Assessing the Efficacy of Test Selection, Prioritization, and Batching Strategies in the Presence of Flaky Tests and Parallel Execution at Scale

Emad Fallahzadeh

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy (Computer Science) at

Concordia University

Montréal, Québec, Canada

July 2023

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Emad Fallahzadeh**

Entitled: **Assessing the Efficacy of Test Selection, Prioritization, and Batching Strategies in the Presence of Flaky Tests and Parallel Execution at Scale**

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

______________________________ Chair
Dr. Mustafa Mehmet Ali

______________________________ External Examiner
Dr. Liam Peyton

______________________________ Examiner
Dr. Nikolaos Tsantalis

______________________________ Examiner
Dr. Peter Chen

______________________________ Examiner
Dr. Yan Liu

______________________________ Supervisor
Dr. Peter C. Rigby

Approved by ______________________________
Dr. Leila Kosseim, Graduate Program Director

9/7/2023 ______________________________
Dr. Mourad Debbabi, Dean
Gina Cody School of Engineering and Computer Science

Abstract

Assessing the Efficacy of Test Selection, Prioritization, and Batching Strategies in the Presence of Flaky Tests and Parallel Execution at Scale

Emad Fallahzadeh, Ph.D.

Concordia University, 2023

Effective software testing is essential for successful software releases, and numerous test optimization techniques have been proposed to enhance this process. However, existing research primarily concentrates on small datasets, resulting in impractical solutions for large-scale projects. Flaky tests, which significantly affect test optimization results, are often overlooked, and unrealistic approaches are employed to identify them. Furthermore, there is limited research on the impact of parallelization on test optimization techniques, particularly batching, and a lack of comprehensive comparisons among different techniques, including batching, which is an effective but often neglected approach.

To address research gaps, we analyzed the Chrome release process and collected a dataset of 276 million test results. In addition to evaluating established test optimization algorithms, we introduced two new algorithms. We also examined the impact of parallelism by varying the number of machines used. Our assessment covered various metrics, including feedback time, failing test detection speed, test execution time, and machine utilization.

Our investigation reveals that a significant portion of failures in testing is attributed to flaky tests, resulting in an inflated performance of test prioritization algorithms. Additionally, we observed that test parallelization has a non-linear impact on feedback time, as delays accumulate throughout the entire test queue. When it comes to optimizing feedback time, batching algorithms with adaptive batch sizes prove to be more effective compared to those with constant batch sizes, achieving execution reductions of up to 91%. Furthermore, our findings indicate that the batching technique is on par with the test selection algorithm in terms of effectiveness, while maintaining the advantage of not missing any failures.

Practitioners are encouraged to adopt adaptive batching techniques to minimize the number of machines required for testing and reduce feedback time, while effectively managing flaky tests. Analyzing historical data is crucial for determining the threshold at which adding more machines has minimal impact on feedback time, enabling optimization of testing efficiency and resource utilization.

Acknowledgments

I would like to take this opportunity to express my heartfelt gratitude to the individuals who have played a significant role in the completion of my Ph.D. journey.

First and foremost, I would like to extend my deepest appreciation to my supervisor, Dr. Peter Rigby, for his exceptional guidance, unwavering support, and invaluable mentorship throughout my research. His vast knowledge, insightful feedback, and dedication to my academic and professional growth have been instrumental in shaping the trajectory of my work. I am truly grateful for his expertise and the countless hours he has devoted to my development.

I am also immensely thankful to my advising and thesis committee for their invaluable contributions and guidance. Their expertise, constructive criticism, and thought-provoking discussions have greatly enhanced the quality of my research. I am deeply appreciative of their time and effort in reviewing my work, providing valuable suggestions, and pushing me to think critically. Their input has significantly strengthened the outcomes of my Ph.D. dissertation.

Furthermore, I am grateful to the funding agencies, including the NSERC and Concordia University FRS, for their financial support. Their assistance has provided me with the opportunity to pursue my research and make significant progress in my academic journey.

Lastly, I want to express my deepest appreciation to my family and friends for their unwavering love, encouragement, and understanding throughout this challenging endeavor. Their continuous support has been a constant source of inspiration and motivation, and I am truly grateful for their belief in me.

I am deeply grateful for the invaluable contributions from all those who contributed to my Ph.D. success, and I feel privileged to have worked with such remarkable individuals.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software systems constantly evolve through continuous changes aimed at integrating new features and resolving bugs. To ensure the integrity of these systems, regression tests are executed. However, the adoption of Continuous Integration (CI) release processes has led to a substantial increase in the number of test executions over time. Consequently, the feedback time for developers has also increased. This situation presents significant challenges and costs for companies, even those with substantial resources like Google, which conducts millions of tests on a daily basis. Despite their best efforts, it remains impractical to execute tests for every single change [1, 2].

To address the challenges of increasing test volumes and feedback time, several test optimization techniques have been proposed. One such technique is test selection, which focuses on running tests that are more likely to uncover failures. This approach improves feedback time and saves on test execution time. However, some companies are hesitant to rely solely on test selection as it may allow bugs to slip through without running all tests.

Test prioritization, on the other hand, ensures that all tests are executed while giving higher priority to tests that are more likely to expose failures. By prioritizing critical tests, feedback time can be reduced without compromising overall test coverage.

Test parallelization is another strategy aimed at minimizing feedback time. It involves distributing tests across multiple machines and running them in parallel. This approach harnesses the power of parallel computing to speed up test execution and provide quicker feedback to developers.

Test batching has gained popularity as a test optimization technique especially when there are overlapping changes that must be tested. It involves merging multiple changes together and running tests for all corresponding changes once. This approach reduces both feedback time and execution time by grouping related changes and their associated tests.

1.1 Research Gap

The existing research on test optimization has primarily focused on small datasets, which leads to impractical solutions for large-scale projects [2, 3]. This gap is particularly relevant because test optimization techniques are more applicable and impactful in the context of large-scale projects. Additionally, previous studies have overlooked important factors and limitations specific to large projects.

The impact of flaky tests on test optimization techniques is often disregarded. While many studies consider all test failures as blocking, meaning they prevent a build from being integrated, our investigation of the Chrome release process reveals that flaky failures are non-blocking tests that do not hinder integration. Moreover, the methods used to identify flaky tests differ from the reality of many large projects. For instance, Alshamari *et al.* [4] ran tests up to 1000 times and identified 67% of them as flaky tests. In contrast, in our analysis of 276 million test results, failing tests were only rerun between 2 to 10 times, with a mere 0.03% labeled as flaky.

Another significant issue is the reliance on expensive and impractical approaches in previous studies, which are not feasible in large projects. Memon *et al.* [2] explicitly stated that costly techniques like code coverage are impractical in Google due to high code churn. The reason behind the usage of these expensive test optimization approaches in the literature is the reliance on small, non-representative datasets. For example, Alshamari *et al.* [4] studied flaky tests using a dataset of only 24 projects, totaling a mere 22,245 test results.

Furthermore, the impact of parallelization on different test optimization techniques has received limited attention in research. While various test optimization techniques have been investigated, there is a lack of comparative studies, particularly with respect to batching, an efficient yet often overlooked technique. Another research limitation is the absence of standardized metrics for comparing different test optimization techniques. For example, it would be valuable to determine the speed at which test batching can detect failing tests and to understand the performance of different techniques in diverse resource execution environments.

1.2 Research Aim

In this research, we aim to understand the efficiency of different test optimization techniques by considering the impact of scale and flaky tests.

To achieve this goal, we first study the impact of flaky tests and scale on test optimization algorithms in the large-scale Chrome project in chapter 2. This chapter answers the question of how the presence of flaky tests and scale affects the performance of these algorithms. Additionally, we analyze the distribution of different types of failures within the Chrome dataset to gain insights into the overall testing landscape.

To evaluate the effectiveness of test batching techniques at scale and considering parallelization, we simulate and assess various batching algorithms, including two novel approaches, on the Chrome project in chapter 3. This chapter explores the influence of parallelism by varying the number of machines used in the simulations. Furthermore, we compare the performance of batching algorithms with test prioritization and selection techniques considering the parallel execution in chapter 4, from various perspectives considering important factors such as feedback time, resource usage, test execution time, and the ability to quickly detect failing tests.

1.3 Research Objectives and Overarching Research Questions

This research addresses three overarching research questions, each corresponding to a specific chapter in the thesis:

Chapter 2 explores the impact of flaky tests and scale on test prioritization in Chrome. The overarching research question for this chapter is: *What is the impact of flaky tests on test prioritization in Chrome?*

This investigation aims to understand how flaky tests and scale influence the prioritization of tests in the context of Chrome testing.

Chapter 3 focuses on evaluating the performance of batching algorithms when tests are executed in scale and in parallel. The overarching research question for this chapter is: *How do batching algorithms perform when testing in parallel?*

This research question aims to assess the effectiveness of different batching algorithms in terms of their impact on build feedback time and test execution time.

Chapter 4 examines the effectiveness of various test optimization algorithms when executed in parallel in Chrome. The overarching research question for this chapter is: *How effective are different test optimization algorithms when executed in parallel in Chrome?*

This research question involves analyzing and comparing the performance of test selection, prioritization, and batching techniques using diverse evaluation metrics.

To address the research questions, we analyze the Chrome release process and collect a comprehensive dataset comprising 276 million test results. We thoroughly examine and characterize the dataset, exploring its features and classifying the test results accordingly.

To conduct our evaluations, we simulate well-known test optimization algorithms and introduce two new algorithms of our own. Additionally, we assess the influence of parallelism by varying the number of machines employed in the simulations. Our evaluation encompasses a range of metrics, including feedback time, the speed at which failing tests are detected, test execution time, and the utilization of machines.

We make the following contributions:

- **Investigating the Chrome release process:** This study explores the Chrome release process, examining the steps involved from the initial change commitment to the final integration into the main Chrome repository. We also explore the complex classification system used in Chrome to categorize test results.

- **Capturing the dataset:** We collect a comprehensive dataset comprising 276 million test results from the Chrome project. This dataset not only includes the immediate pass/fail outcomes of each test, but also incorporates the final verdicts determined by the Chrome testing infrastructure, such as flaky labels. The data and scripts utilized in this research are made publicly available in our replication package [5].

- **Simulating test optimization algorithms:** To assess the effectiveness of test optimization algorithms, we replicate prominent existing algorithms and introduce two novel batching techniques. These algorithms account for the presence of flaky failures.
- **Evaluating parallelization effects:** We evaluate the performance of the test optimization algorithms by simulating their execution on different numbers of machines, enabling us to analyze their efficacy under varying resource availability scenarios.
- **Addressing flaky tests:** A significant focus of our study is dedicated to understanding and addressing the challenges posed by flaky tests. We incorporate realistic conditions associated with flaky tests into our algorithm simulations.
- **Comparing test optimization algorithms:** We conduct a comparative analysis of various test optimization algorithms to identify their strengths and weaknesses. This evaluation provides valuable insights for practitioners seeking to determine the most effective techniques, taking into account different resource availability situations.
- **Evaluation metrics:** To assess the performance of the test optimization algorithms, we utilize metrics such as feedback time, failure detection speed, test execution time, and machine utilization.

This thesis is structured as follows: Chapter 2 examines the impact of flaky tests on test prioritization in Chrome, which is based on our published paper at ICSE-SEIP 2022 [6]. Chapter 3 investigates the performance of parallel batch testing in Chrome and corresponds to our manuscript currently undergoing the second round of review at the FSE 2023 conference after major revisions. Chapter 4 explores the effectiveness and comparison of different test optimization techniques when executed in parallel in Chrome, representing the manuscript that is ready for submission to the EMSE journal. Finally, Chapter 5 concludes the study and provides insights into potential future research directions. In the appendices, Appendix A provides a guideline for practitioners on how to use the results of this study. Appendix B showcases different samples of test results from the Chrome website and Chrome logs, while Appendix C presents the infrastructure we developed to retrieve Chrome data.

Chapter 2

The Impact of Flaky Tests on Test Prioritization

This chapter is a verbatim copy of the paper published in the 44th International Conference on Software Engineering, Software Engineering in Practice Track (ICSE-SEIP 2022) [6], preserving the original content without any modifications.

Abstract

Test prioritization algorithms prioritize probable failing tests to give faster feedback to developers in case a failure occurs. Test prioritization approaches that use historical failures to run tests that have failed in the past may be susceptible to flaky tests as these tests often fail and then pass without identifying a fault. Traditionally, flaky failures like other types of failures are considered blocking, *i.e.* a test that needs to be investigated before the code can move to the next stage. However, on Google Chrome, flaky failures are non-blocking and the code still moves to the next stage in the CI pipeline. In this work, we explain the Chrome testing pipeline and classification. Then, we re-implement two important history based test prioritization algorithms and evaluate them on over 276 million test runs from the Chrome project. We apply these algorithms in two scenarios. First, we consider flaky failures as blocking and then, we use Chrome's approach and consider flaky failures as non-blocking.

Our investigation reveals that 99.58% of all failures are flaky. These types of failures are much more repetitive than non-flaky failures, and they are also well distributed over time. We conclude that the prior performance of the prioritization algorithms have been inflated by flaky failures. We release our data and scripts in our replication package [5].

2.1 Introduction

In large projects, developers make many code changes every minute [7]. To make sure that these changes do not introduce a regression into the codebase, regression tests are run. In the early era of regression testing, the number of regression tests was increasing in each individual build, delaying feedback to developers. Kim and Porter [8] observed that tests that failed in the past tend to be more likely to fail in the future, which was also confirmed in the more recent articles [9,10]. Their method is much simpler and more cost-effective in comparison with more time-consuming approaches such as code coverage techniques for test prioritization. Statistically, their approach will provide faster feedback about failures to developers and help identify bugs in a release. With the advent of Continuous Integration (CI) and monorepos, the number of changes increased dramatically. Elbaum *et al.* adapted Kim and Porter's approach to modern CI environments [1]. Instead of considering a single build, Elbaum *et al.* considered Google's test environment, where multiple builds are being tested simultaneously by multiple machines. Consequently, they implemented the test prioritization based on the historical test failure across builds.

In this paper, we address the issue of flaky tests in a large scale test environment. Flaky tests fail and pass non-deterministically, and few prior studies explicitly deal with. For example, Elbaum *et al.* [1] do not differentiate flaky vs real test failures, but mention that flaky failures should be removed when using historical test case prioritization. Machalica *et al.* [11] filter out flaky test failures by re-running tests that fail ten times. They found that test selection using historical and other factors was less effective when flaky test failures were included. Peng *et al.* [12] reran 252 failing Travis CI jobs and found a total of 29 flaky failures. On this tiny dataset, they found that historical test prioritization is negatively impacted by the existence of flaky tests. In our work, we examine 276 million tests on CHROME to understand the impact of flaky tests on test prioritization at scale.

In this research, we study the release process of Chrome project as a representative of a large scale testing system, and we describe the process from a change commit until it is landed on the Chrome main repository. We explain the complex test status classification adopted in Chrome, and the concept of test expectations and its usage in determining the final outcome of a test. We capture the results of 276 million test runs from the Chrome project to conduct our study. This dataset not only includes the immediate results of running tests, but also the final flags chosen by the Chrome testing infrastructure, such as the flaky flags. We re-implement two of the prominent historical-based test prioritization techniques as two applicable approaches in this scale to apply on our dataset to conduct the study. We select Kim and Porter's [8] approach as one of the most effective test prioritization techniques within a single build, and we refer to it as the KIMPORTER algorithm. We choose Elbaum's *et al.* [1] approach as state-of-the-art for test prioritization using historical failures across builds, and we refer to it as ELBAUMPRIORITIZATION algorithm. We reassess test prioritization in the context of Non-Blocking Flaky Failures (NBFF) where flaky failures do not block the builds, and we compare its results with the case of Blocking Flaky Failures (BFF). As a result, we see that the performance of the prioritization algorithms is much lower in the NBFF scenario in comparison with BFF case where the results are distorted with flaky failures. We then investigate the CHROME dataset to see the difference between the rate and distribution of different types of failures. The combination of the results and the statistical data about the flaky failures convince us that the test prioritization algorithms mostly had been prioritizing flaky failures.

