the number of test stations are 3 and 5. We compare the efficiency and effectiveness of our approach with respect to a greedy approach [20] as the baseline. Results show that the overall test execution time can be reduced by up to 16% even when only 3 test stations are available. Test effectiveness is also improved as a result of optimal scheduling of test cases when the rate of fault detection is considered. The amount of improvement ranges between 3.06% and 34.67%.

The remainder of this thesis is organized as follows. In Chapter 2, we provide background information on existing metrics for measuring test effectiveness and we introduce our variants. In Chapter 3, we present our optimization model used for assigning test cases to multiple test stations. In Chapter 4, we introduce the experimental setup for our industrial case study. In Chapter 5, We discuss the obtained results. we summarize related studies in Chapter 6. Finally, we conclude the thesis in Chapter 7.

CHAPTER II

METRICS FOR EVALUATING THE EFFECTIVENESS OF TEST CASE PRIORITIZATION

The basic goal of test case prioritization is to detect faults as early as possible. Hence, several metrics have been proposed to quantify how quickly a prioritized test suite detects faults. In this chapter, we first explain APFD and APFD_c metrics that were previously proposed. Then, we introduce two variants of these metrics, LAPFD and APFD_{pc} that are tailored for our application context.

2.1 APFD

The mostly cited and well-known metric [6]. is the Average of the Percentage of Faults Detected (APFD) [18]. Most of the other metrics are introduced as variants of this one. The basic idea is to calculate the weighted average of the percentages of faults detected by test cases. APFD values range between 0 and 100 and higher values indicate a faster rate of fault detection. The metric calculation can also be visualized with a chart where x-axis corresponds to the percentage of test cases executed so far and y-axis corresponds to the number of distinct faults detected so far. The area under the curve gives the APFD value.

A toy example is presented in Table 1 to illustrate the calculation of APFD. Hereby, there are 4 test cases listed. There are 4 distinct faults detected by these test cases. APFD for this example is visualized in Figure 1, assuming that the test cases are executed in alphabetical order, A-B-C-D. Half of the faults (1 and 2) are detected after half of the test cases (A and B) are executed. All the faults are already detected after the execution of C. APFD for this example turns out to be 56.25. It would be

75, if test cases were executed in the order B-C-A-D.

Table 1: A motivating toy example.

Test case (duration)	fault # (severity level)			
	1 (3)	2 (4)	3 (3)	4 (3)
A (1 hour)	✓			
B (2 hours)	✓	✓		
C (6 hours)			✓	✓
D (1 hour)		✓	✓	

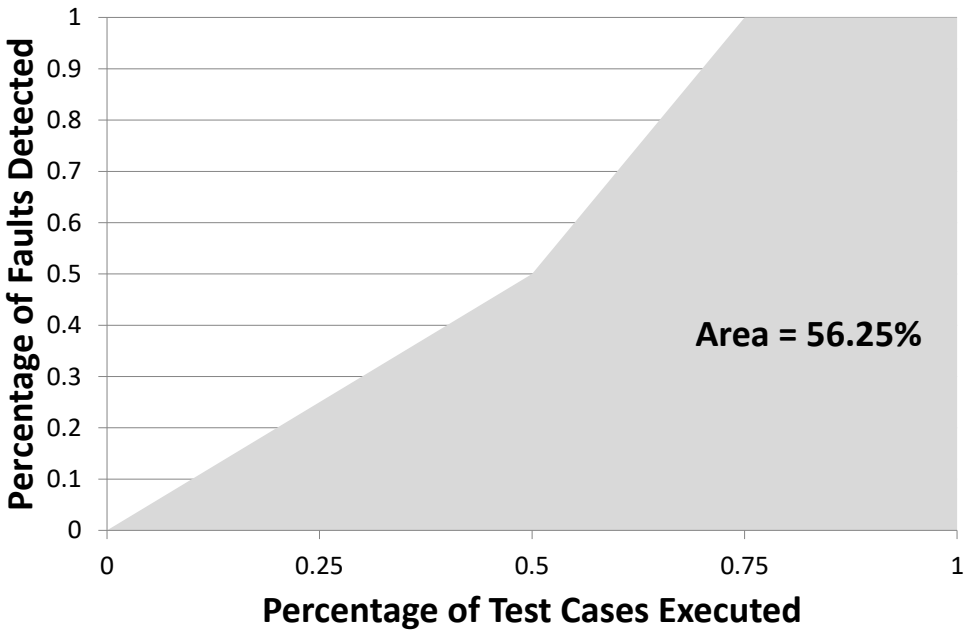

Figure 1: APFD for the toy example, when test cases are executed in the order A-B-C-D.

APFD metric has two shortcomings. First, it treats the cost of each test case to be equal. For instance, both test cases B and C detect two faults. However, it

takes 2 hours to execute B, whereas it takes 6 hours to execute C. Second, it treats the severity of each fault to be the same. For instance, the severity of one of the faults detected by B is higher than those of all the other faults. These shortcomings were previously identified and a variation of the metric, $APFD_c$ was proposed [22] as explained in the following section.

2.2 APFDc

$APFD_c$ rewards units of fault severity detected per unit test cost. The contribution of the test cases along the x-axis is weighted according to the test cost. The x-axis can simply represent the percentage of the total test time elapsed, for instance. The contribution along the y-axis is also weighted in terms of the cumulative severity of faults a test reveals.

Although it addresses the shortcomings of APFD, $APFD_c$ is still not applicable for our application context. In our case, test cost is measured in terms of the time it takes to execute a test case. Hence, the contribution of a test case can be weighted along the x-axis, in terms of the time elapsed. However, the completion times of test cases are calculated by assuming that these test cases are executed sequentially. We introduce $APFD_{cp}$ later in this Chapter to address this issue. Another issue is regarding the consideration of fault severity. $APFD_{cp}$ does not distinguish faults in terms of their severity and it only considers faults that have the highest severity, which are named as *showstopper* faults [14]. $APFD_c$ employs a weighted calculation based on the severity values of detected faults. However, fault severity is measured in ordinal scale in our context. Hence, calculating the cumulative severity of detected faults is not a proper way of quantifying a kind of weighted contribution. First of all, addition, subtraction and other arithmetic operations have no meaning in ordinal scale [24]. So, adding up severity values do not make sense mathematically. Second, this is also not meaningful in terms of the application context. For instance, a test

case can detect 4 faults with severity scale 1. Another test case can detect a single fault with a severity scale 4. If the costs of these two test cases are equal to each other, $APFD_c$ metric would be equal for two different orderings of these test cases. However, in practice, detecting a fault with severity scale 4 is more preferable than detecting dozens - if not hundreds - of faults with severity scale 1. We introduce the LAPFD metric [14] to address this issue.

2.3 LAPFD

LAPFD weights the cost of each test case just like $APFD_c$. However, we do not add up the severity values of different faults that are detected by a test case. We calculate the $APFD_c$ metric for each severity class separately. Then, we perform a lexicographic ordering based on the calculated values. Hereby, the largest $APFD_c$ value for the faults with the highest severity is considered first. If there is a tie, the metric value for the less severe faults are considered in the order of their severity. To explain through an example, consider the toy example in Table 1 with 4 test cases and 4 faults.

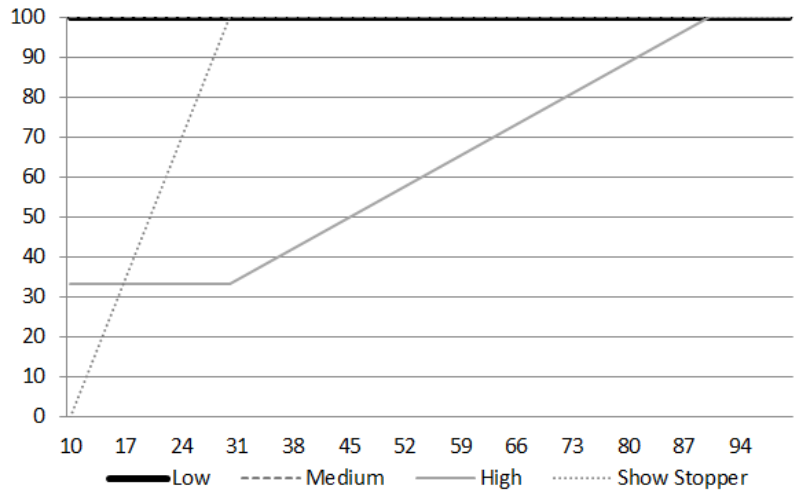

Figure 2: LAPFD for the toy example, when test cases are executed in the order alphabetical as A-B-C-D

LAPFD is visualized in Figure 2, assuming that the test cases are executed in

alphabetical order, A-B-C-D.The 4 severity levels are shown in the same graph. Low and Medium severity levels graphs started from the 100 because there are no issue of that severity was detected. The area under the graphic is accepted as 100.In this case, LAPFD values respectively starting from ShowStopper,high,medium and low. In this case LAPFD values[80,56.66,100,100]. If test cases were ordered in the greedy approach as A-D-B-C. You can see the Figure 3. It would be in the values of LAPFD values [85,68.33,100,100]. So, we can observe that the greedy approach LAPFD values of the Showstopper and high severity issues for bigger than the more alphabetical ordering.Even in this example, we can see the importance of test prioritization.

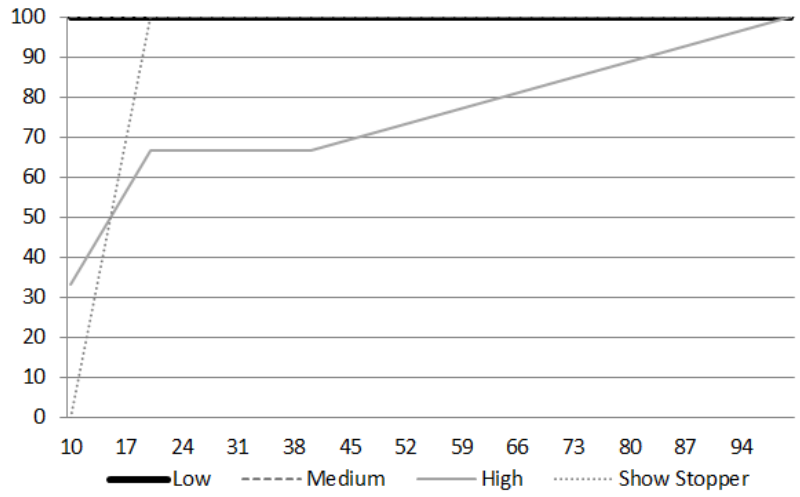

Figure 3: LAPFD for the toy example, when test cases are executed in the order greedy approach as A-D-B-C

2.4 APFDpc

We introduce a variant of the $APFD_c$ metric, namely $APFD_{pc}$. We cannot directly adopt the equation proposed for calculating $APFD_c$ [22] in a scenario with parallel test execution since that equation assumes a sequential execution of test cases. The time elapsed after the completion of a test is calculated by summing up the test execution times of all the tests preceding this test and the execution time of the test itself. In this work, we focus on prioritizing test cases that are to be executed on

multiple test stations in parallel. This changes the metric calculation and the results significantly. For instance, even if we have two test stations available, all the test cases listed for the toy example in Table 1 can be completed within 6 hours instead of 10 hours (See the first part, S1 in Figure 4). One of the test stations can be allocated for C. The other 3 tests can be completed on the other test station within 4 hours, at which point, 75% of the faults would already be detected. Hence, we calculate the actual completion time of each test case, which depends on the test station they are allocated as well as their execution order in that station.

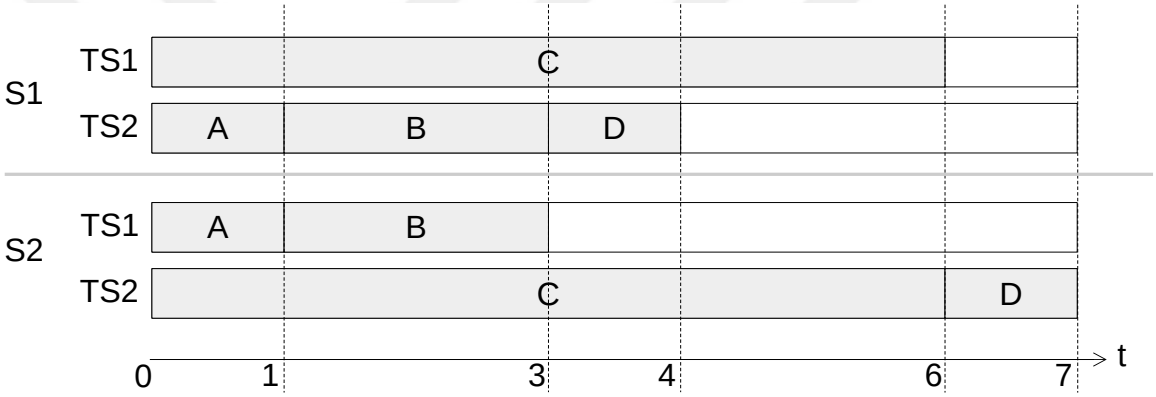

Figure 4: Two strategies, S1 and S2 for allocating test cases in the toy example in Table 1 on two test stations, TS1 and TS2.

The total elapsed time changes based on the allocation of tests among test stations. However, the x-axis is normalized in $APFD_c$ since we consider the percentage of time elapsed. Therefore, a comparison between two alternative allocations and test prioritization results would be unfair. We normalize the x-axis based on the longest total time elapsed for the pair of compared alternatives to address this issue. Consider, for instance, two strategies for the case with two available test stations for the toy example in Table 1. These strategies are depicted in Figure 4 and named as S1 and S2. S1 allocates C on the first test station. The other tests execute on the second station in alphabetical order. Test cases A, B, C and D will be completed after 1, 3, 6 and 4 hours, respectively. The total elapsed time is 6. S2 allocates A and B on the

first test station, while allocating C and D on the second one. Test cases allocated for each test station are executed in alphabetical order. As a result, test cases A, B, C and D will be completed after 1, 3, 6 and 7 hours, respectively. The total elapsed time turns out to be 7. We calculate APFD_{pc} for S1 and S2 as the areas under the corresponding curves depicted in Figure 5.

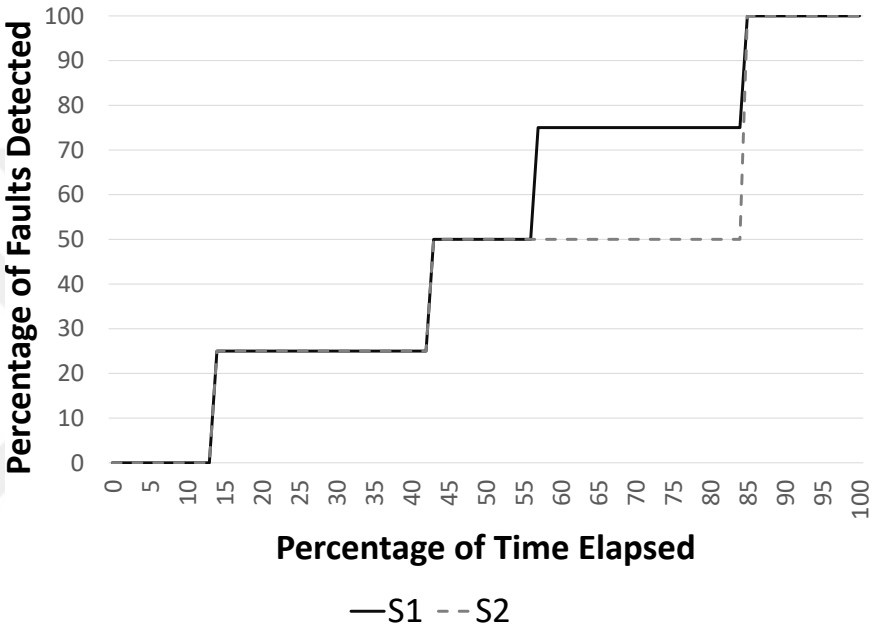

Figure 5: APFD_{pc} for the toy example, when test cases are executed in parallel with two strategies, S1 and S2.

Tests with S1 and S2 are completed in 6 and 7 hours, respectively. Hence, the x-axis in Figure 5 is normalized with respect to 7 hours of total test execution time. A is completed after the first 1 hour for both S1 and S2, when 14.29% of the total time (1/7) is elapsed. 1 of the 4 faults (25%) is detected at this time. We assume that all the faults detected by a test case are revealed when the execution of the test case is completed since there is no intermediate reporting during the execution of test cases. The second milestone for both strategies is when B is completed after 3 hours, at which point, half of the faults are detected. D is completed after 4 hours for S1 and 75% of the faults are detected. This is not the case for S2 since the execution of D waits for the completion of C. C is completed for both strategies after 6 hours,

where all the faults are detected. The remaining 1 hour does not contribute to the percentage of faults detected. S2 is still busy waiting for the completion of D although no new faults are revealed at the end. We use Algorithm 1 for calculating APFD_{pc} by summing up piece-wise areas.

$APFD_{pc} \leftarrow 0$;
$t \leftarrow 0$;
$t_p \leftarrow 0$;
while $t < 100$ **do**
  if $\exists$ *test completed at* t **then**
    $pa \leftarrow (t - t_p) \times (PFD_t + PFD_{t_p})/2$;
    $APFD_{pc} \leftarrow APFD_{pc} + pa$;
    $t_p \leftarrow t$;
    $t \leftarrow t + 1$;
  end
  $pa \leftarrow (t - t_p) \times (PFD_t + PFD_{t_p})/2$;
  $APFD_{pc} \leftarrow (APFD_{pc} + pa)/100$;
end

Algorithm 1: Calculation of APFD_{pc} for a particular test allocation and execution order.

Hereby, t and t_p represent time instances that are normalized between 0 and 100. APFD_{pc}, t and t_p are all initialized as 0. Each time whenever a test execution is completed, piece-wise area (i.e., pa) is calculated between t_p and t. PFD_t represent the percentage of faults detected at time t. The calculated value is added to APFD_{pc}. Note that there might also be more than one test case that is completed at any time instant. APFD_{pc} values for S1 and S2 are calculated as 60.72 and 58.93, respectively. Figure 5 also graphically confirms that S1 is slightly better than S2 in detecting faults earlier. Results would be even better if B and D were executed before A in S1. Hence, the ordering of test cases is important but they should be optimally allocated to test stations first. In the following, we discuss our optimization model for allocating test cases to multiple test stations.

CHAPTER III

THE OPTIMIZATION MODEL

We propose a mixed Integer Linear Programming (ILP) formulation to optimize the assignment of test cases to a given set of test stations. The proposed model has two major advantages: *i)* as it is based on an exact solution methodology, it is guaranteed to yield the optimal solution as long as the given instance is solvable in a reasonable amount of time, *ii)* even if the instance parameters are updated, the proposed solution can be used without any modification/calibration requirement, unlike certain heuristic methods. The main shortcoming of the exact methods is their computational complexity, however in our setting, we were able to solve the given instances within seconds on a commodity laptop computer.

In our problem setting, the objective is to complete the execution of all test cases in the minimum total amount of time, which is usually referred as *minimizing the makespan* in the scheduling domain [23]. We assume that the duration of each test case is deterministic and it does not depend on the assigned test station. Finally, we assume that there are no dependencies among the test cases.

$I =$ set of test cases

$J =$ set of test stations

$p_i =$ duration of test case i, $\forall i \in I$

$$x_{ij} = \begin{cases} 1, & \text{if test case } i \text{ is assigned to test station j} \\ 0, & \text{otherwise} \end{cases}$$

$$, \forall i \in I, j \in J$$

$f_j =$ completion time of cases assigned to test station j

$$, \forall j \in J$$

$z =$ makespan

$$AP : \min z, \tag{1}$$

$$s.t.$$

$$\sum_{j \in J} x_{ij} = 1, \quad \forall i \in I, \tag{2}$$

$$\sum_{i \in I} p_i x_{ij} = f_j, \quad \forall j \in J, \tag{3}$$

$$f_j \leq z, \quad \forall j \in J, \tag{4}$$

$$x_{ij} \in \{0, 1\}, \quad \forall i \in I, \forall j \in J, \tag{5}$$

$$f_j \geq 0, \quad \forall j \in J. \tag{6}$$

The objective is to minimize the makespan, the total time to complete all test cases given the number of test stations. Note that minimizing the makespan does not necessarily minimize the total time on each test station individually. That is, given the makespan is equal to z^*, the total time on each test station can be equal to z^* or it can be equal to z^* for one test station, while it can be strictly less than z^* for all others. The formulation above does not discriminate between the former and the latter cases as the makespan in both cases are equal to each other. If this is not the preferred solution and individual minimization of the total completion time is also

desired, then a lexicographical approach might be used instead; however, the computational time complexity of that approach will be significantly higher. Constraints (2) ensure that each test case is assigned to exactly one test station. Constraints (3) guarantee that the total completion time on a given test station is the summation of the execution times of all the test cases assigned to that station. Constraints (4) ensure that the makespan is greater than or equal to the total completion time of each and every test station. This set of constraints will act to *minimize the maximum* type of restriction since the problem is a minimization problem. Finally, constraints (5) and constraints (6) are the binary and non-negativity restrictions for the decision variables, respectively. The optimal solution to AP yields the best solution for our problem.