The major contributions of this paper are:

- **Chrome release process:** In this research, we study the release process and steps included in Chrome from committing a change to finally, landing the change on the main Chrome repository. We also investigate the complex classification system which is used to categorize test results in Chrome.

- **Dataset:** We capture 276 million test results from the Chrome project to conduct this study. In addition to the immediate results of a test such fail and pass, this dataset consists of the final verdict which is concluded by the Chrome testing infrastructure, e.g. flaky labels. We release our data and scripts in our replication package [5].

- **Simulation:** We run two prominent history-based test prioritization algorithms on the enormous Chrome dataset by considering flaky failures as blocking versus non-blocking.
- **Outcome:** We reach the following conclusions.
  (1) Historical test prioritization algorithms are largely impacted by flaky tests.
  (2) The ratio of flaky failures is much higher than non-flaky failures, and they are much more repetitive than non-flaky failures.
  (3) Flaky failures are well distributed over time.
  (4) Previous test prioritization results are probably distorted with flaky tests.

The remainder of this paper is structured as follows. Section 2.2 describes the CHROME release process, its test status classification, and the way we capture data. Section 2.3 explains the adopted test prioritization approaches in this study. Section 2.4 determines how we set up the parameters to run test prioritization algorithms and how the outcomes are evaluated and interpreted. Section 2.5 displays the results of running the algorithms on the CHROME dataset in two different scenarios. Section 2.6 investigates the CHROME dataset to explain the reasons for the substantial differences in the results of these scenarios. Section 2.8 discusses the related works to this study. Finally, Section 2.9 concludes the paper.

2.2 Background on Data

The CHROME browser is a popular open-source web browser that has 63.64% of market share of all web browsers as of February 2021[13]. We choose this project in our experiment because the software development data is publicly available, but the process is similar to that used inside Google. The scale testing at Chrome is many order of magnitude larger than the Travis CI projects, which have been the subject of recent test prioritization and flaky test papers [12, 14].

2.2.1 CHROME Release Process

The Chrome development team follows the following process that we outline in Figure 2.1.

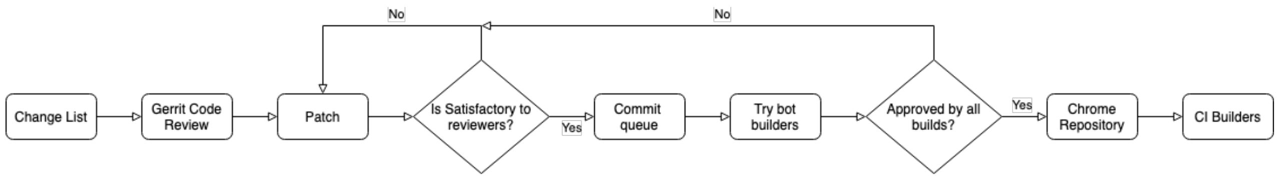


Figure 2.1: Chrome release process

(1) A developer uploads their code change, called a Change List (CL) at Google and CHROME, to the Gerrit code review tool.

(2) Reviewers can comment and suggest changes, which may lead to one or more new versions of the patch.

(3) Once the reviewers are satisfied with the change, it will be sent to the Commit Queue (CQ) to be emulated and tested on different platforms, *i.e.* builders, by try bots that ensure the integrity of the change through pre-submit testing.

(4) If the change is approved by all the try bots, the CQ will automatically commit the change to the CHROME main repository.

(5) Once the change is committed, also landed, the CHROME Continuous Integration system groups multiple changes made to trunk and does post-submit testing.

2.2.2 Test Status Classification in CHROME

Chrome is a large and complex project that does not have simple binary test outcomes. Instead, the outcomes are specified based on a predetermined set of allowable "expected results" for each test. These expected results are defined by CHROME developers on the basis of their expectations of the test results on a particular platform and their interpretation of the prior results of that test.

The most common expected results for tests are described below:

- Pass: by default, tests are expected to pass.
- Skip: tests that are disabled because they are expected to fail, especially on a particular platform. For example, a test that is specific to Windows will be skipped when run for a Linux builder.

- Fail: tests that are expected to fail.
- Timeout: tests that are expected to time out without producing any results.
- Crash: tests that are expected to crash while running.

Any expected result other than Pass or Skip is a failure expectation. Failure expectations are usually used in two cases. First, when a test is developed for a particular platform like Windows, and it is expected to fail on another platform such as Linux. In this scenario, Chrome developers prevent a failure of this test on Linux to cause multiple test reruns and blocking the build. Second, it is used for the case of flaky failures. For instance, if a test produces timeout flakiness, Chrome developers might make timeout as an expected result. Consequently, if this test time out in the next builds, it does not cause multiple test reruns and build breakage. Meanwhile, other types of failures such as crash can result in multiple test reruns and build breakage.

Based on the expected results, the final test outcome is determined as follows.

- Expected: if the result of a test is the same as its expected result, it is categorized as an expected result. Expected results do not block a build, even if they are a failure.
- Flaky: if a test fails non-deterministically, it is a flaky test. A flaky test has multiple different outcomes as a result of running the test multiple times on the same build. It might also lead to multiple expected results in the future.
- Unexpected: if a test fails multiple times unexpectedly, its result is considered an Unexpected outcome. This type of result will usually break the build.

Table 2.1 displays a summary of the final test outcomes, and Figure 2.2 shows a decision tree for how the decisions are made and how the final outcome is concluded based on the test results and the expected results. An allowable set of expected results is created for a test based on developer's expectations and may include: Pass, Skip, Fail, Timeout, Crash. If the outcome exists inside the list, it is considered Expected; otherwise, it is an unexpected outcome. Unexpected pass is automatically categorized as Flaky, but unexpected failure is automatically retried n times. If the result is still an unexpected failure after n retries, the test is categorized as Unexpected. If any of the retries results in an expected result, *e.g.,* a Pass, then the test failure is considered flaky and is non-blocking.

Table 2.1: Final test outcomes for CHROME

Category	**Actual Result**
Expected	Expected failure / Pass
Flaky	Multiple different results / Unexpected pass
Unexpected	Unexpected failure

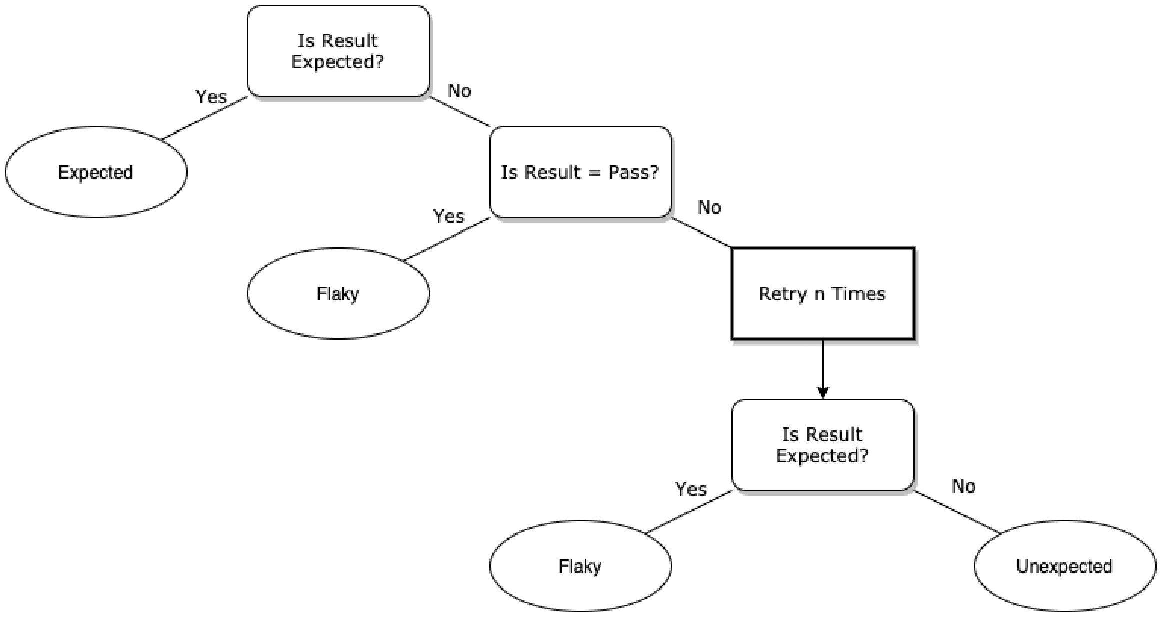


Figure 2.2: A decision tree for how chrome testing infrastructure decides and determines the test outcomes.

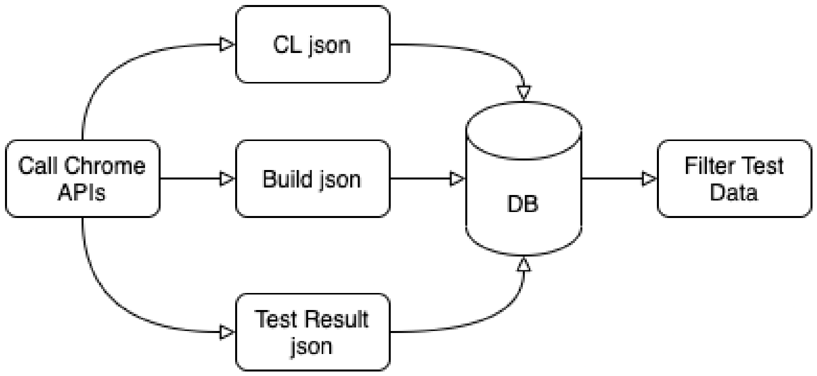


Figure 2.3: Data processing pipeline

2.2.3 Data Processing Pipeline

Figure 2.3 shows the processing pipeline that we developed to gather data. It has the following steps:

(1) We called the Chrome Gerrit code review [15] APIs with a date range from 2021-01-01 to 2021-01-31 for the CHROME pre-submit testing infrastructure.

(2) We captured and stored the JSON raw data of change lists, builds, and tests.

(3) Since we are more interested in the testing results, we then filtered out the data to capture testing data for a particular builder, and it includes 276,550,812 test run verdicts for the entire period.

(4) This filtered data consists of test id, test name, test suite, build id, build start time, build end time, result id, status, final result, and test duration. We release this data in a replication package [5].

2.2.4 Descriptive Statistics

For the captured change lists and builds we have the following information:

- There are 9524 change lists, and an average of 307 change lists per day.
- There are 19,045 builds, and an average of 614 builds per day.

- On average, it takes about 4.06 days for a change list on the Gerrit review to be submitted to the main repository.
- There are 49,932 test suites
- There are 276,550,812 test cases for the entire period and an average of 8,920,993 per day.

2.3 Background on Test Prioritization Algorithms

Many projects deal with a large number of tests on a daily basis. To optimize the testing process, studies have proposed various test prioritization techniques. These approaches aim to increase testing performance and reduce feedback time to developers. One of the most cost-effective test prioritization methods has been the usage of historical test data. In the following, we review two historical based test prioritization techniques which are used in our experiment.

2.3.1 RA 1. Individual test failures within a single build

At the early stage of software development and release, the build frequency was low, but the number of tests within each build was increasing which lead to late feedback delivery to developers. As a result, some studies proposed test prioritization with the goal of prioritizing tests within each build to detect failing tests sooner [16, 17]. With this goal, some test prioritization approaches were suggested based on code coverage and the dependency of tests to the changing code [18, 19]. Meanwhile, Kim and Porter were pioneers in proposing the usage of historical test failure information to prioritize tests [8].

Since their approach does not rely on the source code dependencies, it performs more efficient than the other techniques depending on source code. The priority in the Kim and Porter algorithm is given to tests with the most recent failures.

To determine the prioritization score they adopted the exponential moving average formula from statistical quality control as in the formula 7.

$$\begin{aligned} &P_0 = 0 \\ &P_k = \alpha * h_k + (1-\alpha) * P_{k-1},\ 0 \leq \alpha \leq 1,\ k \geq 1 \end{aligned} \tag{1}$$

Where P_k is the probability of a test to fail at the kth observation based on its prior failures, and α is the smoothing constant adopted to weight each individual historical observation. A larger α makes recent observations more influential in the probability score. h_k shows whether the test has been failed at the kth observation, and it is assigned 1 if it failed and 0 if it passed. For the value of P_0, we followed the correction that was made by Mattis *et al.* [14] as there is no observation at P_0. We re-implement the Kim and Porter algorithm as one of the most effective test prioritization techniques, and we refer it as KIMPORTER algorithm in our study. We show the pseudocode implementation of KIMPORTER in Algorithm 3, and release our code in the replication package [5].

Algorithm 1: KIMPORTER

```
Result: KIMPORTER takes tests from 1 build into the $dispatchQueue$ ordering tests inside
        the build based on their previous failures.
1  while There are more builds do
2      fill $dispatchQueue$ with tests from 1 build;
3      calculate $p_k$ for each test inside the queue;
4      prioritizeTests(dispatchQueue);
5      while $dispatchQueue$ is not empty do
6          test = $dispatchQueue$.getNextTest();
7          result = run(test);
8          if result == failure then
9              update $h_k$ for the test
10         end
11     end
12 end
```

The Kim and Porter approach relies on the idea that tests that fail frequently in the past will fail in the future. Furthermore, usually only a few tests fail, which makes this prioritization approach even more effective. If we run this small set of frequently failing tests first, finding the first failure of a build should be more efficient.

2.3.2 RA 2. Individual test failures across the CI builds

Continuous integration has become a dominant release engineering practice, and the increased number of change requests has increased the number of builds and test runs. In a sample GOOGLE dataset, the number of test requests reaches up to 2,461 per minute [10]. As a result, they have adopted multi-machine and multi-release environment to run their enormous number of tests simultaneously [1]. This in turn has brought the concept of running and prioritizing tests *across* builds into practice.

Elbaum *et al.* introduce this challenge, and they propose a test prioritization approach which uses the same idea as Kim and Porter, but they adapt it across the builds in Google's continuous integration environment [1].

They used three time windows for implementing their algorithm. w_p as a time window for tests to be prioritized, w_f as a time window for counting the number of previous failures, and w_e as a previous execution time window to prevent low-priority tests from starvation. We re-implement their approach in Listing 2.

Algorithm 2: ELBAUMPRIORITIZATION

```
Result: ELBAUMPRIORITIZATION algorithm takes tests from b builds into the
        $dispatchQueue$ and prioritizes them across the builds based on three conditions:
        1. The test has recent failures 2. it is new 3. it has not been run for a long time.
while There are more builds do
    fill $dispatchQueue$ with tests from the next b builds;
    foreach $t_i \in dispatchQueue$ do
        if $t_i \in w_f$ or $t_i \notin w_e$ or $t_i$ is new then
            $t_i.priority = 1$
        else
            $t_i.priority = 0$
        end
    end
    prioritizeTests($dispatchQueue$);
    while $dispatchQueue$ is not empty do
        test = $dispatchQueue$.getNextTest();
        run(test);
    end
end
```

Elbaum *et al.*'s approach has the same reliance on historical test failures as the KIMPORTER algorithm, but it assumes multiple machines are testing builds simultaneously. The ELBAUMPRIORITIZATION algorithm can perform more effectively in the continuous integration environment by prioritizing failing tests across builds and running them upfront.