As the proposed model is a formulation based exact model, it can easily handle different/additional objective functions and additional restrictions. However, such modifications might affect the computational efficiency of the proposed approach. We implemented the model in Python, using CPLEX [25] as the solver.

We present an industrial case study in the following chapter, where we evaluate the impact of the use of our model on test case effectiveness.

CHAPTER IV

INDUSTRIAL CASE STUDY

In this chapter, we introduce the experimental setup for our case study, where experimental objects and certification test suites constitute real artifacts used at Vestel[1]. Vestel is a consumer electronics company that produces Smart TV systems for 157 different brands from 145 different countries [26] in addition to the products of its own.

We define the following five research questions to evaluate the effectiveness of our approach

- *RQ1*: Can we find critical issues earlier by test case prioritization?
- *RQ2*: How much time can be saved by employing parallel test execution and an optimal test allocation during certification tests?
- *RQ3*: How the test effectiveness is affected by employing parallel test execution and an optimal test allocation during certification tests?
- *RQ4*: What is the cost of the optimization process?

The main goal of test case prioritization is to detect the most critical issues as fast as possible even in the case of sequential test execution. *RQ1* is defined for investigating to what extent this goal can be achieved. We also expect a reduction of overall test duration in case of parallel execution of tests. *RQ2* is defined for investigating the amount of gain in time, if any. In addition to overall test duration, we are also interested in how early faults can be detected by parallel test execution

[1] http://www.vestel.com.tr

and an optimal test allocation. *RQ3* is defined for evaluating our approach from this aspect. We use the $APFD_{pc}$ metric for quantifying test effectiveness as introduced in chapter 2. The last research question, *RQ4* is about the cost of the approach. In particular, we would like to know if the approach is scalable in an industrial context. In the following, we introduce experimental objects and test suites.

4.1 Experimental Objects and Test Suites

In our case study, we used 3 Smart TV software development projects, enumerated as *Project 1*, *Project 2* and *Project 3*, as experimental objects. We observed the effectiveness of our approach for two successive test cycles, *Cycle 1* and *Cycle 2* for each of these projects. Hence, our evaluation is based on 6 test cycles in total. We applied our approach to the execution of 3 test suites for each of these 6 test cycles, as explained in the following.

We used test suites regarding the certification of 3 Smart TV applications: Netflix[2], YouTube[3], and Prime Video[4]. Table 2 lists the number of test cases and the time it takes to execute these test cases sequentially during a single test cycle.

Table 2: Certification test suites employed in the case study

Application	**# of test cases**	**total test time (hrs)**
Netflix	213	40.75
YouTube	145	72.7
Prime Video	123	28.73

We removed two of the test cases from the scope of our study. These test cases did not detect any faults under our observation. Moreover, their execution times were orders of magnitude larger than the other test cases. Hence, we considered them as

[2]https://www.netflix.com/
[3]https://www.youtube.com/
[4]https://www.primevideo.com/

outliers. For example, one of these test cases is called the *dial test*, for which the TV is kept in standby mode for 2 days and then it is checked whether the TV can be waken up with a mobile phone. As a result, the execution of this test case inevitably takes 2 days. It has to be executed anyhow but one can consider its scheduling separately.

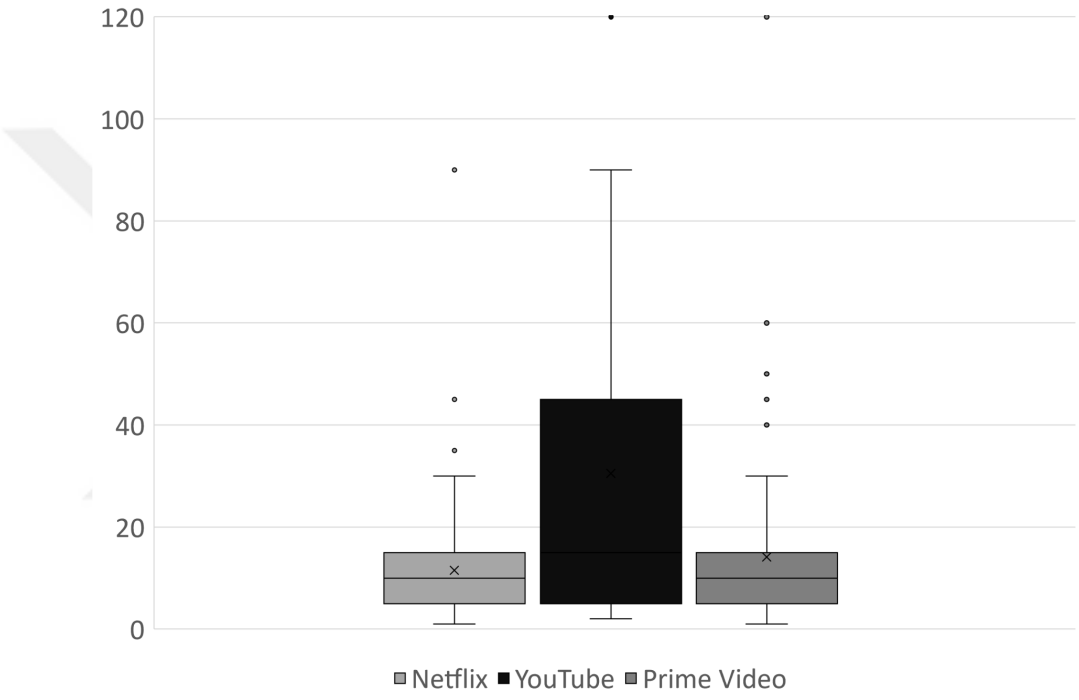

Figure 6: Test execution times (measured in minutes) for the test cases that take place in the 3 certification test suites.

Even after removing two outlier test cases from our dataset, we see in Table 2 that certification testing for the 3 applications takes 142.18 hours in total. This is approximately 2.5 days. Moreover, there still exists a high variation among the execution times of test cases. The box-plot shown in Figure 6 clearly depicts this variation. Hereby, the y-axis corresponds to the test execution time measured in minutes. In fact, there exist a few test cases that take up to 180 minutes to complete and as such, the corresponding data points are not visible in the chart. We set the

maximum value of the y-axis as 120 minutes to clearly depict the distribution among the remaining, majority of the test cases. These observations already indicate how valuable and relevant test case prioritization and parallel test execution would be for improving test efficiency in the context of certification testing in the consumer electronics domain.

We collected data regarding faults detected by each test case during each test cycle of each project. These faults are categorized according to their severity levels. As defined and used by Vestel, fault severities are measured in an ordinal scale as listed in Table 3. Hereby, faults that are associated with scale 4 must be immediately fixed as they might stop the production process; hence, the name, *showstopper*.

Table 3: Fault severity levels

Scale	**severity**
4	showstopper
3	high
2	medium
1	low

The ordinal scale just indicates the order of importance [24]. We can not infer a ratio or magnitude based on the enumeration of the scale. For instance, a fault that is associated with the *low* severity level is not exactly four times less important than another one that is categorized as a *showstopper* fault, just because these severity levels correspond to scales 4 and 1, respectively. Therefore, when evaluating the effectiveness of test prioritization in the case of sequential test execution (i.e., when there exist only one test station), we used LAPFD. We observed variance in the number of showstopper bugs detected by the compared alternatives in which case, this number directly determines the result of the comparison. Hence, we used APFD_{pc} when evaluating the effectiveness of test prioritization in the case of parallel test

execution, and we only considered *showstopper* faults.

Table 4: Number of *showstopper* faults detected for 2 test cycles of 3 projects.

Application	**Project 1**		**Project 2**		**Project 3**	
	Cycle 1	Cycle 2	Cycle 1	Cycle 2	Cycle 1	Cycle 2
Netflix	21	6	69	10	19	5
YouTube	21	7	5	3	6	3
Prime Video	11	3	3	2	3	3

We can observe from Table 4 that the number of showstopper faults detected in the second cycle is less than the number of those detected in the first cycle for all the projects. Such a result is expected since the system becomes more mature in time. However, there is no significant correlation among the fault detection ability of test cases either among two successive test cycles of a project or test cycles of two different projects.

Table 5: Correlation results of issues for Cycle-1 to Cycle-2 between same projects.

Cycle-1 to Cycle 2	**P1 to P1**	**P2 to P2**	**P3 to P3**
Netflix	0.12	0.126	-0.01
YouTube	0.045	0.135	0.353
Prime Video	0.034	-0.028	-0.024

We evaluated the correlation among various test cycles and projects in terms of the number of bugs detected by test cases. We measured correlation with Pearson correlation coefficient[5]. Results are listed in Table 5, Table 6 and Table 7. The measurement takes values between 1 and -1. A correlation value that is close to 1 or -1 indicates a significant correlation. Unfortunately, the values we obtained

[5]We used the CORREL function implemented as part of MS Excel 365.

Table 6: Correlation results of issues for Cycle-1 between different projects.

Cycle-1	P1 to P2	P2 to P3	P1 to P3
Netflix	0.024	-0.0022	-0.0624
YouTube	-0.086	0.085	0.075
Prime Video	0.0301	-0.020	0.13

Table 7: Correlations of detected faults by test cases among various projects.

Cycle-2	P1 to P2	P2 to P3	P1 to P3
Netflix	0.071	0.007	0.142
YouTube	0.22	-0.079	0.058
Prime Video	0.051	-0.091	0.019

are close to 0 and they are far from these boundaries. Our initial idea was to also incorporate information regarding the fault detection ability of test cases in previous test cycles and/or projects while prioritizing test cases. However, we ignored this information after observing statistically insignificant correlations. For this reason, we only consider test execution times in our current approach to optimize the allocation of test cases for parallel execution. The data presented in Table 4 is used only for the evaluation of alternative approaches. In the following, we discuss test stations and scenarios employed in our study.

4.2 Test Stations and Scenarios

Constructing and maintaining a test station is very expensive. This is not due to hardware costs but mainly due to the required human resources. Many of the test cases in certification test suites cannot be automated and there is a need for a tester dedicated for each test station to coordinate and observe test executions. Therefore, the number of test stations can be very limited for these certification tests in practice.

In our evaluation, we considered 3 scenarios, where the number of test stations is 1, 3 and 5.

Test execution is sequential when there is only one test station. We compared two test case prioritization alternatives for this scenario. The first one is random ordering, which reflects the current state-of-the-practice. The second one is a greedy approach [20] that was proposed before. In this approach, test cases are sorted in increasing order of their execution times. The comparison is performed with LAPFD for this scenario. We also evaluated test effectiveness with APFD_{pc}, by considering the showstopper faults only. We used these results to compare them with the other scenarios in which parallel test execution is possible.

Test cases can be executed in parallel when the number of available test stations is 3 or 5. This time, the baseline prioritization method is basically the implementation of the greedy approach [20] that was proposed before. Hereby, the test cases are sorted in increasing order of their execution times. Then, each test case is assigned to the next available station in this order. Hence, the i^{th} test case is assigned to the j^{th} station, where $j = imod(n) + 1$ and n is the total number of stations. We compared the effectiveness of the resulting allocation with respect to the optimal one that is obtained as explained in chapter 3. Test cases that are allocated to a particular station are executed in increasing order of their execution times. There is no allocation step when the number of test stations is 1. In this case, the baseline method implements the shortest-process-first scheduling. Hereby, we compared the resulting test execution order with respect to a random order (tests are executed in the order as they are listed in the given test suite) to observe the impact of ordering based on test execution times on the test effectiveness. We present and discuss the results in the following chapter.

CHAPTER V

RESULTS AND DISCUSSION

In the following, we present and discuss results regarding various scenarios. Then we summarize and interpret the overall results for answering research questions. Finally, we discuss the limitations of the approach and threats to the validity of our evaluation.

5.1 Sequential Test Execution (LAPFD)

In this section, we evaluate the test case ordering of our certification test suites using the LAPFD metric. Figure 7 depicts the corresponding chart for the *Random* prioritization of the Netflix test suite as the baseline for the project 3. Figure 8 depicts the corresponding chart for the *Greedy* approach prioritization of the Netflix test suite as the baseline for the project 3. The x-axis is normalized according to the total test duration. So, it reflects the percentage of the test duration elapsed. Similarly, y-axis is normalized according to the total number of faults detected for each severity type separately. Hence, it represents the percentage of faults detected so far in the corresponding severity scale. LAPFD is determined by calculating the area under the curve for each fault severity class separately.

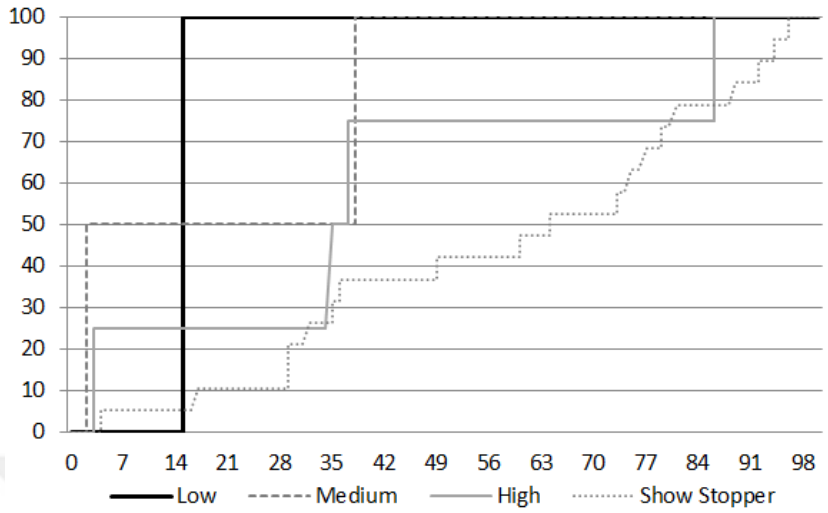

Figure 7: The calculation of the LAPFD metric for *Random* ordering regarding the execution of the *Netflix* test suite in *Project 3* in the first cycle.

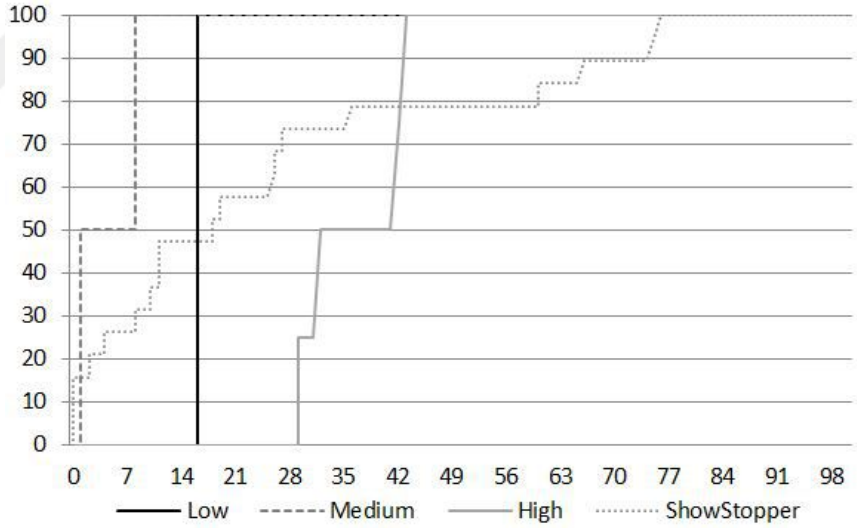

Figure 8: The calculation of the LAPFD metric for the *Greedy* approach regarding the execution of the *Netflix* test suite in *Project 3* in the first cycle.

Higher LAPFD values are emphasized with a bold font for each pair-wise comparison of *Random* and *Greedy* approaches. The *Random* approach evaluates to [66.7 ,73.09, 84.85,76.85] for the first test cycle of YouTube for *Project 1*. The *Greedy* approach evaluates to [85.29, 90.29, 94.57, 89] for the same test cycle. In our comparison, *Greedy* approach stands out due to the earlier detection of *showstopper* faults.

Table 8: LAPFD values for *Project 1*

	Cycle 1		Cycle 2	
	Random	Greedy	Random	Greedy
Netflix	**78.23**,76.16, 59.98, 100	73.59, 75.79, 80.73, 100	62.96,64.04,69.34,72.69	**72.16**,78.81,66.29,53.06
YouTube	66.7 ,73.09, 84.85,76.85	**85.29**, 90.29, 94.57, 89	67.23,66.66,60.06,58.55	**85.82**,71.88,75.45,87.19
Prime Video	70.49, 76.48, 64.72, 100	**78.60**, 65.79, 81.51, 100	90.7,69.66,60.39,100	**95.5**,94.54,93.67,100

Table 9: LAPFD values for *Project 2*

	Cycle 1		Cycle 2	
	Random	Greedy	Random	Greedy
Netflix	39.69,76.06,73.24,32.96	**63.12**,84.92,87.96,78.32	62.08,64.69,40.13,83.42	**82.04**,74.72,47.54,94.06
YouTube	57.16,63.52,29.76,23.84	**73.46**,82.43,86.29,73.45	37.98,93.70,61.71,100	**82.08**,93.04,93.79,100
Prime Video	81.84,83.67,61.28,51.82	**93.97**,94.54,71.37,72.94	81.23,99.79,58.78,100	**94.25**,99.79,72.07,100

LAPFD results for the two test cycles of *Project 2* are listed in Table 9. The *Greedy* approach leads to higher LAPFD values for all the cases this time. We can conclude that if *Greedy* approach was used for prioritizing the test cases, *showstopper* issues could be found earlier than the state-of-the-practice.

Table 10: LAPFD values for *Project 3*

	Cycle 1		Cycle 2	
	Random	Greedy	Random	Greedy
Netflix	41.09,59.43,79.57,84.53	**74.65**,63.31,95.13,83.43	**67.10**,86.58,97.96,27.6	62.59,92.05,93.04,33.43
YouTube	55.28,52.16,88.42,39.43	**85.27**,84.69,72.03,88.27	84.08,95.11,38.06,100	**94.18**,96.61,93.27,100
Prime Video	**58.73**,86.79,67.79,82.77	32.72,89.44,85.16,93.67	85.85,100,33.84,75.49	**89.78**,100,63.37,99.50

Once and for all, we can see the LAPFD values for the two test cycles of *Project 3* in Table 13. Although the *Greedy* approach leads to better results for the majority of the cases for this project, there are two cases where, the random ordering performs better. One of them is the second test cycle of the *Netflix* test suite. The other one

is the first test cycle of the *Prime Video* test suite. We observed that the number of showstopper bugs detected by the compared alternatives always varies. As a result, this this number directly determines the result of the comparison. Hence, we also calculated $APFD_{pc}$ values by considering *showstopper* faults only for the sequential test execution scenario as explained in the following. These values are used later for comparison with respect to those obtained for the parallel test execution scenarios.

5.2 Sequential Test Execution ($APFD_{pc}$)

We compared two test case prioritization alternatives just like our evaluation with LAPFD as presented in the previous section. The first one is *Random* execution, where tests are executed in the order as they are listed in the given test suite. The second one is the *Greedy* approach, where test cases are sorted in increasing order with respect to their execution times. The overall test duration (See Table 2) is the same for both approaches in this scenario. $APFD_{pc}$ values[1] for the two test cycles of *Project 1* are listed in Table 11.

Table 11: $APFD_{pc}$ values for *Project 1* with only a single station, where all the test cases are executed sequentially.

	Cycle 1		Cycle 2	
	Random	Greedy	Random	Greedy
Netflix	**78.23**	73.59	62.96	**72.16**
YouTube	66.7	**85.29**	67.23	**85.82**
Prime Video	70.49	**78.60**	90.7	**95.5**

Higher $APFD_{pc}$ values are emphasized with a bold font for each pair-wise comparison of *Random* and *Greedy* approaches. We can see higher values for almost all the applications and all the test cycles when the *Greedy* approach is applied. The

[1]We could also use the $APFD_c$ metric for this scenario that involves sequential test execution; however, we adopted $APFD_{pc}$ for consistency.

first cycle of the execution of the *Netflix* test suite is the only exception to this observation. Recall that we assume no prior information regarding the faults detected by test cases. Therefore, the *Greedy* approach suggests a prioritization based on test execution times only. As a result, an optimal ordering based on test execution times does not necessarily lead to the optimal (or better) $APFD_{pc}$ value. There might be a test case that detects many faults in a test cycle although its execution is deferred since it is costly, taking too much time to complete.