2.4 Evaluation Setup

To find the difference in the performance of the prioritization algorithms under study, we require metrics that can show how successful these prioritization algorithms are in finding failing tests faster in comparison to their competitor algorithms.

In this work, we are interested in evaluating the overall change in test run time at a global level where the same test may be run multiple times. Despite the dominant APFD metric [20], Elbaum *et al.* as one of the proponents of this metric for test prioritization [18] find it inappropriate to use in the CI and large scale environment [1] because it does not consider tests *across* builds. To deal with this issue, Elbaum *et al.* introduced the gain in hours for approach, A, relative to TESTALL as defined as:

$$\text{GAINEDHOURS (A)} = \text{FailTime}(\text{TESTALL}) - \text{FailTime}(A) \tag{2}$$

We calculate the differences between the run time of each individual failing test detected by the KIMPORTER, and ELBAUMPRIORITIZATION algorithms in comparison with TESTALL which is the FIFO time order of test for each build request. This will lead us to achieve the time gained for each failing test by each algorithm. To have a better insight about the distribution of the time gained by each prioritization algorithm, we use violin plots. The plot displays the median gain, first and third quartiles, outliers as well as the density distribution of data.

2.4.1 Evaluation Parameters

In this study, we configure our algorithms to run with the following parameters.

For the KIMPORTER algorithm, we consider the α = 0.8 as in [14] to give 80% of the priority to the most recent failures. For the ELBAUMPRIORITIZATION algorithm, we adopt prioritization windows based on time or the number of tests [1]. We determine the prioritization windows to be a set of builds. Therefore, we set w_p to include 2 builds, w_f to consist of 48 builds, and w_e to have 96 builds. We determine these numbers relative to the time windows chosen in the original Elbaum *et al.* paper [1].

2.5 Results and analysis

In this section, we aim to answer the following research questions:

RQ1: What is the impact of flaky tests on history-based test prioritization algorithms?

RQ2: What is the rate of flaky failures and how repetitive are they relative to non-flaky failures?

RQ3: How are flaky and non-flaky failures distributed over time in Chrome?

2.5.1 RQ1: Impact of Flaky Tests on Test Prioritization

We apply the presented prioritization algorithms to the CHROME dataset in two scenarios.

First, we run the test prioritization algorithms on the 276 million test cases given the condition that flaky failures are blocking like other types of failures, and we refer it as Blocking Flaky Failures (BFF) scenario. Then, we run the same test prioritization algorithms on the same dataset considering the flaky failures as non-blocking and we refer it as Non-Blocking Flaky Failures (NBFF) case. The latter, non-blocking flaky failures, is how Chrome developers actually treat flaky failures (see Section 2.2.2).

Figure 2.4 is the violin plot showing the distribution of the GAINEDHOURS of all failing tests as a result of running the KIMPORTER and ELBAUMPRIORITIZATION test prioritization algorithms against the TESTALL algorithm in the BFF scenario. This plot shows the overall performance of these prioritization algorithms. The solid vertical line in the density plot shows the median, and the dashed vertical lines display the quartiles in the distributions.

As can be seen in this plot, the first quartile is at 0.37 hours for the KIMPORTER algorithm, which means that 75% of failing tests are detected at least 22.2 minutes faster. The median is at 3.07 hours, and the third quartile is at 6.85 hours. In the ELBAUMPRIORITIZATION approach, the first quartile is less than KIMPORTER algorithm, near zero. However, the median and the third quartile are higher than KIMPORTER algorithm at 5.66 hours and 10 hours respectively. In KIMPORTER

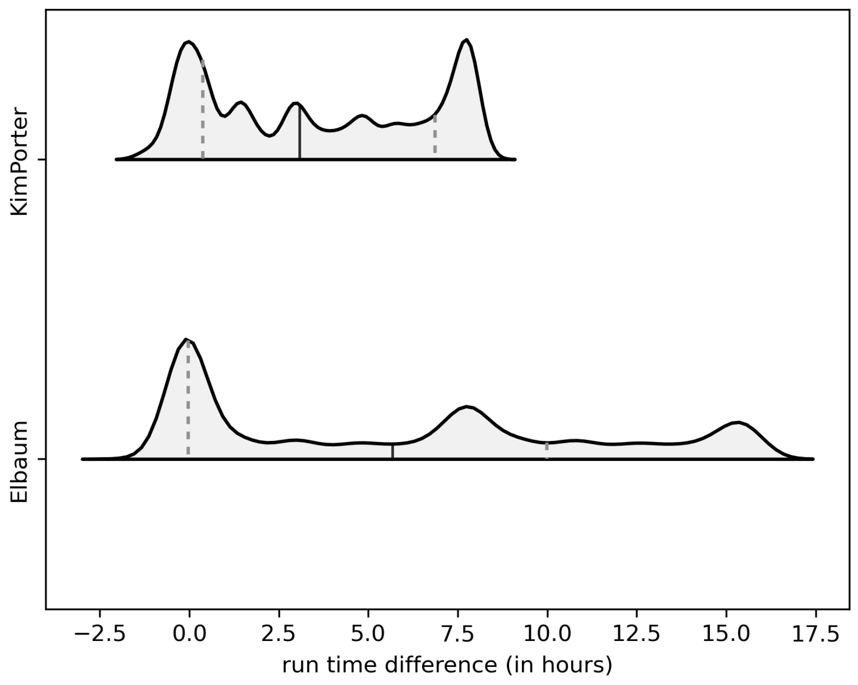


Figure 2.4: Violin plot showing the GAINEDHOURS of failing tests in KIMPORTER, and ELBAUMPRIORITIZATION algorithms against TESTALL in the BFF scenario. The solid vertical line in the density plot shows the median, and the dashed vertical lines display the quartiles in the distribution.

and ELBAUMPRIORITIZATION algorithms there are about 18%, and 25% of failing tests that are delayed in comparison to TESTALL respectively. However, the median, quartiles, and most of the body of the violin plots for both KIMPORTER and ELBAUMPRIORITIZATION algorithms are on the positive sides of the distributions. This means that for most of the tests, these algorithms were able to detect the failures faster in comparison to TESTALL algorithm in the BFF scenario.

Figure 2.5 shows the violin plot distribution of the GAINEDHOURS by all failures in the NBFF scenario from running the KIMPORTER and ELBAUMPRIORITIZATION test prioritization algorithms against TESTALL approach. In the KIMPORTER approach, the first quartile is near zero which means that 25% of failures are detected by delay in comparison to TESTALL. The median is at 0.53 of an hour, and the third quartile is at 3.72 hours. For the ELBAUMPRIORITIZATION algorithm, the first quartile and the median are near zero, which means that the algorithm delayed half of the failures, and runs the other half faster. The third quartile is at 3.8 hours in GAINEDHOURS.

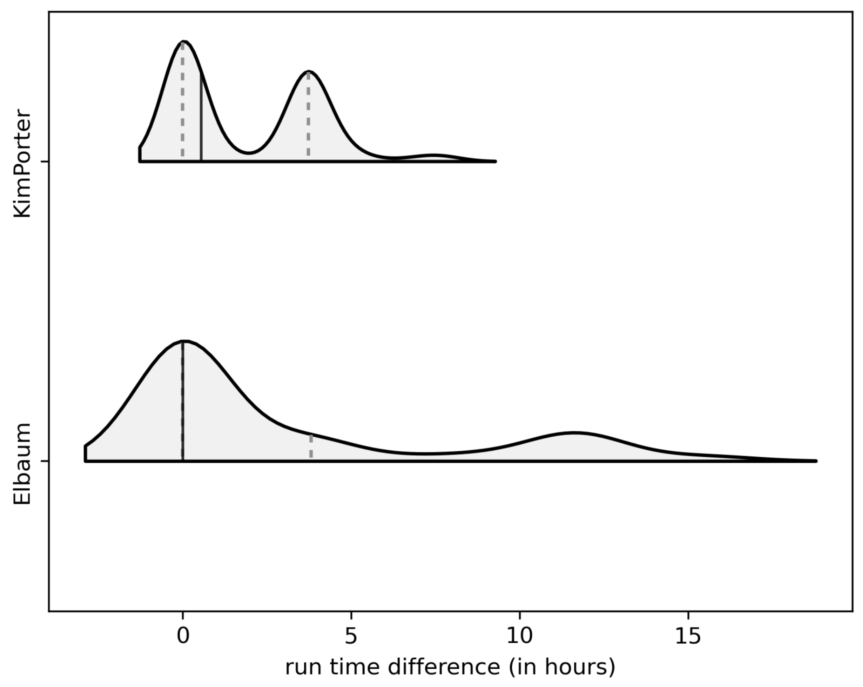


Figure 2.5: Violin plot showing the GAINEDHOURS of failing tests in KIMPORTER, and ELBAUMPRIORITIZATION algorithms against TESTALL in the NBFF scenario. The solid vertical line in the density plot shows the median, and the dashed vertical lines display the quartiles in the distribution.

It is obvious in the violin plots of both KIMPORTER and ELBAUMPRIORITIZATION algorithms in the NBFF scenario that the median, quartiles, and the body of these violin plots shifted toward zero in comparison to Figure 2.4. This means that the performance of these prioritization algorithms deteriorated significantly when the flaky failures are considered as non-blocking failures.

Tables 2.2, and 2.3 display a summary of the GAINEDHOURS and the percentage of failures that are delayed by the KIMPORTER and ELBAUMPRIORITIZATION test prioritization algorithms against TESTALL approach in both BFF and NBFF scenarios.

RQ1: History-based test prioritization algorithms are ineffective when flaky tests are present.

In the following, we investigate the distribution of different types of failures in the Chrome dataset to better understand this phenomenon.

Table 2.2: The GAINEDHOURS and the percentage of delayed failures for the KIMPORTER algorithm in comparison with TESTALL in the BFF and NBFF scenarios in 25th, 50th, and 75th percentiles.

	25th	50th	75th	% Delayed
BFF	0.37	3.07	6.86	18
NBFF	0.00	0.54	3.72	25

Table 2.3: The GAINEDHOURS and the percentage of delayed failures for the ELBAUMPRIORITIZATION algorithm in comparison with TESTALL in the BFF and NBFF scenarios in 25th, 50th, and 75th percentiles.

	25th	50th	75th	% Delayed
BFF	0.00	5.66	10	25
NBFF	0.00	0.00	3.8	50

2.5.2 RQ2: Rate and repetitiveness of Failures

One of the important statistics we have to consider is the rate of each type of failures. For the entire period we have 276,550,812 test cases. Out of all these test cases there are 100,456 failures, and among all failures we have 416 non-flaky failures and 100,037 flaky failures. This means that the rate of non-flaky failures among failures is 0.41%, whereas the rate of flaky failures among all failures is 99.58%. It is obvious that the majority of failures are flaky failures in CHROME, *i.e.* on failure, a build passes on one of its reruns.

Another important factor we have to consider is how repetitive flaky failures are in comparison to non-flaky failures. History-based test prioritization algorithms perform rely on repetitive past failure predicting future failures. To measure the concentration of test failures, we use the Lorenz plot and Gini coefficient, which were designed to measure the concentration of wealth in a population.

Figure 2.6 shows the Lorenz plot for the distribution of the non-flaky failures over tests containing non-flaky failures versus the Lorenz plot for the distribution of the flaky failures over tests consisting of flaky failures and also perfect equality. The non-flaky Lorenz curve displays the cumulative growth in the number of non-flaky failures regarding the cumulative growth in the number of tests having non-flaky failures, and the flaky Lorenz curve shows the cumulative growth in the number of flaky failures considering the cumulative growth in the number of tests containing flaky failures. The straight line shows the perfect equality line, where each test passes and fails the same

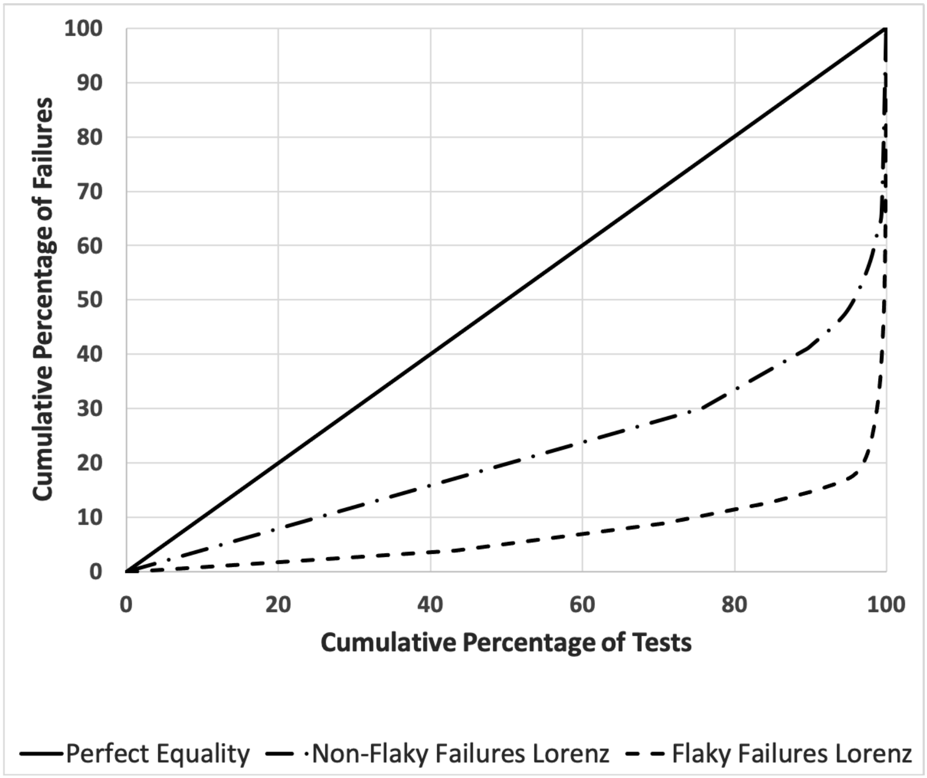


Figure 2.6: Lorenz plot showing the cumulative growth in the number of non-flaky failures regarding the cumulative growth in the number of tests containing non-flaky failures versus Lorenz plot displaying the cumulative growth in the number flaky failures considering the cumulative growth in the number of tests consisting of flaky failures and perfect equality on CHROME dataset.

number of times. A larger area between a curve and the equality line results in a larger Gini coefficient value, which indicates that the distribution is more skewed. The Gini coefficient value for the non-flaky Lorenz plot is 0.57, while the Gini coefficient for the flaky Lorenz plot is 0.86.

As we can see, the Gini coefficient regarding the distribution of the flaky failures over the tests containing flaky failures is about 30% more than the Lorenz plot showing the distribution of non-flaky failures over the tests consisting of non-flaky failures. This reveals that flaky failures are much more unevenly distributed, with a smaller set of tests failing frequently. In Figure 2.6 we can see that about 80% of flaky failures are concentrated on only 3% of the tests containing flaky failures; whereas 80% of non-flaky failures are distributed among about 50% of tests consisting of non-flaky failures. This significant concentration of flaky failures over a minority of tests increases the repetition of these flaky failures which in turn improves the distorts the effectiveness of test prioritization algorithms that consider flaky failures.