Table 12: $APFD_{pc}$ values for *Project 2* with only a single station, where all the test cases are executed sequentially.

	Cycle 1		Cycle 2	
	Random	Greedy	Random	Greedy
Netflix	39.69	**63.12**	62.08	**82.04**
YouTube	57.16	**73.46**	37.98	**82.08**
Prime Video	81.84	**93.97**	81.23	**94.25**

Results for the two test cycles of *Project 2* are listed in Table 12. The *Greedy* approach leads to higher $APFD_{pc}$ values for all the cases this time. Finally, we can see the results for the two test cycles of *Project 3* in Table 13. Although the *Greedy* approach leads to better results for the majority of the cases for this project, there are two cases where, the *Random* approach performs better. One of them is the second test cycle of the *Netflix* test suite. The other one is the first test cycle of the *Prime Video* test suite.

Overall, we observe that ordering of tests based on test execution times improves $APFD_{pc}$ in 15 out of 18 test cycles of 3 projects for 3 certification test suites. We do not exploit parallelism, we do not perform any optimization and we cannot reduce the overall test duration in this basic scenario, where tests are executed sequentially. In the following, we present the results for the scenario, where test cases are executed

Table 13: $APFD_{pc}$ values for *Project 3* with only a single station, where all the test cases are executed sequentially.

	Cycle 1		Cycle 2	
	Random	Greedy	Random	Greedy
Netflix	41.09	**74.65**	**67.10**	62.59
YouTube	55.28	**85.27**	84.08	**94.18**
Prime Video	**58.73**	32.72	85.85	**89.78**

in parallel on 3 test stations.

5.3 Parallel Test Execution with 3 Stations

In our second scenario of the parallel test execution, we exploit parallelism by using 3 test stations. We compare two approaches. One of them employs the optimal allocation of test cases on test stations to minimize the overall test duration as explained in chapter 3. The other one employs a greedy approach for test allocation as explained in section 4.2. Test cases that are allocated to a particular test station are executed in increasing order of their execution times in both approaches. We perform comparisons with respect to two aspects: *i)* Overall test duration, and *ii)* $APFD_{pc}$ metric value.

Table 14: The overall test duration, when test cases in 3 test suites are executed in parallel on 3 test stations and when they are allocated with the *Greedy* and *Optimum* approaches.

	Test Duration (minutes)	
	Greedy	Optimum
Netflix	858	**817**
YouTube	1,624	**1,540**
Prime Video	684	**575**

Table 14 lists the overall test duration in minutes for 3 test suites when test allocation is performed with *Greedy* and *Optimum* approaches. Recall that the total duration for executing *Netflix*, *YouTube* and *Prime Video* test suites sequentially are 2,445, 4,362 and 1,724 minutes, respectively (See Table 2). Hence, parallel test execution on just 3 test stations already leads to a significant reduction in test duration. Table 14 also shows that *Optimum* test allocation gains at least 40 minutes of additional time when compared with the *Greedy* approach. *Netflix*, *YouTube* and *Prime Video* test suites are completed 41, 84 and 109 minutes earlier, respectively.

Although the optimization is performed for minimizing the test execution time only, it also has a positive effect on the time of fault detection. $APFD_{pc}$ values for the two test cycles of *Project 1* are listed in Table 15. We can see that $APFD_{pc}$ values are improved for all the test suites when the optimal allocation is employed. The minimum and maximum amount of improvements are 3.06 (*Netflix*, Cycle 2) and 12.42 (*YouTube*, Cycle 2), respectively.

Table 15: $APFD_{pc}$ values for *Project 1* when 3 test stations are used for executing test suites.

	Cycle 1		Cycle 2	
	Greedy	Optimum	Greedy	Optimum
Netflix	71.71	**75.82**	63.27	**66.33**
YouTube	69.00	**77.71**	62.89	**75.31**
Prime Video	66.02	**71.98**	90.96	**97.29**

Results regarding *Project 2* are listed in Table 16. Like the results we have for *Project 1*, $APFD_{pc}$ values are improved for all the test suites when the optimal allocation is employed. However, the variation in the amount of improvement is higher. The minimum and maximum amount of improvements are 4.73 (*Prime video*, Cycle 1) and 26.72 (*YouTube*, Cycle 1), respectively.

Table 16: $APFD_{pc}$ values for *Project 2* when 3 test stations are used for executing test suites.

	Cycle-1		Cycle-2	
	Greedy	Optimum	Greedy	Optimum
Netflix	43.12	**63.36**	65.19	**78.57**
YouTube	60.97	**87.69**	54.70	**68.14**
Prime Video	78.72	**83.45**	76.42	**86.11**

We can see in Figure 9, the calculation of the $APFD_{pc}$ metric for *Greedy* and *Optimum* approaches regarding the execution of the *Youtube* test suite in parallel on 3 test stations for *Project 2* in the first test cycle. The optimal test allocation leads to the maximum amount of improvement (26.72) for *Project 2* for this case. $APFD_{pc}$ values are calculated as 60.97 and 87.69, respectively.

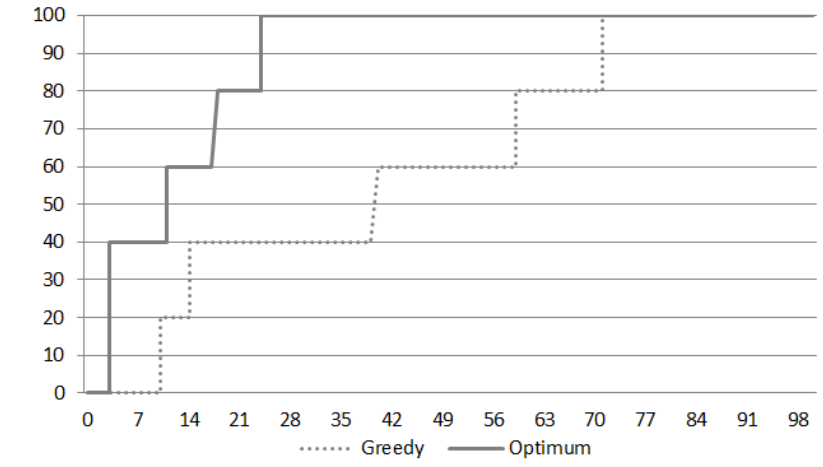

Figure 9: The calculation of the $APFD_{pc}$ metric for *Greedy* and *Optimum* approaches regarding the execution of the *Youtube* test suite in parallel on 3 test stations for *Project 2* in the first test cycle.

Finally, we can see the results for the two test cycles of *Project 3* in Table 17. $APFD_{pc}$ values are also improved in general when the optimal allocation is employed.

The variation in the amount of improvement is high. The minimum and maximum amount of improvements are 3.31 (*YouTube*, Cycle 2) and 27.32 (*Netflix*, Cycle 1), respectively. There is also an exceptional case (*Netflix*, Cycle 2), where the $APFD_{pc}$ value is decreased by 5.63 after test allocation is altered for minimizing test duration. We investigated the reason for this result and found out that there are just 5 faults detected in the corresponding test cycle and all these faults are detected in just a couple of test cases, which were scheduled earlier by the *Greedy* approach. As a result, although the tests are completed 40 minutes earlier with optimal test allocation, the time of detection of faults is delayed for this particular case.

Table 17: $APFD_{pc}$ values for *Project 3* when 3 stations are used for executing test suites.

	Cycle 1		Cycle 2	
	Greedy	Optimum	Greedy	Optimum
Netflix	45.65	**72.97**	**69.32**	63.69
YouTube	60.8	**73.02**	78.56	**81.87**
Prime Video	51.55	**65.83**	88.59	**93.71**

Overall, we observed that parallel test execution on just 3 test stations already leads to a significant reduction in test duration. Optimal test allocation further reduces this duration by up to 109 minutes, when compared with the *Greedy* approach. Although the optimization is performed for minimizing the test execution time only, it also improves $APFD_{pc}$ in 17 out of 18 test cycles of 3 projects for 3 certification test suites. The amount of improvement ranges between 3.06 and 27.32. There is only one case, where the $APFD_{pc}$ is decreased by 5.63. In this case, there exists a low number of faults that are altogether detected by a couple of test cases. In the following, we present the results for the scenario, where tests are executed in parallel on 5 test stations.

5.4 Parallel Test Execution with 5 Stations

In our third scenario of the parallel test execution,, we exploit parallelism by using 5 test stations. We again compare two approaches, *Greedy* and *Optimal*, with respect to overall test duration and $APFD_{pc}$ metric value. Table 18 lists the overall test duration in minutes for 3 test suites. We can see that the overall duration is further reduced. The difference in reduction between the *Greedy* and *Optimum* approaches is higher in general, when compared with the second scenario with 3 test stations. *Netflix*, *YouTube* and *Prime Video* test suites are completed 25, 264 and 119 minutes earlier with optimal test case allocation, respectively.

Table 18: The overall test duration, when test cases in 3 test suites are executed in parallel on 5 test stations and when they are allocated with the *Greedy* and *Optimum* approaches.

	Test Duration (minutes)	
	Greedy	Optimum
Netflix	515	**490**
YouTube	1137	**873**
Prime Video	465	**346**

We also observe an improvement in the time of fault detection. $APFD_{pc}$ values for the two test cycles of *Project 1* are listed in Table 19. We can see that $APFD_{pc}$ values are improved for all the test suites when the optimal allocation is employed. The minimum and maximum amount of improvements are 3.43 (*Netflix*, Cycle 2) and 17.98 (*YouTube*, Cycle 2), respectively. The corresponding cases are the same as those associated with the minimum and maximum amount of improvements for the second scenario, where 3 test stations are used.

Results regarding *Project 2* are listed in Table 20. Like the results we have for

Table 19: $APFD_{pc}$ values for *Project 1* when 5 test stations are used for executing test suites.

	Cycle 1		Cycle 2	
	Greedy	Optimum	Greedy	Optimum
Netflix	75.28	**79.89**	63.83	**67.26**
YouTube	75.53	**83.24**	64.05	**82.03**
Prime Video	76.46	**85.01**	91.7	**98.85**

Project 1, $APFD_{pc}$ values are improved for all the test suites when the optimal allocation is employed. However, the amount of improvement is higher. The minimum and maximum amount of improvements are 9.39 (*Prime video*, Cycle 1) and 20.81 (*YouTube*, Cycle 2), respectively.

Table 20: $APFD_{pc}$ values for *Project 2* when 5 test stations are used for executing test suites.

	Cycle 1		Cycle 2	
	Greedy	Optimum	Greedy	Optimum
Netflix	42.25	**61.92**	66.41	**77.41**
YouTube	61.49	**79.25**	53.25	**74.06**
Prime Video	82.50	**91.89**	79.03	**88.92**

Finally, we can see the results for the two test cycles of *Project 3* in Table 21. $APFD_{pc}$ values are also improved in general when the optimal allocation is employed. The variation in the amount of improvement is very high when compared with the previous projects. The minimum and maximum amount of improvements are 3.51 (*Prime Video*, Cycle 2) and 34.67 (*Netflix*, Cycle 1), respectively. The second test cycle of the *Netflix* test suite for *Project 3* is subject to an exception as it was the case for the second scenario with 3 test stations. The $APFD_{pc}$ value is decreased

when tests are allocated for minimizing the overall test execution time for this case. However, the difference between the $APFD_{pc}$ values is below 1%.

Table 21: $APFD_{pc}$ values for *Project 3* when 5 test stations are used for executing test suites.

	Cycle 1		Cycle 2	
	Greedy	Optimum	Greedy	Optimum
Netflix	42.41	**77.08**	**69.47**	68.54
YouTube	54.14	**83.80**	64.74	**75.27**
Prime Video	56.98	**76.12**	88.74	**92.25**

We can see in Figure 10, the calculation of the $APFD_{pc}$ metric for *Greedy* and *Optimum* approaches regarding the execution of the *Netflix* test suite in parallel on 5 test stations for *Project 3* in the first test cycle. $APFD_{pc}$ values are calculated as 42.41 and 77.08, respectively.

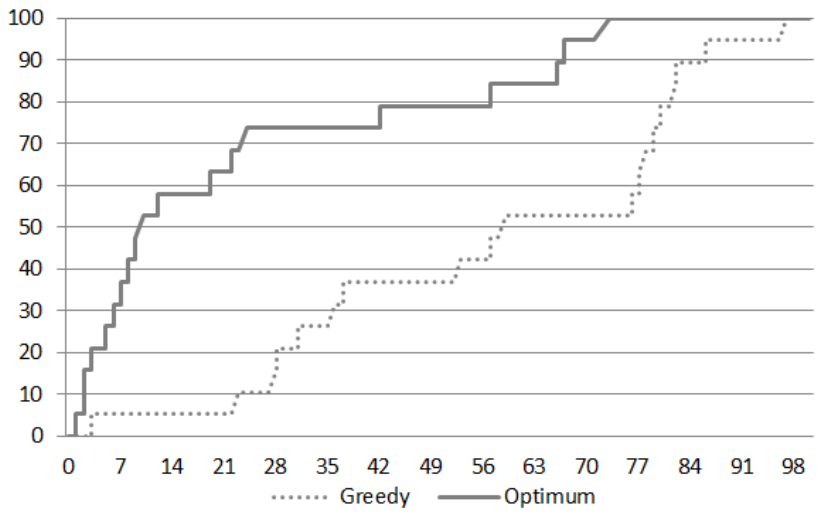

Figure 10: The calculation of the $APFD_{pc}$ metric for *Greedy* and *Optimum* approaches regarding the execution of the *Netflix* test suite in parallel on 5 test stations for *Project 3* in the first test cycle.

Overall, we observed that parallel test execution on 5 test stations reduces the test

duration by 32.6 hours when optimal test allocation is used. Sequential execution of the *Netflix* test suite takes 40.75 hours, whereas the execution of test cases that are optimally allocated to 5 parallel test stations complete within 8.2 hours (490 minutes). The difference in reduction between the *Greedy* and *Optimum* approaches is also increased when compared with the second scenario with 3 test stations. The difference in test completion times are measured as 25, 264 and 119 minutes for *Netflix*, *YouTube*, and *Prime Video* test suites, respectively. APFD_{pc} values are also improved in 17 out of 18 test cycles of 3 projects for 3 certification test suites. The amount of improvement ranges between 3.43 and 34.67. There is only one case, where the APFD_{pc} is decreased by 0.93. In the following, we further discuss the obtained results and answer the research questions.

5.5 Discussion

For the first test experiment scenario, we observed that the greedy approach ordering of test cases consistently outperformed random ordering and unit time for three experimental objects. We observed regarding *RQ1* that even the greedy approach can improve the effectiveness of the process in terms of detecting critical faults earlier. APFD_c values for faults with the highest severity levels turned out to be always larger. These values were compromised to some extent in some cases for faults associated with lower severity. However, these values are considered to be low-order terms by LAPFD and in our application context.

We can conclude regarding *RQ2* that significant time saving is possible by the parallel execution of certification tests. Figure 11 summarizes the results regarding test execution times. Hereby, the total completion time is depicted in minutes. The first bar for each test suite represents the time required for sequential execution. The second and third bars represent parallel execution, when the number of test stations is 3 and 5, respectively.

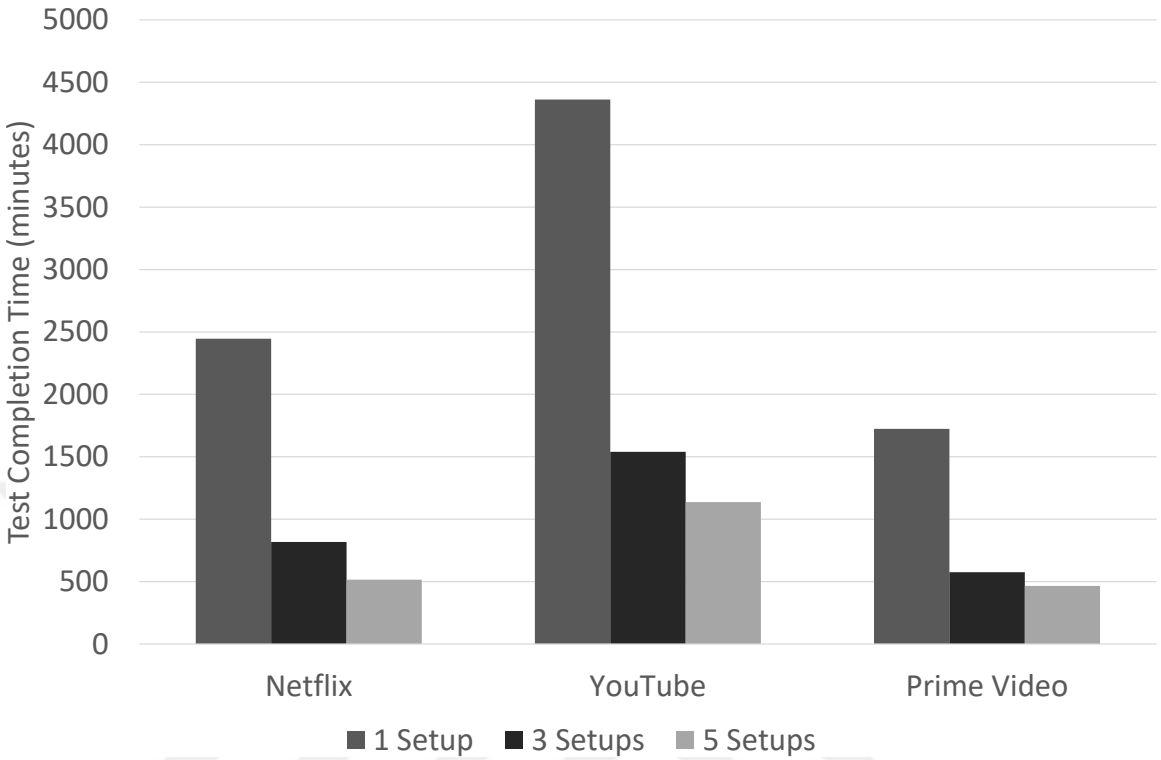

Figure 11: The overall test execution times for the 3 test suites.

The execution time can be reduced by more than a half already by employing 3 test stations. The amount of reduction in execution time is not in the orders of magnitude as we increase the number of test stations. Nevertheless, we continue to have significant time savings. The overall test duration for the *YouTube* test suite, for instance, is reduced by 11 hours, when the number of test stations is increased by two and tests are allocated on stations optimally. The impact of test allocation increases as the number of test stations increases. Hence, optimal allocation becomes more important. Figure 12 depicts the overall test execution times achieved with the *Greedy* and *Optimal* test allocation approaches for the 3 test suites when 5 test stations are used for parallel execution of test cases. The difference between the two is the highest (4.4 hours) for the *YouTube* test suite.

The optimization is performed for minimizing the test execution time only. Hence, the effectiveness of the solution in terms of the time of fault detection is questionable

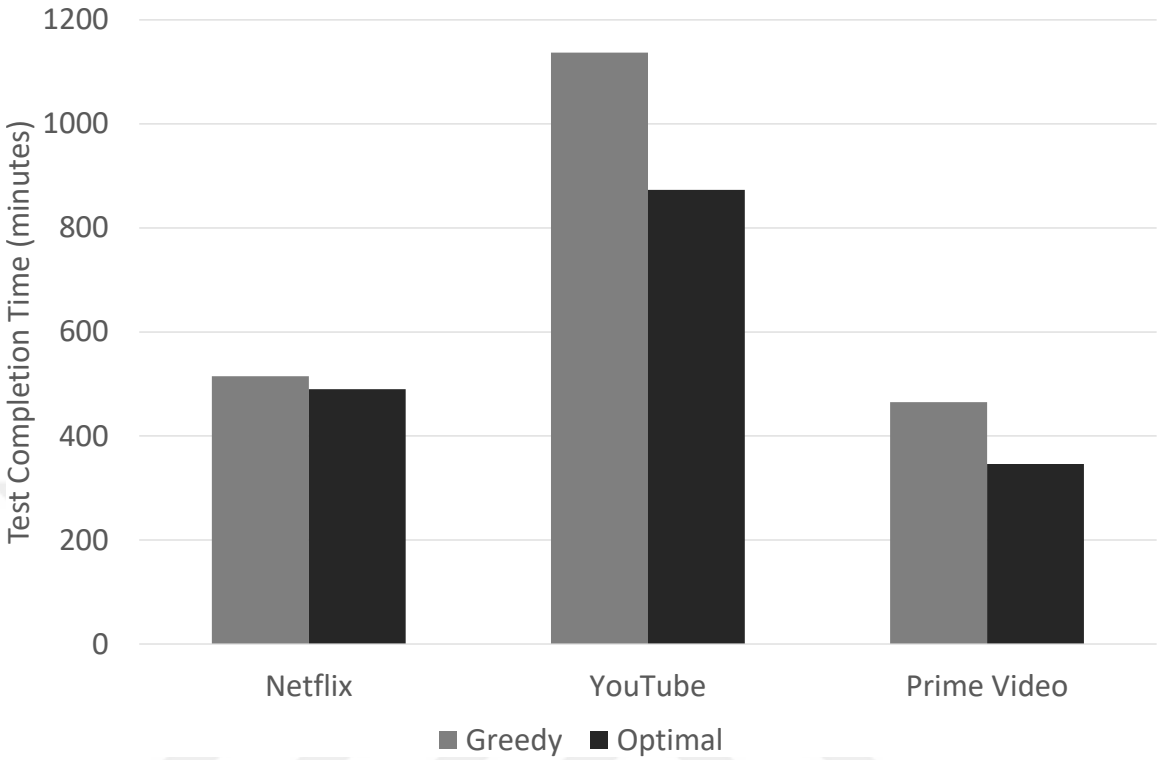

Figure 12: The overall test execution times achieved with the *Greedy* and *Optimal* test allocation approaches for the 3 test suites when 5 test stations are used for parallel execution of test cases.