The rate of repetition in different types of failures also confirms this phenomenon. Out of 165 tests containing non-flaky failures there are only 40 tests consisting of repetitive non-flaky failures, which stands for 24%. This low percentage of repetitive non-flaky failures can be one of the reasons for the poor performance of test prioritization algorithms in the NBFF scenario. On the other hand, from 9119 tests consisting of flaky failures about 57% are repetitive, which supports the strong performance of the prioritization algorithms in the BFF case.

RQ2: The rate of flaky failures is much higher than non-flaky failures, and they are much more repetitive than non-flaky failures.

2.5.3 RQ3: Failure Distribution Over Time

One of the other aspects which matters for the performance of test prioritization algorithms is the repetitive occurrence of the failing tests over time. Since they rely on the historical failures, the more repetitive the failures are over time, there is a higher chance that the failures be detected faster.

Figure 2.7 displays the number of repetitive non-flaky failures for the whole month of January of 2021 on a daily basis. We can see in this chart that the number of repetitive non-flaky failures is very limited ranging from 0 to 26 daily. What is also important in this figure is that 10 days in the whole month which stands for 32% of days in the month lack repetitive non-flaky failures. The existence of very few non-flaky failures over time can dramatically impact the performance of the test prioritization algorithms.

Figure 2.8 shows the number of repetitive flaky failures for the whole month of January of 2021 on a daily basis. The number of repetitive flaky failures in this period ranges from 164 to 8047 per day, and there is no day without repetitive flaky failure. As we can see, in contrast to repetitive non-flaky failures, repetitive flaky failures are numerous for the whole period. This can significantly improve the performance of the history-based test prioritization algorithms.

RQ3: Flaky failures occur repetitively over time, while there are few repetitive non-flaky failures.

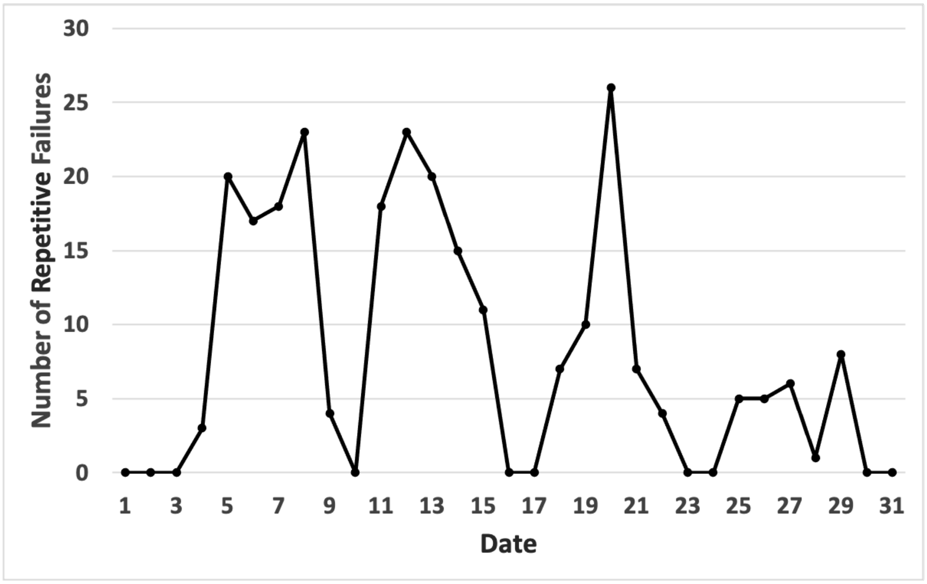


Figure 2.7: Time series showing the repetitive occurrence of non-flaky failures over time for the January of 2021 on a daily basis on the CHROME dataset.

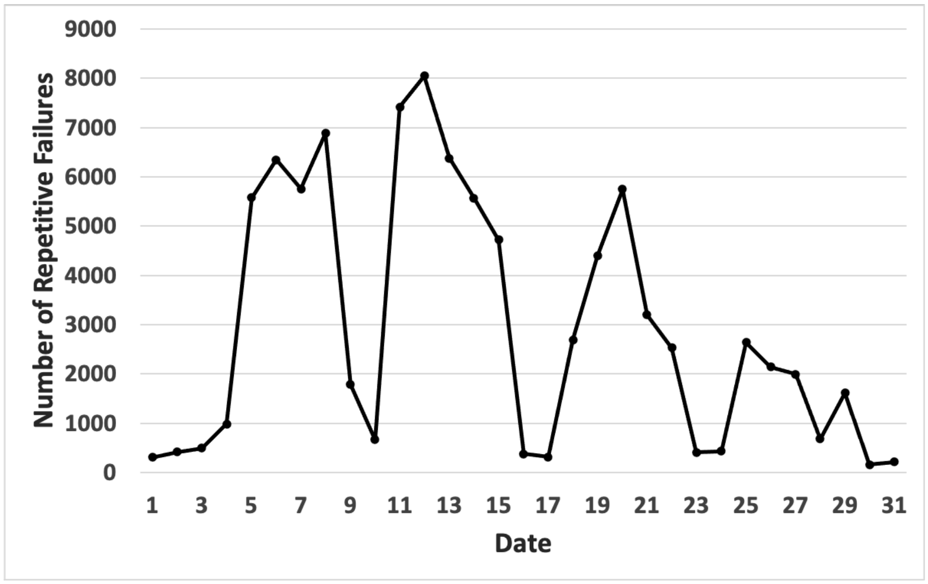


Figure 2.8: Time series showing the repetitive occurrence of flaky failures over time for the January of 2021 on a daily basis on the CHROME dataset.

2.6 Discussion

In the previous section, we saw that once the flaky failures are acknowledged as non-blocking as Chrome does, there are few non-flaky test failures, and the history based test prioritization algorithms are ineffective. We see that the ELBAUMPRIORITIZATION algorithm becomes ineffective with the median GAINEDHOURS of 0, and the KIMPORTER approach is much less effective.

One other question to answer is that whether there are any benefits in prioritizing flaky failures. Although there might not be a clear answer to this question in the literature and there is a need for further study on this matter, according to the testing pipeline in Chrome, we can say that the moment a test is detected as a flaky failure it is ignored and, it will not block the build. There are long-term procedures that CHROME testing infrastructure follows to find the cause of the flakiness and to fix it. However, there is not an immediate treatment for the flaky tests, and the goal of fast feedback might not be the case for flaky failures as they do not have any influence on the destiny of a build.

We can conclude our findings as follows. We saw that the test prioritization algorithms can become completely ineffective when flaky failures are admitted as non-blocking. We also showed that the rate of flaky failures are much higher than the non-flaky failures, and they are 99.58% of failures in Chrome. Moreover, flaky failures are much more repetitive than non-flaky failures as their failure distribution is more towards a minority of tests. Failure time series also display flaky failures as being well distributed over time, while non-flaky failures are not. All of these convince us that the test prioritization algorithms under the study had mostly prioritized flaky failures in the first place. This is due to the fact that most of the failures are flaky failures, and they have a much more repetitive nature than non-flaky failures.

The implications of this finding questions the efficacy of prior studies, that did not separate flaky failures from other types of failures, *e.g.,* Elbaum *et al.* [1] on the Google dataset.

Historical test prioritization algorithms prioritize flaky failures as they fail repetitively.

2.7 Threats to Validity

2.7.1 External Validity

The focus of this study is on the CHROME testing data which we scraped from the CHROME website. As far as we know, there is no other publically available dataset consisting of the large number of tests as this dataset, and more importantly, having flaky labels. As a result, this dataset has been the only one we were able to work on, and we release it in our replication package for other researchers [5]. We hope developers will evaluate the impact of flaky tests failures on test case prioritization on other industrial projects.

This study concentrates on the performance of test prioritization algorithms using the historical test failures. Many other test prioritization techniques such as code coverage based techniques are not suitable and practical for the large datasets like CHROME [2]. It is also impractical to rerun tests without the massive infrastructure available to Google Chrome. We hope that future work will the impact of flaky tests on other types of test case prioritization.

2.7.2 Construct Validity

The metrics we use in this study are the time differences between the actual test runs in TESTALL order and the order suggested by our KIMPORTER and ELBAUMPRIORITIZATION algorithms. We simply plot the violin plot for each test and show the distribution. We believe that these measures are effective enough for the purpose of displaying the performance of prioritization algorithms on the CHROME dataset. We also investigated the APFD metric [20] as one of the dominant test prioritization measures; however, as Elbaum *et al.* [1] mentioned in their study, this metric does not work properly in the continuous integration environment where the focus is on the performance of detecting failures across builds.

2.7.3 Internal Validity

This work shows that the test prioritization algorithms gain their performance advantage from the existence of flaky tests. Test prioritization algorithms rely on the repetitive failures over time, and this aligns well with the nature of flaky tests being repetitive. Consequently, as the results also confirm, test prioritization algorithms mostly prioritize flaky tests. We hope future work will look at other factors relevant to large-scale CI systems.

2.8 Related Works

Many test prioritization techniques have been proposed to optimize the regression testing process and deliver faster feedback to developers [21–24]. At the early stage of regression test prioritization, the literature offers test prioritization within a single build along with the software release environment of the time. Kim and Porter [8] are the first to consider historical test failures in test prioritization. They take advantage of the exponential moving average formula to increase the weight assigned to recent test failures in their test prioritization approach. In a more recent work, Marijan *et al.* [25] explain the need for having a short feedback loop in the continuous integration development cycle. They propose a system that prioritizes tests considering the time limitation to expose the most failures.

On the other hand, big companies like Google are facing a more complex problem. They receive multiple change requests simultaneously, which requires a more advanced test prioritization. Elbaum *et al.* [1] acknowledge this problem and devise an algorithm to prioritize tests across continuous integration's builds on a sample Google dataset. They propose a waiting queue in which tests from various builds are lined up for prioritization. They adopt some time windows to control prioritization, and starvation. Most studies assume that tests are independent. However, Zhu *et al.* [10] use the previous co-failure test results to prioritize them and also re-prioritize tests once a failure occurs. They use multiple queues to control starvation and compare their algorithm with the Elbaum *et al.*'s approach by using GOOGLE and CHROME datasets. This work is based on the idea that test failures might occur together. Najafi *et al.* [26] factors in the time to run a test, longer running tests get lower priority. They design their prioritization algorithm based on the historical test failure frequency, test failure association, and the cost of the testing process. They apply their algorithm to the multi-request test environment in Ericsson. None of these prior works considered whether the failure was a flake or a true failure.

Various datasets are used for the purpose of test prioritization. Elbaum *et al.* [1] provided a sample from Google test results, which contains 3.5 million test verdicts for a period of 30 days. There are also some studies which adopted datasets taken from Travis CI to conduct their research about testing [14, 27, 28]. Recently, Matiss *et al.* [14] provided a dataset containing test results belonging to 20 Travis CI Java projects. They published their dataset to be used for test prioritization studies. Our observation shows that many studies use small projects for test optimization and flaky tests. However, the problem of test optimization and flaky tests are more relevant to the massive projects such as CHROME that have resource constraints. Another important issue is that because most of the previous studies are working on small projects they adopt approaches such as code coverage techniques that are not pragmatic in the large projects [2]. In this research, we captured and evaluated about 276 million test results from the Chrome project which presents the real testing environment in a massive project.

Flaky tests has been the subject of many studies recently [27, 29, 30]. However, there are few studies that worked on the impact of flaky tests on the release pipeline. Rahman *et al.* [31] investigate the impact of failures including flaky failures on the number of crash reports associate with Firefox builds, and they find that builds with more failures end up having higher number of crash reports. Martinez *et al.* [32] study the automatic repair system on the Defects4j project, and they reported flaky tests as one of the challenges that might mislead the automatic repair systems. Peng *et al.* [12] try to improve the IR-based test-case prioritization approaches on Travis CI projects, and they find that flaky tests can impact the test prioritization approaches. However, they do not investigate the reasons behind this impact, and the statistics and distribution of flaky tests. Moreover, most of the previous studies recognize flaky tests by re-runs. Nevertheless, they might fail to simulate the real testing environment and be subjective and unrealistic in number of re-runs and determining flaky tests. For example, Chrome usually re-runs tests twice, while FlakeFlagger baseline [4] reruns tests 1000 times. In this work, however, we are using the real flaky labels discovered by the Chrome testing infrastructure.

2.9 Conclusion

Regression testing is costly in the large continuous integration environments. To ensure appropriate scale, we captured more than 276 million test case data from the Google Chrome project, and we applied the KIMPORTER and ELBAUMPRIORITIZATION historical-based test prioritization algorithms to the dataset by first considering flaky tests as blocking, following prior work [1,8], and then non-blocking, following Chrome's process. These algorithms rely on repetitive past failures to predict future failures. Unfortunately, flaky failures are much more common and repetitive than non-flaky failures, making these algorithms prioritize flaky tests ahead of non-flaky tests.

Chapter 3

Parallel Batch Testing

This chapter is a verbatim copy of the paper that is accepted for publication in Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23) [33], preserving the original content without any modifications. The presentation of the batching algorithms, the idea for the *TestCaseBatching*, and the calculation of the results for Ericsson are credited to Bavand and are included here for context. I reworked the paper for submission and performed major revisions, including adding Chrome to make the results more generalizable.

Abstract

Continuous integration at scale is costly but essential to modern software development. Different test optimization techniques including test selection and prioritization have been proposed to reduce the cost. Test batching is an effective alternative, but often overlooked test optimization technique. In this study, we evaluate the impact of parallelization by varying the number of machines on test batching and propose two new test batching approaches.

We use the default *TestAll* approach as a baseline to examine the impact of parallelism and machine count on feedback time, which is the time taken for test verdicts to be available for each change. Additionally, we re-evaluate the *ConstantBatching* technique and introduce the *BatchAll* algorithm, which adjusts batch size based on the remaining changes in the queue. We also propose

TestCaseBatching, which allows new builds to be added to a batch before all tests are executed, accelerating continuous integration. The evaluations are conducted using test results from Ericsson, a proprietary application, and 276 million test outcomes from the open-source Chrome project. Our assessments focus on feedback time and execution reduction, and we provide access to our scripts and data for the Chrome project.

The results reveal a non-linear impact of test parallelization on feedback time, as each test delay compounds across the entire test queue. *ConstantBatching*, with a batch size of 4, utilizes up to 72% fewer machines to maintain the actual average feedback time and provides a constant execution reduction of up to 75%. Similarly, *BatchAll* maintains the actual average feedback time with up to 91% fewer machines and exhibits variable execution reduction of up to 99%. *TestCaseBatching* holds the line of the actual average feedback time with up to 81% fewer machines and demonstrates variable execution reduction of up to 67%. We recommend practitioners use *BatchAll* and *TestCase-Batching* as both can reduce the number of machines needed for testing. It is crucial to first examine historical data and determine the threshold at which increasing the number of machines minimally affects feedback time to ensure efficient testing and resource utilization.

3.1 Introduction

Testing is both time-consuming and resource intensive. To reduce both resource consumption and provide earlier feedback, test selection has been widely adopted in the industry and extensively studied [34–36]. Test selection's inherent trade-off is that not all tests are run, and some test failures may be missed. On the other hand, test prioritization guarantees that all tests will be run, but those that are more likely to reveal faults will be run first, reducing feedback time on test failures, but not reducing the resource usage in testing [37–39]. In contrast, batch testing which groups builds for testing and bisects on failure is conceptually better than both test selection and prioritization because all the tests are run with less resource consumption and much faster overall feedback times [40,41]. Most changes require similar test sets, and the saving in batch testing is achieved because batching groups changes and tests to reduce the number of redundant test runs among builds. For example, if we are testing four builds in a batch that request the same tests, and the batch passes, we will save three build test executions. However, when a batch fails, a bisection algorithm must be run to identify the culprit build that is causing the failure. Bisection can slow down individual builds, but provided that less than 40% of builds fail, batching and bisection are effective at reducing overall feedback time [41].

Despite its effectiveness, only few studies examine batch testing. Najafi *et al.* [40] studied batching and bisection techniques at Ericsson and found that constant batch sizes can reduce the resource usage, *i.e.* execution time, necessary to run all the required tests by up to 42%. Beheshtian *et al.* [41] evaluated open-source Travis torrent projects and proposed *BatchStop4*, which can reduce build test executions on average by 50%. Bavand *et al.* [42] used a dynamic batch size approach using the historical failure rate of the projects and reduced the execution time by about 47% against their baseline. Previous batch testing works make three assumptions that are unrealistic on large software systems. First, they do not run tests in parallel and implicitly use a single machine. Second, after the failure of a batch, they rerun all tests. This is inefficient because we know which tests failed on a batch, and we do not need to bisect and re-run the passing tests. Third, previous researchers only focus on reducing resource consumption and do not investigate the feedback time outcomes for different batching algorithms.

In this study, we address the limitations of prior work, and study systems with a much larger scale of CI builds and tests: Ericsson and Chrome. To understand the impact of parallelization on testing, we replicate the *TestAll* algorithm which simply runs all tests without applying any batching technique, but unlike prior work, we vary the number of machines available for testing and run tests in parallel. Our outcome measures are the feedback time, *i.e.* wall time per test, and total test execution time reduction for all the tests. We replicate *ConstantBatching* from the Beheshtian *et al.*'s study [41] which uses a constant batch size for batching, and we introduce two novel *BatchAll* and *TestCaseBatching* techniques. *BatchAll* algorithm adapts the batch size to the number of remaining changes inside the queue to batch them all at each time. *TestCaseBatching* approach works the same as *BatchAll* except that it also accepts new changes while running the batch. We adopt different numbers of machines on these batching algorithms and evaluate their performance by using feedback time and execution reduction.

We provide results for the following research questions:

RQ1: How does parallelization affect the feedback time performance of TestAll with varying numbers of machines?

RQ2: How effective is ConstantBatching in terms of feedback time and execution reduction when executed in parallel with varying numbers of machines?

RQ3: How effective is BatchAll in terms of feedback time and execution reduction when executed in parallel with varying numbers of machines?

RQ4: How effective is TestCaseBatching in terms of feedback time and execution reduction when executed in parallel with varying numbers of machines?

Our major contributions to this study are as follows.

- **Parallelism in batching:** we study the impact of parallelization on testing in general and batching.

- **Datasets:** we use two large-scale Ericsson and Chrome datasets which run millions of tests per day and are suitable to run our batching algorithms and apply parallelism.

- **Simulation:** we replicate *TestAll* and *ConstantBatching* algorithms and introduce two novel *BatchAll* and *TestCaseBatching* approaches.

- **Outcome measure:** we use feedback time and execution reduction measures to evaluate the performance of batching algorithms.

- **Results:** we reach the following conclusions.

  (1) The impact of parallelism on the performance testing algorithms exhibits a non-linear relationship.

  (2) *ConstantBatching* with a batch size of 4, utilizes up to 72% fewer machines to maintain the actual average feedback time and provides up to 75% in execution reduction, regardless of the number of machines utilized.

  (3) *BatchAll* maintains the actual average feedback time with up to 91% fewer machines and exhibits variable execution reduction of up to 99% depending on machine utilization and the dataset.

(4) *TestCaseBatching* holds the line of the actual average feedback time with up to 81% fewer machines and demonstrates variable execution reduction of up to 67% depending on machine utilization and the dataset.

3.2 Background and Methodology

We study a proprietary project at Ericsson and the Google-led open source project Chrome. The project we examine at Ericsson tests the software that runs on cellular base stations. In this context, the machines used for testing are extremely expensive and limited in number. Ericsson spends millions on testing infrastructure and still needs to batch tests in order to test all the changes. Test resources are scarce, and managers discuss the tradeoff of buying new test machine resources for having slower feedback or removing more tests through selection and risking failures slipping through.

Testing at Ericsson involves a multistage process, including various testing levels from unit tests to integration tests, before changes are integrated into the released product. Our study focuses on confidence levels 2 and 3, primarily consisting of integration tests, which significantly contribute to the overall testing costs at Ericsson compared to other levels like unit testing. Thus, we capture the test results, execution time, and change timestamps for each test case during integration testing to gain insights and optimize this critical stage.

To generate our sample dataset, we evaluate the integration testing of a project at Ericsson. We capture the test results for the period of six weeks from January to February 2021. For this duration, we observe over 11,000 changes. For confidentiality reasons, we do not disclose other data or aspects of the testing process and use the data to simulate batching scenarios where machines are very expensive and highly utilized leading to strong resource constraints.

Chrome is one of the most popular browsers in the world, and it is representative of a large-scale project. There are millions of tests run each day, and there are a massive number of builds and tests running in parallel. The testing process used by Chrome was described in detail by Fallahzadeh *et al.* [6], we summarize briefly below. A change list, *i.e.* pull request, is committed to the Gerrit code review tool for revision. Reviewers may suggest changes to the change list to improve the code or when they find issues. If the change is satisfactory to the reviewers, it will be sent for testing on the try bot builders. After being approved by the builders, the change is merged into the main Chrome repository.

For the Google Chrome case, we use the Chrome test results published by Fallahzadeh *et al.* [6]. This publicly available dataset is captured by calling the Gerrit code review APIs from Chrome. This dataset consists of 276 million test cases for the month of January 2021 for Chrome. These test runs are for 9,524 changes across 49,932 distinct test suites for an average of 8.9 million test case runs per day. The rate of failures for the builds in this project is about 8.5%. We describe how this data is used in our simulation below.

For both Ericsson and Chrome, the data of interest are the test id, test name, build id, build start time, build end time, status, final result, and test duration. We use these attributes to implement our various batching algorithms which will be discussed in the following. This methodology can be applied to any project that collects this basic data. We release our scripts and data for Chrome in the supplemental material in an anonymized replication package.

3.2.1 Simulation Method and Outcome Measures

In this study, we use the record of real historical test runs and only vary the number of machines to evaluate different batching algorithms. We do *not* re-run any tests. Instead, the process involves determining tests for a batch based on the historical test results of the included builds, selecting the maximum execution time among the corresponding builds for each test to capture the worst-case execution time. The simulated batch is then dispatched across available machines, optimizing test allocation for parallelism and resource utilization, considering the varying number of machines in different scenarios. During the simulation, we assume that a test will fail in the simulated batch if it failed in any of the builds included in the batch, and a test is considered to pass in the simulated batch only if it passed in all the builds. This approach ensures an accurate reflection of test behavior and outcomes in the simulation. We preserve the order of test runs and discuss the limitations of simulation in Section 3.6.

The data necessary to conduct the simulation is not company specific and the simulations can be applied to the test results of the other projects as well. The data includes the code change under test, the time the change was available for testing, the requested tests for the change, and the duration that each test took to run. The changes are then queued based on their arrival time and simulated using the batching algorithm, *i.e.* how efficiently would we have been able to process the same changes and requested tests?

Feedback time: One of the most important factors in designing a continuous integration testing infrastructure is giving fast feedback on test outcomes for each change. The time between committing a change and receiving all test verdicts is defined as feedback time.

$$\text{FeedbackTime} = \text{Time}_{\text{TestVerdicts}} - \text{Time}_{\text{Commit}} \tag{3}$$

For example, if a developer commits a change at 9 am and receives the feedback that the tests passed successfully on that change at 10 am, the feedback time will be 1 hour for that change. In contrast, if the change was queued for 1 hour, then the feedback time would be 2 hours, a doubling in feedback time. In the examples, we use a unit T to represent time, but in the study, we use the actual time each test takes to run. The time a test will be queued depends on the available resources and batching algorithm.

To contrast batching algorithms, we use the *AvgFeedback* as the sum of the feedback times for each change in the project divided by the total number of changes for the project. The equation below shows the *AvgFeedback* for batching algorithm A across C changes with m machines.

$$\text{AvgFeedback}_m(A) = \frac{\sum_{c=1}^{C} \text{Feedbacktime}_c(A, m)}{C} \tag{4}$$

To contrast batching algorithms, we calculate the *FeedbackReduction* as the percentage decrease in *AvgFeedback* of batching algorithm, A1, compared to the *AvgFeedback* of the algorithm, A2, with m machines.

$$\text{FeedbackReduction}_m(A1, A2) = (1 - \frac{\text{AvgFeedback}_m(A1)}{\text{AvgFeedback}_m(A2)}) * 100 \tag{5}$$

Execution Reduction: The EXECUTIONREDUCTION metric quantifies the performance of different batching algorithms in terms of saving execution time with varying numbers of machines. It is calculated using Equation 10.

$$\textsc{ExecutionReduction}_m(A) = 1 - \frac{\sum_{t=1}^{K} \text{Test Execution Time (A, m)}}{\sum_{t=1}^{N} \text{Test Execution Time } (\textit{TestAll})} \tag{6}$$

This formula calculates the execution reduction in test executions achieved by a batching algorithm (A) with m machines compared to running all tests in their original order (*TestAll*). Here, K represents the number of tests executed by the batching algorithm (A) with m machines, and N is the total number of tests executed by *TestAll*. The formula calculates the percentage of time required for running the tests using approach A with m machines relative to *TestAll*. Subtracting this percentage from 1 provides the percentage of execution time that was reduced. For example, if approach A with m machines takes 40% less time to run tests than *TestAll*, the EXECUTIONREDUCTION by A with m machines is calculated as 60%, indicating that approach A with m machines saves 60% of the time in test execution compared to *TestAll*.

By using this formula as a metric, we can compare the performance of different batching algorithms and determine their efficiency in saving time during test execution.

Simulation Setup and Plateau Thresholds. To evaluate the batching algorithms' performance in terms of feedback time, we compare them to the actual average feedback time as the baseline. Due to the complexity and unavailability of the exact number of machines required to achieve the actual average feedback time in both Ericsson and Chrome, we use the baseline number of machines needed to maintain the actual average feedback time for *TestAll*. Our simulations vary the available resources (machines) from 1 to 9 for Ericsson and from 1 to 375 for Chrome, where the average feedback time for all algorithms reaches a plateau.

Determining a plateau often involves the expertise of domain experts who suggest appropriate thresholds. For Ericsson, we establish a threshold of 6 percent improvement from the actual feedback baseline, indicating that beyond this point, the improvement in feedback time becomes insignificant with a unit increase of 1 machine. In the case of Chrome, the threshold is set at 2 percent improvement from the baseline, considering a unit increase of 25 machines. These thresholds are carefully determined to account for the specific requirements and trade-offs in each domain. Ericsson, with its high-cost machines, necessitates a more stringent plateau criterion, while Chrome, benefiting from machine farms, adopts a lower threshold to optimize performance.

3.3 Batching Algorithms

Although parallel testing can help to reduce the feedback time, even at large companies using farms of servers to run tests in parallel, they still need batch changes to further reduce resources [43]. In the following, we describe the definitions and formulations for each batching algorithm.

3.3.1 *TestAll*

Ideally, each change would be tested immediately and in isolation, running all the requested tests independently of other changes. This approach works well on small projects that are not resource constrained. The feedback time for each change varies and depends on the time that a change waits in the queue.

Figure 3.1 describes the testing process for *TestAll* algorithm with 1 and 2 machines scenarios. There are two changes that arrive at $T = 0$ and $T = 1$ times respectively. When there is only one machine available, all tests are running on a single machine and subsequent changes have to wait in a queue. At the time $T = 2$, the tests belonging to the first change are executed with a feedback time equal to 2 units of time. Test execution for change 2 is finished at $T = 4$ resulting in the feedback time of 3 units of time. In the 2 machines scenario, the tests belonging to the changes are distributed between the two machines. At the time $T = 1$ both tests A and B for change 1 are executed by the machines M1 and M2, leading to 1 unit of feedback time. At the time $T = 2$, tests are run for all changes, making the average feedback time of 1 in comparison with the previous average feedback time of 2.5 units of time.

3.3.2 Batch Testing and *BatchStop4*

TestAll is expensive and sometimes infeasible at large companies, *e.g.,* Ericsson [40] or Google [43]. Instead of testing every change individually, we can combine multiple changes and run the union of their requested tests in a batch. If a batch passes, we save resources and provide feedback more quickly. However, if a test fails, we need to find the culprit change(s) that are responsible for the failure. If the intersection of the requested tests for the batched changes is large, and most tests pass, the saving could be substantial. In an extreme example, if we batch 50 changes, and each change

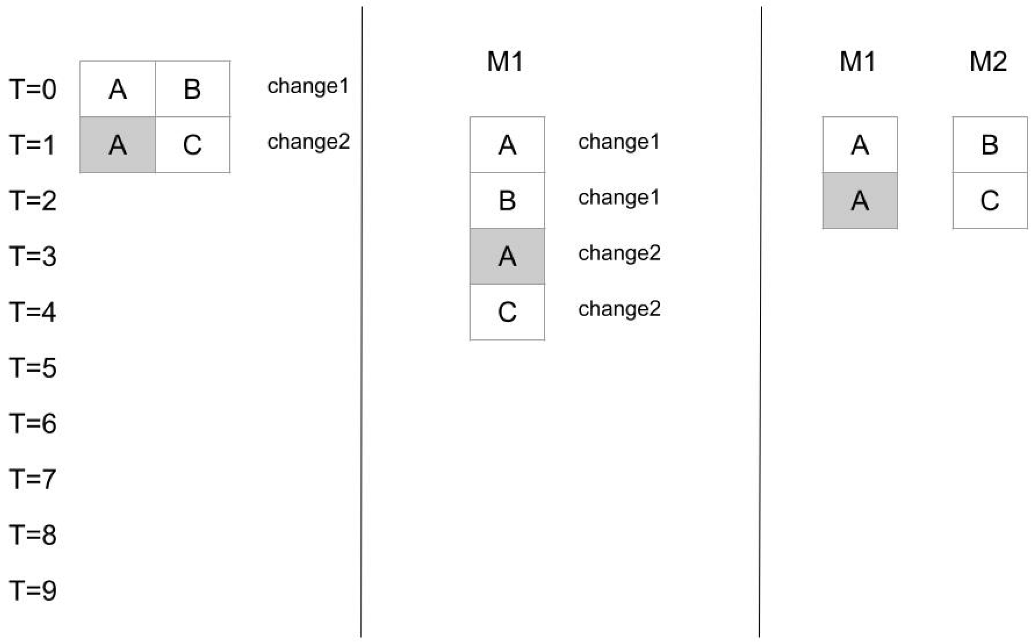


Figure 3.1: A sample of *TestAll* algorithm. In this example, there are two subsequent changes and their requested tests A, B, and C. The sequence of events is displayed by T, and the machines are shown by M.

requests the same tests when the batch passes we save 49 build test executions.

However, when the batch fails, we need to find the culprit change(s) responsible for the failure. Out of different culprit-finding approaches like Dorfman and bisection [40, 44], we use the *BatchStop4* which has been shown both mathematically and empirically to be the top performing approach [41]. Figure 3.2 displays the batching process of 8 changes consisting of different tests with the same execution time. To create the batch, a union of the tests across all the changes is used, which gives 6 distinct tests of A, B, C, D, E, and F to run. Executing the batch fails, leading to the culprit-finding process that requires additional 6 test executions. Since we know that test A failed, we only run this test in the subsequent builds. This ends up running 12 test runs, which is 50% less than the *TestAll* approach that requires 24 test executions.