(*RQ3*). However, we empirically observed that APFD_{pc} values are also improved. We measured this value in two scenarios (when the number of test stations is 3 and 5), for test cycles of each of the 3 projects, and for each of the 3 test suites. APFD_{pc} was improved in 34 out of 36 cases. In fact, the two exceptional cases correspond to the same test cycle (i.e., the second test cycle of the *Netflix* test suite executed on *Project 3*) for two scenarios, where 3 and 5 test stations are used. APFD_{pc} values are decreased by 5.63 and 0.93 for these scenarios, respectively, after test allocation is altered for minimizing test duration. The reason is the existence of a low number of faults that are detected by a low number of test cases, which are scheduled earlier by the *Greedy* approach. The amount of improvement for the remaining 34 cases ranges between 3.06 and 34.67.

The last concern is the cost of the approach (*RQ4*). The optimization process can

be completed within seconds on a commodity laptop computer. Therefore, the cost of the optimization process is negligible. In fact, the optimization problem that we are solving is an NP-complete problem [23]. However, in our context, the number of test cases is in the order of a couple of hundreds. There are also no dependencies among the test cases. These properties of the problem domain make it feasible to achieve the optimal solution. The number of test stations is also limited due to resource constraints (i.e., limitation of human resources). Therefore, it is preferable to find and use the optimal solution rather than employing a greedy approach. The cost is negligible with respect to the obtained benefits.

5.6 Limitations and Threats to Validity

Our evaluation is subject to *external validity threats* [27] since our study focuses on certification tests in the consumer electronics domain and all the experimental objects are from this domain. Hence, our results cannot be generalized for other settings, where our assumptions do not hold. For instance, there might be dependencies among the test cases, which impose ordering constraints. The characteristics of test suites can also be different for certification tests performed in other domains.

There exist *internal validity threats* [27] regarding the calculation of the APFD_{pc}. We assumed that all the faults detected by a test case are revealed at the time of completion of that test case. In fact, some of the faults can be detected earlier. For instance, a fault can be revealed in the first couple of seconds of a test that takes several hours to complete. However, we cannot know the exact time of fault detection since there is no reporting for intermediate results.

The calculation of the APFD_{pc} metric is subject to *construct and conclusion validity threats* [27] since it depends on the fault detection ability of test cases. Our approach performs optimization based on test execution times only and there is no guarantee that it will also lead to early detection of faults. However, it leads to a

clear advantage in terms of total execution time when tests are executed in parallel. In addition, we observed that it also leads to an improved APFD_{pc} in the majority of cases, where 3 scenarios, 3 projects, 2 test cycles and 3 applications are involved.

In our evaluations, we distributed test executions on 3 and 5 test stations to exploit parallelism. We did not perform evaluations for a larger number of test stations since such scenarios are not feasible in practice. In fact, we were informed by our industrial partners that the allocation of 5 test stations is already hard to achieve. Many of the test cases in certification test suites cannot be automated. As such, there is a need for a tester dedicated to each test station. There are certification tests that need to be performed for many TV projects at the same time. There are simply not enough human resources to support more than 5 test stations for each project.

CHAPTER VI

RELATED WORK

Test case prioritization has been mainly studied and applied in the context of regression testing. In fact, it was explicitly defined as a *regression technique* in previous systematic mapping studies [28] as well in recent literature surveys [6]. The proposed approaches are further categorized according to alternative criteria used for prioritization. These criteria are based on various measures regarding test cases such as their code coverage [15] or importance of requirements verified by these [16]. Some approaches [17] use historical data regarding fault detection effectiveness of test cases. The desired outcome of prioritization is mainly to detect faults as early as possible [6]. In this study, we consider test execution time as the main criterion for test case prioritization. We aim at minimizing the total duration and detected high-level issues as early as possible. We could not exploit historical data in our application context since fault detection abilities of test cases were not correlated among subsequent test cycles or across various projects. Therefore, we assume that it is not possible to predict the fault detection ability of a test case accurately. However, we expect that minimizing the overall test execution time would also lead to the detection of faults earlier. Our results support this expectation for all but one exceptional test instance.

There have been many test case prioritization approaches proposed so far. The majority of them adopt greedy, meta-heuristic [29–31], and evolutionary search algorithms [32]. Some of these techniques consider multiple objectives [33] and some of them adopt lexicographical ordering [34]. However, the underlying heuristics mainly rely on coverage information [34] rather than test cost [35] together with fault severities. These factors are also not reflected to the metrics used for evaluating test

effectiveness. We introduced LAPFD [14] to fill this gap.

The use of ILP was proposed before; however, it was used either to minimize the test suite [8] or to find an optimal ordering among the test cases [36] to be executed on a single test station. There exists a study [20] that considers a scenario, where parallel execution of test cases on multiple test stations is possible. However, a greedy algorithm is used for the allocation of test cases, where test cases are ordered sequentially first. Then, each test case is assigned to the next available station in this order. Assuming that the cost of each test case is equal, the i^{th} test case is assigned to the j^{th} test station, where $j = imod(n) + 1$ and n is the total number of test stations. We consider this approach as a baseline; however, we take varying test costs into account. A greedy next fit algorithm is applied after test cases are sorted in ascending order with respect to their execution times. In our approach, we employ ILP to minimize the overall test execution time.

Only a few industrial case studies are published in the literature [37–39] despite many studies published on test case prioritization. Moreover, these studies mainly focus on regression testing. The context and application domains of existing studies are also different than ours. To the best of our knowledge, there has been no industrial case study focusing on certification testing in the consumer electronics domain. Moreover, the availability of multiple test stations for testing has been ignored while addressing the test case prioritization problem.

CHAPTER VII

CONCLUSION

In this thesis, we focus on the test case prioritization problem in the context of certification testing. We proposed two variants of existing metrics for evaluating the effectiveness of test prioritization. The first one, LAPFD (Lexicographic ordering of Average of the Percentage of Faults Detected), calculates the average of the percentage of faults detected over time, during the execution of test cases. This calculation is performed separately per severity class. Then, a lexicographic ordering is performed based on these classes. The second metric, APFD_{pc}, focuses on faults with the highest severity class only. However, it takes parallel execution of test cases into account while calculating the completion times of test cases.

We also proposed an Integer Linear Programming (ILP) model for prioritizing test cases that can be executed on multiple test stations in parallel. The model can be used for minimizing the overall test execution time, i.e., the makespan of the overall certification testing process.,

We conducted an industrial case study in the consumer electronics domain. We used certification test suites of 3 Smart TV applications applied on 3 TV software projects as real experimental objects. We repeated our measurements for two test cycles for each project and as such, we collected measurements from 18 test sessions in total. We evaluated 3 scenarios for each of these sessions, where the number of available test stations are 1, 3 and 5.

We compared a random ordering of test cases as the state-of-the-practice with respect to a greedy approach as the baseline for the sequential test execution. These alternative orderings are evaluated by using both LAPFD and APFD_{pc} metrics. We

observe that greedy approach ordering of test cases consistently outperformed random ordering. We also observe that there is a large room for improvement regarding the effectiveness of test case prioritization. We performed another evaluation by using the APFD_{pc} metric for scenarios where the number of test stations are 3 and 5. We compared the efficiency and effectiveness of our approach with respect to a greedy approach as the baseline. Results showed that the overall test execution time can be reduced by up to 16% even when only 3 test stations are available. The amount of reduction is up to 4.4 hours when 5 test stations are used. Test effectiveness was improved in 34 out of 36 test cycles, where the amount of improvement ranges between 3.06% and 34.67%. Test effectiveness was decreased in only two test cycles by 5.63% and 0.93%. These were caused by the existence of a low number of faults that are detected by a low number of test cases.

APPENDIX A

LAPFD RESULTS

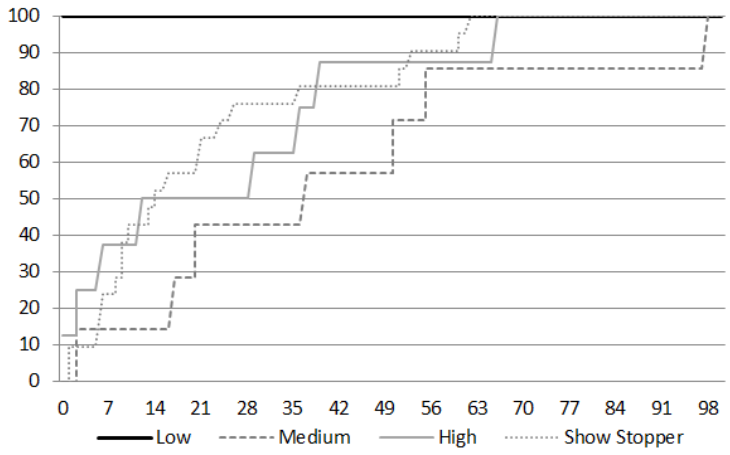

Figure 13: The calculation of the LAPFD metric for Random ordering regarding the execution of the *Netflix* test suite in *Project 1* in the first cycle.

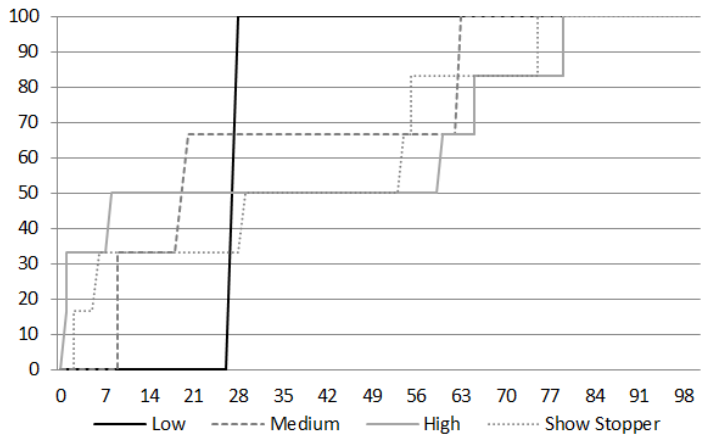

Figure 14: The calculation of the LAPFD metric for Random ordering regarding the execution of the *Netflix* test suite in *Project 1* in the second cycle.

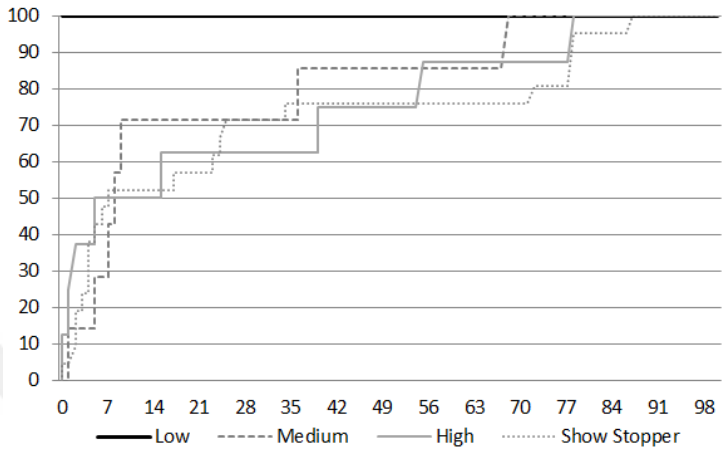

Figure 15: The calculation of the LAPFD metric for *Greedy* approaches regarding the execution of the *Netflix* test suite in *Project 1* in the first cycle.

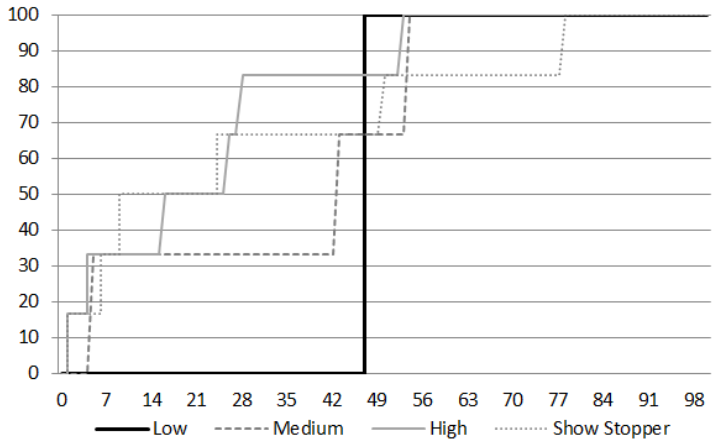

Figure 16: The calculation of the LAPFD metric for *Greedy* approaches regarding the execution of the *Netflix* test suite in *Project 1* in the second cycle.

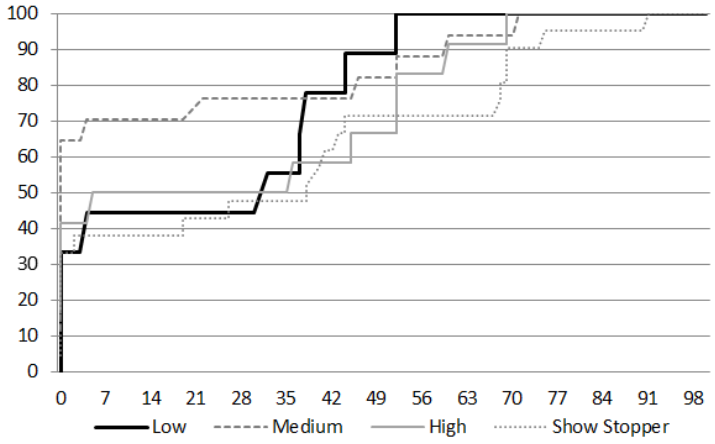

Figure 17: The calculation of the LAPFD metric for Random ordering regarding the execution of the *YouTube* test suite in *Project 1* in the first cycle.

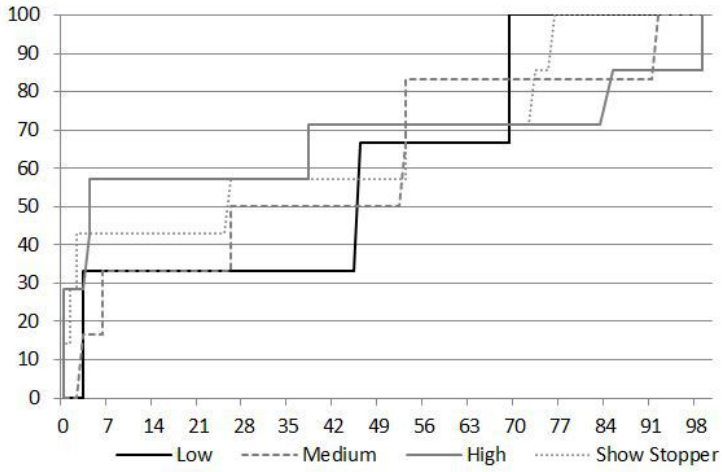

Figure 18: The calculation of the LAPFD metric for Random ordering regarding the execution of the *YouTube* test suite in *Project 1* in the second cycle.

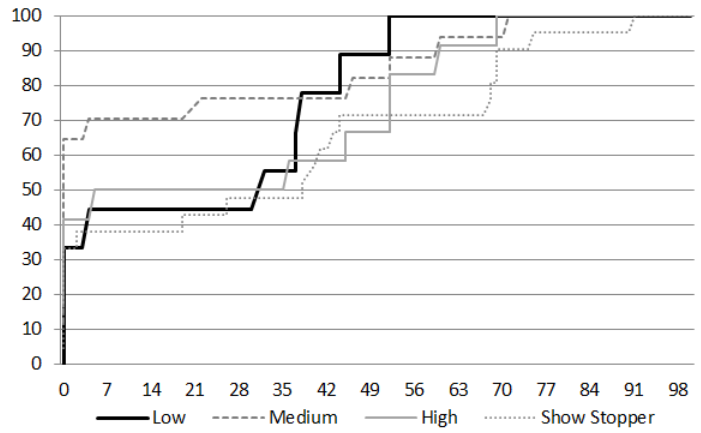

Figure 19: The calculation of the LAPFD metric for *Greedy* approaches regarding the execution of the *YouTube* test suite in *Project 1* in the first cycle.

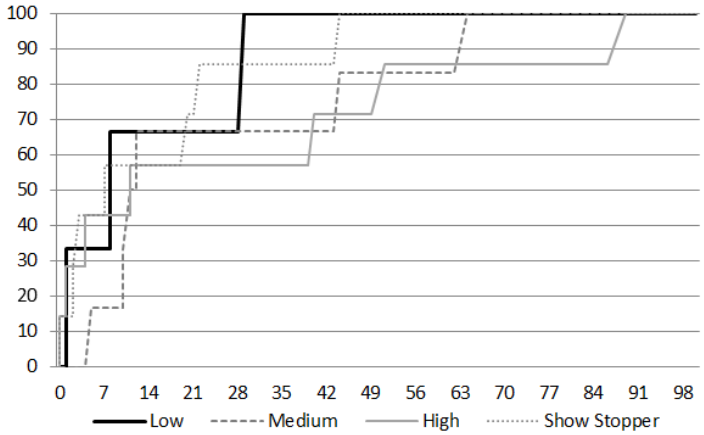

Figure 20: The calculation of the LAPFD metric for *Greedy* approaches regarding the execution of the *YouTube* test suite in *Project 1* in the second cycle.

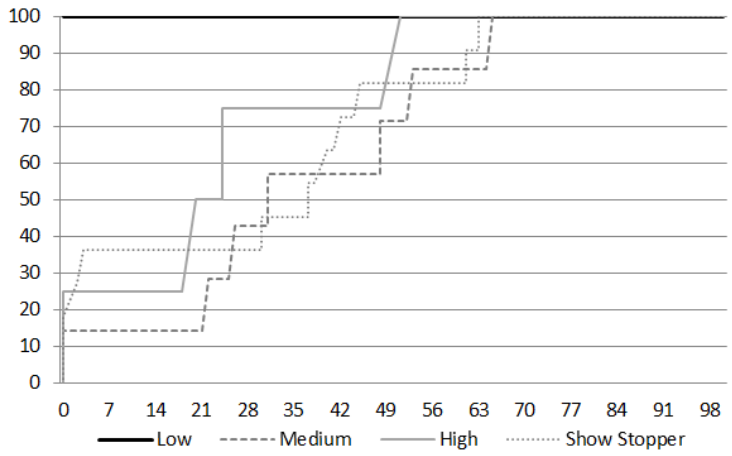

Figure 21: The calculation of the LAPFD metric for Random ordering regarding the execution of the *PrimeVideo* test suite in *Project 1* in the first cycle.

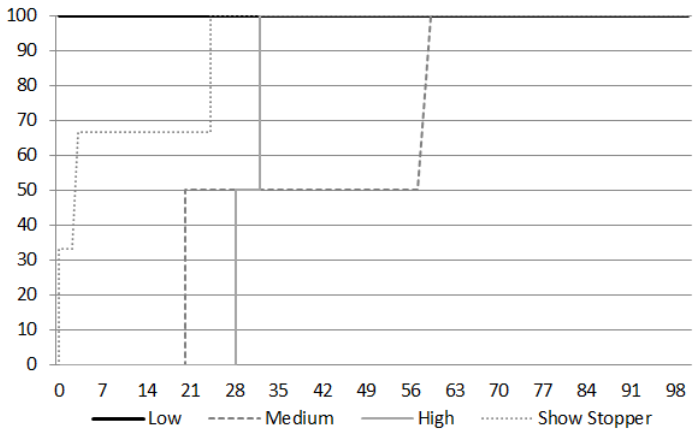

Figure 22: The calculation of the LAPFD metric for Random ordering regarding the execution of the *PrimeVideo* test suite in *Project 1* in the second cycle.