3.3.3 *ConstantBatching*

Prior works have selected a constant batch size for testing [40, 41]. In the *ConstantBatching* technique, we group n changes together and test them in a batch. For example, with $n = 8$, we batch

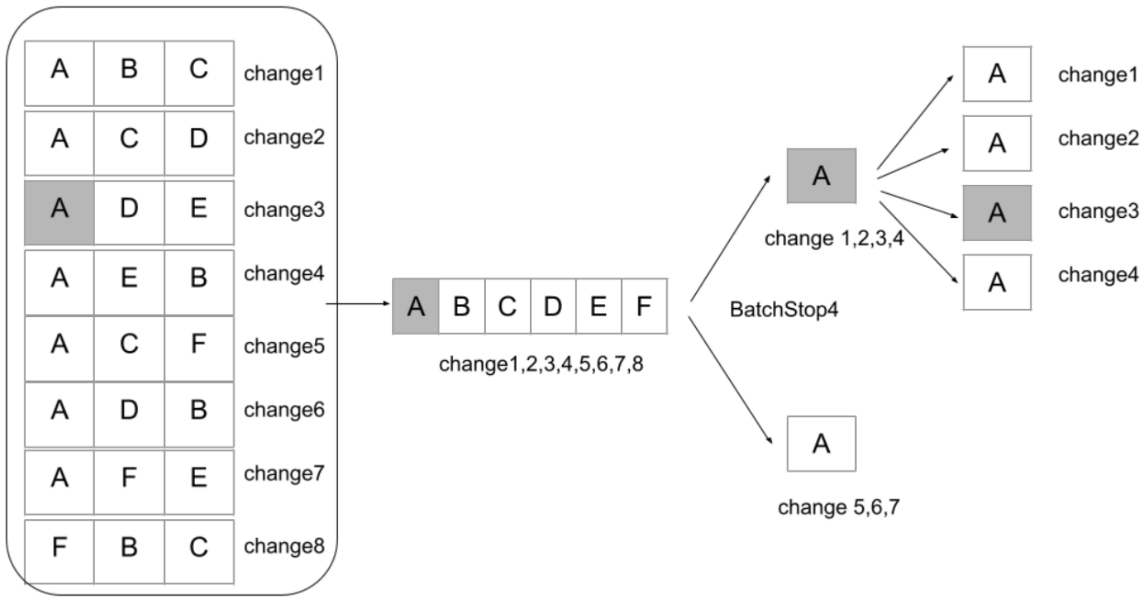


Figure 3.2: A sample of batch testing using the *BatchStop4* culprit approach. In this example, there are eight subsequent changes and their requested tests A, B, C, D, E, and F. The failing test A is determined by a gray colour.

every 8 changes together for testing. Figure 3.3 shows an example of *Batch2* using a single machine as well as having two parallel machines. There are in total 4 changes. The time of committing the changes is equal to T=0, T=2, T=4, and T=6 respectively. For simplicity, we assume each test execution takes 1 unit of time. In a single machine configuration, the feedback time for each build would be 5, 4, 4, and 2 units of time respectively. By using two machines, the feedback time would be 4, 2, 3, and 1 unit of time respectively. We can see when using two parallel machines, there is a time when machines are free and no test has been assigned to them for execution as they have to wait for 2 changes to be available. This affects the feedback time for Change 3.

3.3.4 *BatchAll*

The assumption of a constant batch size introduces problems. First, the rate of committed changes varies over time. For example, during the peak of the workday, there may be 1000's more commits than at night. We need to vary the batch size based on the change queue. In *BatchAll*, when there are resources available, all the waiting changes, are grouped and the union of required tests is run for the batch. When the testing process of the batch finishes and the corresponding resources are

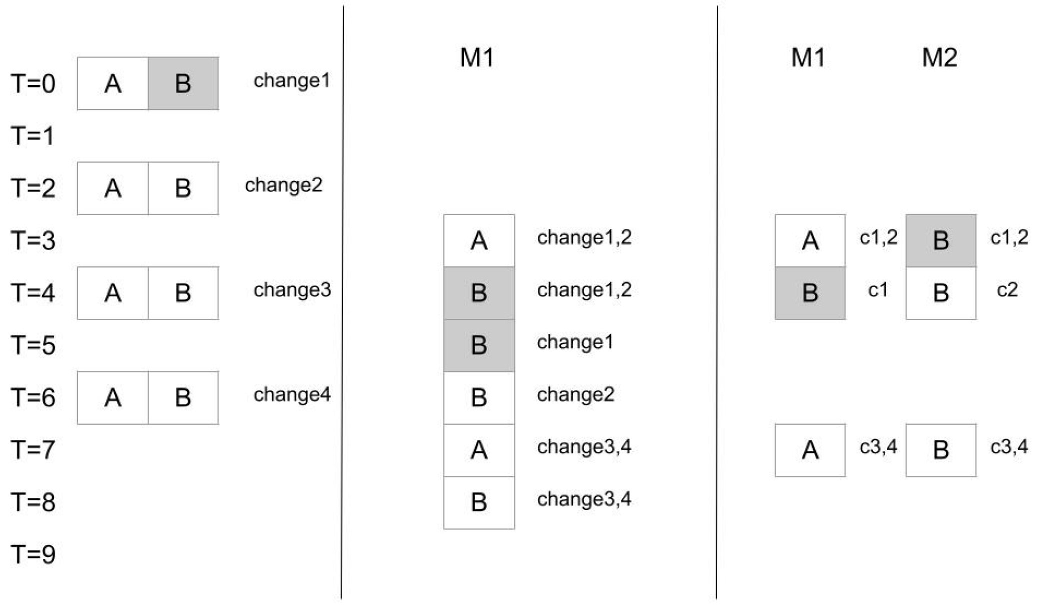


Figure 3.3: A sample of *ConstantBatching* algorithm *i.e. Batch2*. In this example, there are four subsequent changes and their requested tests A, and B. The sequence of events is displayed by T, and the machines are shown by M.

free, another batch is created using all the current waiting changes and the resources are allocated to the new batch.

Figure 3.4 shows an example of *BatchAll* with a single machine. We assume that there is no failure and all the changes pass the tests. Using *BatchAll*, after committing the first change, the resources are immediately allocated to it for testing. Change 1 arrives first, and only after it finishes testing Change 1, it batches all the changes that are now waiting, *i.e.* changes 2, 3, and 4. After testing the second batch of size 3, it runs the tests for Change 5 in a batch of size 1 because no other changes are waiting for testing. The feedback time for each change will be 3, 5, 4, 3, and 5 respectively.

3.3.5 TestCaseBatching

BatchAll can decrease the feedback time by reducing the idle time of resources. However, when all resources are utilized for testing, new changes must be queued until *all the tests* for the current batch are complete. With *TestCaseBatching* new changes are added to the batch when any test finishes rather than having to wait for all the tests to finish. This approach requires the requested

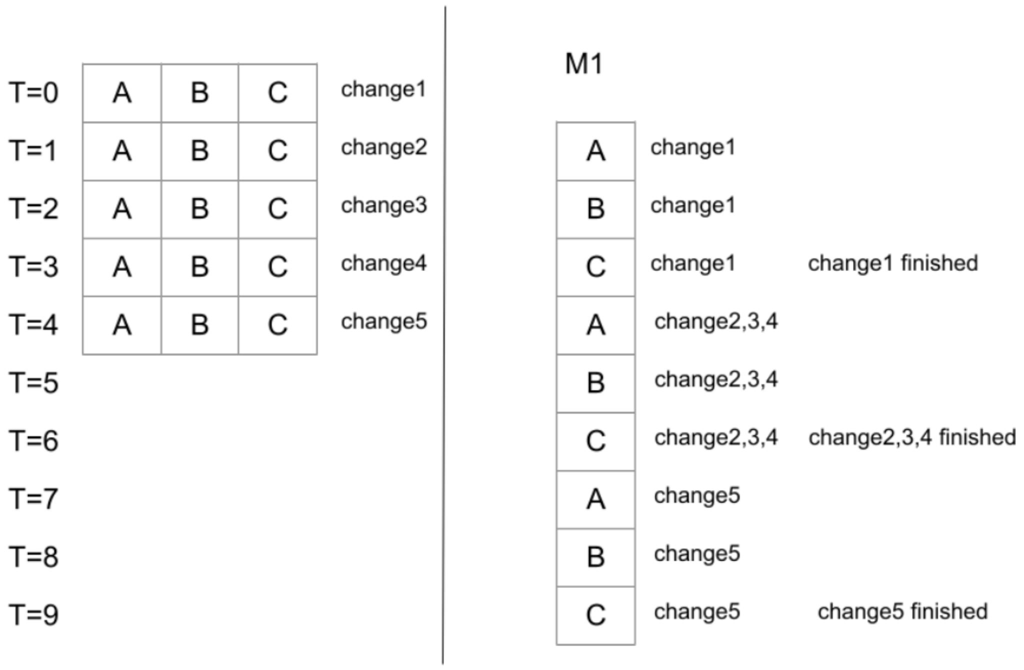


Figure 3.4: A sample of *BatchAll* algorithm. In this example, there are five subsequent changes and their requested tests A, B, and C. The sequence of events is displayed by T, and the machines are shown by M.

tests to be queued. To manage the test queue, the requested test cases for each change are added to the queue, *i.e.* the ChangeID, and TestID. When a test finishes, any new changes are added to the batch and the next test in the queue is run. Once a change has had all its tests run, the results are reported.

In Figure 3.5 we provide an example of *TestCaseBatching*. After each test finishes, *TestCase-Batching* includes any waiting builds and runs the next requested test in the test queue. *TestCase-Batching* has to run Test A three times because it has finished for Change 1 before Changes 2, 3, and 4 and the algorithm has to run it independently for Change 5. In contrast, *TestCaseBatching* must only run B and C twice as they overlap when more changes are available. We see that the average feedback time is reduced to 3 compared to the 4 needed for *BatchAll*, meaning that we get feedback to developers 25% sooner.

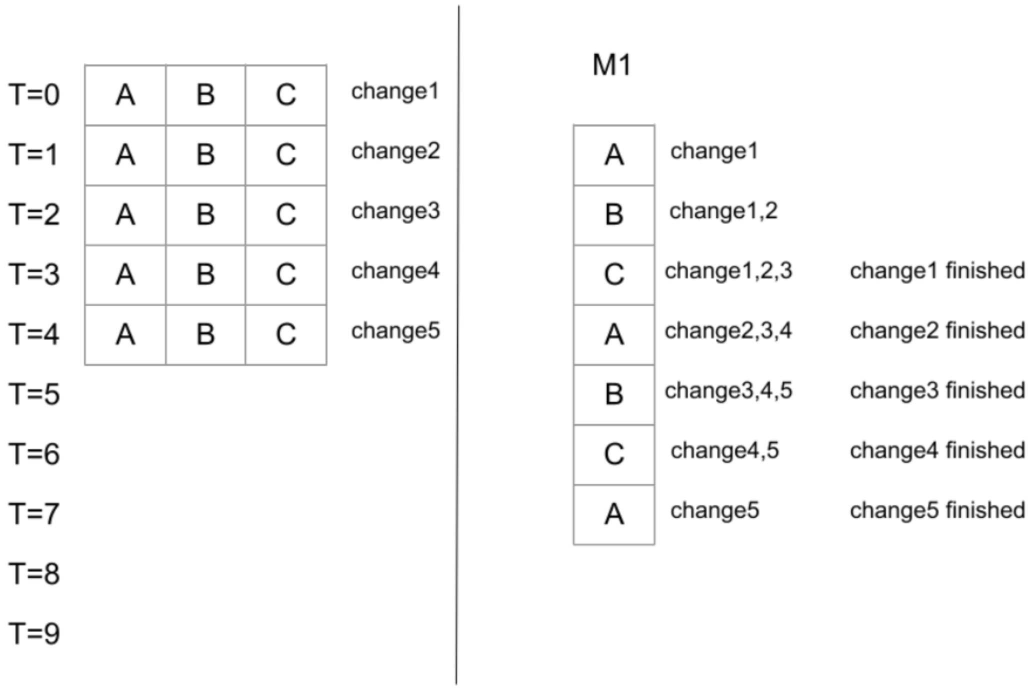

T=0
T=1
T=2
T=3
T=4
T=5
T=6
T=7
T=8
T=9
A B C change1
A B C change2
A B C change3
A B C change4
A B C change5
M1
A change1
B change1,2
C change1,2,3 change1 finished
A change2,3,4 change2 finished
B change3,4,5 change3 finished
C change4,5 change4 finished
A change5 change5 finished

Figure 3.5: A sample of *TestCaseBatching* algorithm. In this example, there are five subsequent changes and their requested tests A, B, and C. The sequence of events is displayed by T, and the machines are shown by M.

3.4 Results

This section presents the results of our evaluation, comparing the performance of different batching algorithms in terms of feedback time, the number of machines utilized, and the extent of execution reduction achieved. We conduct these evaluations using the Ericsson and Chrome datasets, considering various numbers of machines.

3.4.1 RQ1: Parallelization with *TestAll*

In the *TestAll* approach, we simulate testing each change individually to understand the impact of parallelization on the testing process. The curves in Figure 3.6 and 3.7 show that increasing the number of machines has a nonlinear impact on feedback time. We see that *TestAll* needs 7 machines to achieve the actual average feedback time of 8.33 hours for Ericsson. For Chrome, we see *TestAll* needs 217 available machines to achieve the 36.48 minutes average feedback time that is actually observed on Chrome.

Substantial improvement is seen by adding additional parallel machines, but the return on investment diminishes at some point. For instance, in Ericsson, when we increase from 6 to 7 machines there is a speedup of 2.65 times in *TestAll* algorithm, but the increase from 7 to 8 machines only adds 1.5 times of improvement in feedback time. In Chrome, *TestAll* algorithm runs 49 times faster when we use 200 machines instead of 100 machines, while when we increase it from 200 to 300 machines it only gets 8.11 times faster. We see that *TestAll* with 9 machines has plateaued for Ericsson, and we see *TestAll* with 350 machines has plateaued for Chrome.

Figure 3.7 displays the 95% confidence intervals for the average feedback time values of all the algorithms analyzed in Chrome.[1]For the *TestAll* algorithm, the confidence interval range becomes narrower, decreasing from 4% to less than 1% of the actual average feedback time, as the number of machines increases from 200 to 275 and beyond. This reduction in the width of the confidence interval indicates a decrease in the range of the feedback time distribution.

To compare the feedback time distributions of different batching algorithms for Chrome, we utilize the Wilcoxon Rank-Sum test, which is suitable for comparing two independent samples without relying on distribution assumptions. To address the issue of multiple comparisons, we apply the Bonferroni correction by dividing the desired overall p-value of 0.05 by the number of comparisons. This adjustment results in a p-value cutoff of 0.0003 for each comparison, ensuring a stringent threshold for statistical significance.

The obtained p-values are all significantly lower than the cutoff value, indicating significant differences in the feedback time distributions between most algorithm comparisons. However, the comparison between *TestAll* and *Batch2* at 300 machines yields a p-value of 0.06, suggesting a lack of significant difference in only this case.