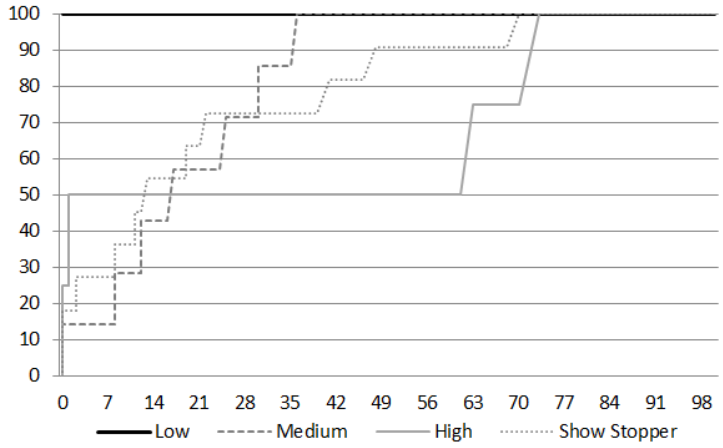

Figure 23: The calculation of the LAPFD metric for *Greedy* approaches regarding the execution of the *PrimeVideo* test suite in *Project 1* in the first cycle.

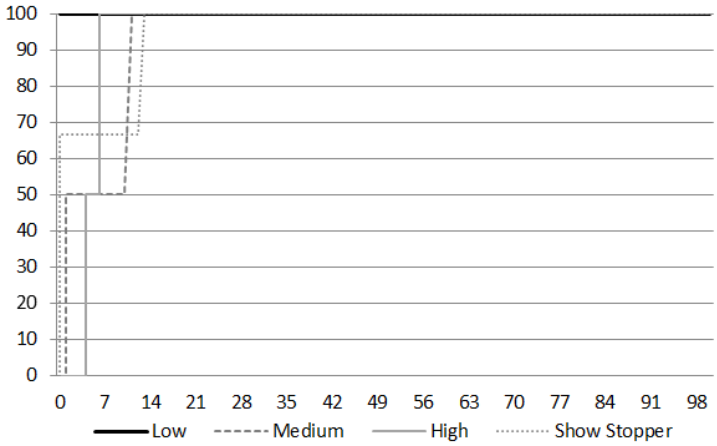

Figure 24: The calculation of the LAPFD metric for *Greedy* approaches regarding the execution of the *PrimeVideo* test suite in *Project 1* in the second cycle.

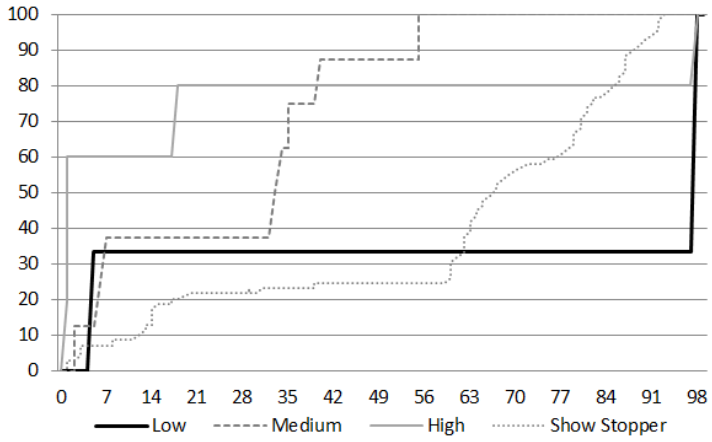

Figure 25: The calculation of the LAPFD metric for Random ordering regarding the execution of the *Netflix* test suite in *Project 2* in the first cycle.

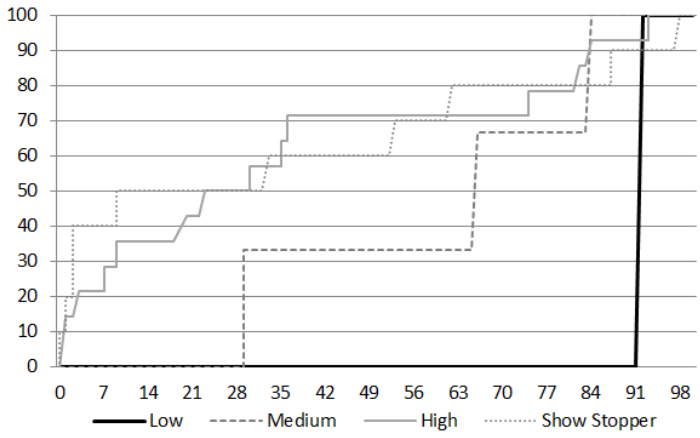

Figure 26: The calculation of the LAPFD metric for Random ordering regarding the execution of the *Netflix* test suite in *Project 2* in the second cycle.

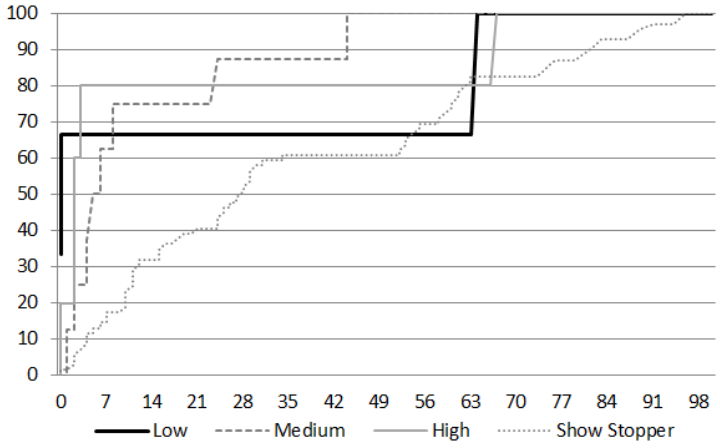

Figure 27: The calculation of the LAPFD metric for *Greedy* approaches regarding the execution of the *Netflix* test suite in *Project 2* in the first cycle.

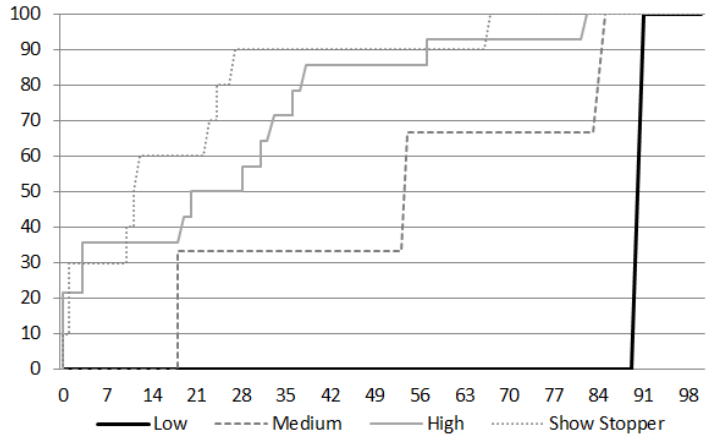

Figure 28: The calculation of the LAPFD metric for *Greedy* approaches regarding the execution of the *Netflix* test suite in *Project 2* in the second cycle.

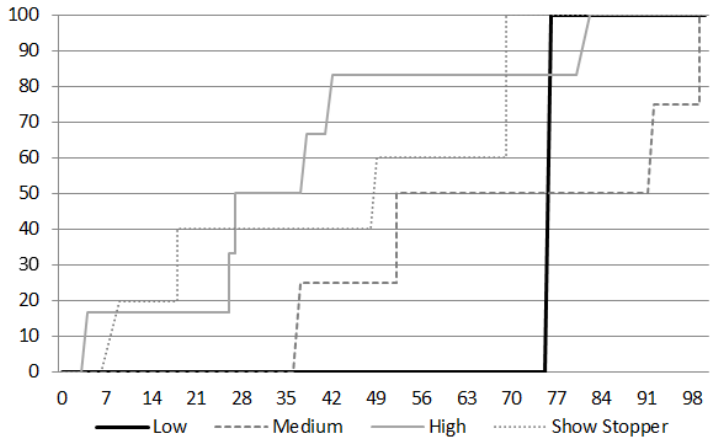

Figure 29: The calculation of the LAPFD metric for Random ordering regarding the execution of the *YouTube* test suite in *Project 2* in the first cycle.

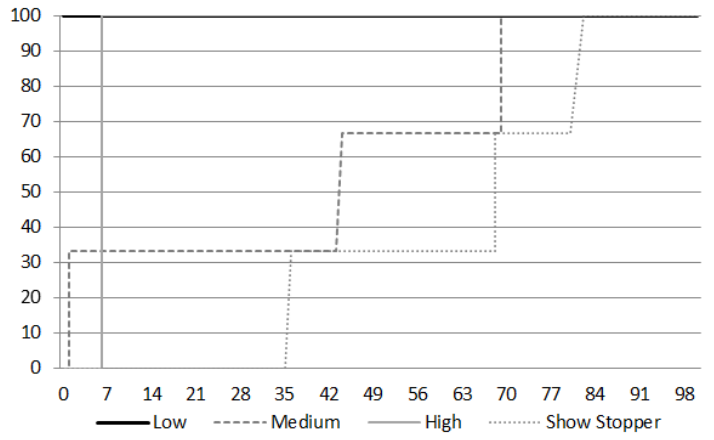

Figure 30: The calculation of the LAPFD metric for Random ordering regarding the execution of the *YouTube* test suite in *Project 2* in the second cycle.

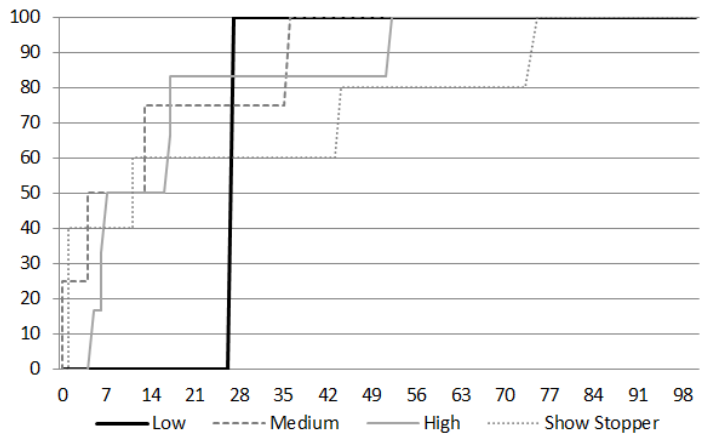

Figure 31: The calculation of the LAPFD metric for *Greedy* approaches regarding the execution of the *YouTube* test suite in *Project 2* in the first cycle.

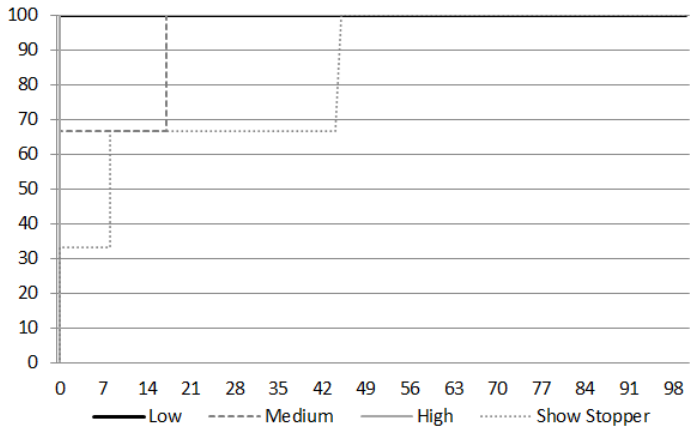

Figure 32: The calculation of the LAPFD metric for *Greedy* approaches regarding the execution of the *YouTube* test suite in *Project 2* in the second cycle.

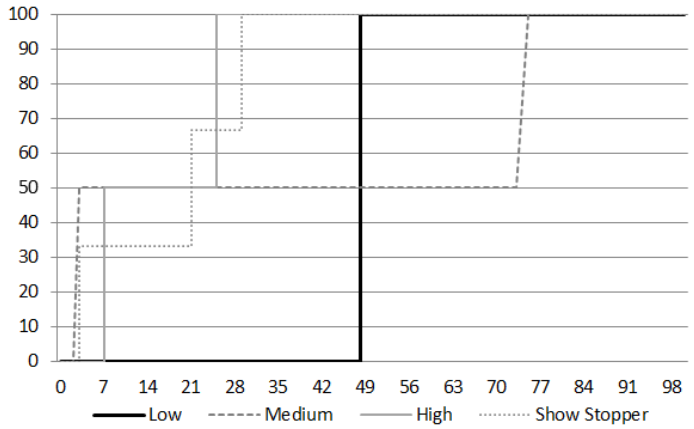

Figure 33: The calculation of the LAPFD metric for Random ordering regarding the execution of the *PrimeVideo* test suite in *Project 2* in the first cycle.

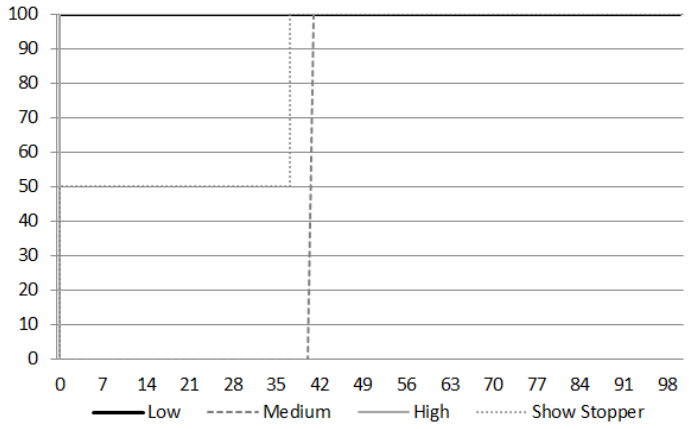

Figure 34: The calculation of the LAPFD metric for Random ordering regarding the execution of the *PrimeVideo* test suite in *Project 2* in the second cycle.

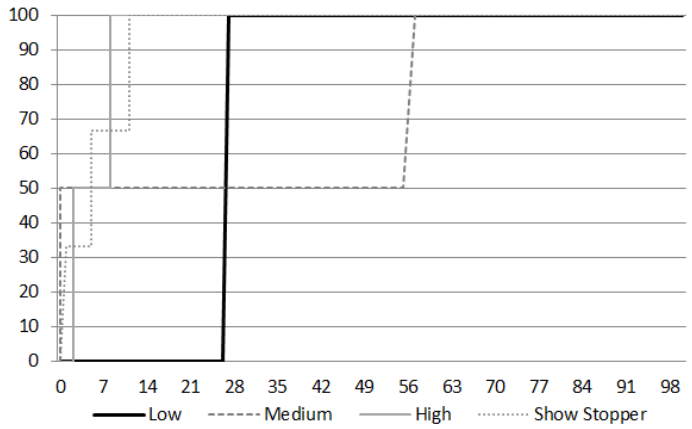

Figure 35: The calculation of the LAPFD metric for *Greedy* approaches regarding the execution of the *PrimeVideo* test suite in *Project 2* in the first cycle.

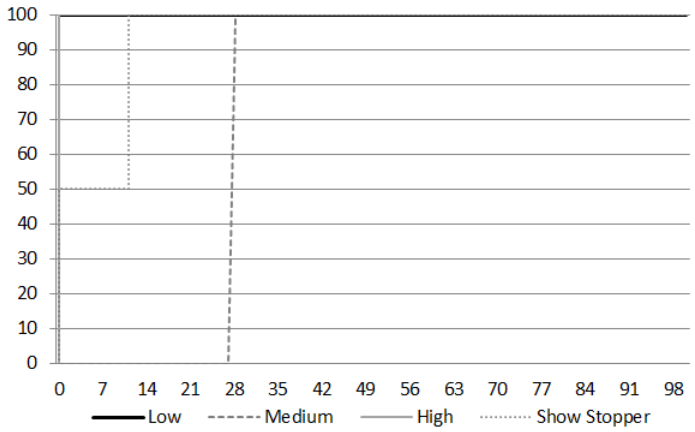

Figure 36: The calculation of the LAPFD metric for *Greedy* approaches regarding the execution of the *PrimeVideo* test suite in *Project 2* in the second cycle.

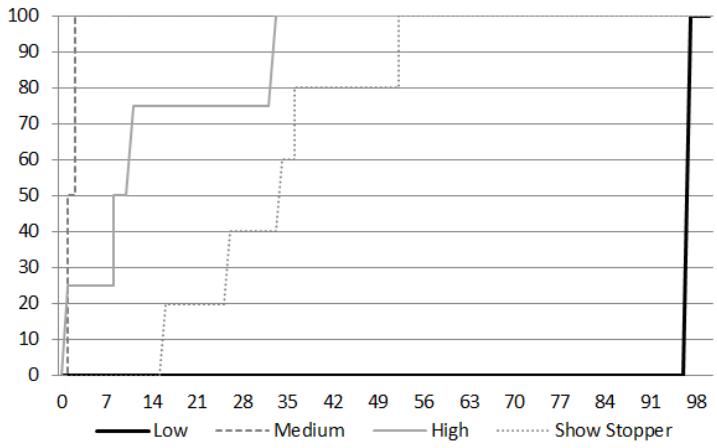

Figure 37: The calculation of the LAPFD metric for Random ordering regarding the execution of the *Netflix* test suite in *Project 3* in the second cycle.

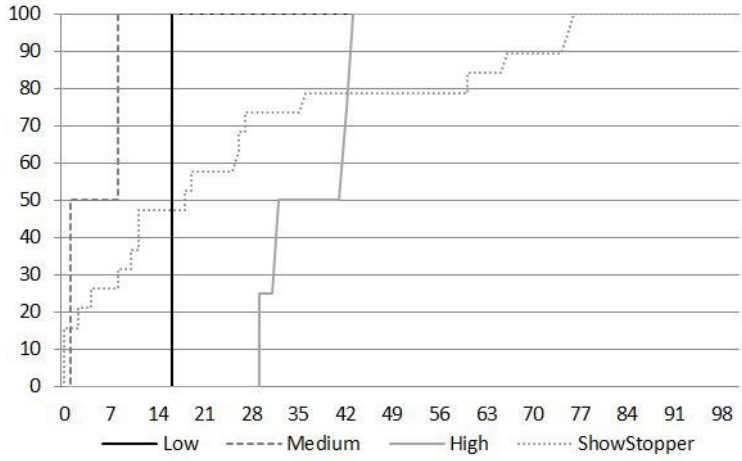

Figure 38: The calculation of the LAPFD metric for *Greedy* approaches regarding the execution of the *Netflix* test suite in *Project 3* in the second cycle.

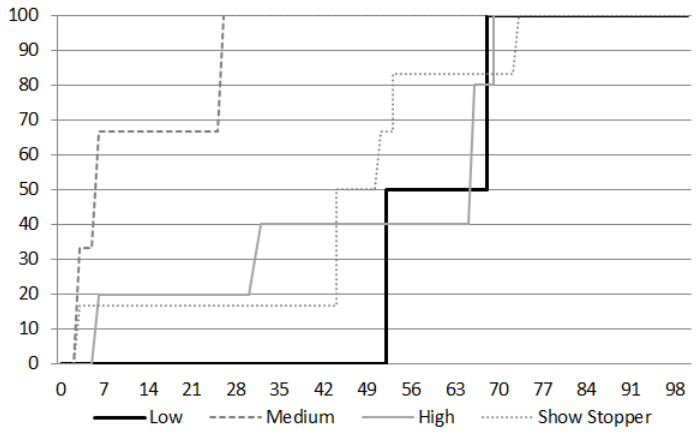

Figure 39: The calculation of the LAPFD metric for Random ordering regarding the execution of the *YouTube* test suite in *Project 3* in the first cycle.

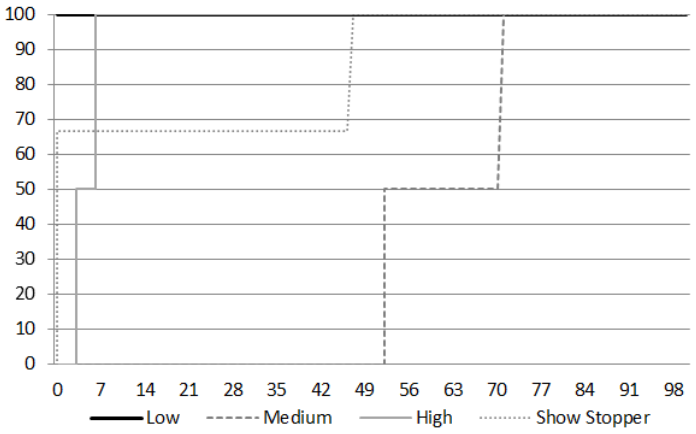

Figure 40: The calculation of the LAPFD metric for Random ordering regarding the execution of the *YouTube* test suite in *Project 3* in the second cycle.

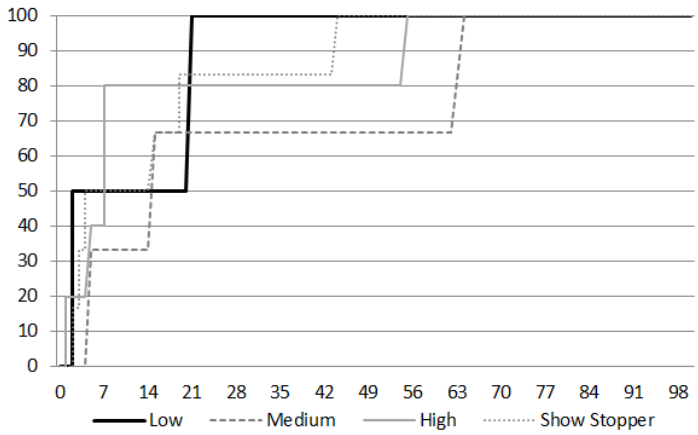

Figure 41: The calculation of the LAPFD metric for *Greedy* approaches regarding the execution of the *YouTube* test suite in *Project 3* in the first cycle.

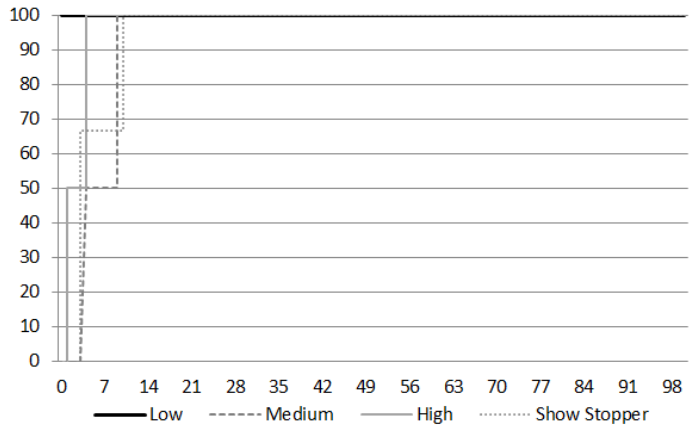

Figure 42: The calculation of the LAPFD metric for *Greedy* approaches regarding the execution of the *YouTube* test suite in *Project 3* in the second cycle.

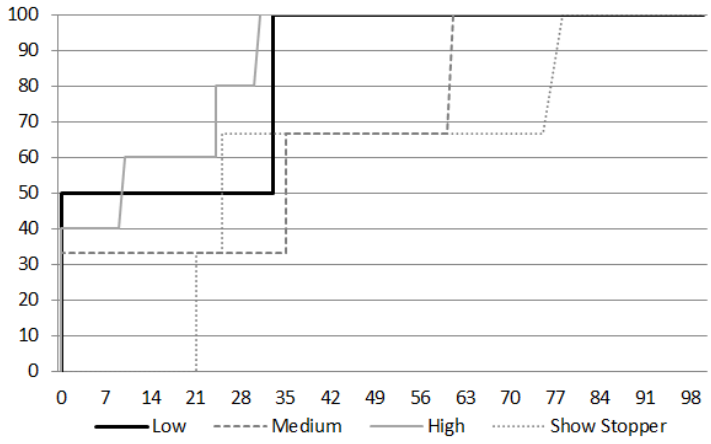

Figure 43: The calculation of the LAPFD metric for Random ordering regarding the execution of the *PrimeVideo* test suite in *Project 3* in the first cycle.

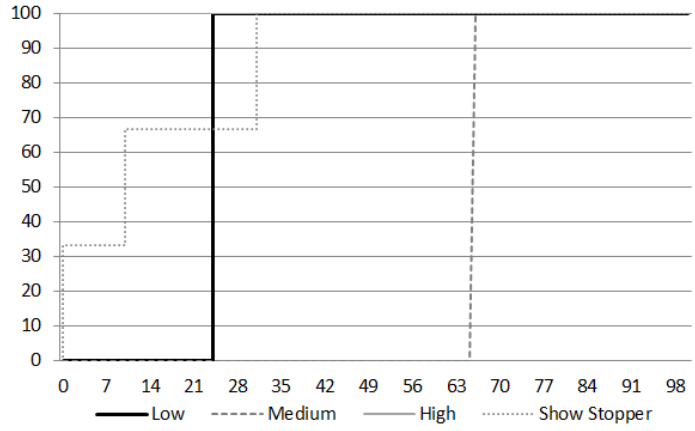

Figure 44: The calculation of the LAPFD metric for Random ordering regarding the execution of the *PrimeVideo* test suite in *Project 3* in the second cycle.

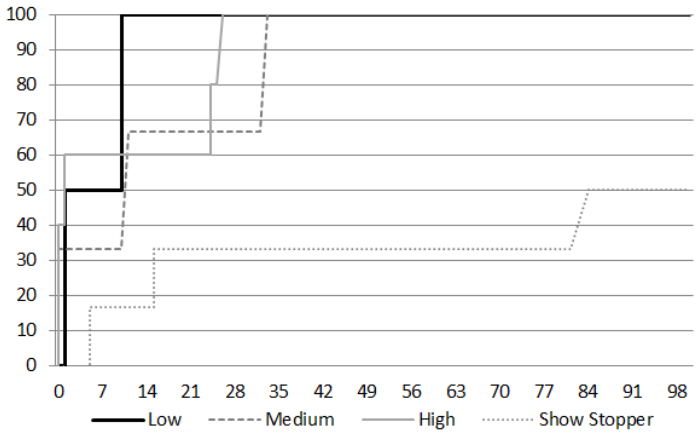

Figure 45: The calculation of the LAPFD metric for *Greedy* approaches regarding the execution of the *PrimeVideo* test suite in *Project 3* in the first cycle.

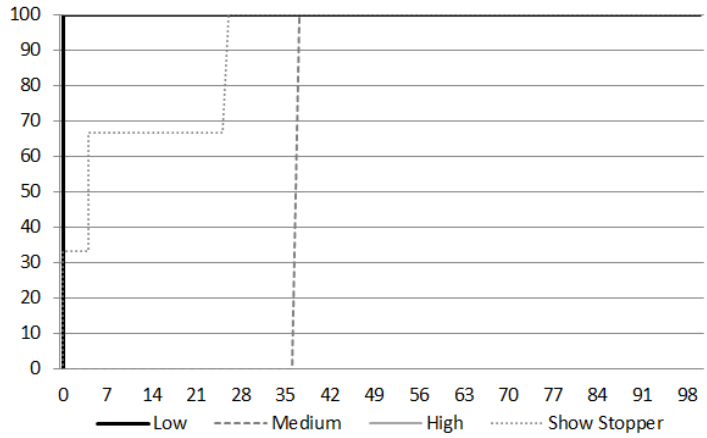

Figure 46: The calculation of the LAPFD metric for *Greedy* approaches regarding the execution of the *PrimeVideo* test suite in *Project 3* in the second cycle.

APPENDIX B

RESULTS FOR PARALLEL TEST EXECUTION ON 3 TEST STATIONS

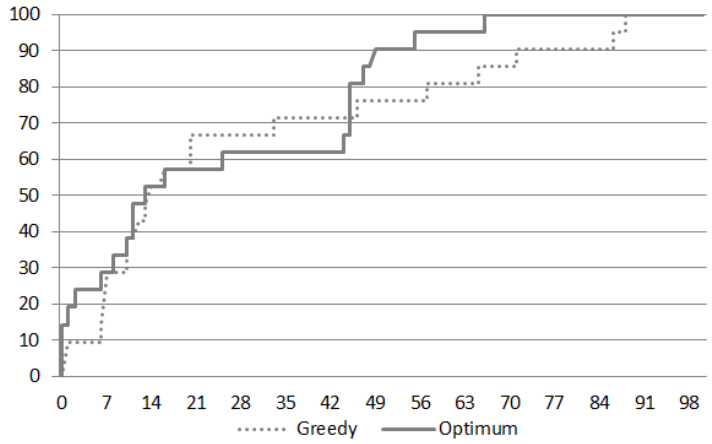

Figure 47: The calculation of the $APFD_{pc}$ metric for *Greedy* and *Optimum* approaches regarding the execution of the *Netflix* test suite in parallel on 3 stations for *Project 1* in the first cycle.

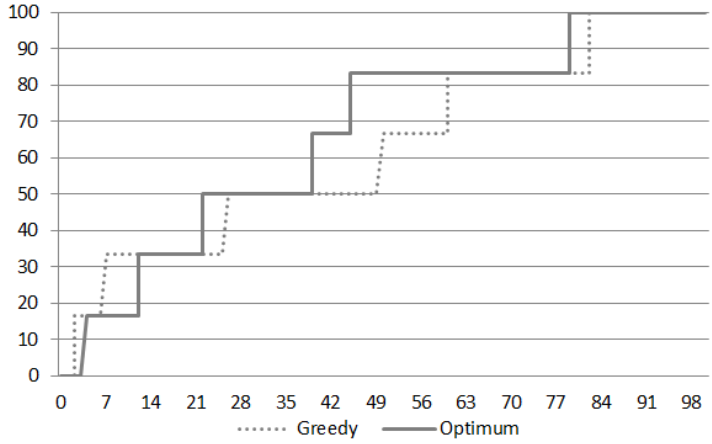

Figure 48: The calculation of the $APFD_{pc}$ metric for *Greedy* and *Optimum* approaches regarding the execution of the *Netflix* test suite in parallel on 3 stations for *Project 1* in the second cycle.

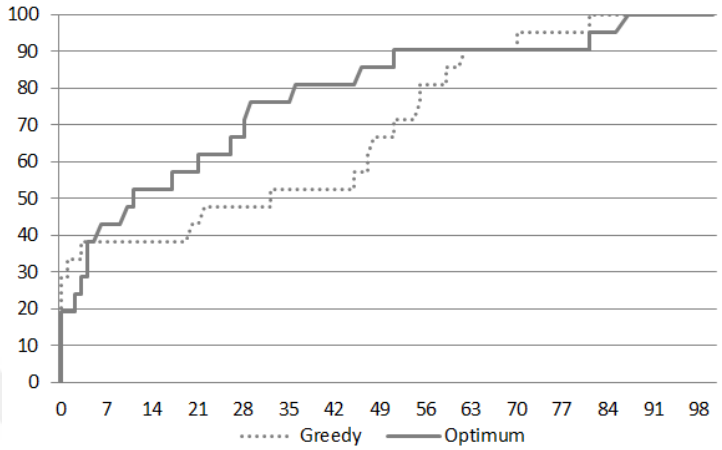

Figure 49: The calculation of the APFD_{pc} metric for *Greedy* and *Optimum* approaches regarding the execution of the *YouTube* test suite in parallel on 3 stations for *Project 1* in the first cycle.

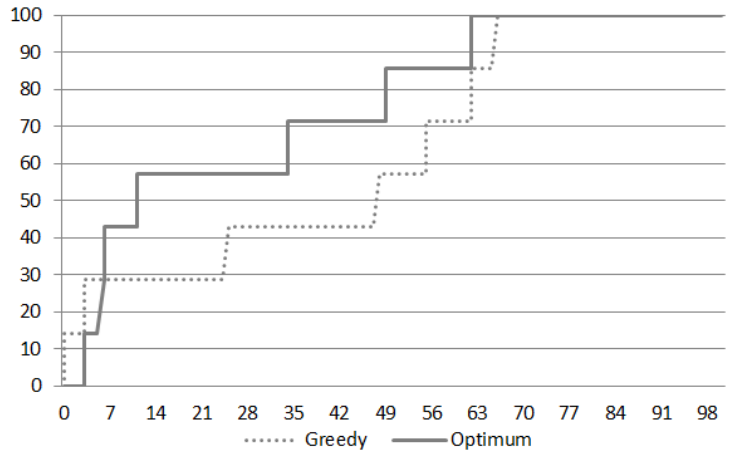

Figure 50: The calculation of the APFD_{pc} metric for *Greedy* and *Optimum* approaches regarding the execution of the *YouTube* test suite in parallel on 3 stations for *Project 1* in the second cycle.

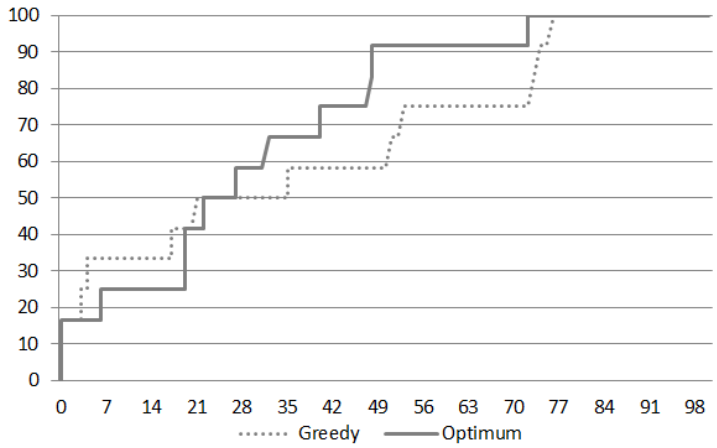

Figure 51: The calculation of the APFD_{pc} metric for *Greedy* and *Optimum* approaches regarding the execution of the *PrimeVideo* test suite in parallel on 3 stations for *Project 1* in the first cycle.

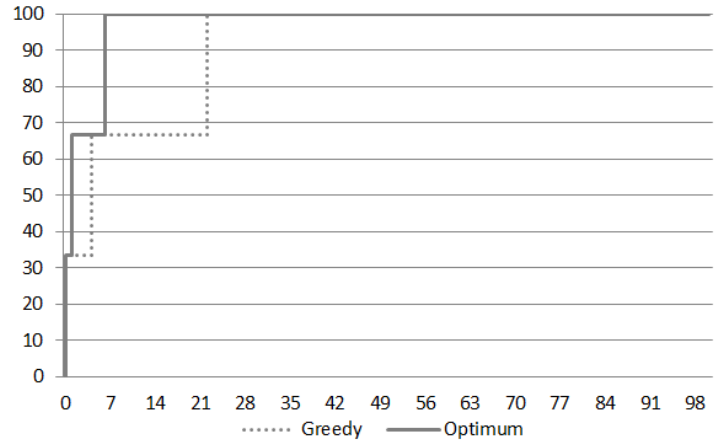

Figure 52: The calculation of the APFD_{pc} metric for *Greedy* and *Optimum* approaches regarding the execution of the *PrimeVideo* test suite in parallel on 3 stations for *Project 1* in the second cycle.

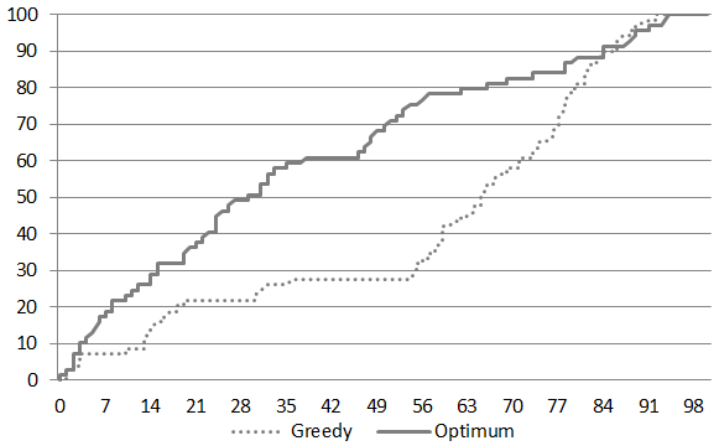

Figure 53: The calculation of the APFD_{pc} metric for *Greedy* and *Optimum* approaches regarding the execution of the *Netflix* test suite in parallel on 3 stations for *Project 1* in the first cycle.

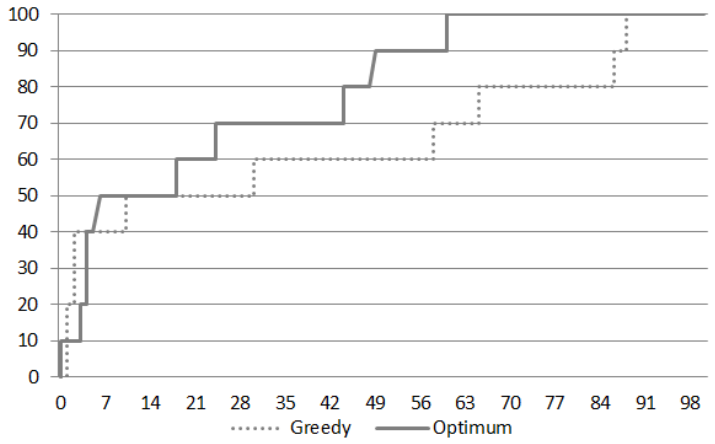

Figure 54: The calculation of the APFD_{pc} metric for *Greedy* and *Optimum* approaches regarding the execution of the *Netflix* test suite in parallel on 3 stations for *Project 1* in the second cycle.

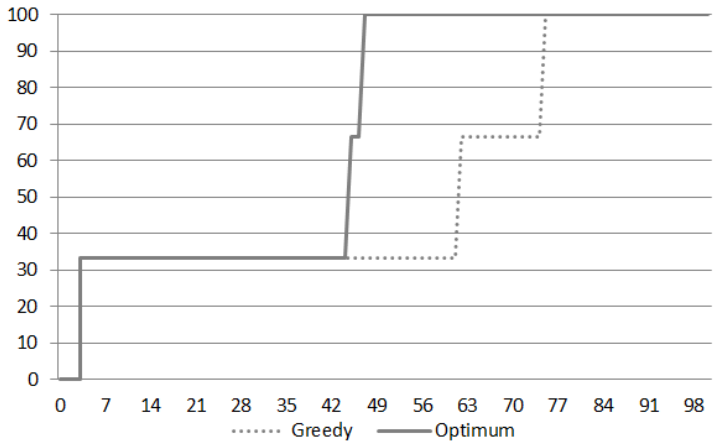

Figure 55: The calculation of the APFD_{pc} metric for *Greedy* and *Optimum* approaches regarding the execution of the *YouTube* test suite in parallel on 3 stations for *Project 1* in the second cycle.

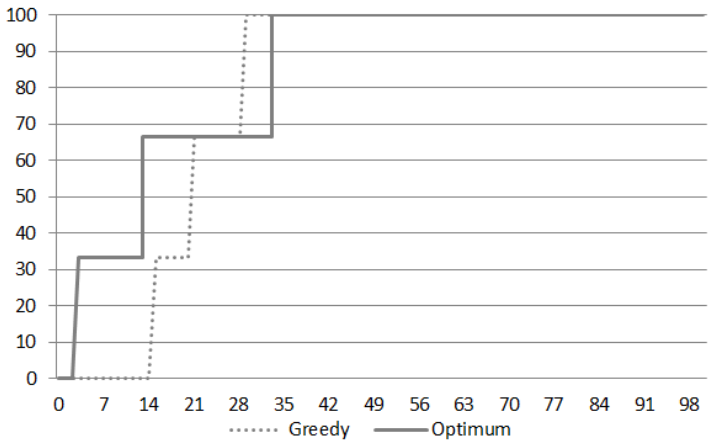

Figure 56: The calculation of the APFD_{pc} metric for *Greedy* and *Optimum* approaches regarding the execution of the *PrimeVideo* test suite in parallel on 3 stations for *Project 1* in the first cycle.

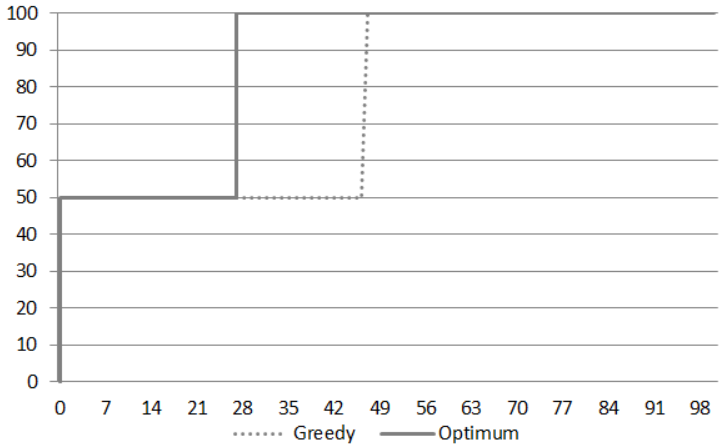

Figure 57: The calculation of the APFD_{pc} metric for *Greedy* and *Optimum* approaches regarding the execution of the *PrimeVideo* test suite in parallel on 3 stations for *Project 1* in the second cycle.