We also calculate Cliff's delta effect size to measure the magnitude and direction of differences between feedback time distributions. In Table 3.4, we present the effect sizes between various algorithms at different machine counts. This data further supports the observed trend and the significance of the differences depicted in the average feedback time plot shown in Figure 3.7.

[1]We are unable to add confidence intervals or compute statistical tests because we no longer have access to the Ericsson data.

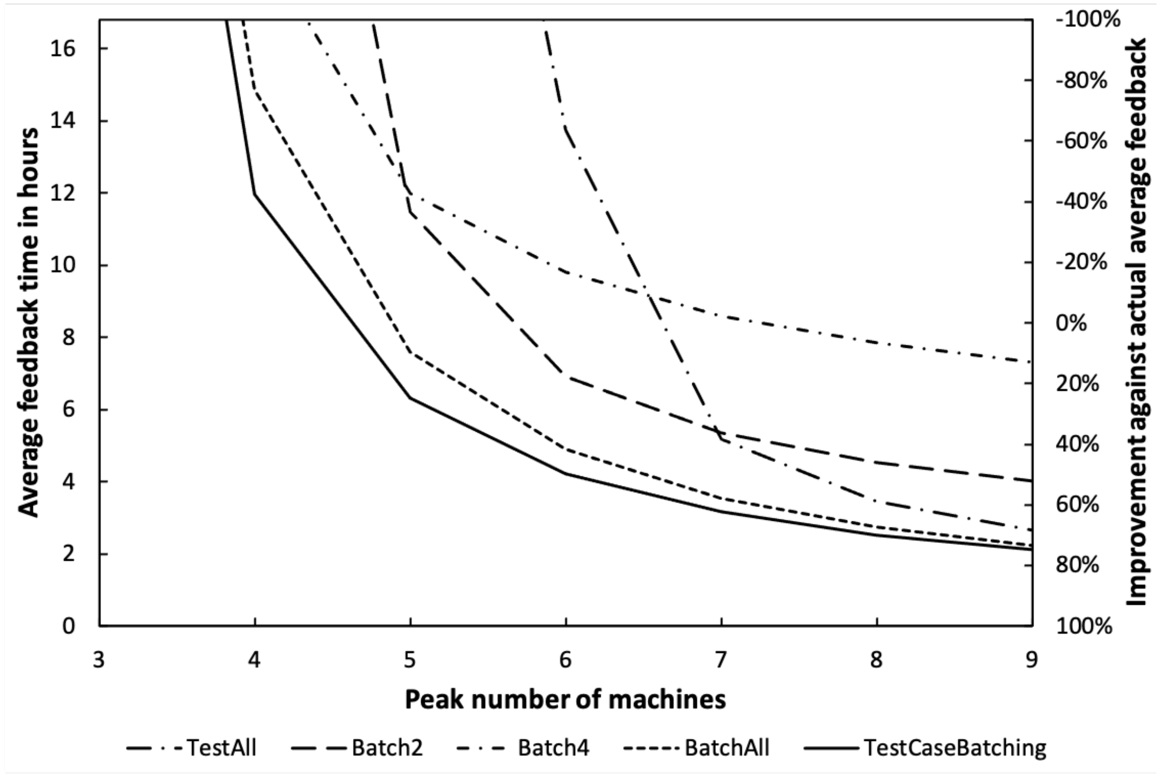


Figure 3.6: Average feedback time and percentage change relative to actual average feedback time for each approach with varying numbers of machines at Ericsson.

3.4.2 RQ2: *ConstantBatching*

Batching reduces the number of test executions when the changes request the same tests. Prior works [40,41] used a constant batch size, and we replicate the state-of-the-art approach on a project at Ericsson. We report the results for batch sizes 2 and 4 in this paper.

When large batches pass, common tests requested by changes are only run one time, dramatically reducing the amount of testing. As noted by prior work [41], the reduction is limited by the number of failing tests because a failure requires additional executions to find the culprit changes. However, on most large projects there are relatively few test failures, making batching highly effective [6].

Prior works have focused on resource savings and largely ignored or simplified feedback time [40, 41]. If we strictly follow a constant batch size, then we see that commits can wait for extended periods of time. Figure 3.6 and 3.7 clearly reveal that constant batching algorithms, *Batch2* and *Batch4*, outperform *TestAll* in the majority of cases, except in highly resource-available environments where

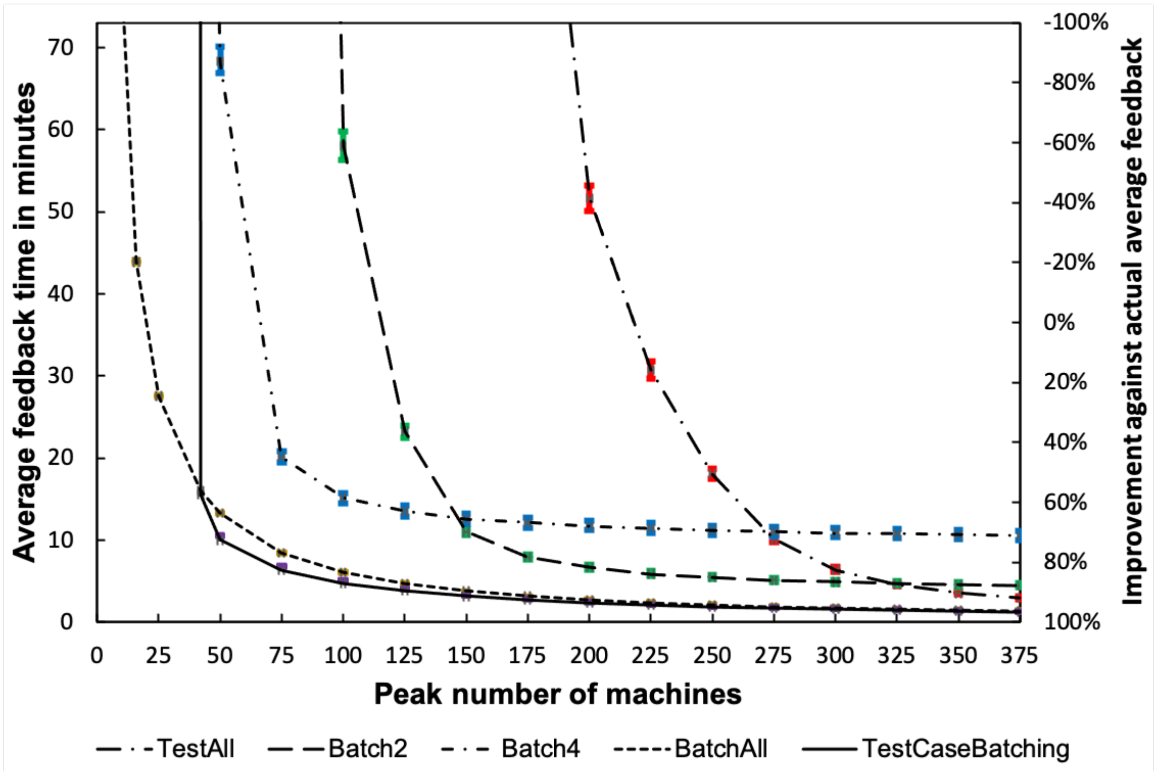


Figure 3.7: Average feedback time with colorful 95% confidence intervals and percentage change relative to actual average feedback time for each approach with varying numbers of machines in Chrome.

the number of machines significantly exceeds the baseline. As the number of resources increase, they plateau with a relatively high feedback time and *TestAll* outperforms them. However, they are simpler to implement than other batching algorithms.

Table 3.1 shows that *Batch2* effectively maintains the actual average feedback baseline of 8.33 hours for Ericsson using 6 machines, resulting in a 14.29% reduction in machine usage compared to the baseline. However, *Batch4* achieves the same average feedback time using 8 machines, resulting in a negative reduction (-14.29%) in machine usage compared to the baseline. For Chrome, *Batch2* can maintain the actual average feedback baseline of 36.48 minutes using 113 machines, resulting in a 47.93% reduction in machine usage compared to the baseline. Similarly, *Batch4* achieves the same average feedback time using 61 machines, resulting in a 71.89% reduction in machine usage compared to the baseline.

Table 3.2 shows that *Batch2* and *Batch4* plateau for Ericsson at 4.52 and 7.86 hours by using

8 machines respectively. This means that *Batch2* and *Batch4* reach a plateau with 70% and 195% longer average feedback time compared to the baseline, respectively. Consequently, increasing the number of machines for both algorithms provides a limited scope for improvement compared to the baseline. For Chrome, Table 4.4 displays that *Batch2* and *Batch4* plateau at 5.88 and 12.6 minutes using 225 and 150 machines. This results in 31% and 255.93% longer plateaued feedback time compared to the baseline, respectively, indicating limited room for improvement relative to the baseline.

The confidence interval values displayed in Figure 3.7 for the *Batch2* algorithm and Chrome reveal that the average feedback time has a range of around 5% when utilizing 100 machines. Notably, as the number of machines increases to 150, this range significantly decreases to less than 1%. Conversely, for the *Batch4* algorithm, the confidence interval values indicate a range of about 5% when using 50 machines. While increasing the number of machines reduces the range, this reduction is constrained, and even with 375 machines, the range remains at 2%.

Figures 3.8 and 3.9 illustrate the percentage of EXECUTIONREDUCTION achieved by various batching algorithms on Ericsson and Chrome, respectively. It is worth noting that the EXECUTIONREDUCTION for the constant batching algorithms remains constant irrespective of the number of machines used. For Ericsson, *Batch2* achieves an EXECUTIONREDUCTION of 23%, while for Chrome, it achieves an EXECUTIONREDUCTION of 50%. Similarly, *Batch4* achieves a EXECUTIONREDUCTION of 37% for Ericsson and 75% for Chrome.

Table 3.1: The number of machines required to maintain the actual average feedback time of 8.33 hours in Ericsson and 36.48 minutes in Chrome, along with the percentage reduction in machine usage compared to the baseline number of 7 machines in Ericsson and 217 machines in Chrome.

Algorithm	***TestAll***	***Batch2***	***Batch4***	***BatchAll***	***TestCase Batching***
Ericsson	7 (0%)	6 (14.29%)	8 (-14.29%)	5 (28.57%)	5 (28.57%)
Chrome	217 (0%)	113 (47.93%)	61 (71.89%)	20 (90.78%)	42 (80.65%)

3.4.3 RQ3: *BatchAll*

ConstantBatching excels in conserving resources and reducing feedback time in highly resource-constrained environments characterized by limited machine availability compared to the baseline

Table 3.2: The number of machines and the average feedback time in hours where each batching algorithm plateaus for Ericsson.

Algorithm	***TestCase Batching***	***BatchAll***	***TestAll***	***Batch2***	***Batch4***
Feedback time	2.53	2.75	2.66	4.52	7.86
Machines	8	8	9	8	8

Table 3.3: The number of machines and the average feedback time in minutes where each batching algorithm plateaus for Chrome.

Algorithm	***TestCase Batching***	***BatchAll***	***TestAll***	***Batch2***	***Batch4***
Feedback time	3.96	3.78	3.54	5.88	12.6
Machines	125	150	350	225	150

and a significant backlog of commits. However, the queue size varies over time, with peak changes happening during working hours. To better utilize the available resources, we suggest *BatchAll* which batches all available changes in the queue.

Figure 3.6 and Figure 3.7 show that *BatchAll* approach always outperforms *TestAll* and *ConstantBatching* algorithms in both Ericsson and Chrome cases. This algorithm performs promisingly both in high resource-constrained and high resource-available conditions compared to the baseline number of machines.

Table 3.1 illustrates that *BatchAll* is capable of maintaining the feedback baseline at 8.33 hours for Ericsson by utilizing 5 machines, leading to a 28.57% reduction in machine usage. Similarly, for Chrome, *BatchAll* achieves the feedback baseline at 36.48 minutes with 20 machines, resulting in an impressive 90.78% reduction in machine utilization.

Table 3.2 shows that *BatchAll* approach plateaus for Ericsson at 2.75 hours by using 8 machines. This represents a 12.5% resource reduction relative to the plateaued feedback baseline. For Chrome, Table 4.4 displays that *BatchAll* plateaus for Chrome at 3.78 minutes using 150 machines. This is a 54.14% reduction in resources to reach the plateaued feedback baseline.

The confidence interval for the average feedback time values depicted in Figure 3.7 for the *BatchAll* algorithm and Chrome indicates a range of approximately 1% compared to the actual average feedback time when using 16 machines. As the number of machines increases to 25, this range narrows significantly, approaching zero, which suggests a more concentrated distribution of

Table 3.4: The Cliff's delta effect size values for pairwise comparisons of different batching algorithms for Chrome at different numbers of machines.

Machines	**TA* vs B2**	**TA vs B4**	**TA vs BA**	**TA vs TCB**	**B2 vs B4**	**B2 vs BA**	**B2 vs TCB**	**B4 vs BA**	**B4 vs TCB**	**BA vs TCB**
8	0.53	0.79	1.00	-0.05	0.57	0.99	-0.30	0.98	-0.51	-0.92
16	0.57	0.84	0.99	-0.04	0.68	0.98	-0.30	0.97	-0.53	-0.87
25	0.63	0.90	0.98	0.08	0.83	0.97	-0.17	0.94	-0.48	-0.67
50	0.83	0.94	0.96	0.97	0.84	0.93	0.95	0.65	0.80	0.51
75	0.90	0.94	0.95	0.96	0.62	0.78	0.85	0.52	0.73	0.43
100	0.83	0.87	0.91	0.92	0.33	0.63	0.75	0.50	0.70	0.35
125	0.65	0.69	0.80	0.84	0.12	0.55	0.67	0.54	0.70	0.29
150	0.60	0.58	0.74	0.78	-0.08	0.49	0.62	0.58	0.70	0.24
175	0.48	0.40	0.66	0.71	-0.19	0.46	0.59	0.61	0.71	0.20
200	0.34	0.20	0.59	0.64	-0.25	0.45	0.56	0.63	0.71	0.16
225	0.24	0.06	0.52	0.58	-0.28	0.45	0.55	0.65	0.72	0.14
250	0.16	-0.06	0.47	0.53	-0.30	0.45	0.54	0.67	0.73	0.12
275	0.06	-0.20	0.42	0.49	-0.32	0.46	0.53	0.68	0.73	0.11
300	-0.02	-0.31	0.39	0.45	-0.33	0.46	0.53	0.69	0.73	0.10
325	-0.09	-0.39	0.35	0.42	-0.34	0.46	0.52	0.70	0.74	0.09
350	-0.14	-0.45	0.33	0.39	-0.34	0.47	0.52	0.70	0.74	0.08
375	-0.18	-0.49	0.30	0.36	-0.35	0.47	0.52	0.71	0.74	0.08

*Please refer to the following abbreviations used in this table: TA: *TestAll*, B2: *Batch2*, B4: *Batch4*, BA: *BatchAll*, TCB: *TestCaseBatching*.

the feedback time values.

Figure 3.8 and 3.9 demonstrate that the *BatchAll* algorithm's ExecutionReduction varies with the number of machines used, decreasing as the number of machines increases. For Ericsson, the ExecutionReduction ranges from 49% to 3% when utilizing 1 to 9 machines. Similarly, for Chrome, the ExecutionReduction ranges from 99% to 9% when employing 1 to 375 machines.

Figure 3.10 depicts the average batch sizes utilized by the *BatchAll* and *TestCaseBatching* algorithms in Chrome, with varying numbers of machines. The figure is displayed on a logarithmic scale to accommodate the substantial difference in batch sizes between a few machines and larger numbers of machines. The average batch sizes for the *BatchAll* algorithm range from approximately 85 batches to nearly 1 batch as the number of machines increases from 1 to 375. Notably, a significant reduction in average batch sizes is observed when the number of machines increases from 1 to 8, resulting in a drop from 85 to 10 batches for *BatchAll*.