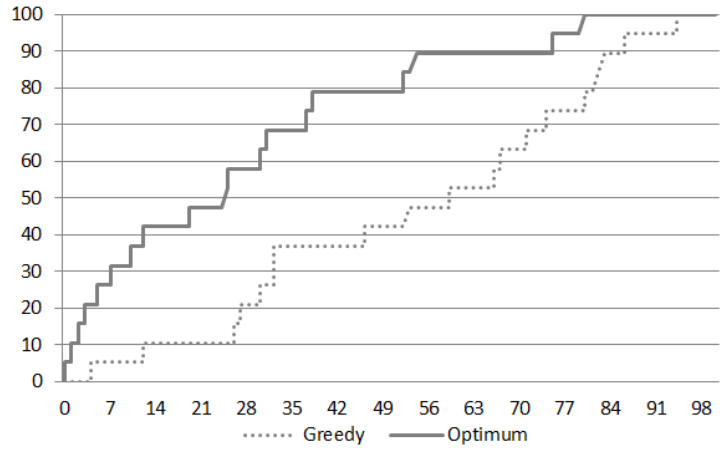

Figure 58: The calculation of the APFD_{pc} metric for *Greedy* and *Optimum* approaches regarding the execution of the *Netflix* test suite in parallel on 3 stations for *Project 1* in the first cycle.

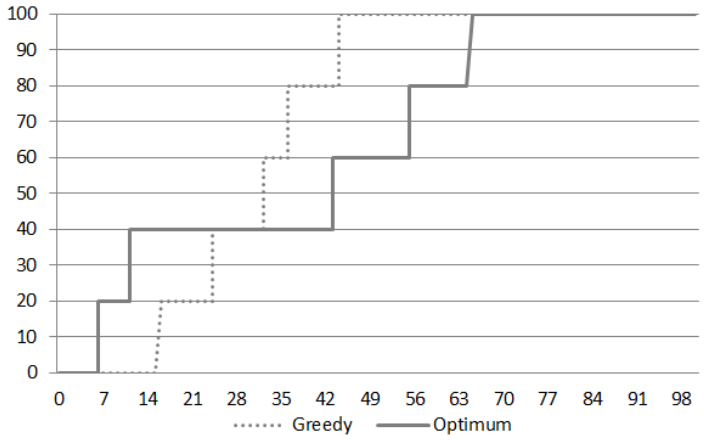

Figure 59: The calculation of the APFD_{pc} metric for *Greedy* and *Optimum* approaches regarding the execution of the *Netflix* test suite in parallel on 3 stations for *Project 1* in the second cycle.

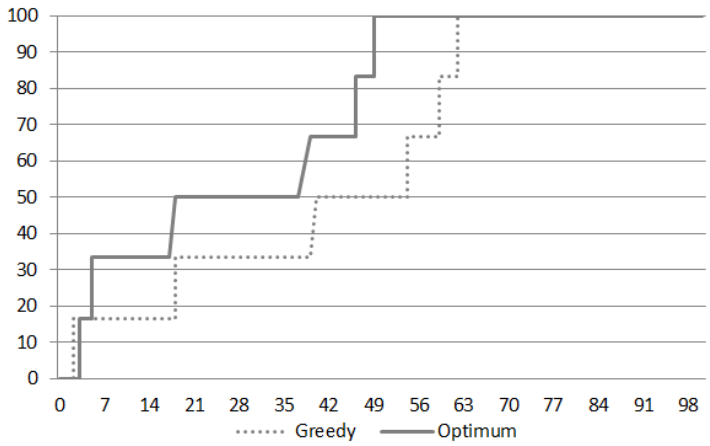

Figure 60: The calculation of the APFD_{pc} metric for *Greedy* and *Optimum* approaches regarding the execution of the *YouTube* test suite in parallel on 3 stations for *Project 1* in the first cycle.

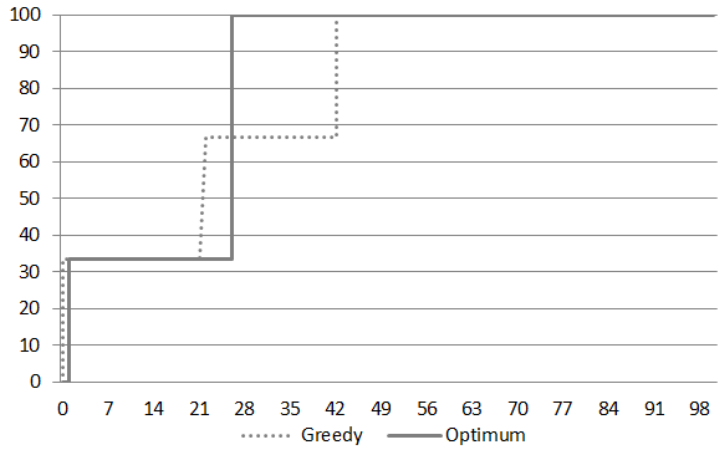

Figure 61: The calculation of the APFD_{pc} metric for *Greedy* and *Optimum* approaches regarding the execution of the *YouTube* test suite in parallel on 3 stations for *Project 1* in the second cycle.

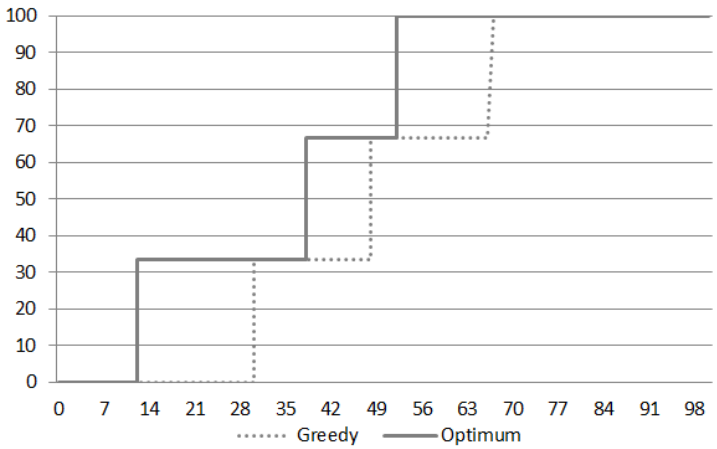

Figure 62: The calculation of the APFD_{pc} metric for *Greedy* and *Optimum* approaches regarding the execution of the *PrimeVideo* test suite in parallel on 3 stations for *Project 1* in the first cycle.

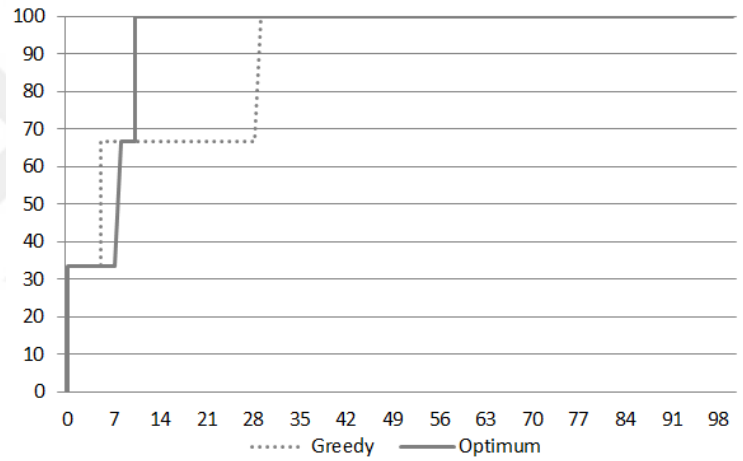

Figure 63: The calculation of the APFD_{pc} metric for *Greedy* and *Optimum* approaches regarding the execution of the *PrimeVideo* test suite in parallel on 3 stations for *Project 1* in the second cycle.

APPENDIX C

RESULTS FOR PARALLEL TEST EXECUTION ON 5 TEST STATIONS

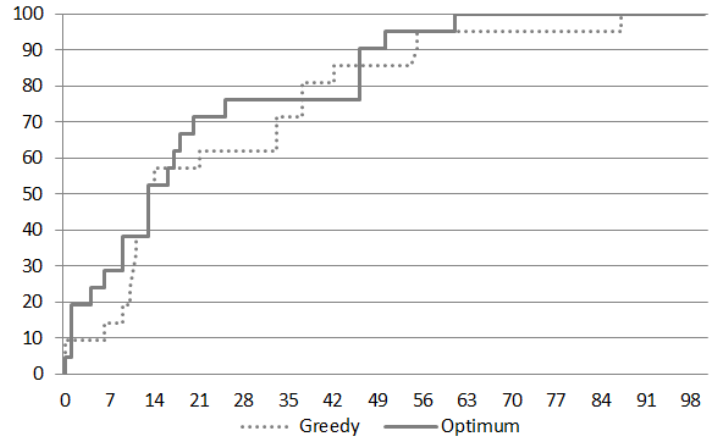

Figure 64: The calculation of the $APFD_{pc}$ metric for *Greedy* and *Optimum* approaches regarding the execution of the *Netflix* test suite in parallel on 5 test stations for *Project 1* in the first test cycle.

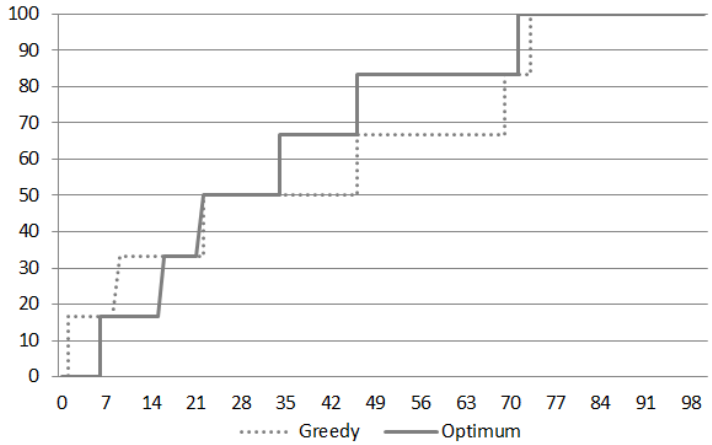

Figure 65: The calculation of the $APFD_{pc}$ metric for *Greedy* and *Optimum* approaches regarding the execution of the *Netflix* test suite in parallel on 5 test stations for *Project 1* in the second test cycle.

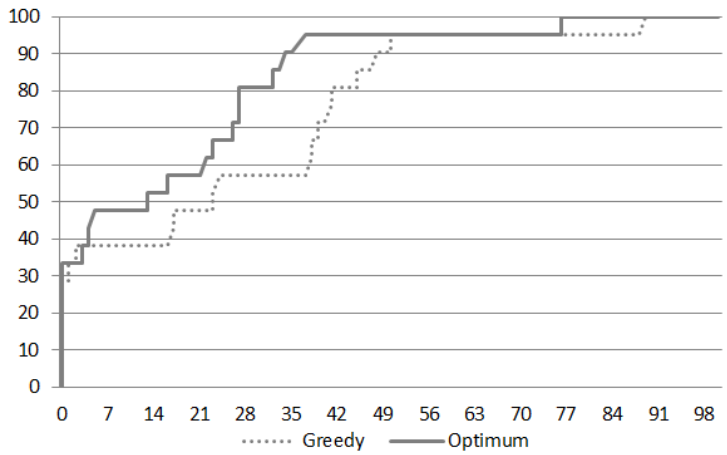

Figure 66: The calculation of the APFD_{pc} metric for *Greedy* and *Optimum* approaches regarding the execution of the *YouTube* test suite in parallel on 5 test stations for *Project 1* in the first test cycle.

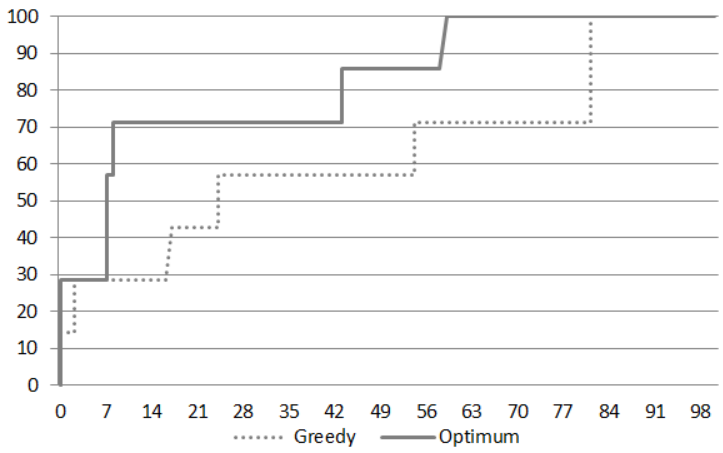

Figure 67: The calculation of the APFD_{pc} metric for *Greedy* and *Optimum* approaches regarding the execution of the *YouTube* test suite in parallel on 5 test stations for *Project 1* in the second test cycle.

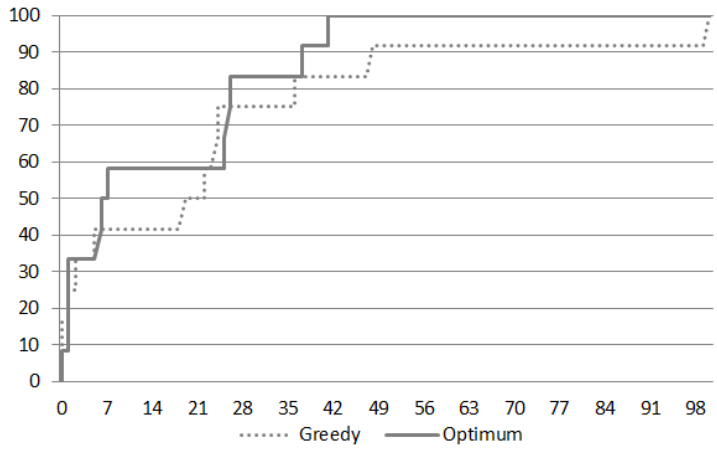

Figure 68: The calculation of the APFD_{pc} metric for *Greedy* and *Optimum* approaches regarding the execution of the *PrimeVideo* test suite in parallel on 5 test stations for *Project 1* in the first test cycle.

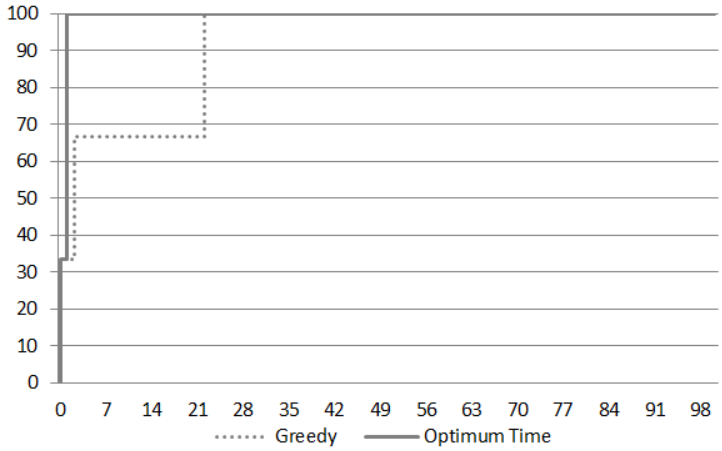

Figure 69: The calculation of the APFD_{pc} metric for *Greedy* and *Optimum* approaches regarding the execution of the *PrimeVideo* test suite in parallel on 5 test stations for *Project 1* in the second test cycle.

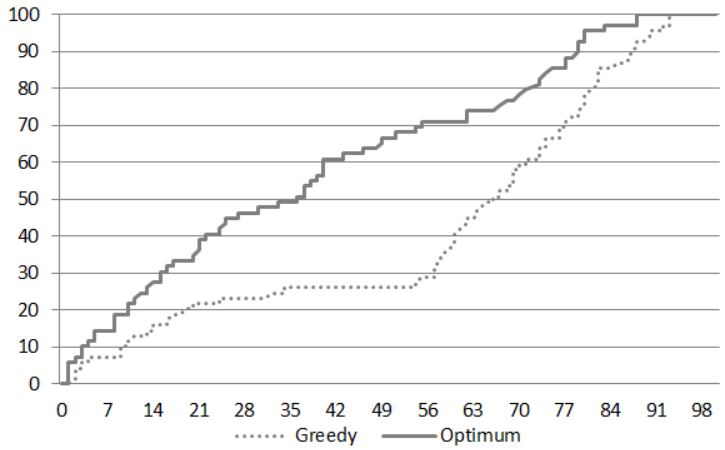

Figure 70: The calculation of the APFD_{pc} metric for *Greedy* and *Optimum* approaches regarding the execution of the *Netflix* test suite in parallel on 5 test stations for *Project 2* in the first test cycle.

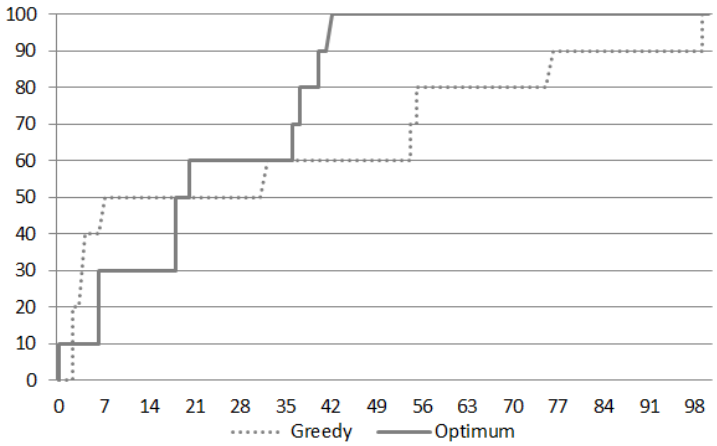

Figure 71: The calculation of the APFD_{pc} metric for *Greedy* and *Optimum* approaches regarding the execution of the *Netflix* test suite in parallel on 5 test stations for *Project 2* in the second test cycle.

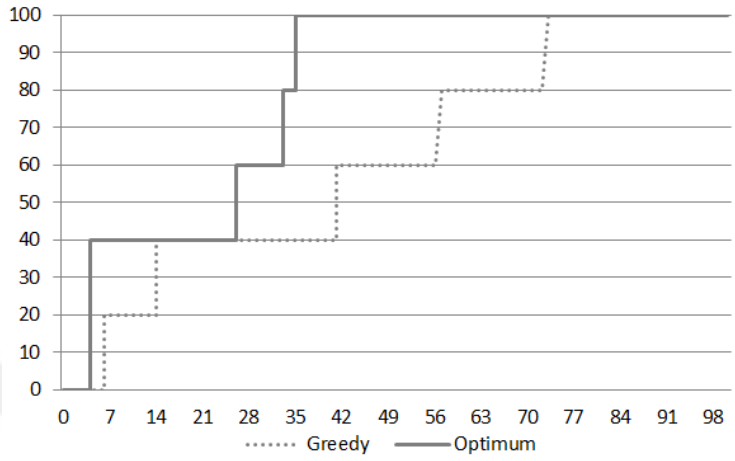

Figure 72: The calculation of the APFD_{pc} metric for *Greedy* and *Optimum* approaches regarding the execution of the *YouTube* test suite in parallel on 5 test stations for *Project 2* in the first test cycle.

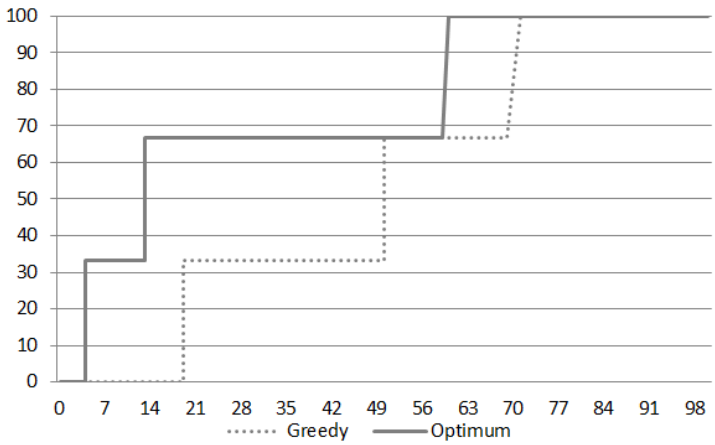

Figure 73: The calculation of the APFD_{pc} metric for *Greedy* and *Optimum* approaches regarding the execution of the *YouTube* test suite in parallel on 5 test stations for *Project 2* in the second test cycle.

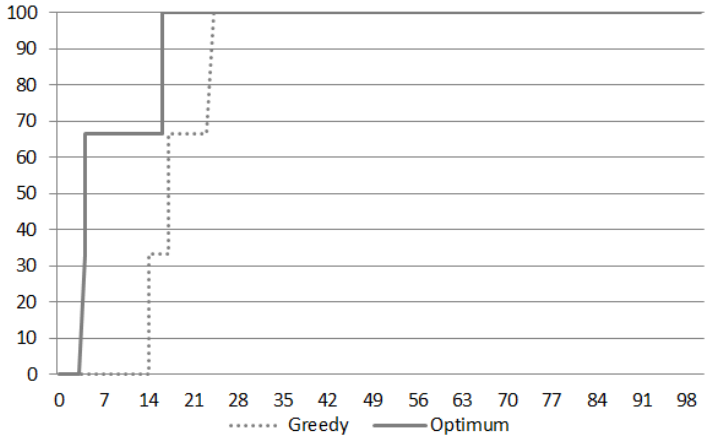

Figure 74: The calculation of the APFD_{pc} metric for *Greedy* and *Optimum* approaches regarding the execution of the *PrimeVideo* test suite in parallel on 5 test stations for *Project 2* in the first test cycle.

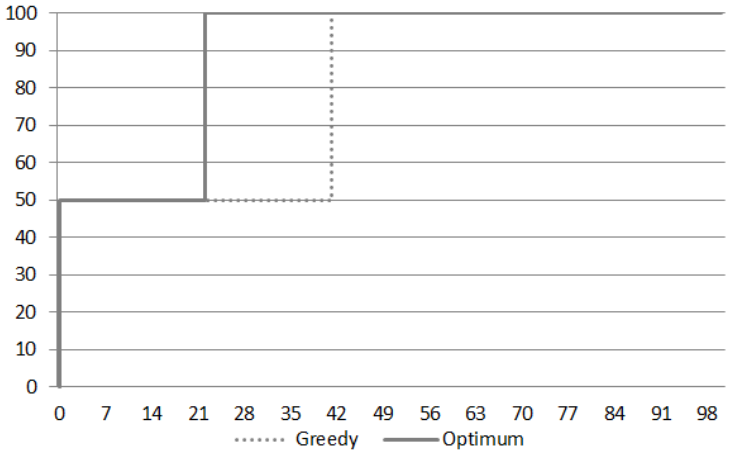

Figure 75: The calculation of the APFD_{pc} metric for *Greedy* and *Optimum* approaches regarding the execution of the *PrimeVideo* test suite in parallel on 5 test stations for *Project 2* in the second test cycle.

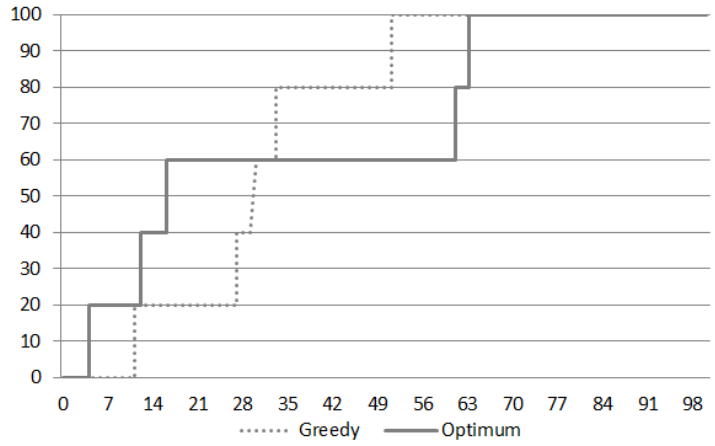

Figure 76: The calculation of the APFD_{pc} metric for *Greedy* and *Optimum* approaches regarding the execution of the *Netflix* test suite in parallel on 5 test stations for *Project 3* in the second test cycle.

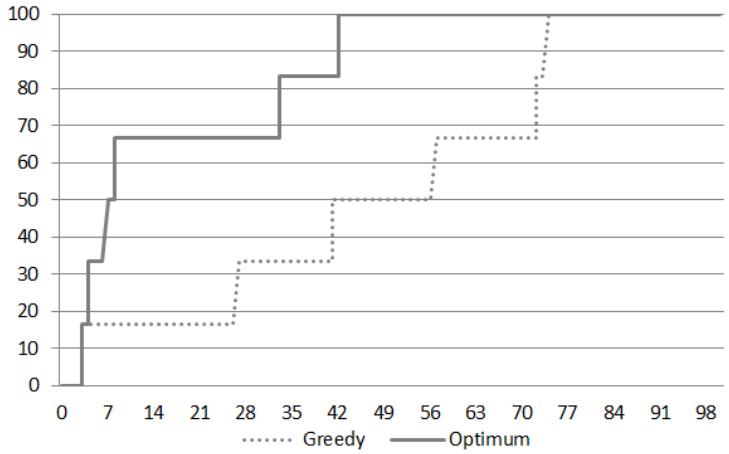

Figure 77: The calculation of the APFD_{pc} metric for *Greedy* and *Optimum* approaches regarding the execution of the *YouTube* test suite in parallel on 5 test stations for *Project 3* in the first test cycle.

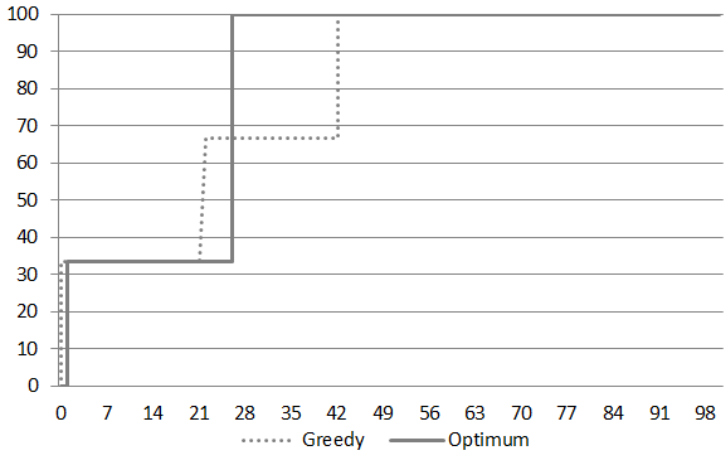

Figure 78: The calculation of the APFD_{pc} metric for *Greedy* and *Optimum* approaches regarding the execution of the *YouTube* test suite in parallel on 5 test stations for *Project 3* in the second test cycle.

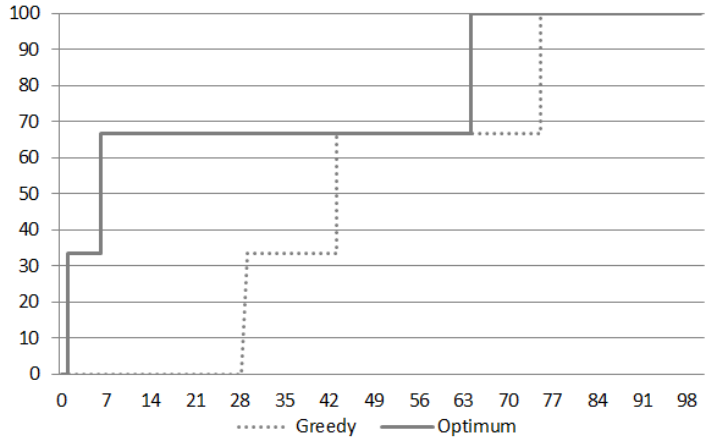

Figure 79: The calculation of the APFD_{pc} metric for *Greedy* and *Optimum* approaches regarding the execution of the *PrimeVideo* test suite in parallel on 5 test stations for *Project 3* in the first test cycle.

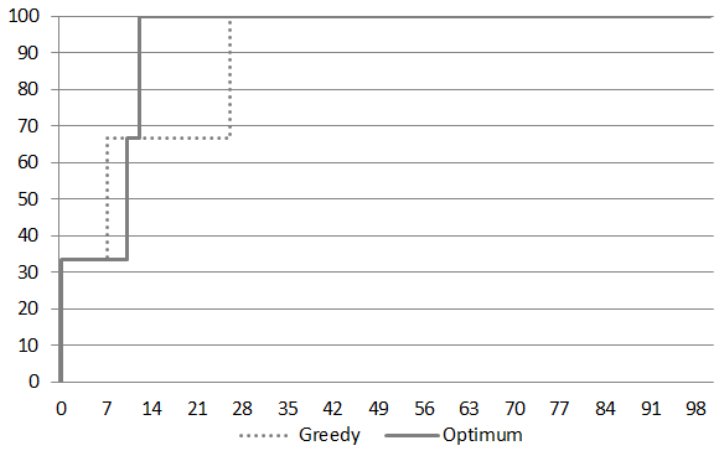

Figure 80: The calculation of the APFD_{pc} metric for *Greedy* and *Optimum* approaches regarding the execution of the *PrimeVideo* test suite in parallel on 5 test stations for *Project 3* in the second test cycle.

Bibliography

[1] V. Hilderman and L. Buckwalter, *Avionics Certification: A Complete Guide to DO-178 (Software), DO-254 (Hardware)*, 1st ed. USA: Avionics Communications Inc., 2007.

[2] "Introduction to Netflix Quality Control (QC) ," https://partnerhelp.netflixstudios.com/hc/en-us/articles/115000353211-Introduction-to-Netflix-Quality-Control-QC-, accessed: 2019-12-01.

[3] H. Leung and L. White, "Insights into regression testing," in *Proceedings of the International Conference on Software Maintenance*, 1989, pp. 60–69.

[4] 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.

[5] D. Garg and A. Datta, "Parallel execution of prioritized test cases for regression testing of web applications," in *Proceedings of the 36th Australasian Computer Science Conference*, 2013, p. 61–68.

[6] M. Khatibsyarbini, M. Isa, D. Jawawi, and R. Tumeng, "Test case prioritization approaches in regression testing: A systematic literature review," *Information and Software Technology*, vol. 93, pp. 74 – 93, 2018.

[7] R. Kazmi, D. Jawawi, R. Mohamad, and I. Ghani, "Effective regression test case selection: A systematic literature review," *ACM Computing Surveys*, vol. 50, no. 2, 2017.

[8] O. Ozener and H. Sozer, "An effective formulation of the multi-criteria test suite minimization problem," *Journal of Systems and Software*, vol. 168, p. 110632, 2020.

[9] G. Rothermel, M. Harrold, J. R. J, and C. Hong, "Empirical studies of test suite reduction," *Software Testing, Verification and Reliability*, vol. 4, no. 2, pp. 219 – 249, 2002.

[10] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid, "An empirical study of junit test-suite reduction," in *Proceedings of the 22nd IEEE International Symposium on Software Reliability Engineering*, 2011, pp. 170–179.

[11] A. Shi, A. Gyori, M. Gligoric, A. Zaytsev, and D. Marinov, "Balancing trade-offs in test-suite reduction," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, 2014, pp. 246–256.

[12] A. Shi, A. Gyori, S. Mahmood, P. Zhao, and D. Marinov, "Evaluating test-suite reduction in real-world software evolution," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2018, pp. 84–94.

[13] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 10, pp. 929–948, 2001.

[14] S. Dirim and H. Sozer, "Prioritization of test cases with varying test costs and fault severities for certification testing," in *Proceedings of the 13th IEEE International Conference on Software Testing, Verification and Validation Workshops*, 2020, pp. 386–391.

[15] D. You, Z. Chen, B. Xu, B. Luo, and C. Zhang, "An empirical study on the effectiveness of time-aware test case prioritization techniques," in *Proceedings of the 2011 ACM Symposium on Applied Computing*, 2011, pp. 1451–1456.

[16] H. Srikanth, L. Williams, and J. Osborne, "System test case prioritization of new and regression test cases," in *Proceedings of the International Symposium on Empirical Software Engineering*, 2005, pp. 64–73.

[17] A. Khalilian, M. Azgomi, and Y. Fazlalizadeh, "An improved method for test case prioritization by incorporating historical test case data," *Science of Computer Programming*, vol. 78, no. 1, pp. 93 – 116, 2012.

[18] G. Rothermel, R. Untch, C. Chu, and M. Harrold, "Test case prioritization: an empirical study," in *Proceedings of the 1999 IEEE International Conference on Software Maintenance*, 1999, pp. 179–188.

[19] S. Elbaum, A. 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.

[20] B. Qu, C. Nie, and B. Xu, "Test case prioritization for multiple processing queues," in *Proceedings of the International Symposium on Information Science and Engineering*, vol. 2, 2008, pp. 646–649.

[21] P. Zoeteweij, R. Abreu, R. Golsteijn, and A. J. C. van Gemund, "Diagnosis of embedded software using program spectra," in *Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, 2007, pp. 213–220.

[22] 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 2001*, 2001, pp. 329–338.

[23] B. T. Eck and M. Pinedo, "On the minimization of the makespan subject to flowtime optimality," *Operations Research*, vol. 41, pp. 622 – 806, 1993.

[24] N. Fenton and S. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*, 3rd ed. USA: CRC Press, 2014.

[25] "IBM CPLEX optimizer," https://www.ibm.com/analytics/cplex-optimizer, accessed: 2019-08-29.

[26] C. S. Gebizli and H. Sozer, "Model-based software product line testing by coupling feature models with hierarchical markov chain usage models," in *roceedings of the 6th IEEE International Workshop on Model-Based Verification and Validation*, 2016, pp. 278–283.

[27] C. Wohlin, P. Runeson, M. Host, M. Ohlsson, B. Regnell, and A. Wesslen, *Experimentation in Software Engineering*. Berlin, Heidelberg: Springer-Verlag, 2012.

[28] C. Catal and D. Mishra, "Test case prioritization: a systematic mapping study," *Software Quality Journal*, vol. 21, pp. 445–478, 2013.

[29] M. M. Islam, A. Marchetto, A. Susi, F. B. Kessler, and G. Scanniello, "MOTCP: a tool for the prioritization of test cases based on a sorting genetic algorithm and latent semantic indexing," in *Proceedings of the 28th IEEE International Conference on Software Maintenance*, 2012, pp. 654–657.

[30] D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, "A test case prioritization genetic algorithm guided by the hypervolume indicator," *IEEE Transactions on Software Engineering*, vol. 46, no. 6, pp. 674–696, 2020.

[31] C. Lu, J. Zhong, Y. Xue, L. Feng, and J. Zhang, "Ant colony system with sorting-based local search for coverage-based test case prioritization," *IEEE Transactions on Reliability*, vol. 69, no. 3, pp. 1004–1020, 2020.

[32] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *IEEE Transactions on Software Engineering*, vol. 33, no. 4, pp. 225–237, 2007.

[33] A. Marchetto, M. M. Islam, W. Asghar, A. Susi, and G. Scanniello, "A multi-objective technique to prioritize test cases," *IEEE Transactions on Software Engineering*, vol. 42, no. 10, pp. 918–940, 2016.

[34] S. Eghbali and L. Tahvildari, "Test case prioritization using lexicographical ordering," *IEEE Transactions on Software Engineering*, vol. 42, no. 12, pp. 1178–1195, 2016.

[35] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel, "The effects of time constraints on test case prioritization: A series of controlled experiments," *IEEE Transactions on Software Engineering*, vol. 36, no. 5, pp. 593–617, 2010.

[36] D. Hao, L. Zhang, L. Zang, Y. Wang, X. Wu, and T. Xie, "To be optimal or not in test-case prioritization," *IEEE Transactions on Software Engineering*, vol. 42, no. 5, pp. 490–505, 2016.

[37] D. D. Nardo, N. Alshahwan, L. Briand, and Y. Labiche, "Coverage-based test case prioritization: an industrial case study," in *Proceedings of the 6th IEEE International Conference on Software Testing, Verification and Validation*, 2013, pp. 302–311.

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

[39] H. Srikanth, C. Hettiarachchi, and H. Do, "Requirements based test prioritization using risk factors: an industrial study," *Information and Software Technology*, vol. 69, pp. 71 – 83, 2016.

[40] R. Malhotra, "A systematic review of machine learning techniques for software fault prediction," *Applied Soft Computing*, vol. 27, pp. 504 – 518, 2015.

[41] C. Catal, "Software fault prediction: a literature review and current trends," *Expert Systems with Applications*, vol. 38, pp. 4626–4636, 2011.

[42] S. McMaster and A. Memon, "Call-stack coverage for gui test suite reduction," *IEEE Transactions on Software Engineering*, vol. 34, no. 1, pp. 99–115, 2008.

[43] R. Steuer, *Multi-Criteria Optimization: Theory, Computation, and Application.* New York: John Wiley, 1986.

[44] S. Sampath, R. Bryce, and A. M. Memon, "A uniform representation of hybrid criteria for regression testing," *IEEE Transactions on Software Engineering*, vol. 39, no. 10, pp. 1326–1344, 2013.

[45] S. Tallam and N. Gupta, "A concept analysis inspired greedy algorithm for test suite minimization," *SIGSOFT Software Engineering Notes*, vol. 31, no. 1, pp. 35–42, 2005.

[46] T. Y. Chen and M. F. Lau, "Dividing strategies for the optimization of a test suite," *Information Processing Letters*, vol. 60, no. 3, pp. 135–141, 1996.

[47] M. J. Harrold, R. Gupta, and M. L. Soffa, "A methodology for controlling the size of a test suite," *ACM Transactions on Software Engineering and Methodology*, vol. 2, no. 3, pp. 270–285, 1993.

[48] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, "Timeaware test suite prioritization," in *Proceedings of the 15th International Symposium on Software Testing and Analysis*, 2006, pp. 1–12.

[49] L. Zhang, S.-S. Hou, C. Guo, T. Xie, and H. Mei, "Time-aware test-case prioritization using integer linear programming," in *Proceedings of the 18th International Symposium on Software Testing and Analysis*, 2009, pp. 213–224.

[50] J.-W. Lin, R. Jabbarvand, J. Garcia, and S. Malek, "Nemo: Multi-criteria test-suite minimization with integer nonlinear programming," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 1039–1049.

[51] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong, "An empirical study of the effects of minimization on the fault detection capabilities of test suites," in *Proceedings of the 6th International Conference on Software Maintenance*, 1998, pp. 34–43.

[52] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur, "Effect of test set minimization on fault detection effectiveness," in *Proceedings of the 17th International Conference on Software Engineering*, 1995, pp. 41–50.

[53] H. Y. Hsu and A. Orso, "MINTS: A general framework and tool for supporting test-suite minimization," in *Proceedings of the 31st IEEE International Conference on Software Engineering*, 2009, pp. 419–429.

[54] D. Jeffrey and N. Gupta, "Improving fault detection capability by selectively retaining test cases during test suite reduction," *IEEE Transactions on Software Engineering*, vol. 33, no. 2, pp. 108–123, 2007.

[55] J.-W. Lin and C.-Y. Huang, "Analysis of test suite reduction with enhanced tie-breaking techniques," *Information and Software Technology*, vol. 51, no. 4, pp. 679 – 690, 2009.

[56] J. Black, E. Melachrinoudis, and D. Kaeli, "Bi-criteria models for all-uses test suite reduction," in *Proceedings of the 26th International Conference on Software Engineering*, 2004, pp. 106–115.

[57] D. Hao, L. Zhang, X. Wu, H. Mei, and G. Rothermel, "On-demand test suite reduction," in *Proceedings of the 34th International Conference on Software Engineering*, 2012, pp. 738–748.

[58] R. Jabbarvand, A. Sadeghi, H. Bagheri, and S. Malek, "Energy-aware test-suite minimization for android apps," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 425–436.

[59] B. Miranda and A. Bertolino, "Scope-aided test prioritization, selection and minimization for software reuse," *Journal of Systems and Software*, vol. 131, pp. 528 – 549, 2017.

[60] D. D. Nardo, N. Alshahwan, L. Briand, and Y. Labiche, "Coverage-based regression test case selection, minimization and prioritization: a case study on an industrial system," *Software Testing, Verification and Reliability*, vol. 25, no. 4, pp. 371 – 396, 2015.

[61] D. Li, Y. Jin, C. Sahin, J. Clause, and W. G. J. Halfond, "Integrated energy-directed test suite optimization," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2014, pp. 339–350.

[62] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy, "The art of testing less without sacrificing quality," in *Proceedings of the 37th International Conference on Software Engineering*, 2015, pp. 483–493.

[63] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y. L. Traon, "Comparing white-box and black-box test prioritization," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 523–534.

[64] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1979.

[65] W. Jin and A. Orso, "Bugredux: Reproducing field failures for in-house debugging," in *Proceedings of the 34th International Conference on Software Engineering*, 2012, pp. 474–484.