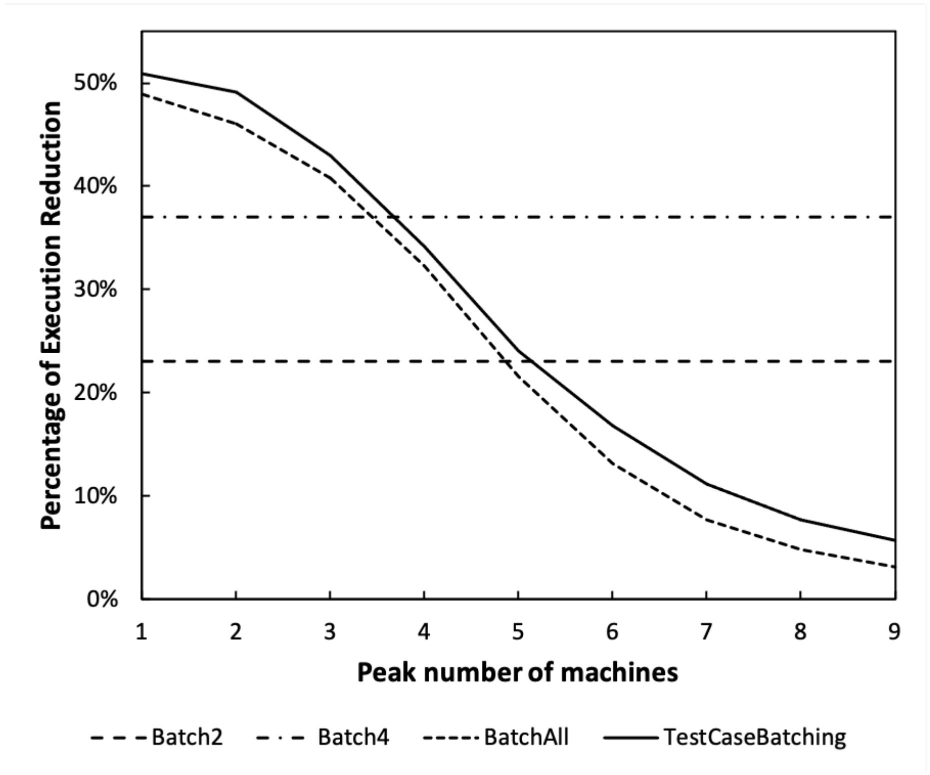


Figure 3.8: The percentage of EXECUTIONREDUCTION achieved by different batching algorithms, relative to the *TestAll*, was calculated for different peak numbers of machines for Ericsson.

3.4.4 RQ4: *TestCaseBatching*

BatchAll processes all available changes. However, any change that arrives has to wait until all the tests for a batch have been completed. In background Section 3.3.5, we introduced *TestCaseBatching* that queues the requested tests across all changes and includes any new change after each test completes (rather than waiting for all tests to complete).

Except when there is an extreme resource constraint, the *TestCaseBatching* approach performs more effectively than other algorithms in terms of feedback time as shown in for Ericsson in Figure 3.6 and Chrome in Figure 3.7. The reasons for the algorithm's poor performance under high resource constraints, as depicted by the parallel line in Figure 3.7 at 42 machines, will be discussed in detail in the discussion section.

Table 3.1 presents that *TestCaseBatching* achieves the feedback baseline at 8.33 hours for Ericsson by utilizing 5 machines, resulting in a 28.57% reduction in machine usage. Similarly, for Chrome, *TestCaseBatching* achieves the feedback baseline at 36.48 minutes using 42 machines,

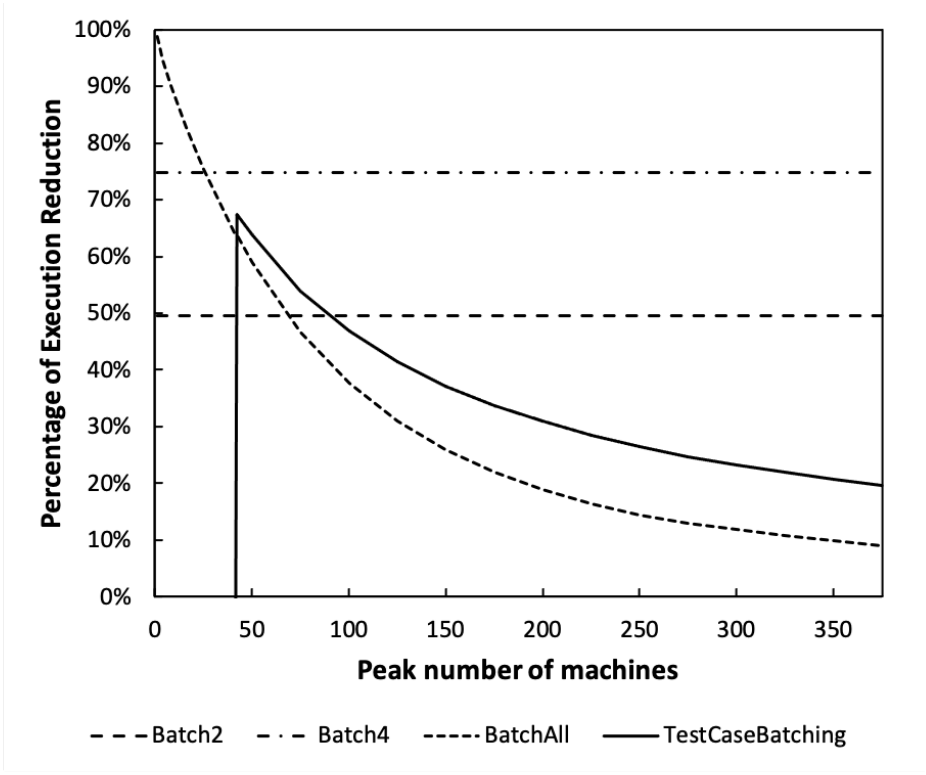


Figure 3.9: The percentage of ExecutionReduction achieved by different batching algorithms, relative to the *TestAll*, was calculated for different peak numbers of machines for Chrome.

leading to a remarkable 80.65% reduction in machine utilization.

Table 3.2 shows that the *TestCaseBatching* approach plateaus for Ericsson at 2.53 hours by using 8 machines. This presents a 12.5% resource reduction to reach the plateaued feedback baseline. For Chrome, Table 4.4 displays that the *TestCaseBatching* plateaus for Chrome at 3.96 minutes using 125 machines. This is a 61.28% reduction in resources to reach the plateaued feedback baseline.

The confidence interval ranges depicted in Figure 3.7 for the *TestCaseBatching* algorithm and Chrome tend to stabilize at approximately 1% of the actual average feedback time as the number of machines increases to 50. Beyond this threshold, the range of confidence intervals decreases significantly, approaching nearly zero. This narrowing range suggests a more concentrated distribution of feedback time values.

Figures 3.8 and 3.9 illustrate the varying percentages of ExecutionReduction achieved by the *TestCaseBatching* algorithm using different numbers of machines. In the context of Ericsson, *TestCaseBatching* achieves an ExecutionReduction ranging from 51% to 6% when employing

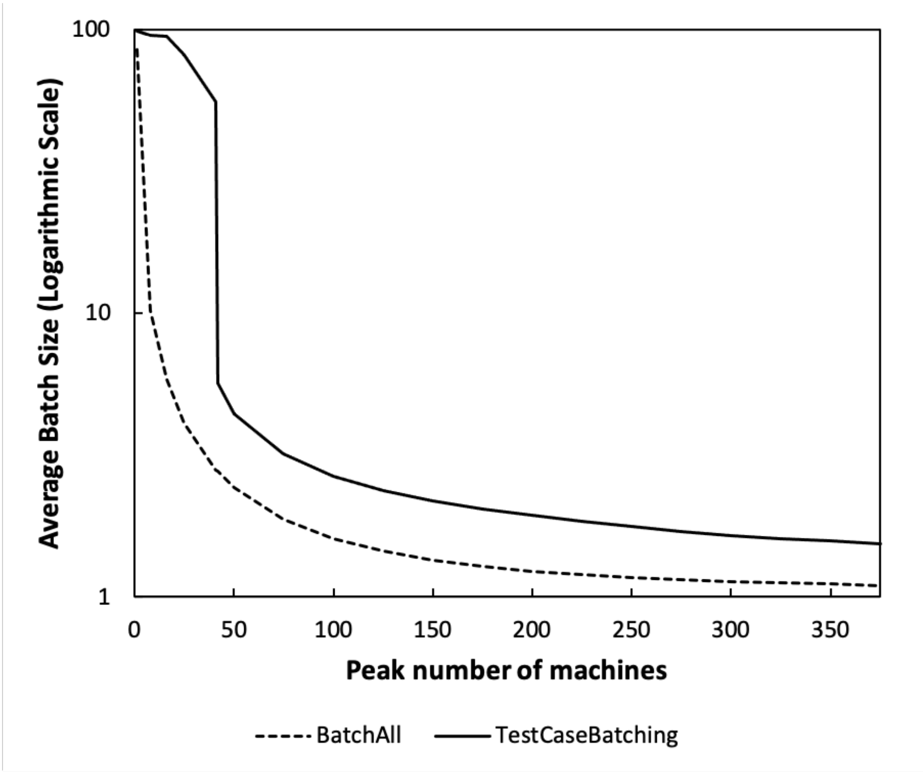


Figure 3.10: The average batch size used by the *BatchAll* and *TestCaseBatching* algorithms was calculated for different peak numbers of machines in Chrome, represented on a logarithmic scale.

1 to 9 machines. On the other hand, in the case of Chrome, *TestCaseBatching* initially exhibits negative performance in EXECUTIONREDUCTION with fewer than 42 machines, as further discussed in the subsequent section. However, with 42 to 375 machines, *TestCaseBatching* achieves an EXECUTIONREDUCTION ranging from 67% to 20%, outperforming the results of *BatchAll*.

Figure 3.10 illustrates the batch size utilization of the *TestCaseBatching* algorithm on Chrome, covering a range of 1 to 375 machines. The average batch sizes vary from approximately 95 to 1.5. In the range of 1 to 41 machines, the average batch size remains relatively high, ranging from 98 to 56. However, at 42 machines, a significant drop occurs, resulting in an average batch size of around 6. Beyond this point, the number of average batch sizes continues to decrease gradually, reaching 1.5 at 375 machines.

3.5 Discussion

Regarding the impact of parallelization on testing with resource constraints, we can interpret from the results that increasing the number of machines can improve the feedback time non-linearly. This is because without sufficient resources the delay is compounded by all subsequent changes, leading to large delays. For example, with only one machine, the queue grows dramatically, and the wait time becomes much longer than the processing time. Once the queue size is reduced, we see smaller improvements because the actual time to run tests becomes the most important factor. However, at some point, the feedback time plateaus in which increasing the number of machines only makes a negligible improvement on the feedback time since the queue size is kept small.

The results for the *ConstantBatching* algorithms of *Batch2* and *Batch4* reveal that running these algorithms in a highly resource-constrained environment can boost their feedback time in comparison with the *TestAll* approach. Moreover, the more resource-constrained the environment is, the more effective larger batch sizes would be. However, in a highly resource-available situation, using a larger constant batch size will deteriorate the feedback time. This is because of the fewer changes remaining in the queue as a result of the faster processing time of the changes. Consequently, *ConstantBatching* with large batch sizes becomes a disadvantage since it increases the wait time for processing the changes. This is also the reason why *ConstantBatching* algorithms reach a plateau at higher feedback time values, leaving little room for improvement with the utilization of additional machines.

BatchAll on the other hand uses the fast unloading advantage of a bigger batch size in a high resource-constrained environment and the low wait time benefit of a smaller batch size in a high resource-available environment. When we simulate a high resource-constrained environment with few machines, it makes large batches that can rapidly reduce the queue, leading to a higher EXECUTIONREDUCTION. Conversely, being in a high resource-available condition with enormous machines, it may reduce the batch size to 1, eliminating build waiting time but resulting in a lower EXECUTIONREDUCTION. This makes *BatchAll* highly effective in terms of feedback time in both of these scenarios.

TestCaseBatching algorithm has the highest performance in terms of feedback time, except in an extremely high resource-constrained environment with few machines, specifically in the Chrome case with fewer than 42 machines. This is because the *BatchAll* algorithm puts some changes in the queue while it is running the current batch, and then it runs all its tests as a batch. Still, the *TestCaseBatching* method always accepts new changes and does not adopt a queue. Since it runs the tests of a change in the absence of potential subsequent changes, when the next changes become available, it has to run some of these tests again as a penalty. When there are enormous overlapping builds, these penalties add up and deteriorate the feedback time and the EXECUTIONREDUCTION of the *TestCaseBatching* algorithm in comparison with the *BatchAll* method. This specifically is the case in Chrome test results, as there are more overlapping builds in the dataset.

The implications of these results for practitioners would be as follows. We recommend using *BatchAll* and *TestCaseBatching* as the best batching algorithms among all the approaches under the study. In a very high resource-constrained environment, *BatchAll* produces a better feedback time and EXECUTIONREDUCTION, otherwise, *TestCaseBatching* performs the best. Despite the testing approach being used, there is a threshold in which increasing the number of machines does not improve the feedback significantly. Hence, it makes sense to recognize this plateau point by exercising the batching approach being used on the corresponding project. This way they can reduce feedback time while saving the number of machines being used.

3.6 Threats to validity

External Validity. The outcomes from this study are from applying various batching techniques on the test results of two real large-scale Ericsson and Chrome projects, and we may not be able to generalize them to all other applications. The performance of these batching algorithms might differ by applying them to various other projects and using different numbers of machines. This is because they probably have different numbers of changes, builds, and test cases. However, the algorithms adopted in this study are not tied to a specific project, and the non-linear relationship between the number of machines and the feedback time in other projects would likely be the same.

Another issue that may threaten the external validity of this work is the failure rate. The failure rate can be different among various applications, and the higher the failure rate is, the higher the cost of culprit finding would become which in turn, can make the batching algorithms ineffective. However, Beheshtian *et al.* [41] show that the batching algorithms can produce savings on the projects with a build failure ratio of below 40%, and among 9 Travis CI projects under their study, 85.5% of them could take advantage of batching techniques. The build failure rate in Chrome dataset is 8.5%, which is well below the 40% threshold, making batching an effective approach for this project. Meanwhile, the focus of this study is more on the impact of parallelism on various batching algorithms and on two very large-scale Ericsson and Chrome projects. At the time of this study, we could not recognize any other available testing datasets for the experiment which are comparable in testing scale with these datasets.

Internal Validity. One of the aspects of test optimization which can compromise the process is test dependencies. Reordering or running tests in parallel when they were not designed to be run in this manner can introduce dependency flaky failures [45]. Therefore, we only batch tests that Ericsson guarantees are independent. In the case of Chrome, we only run tests in parallel and independently if Chrome developers were already running these in parallel on multiple shards.

The historical simulation simplified parts of the Ericsson's and Chrome's testing processes. For example, developers can stop testing a build or manually batch-select changes for testing. Since we cannot model these manual interventions, we exclude them from our simulation.

In our experiment, we assumed that all changes can be batched and none lead to merge conflict. Since each must be ultimately merged into the main branch, we do not introduce any new conflicts because any conflict would have been dealt with when the developer ensures that the code can be merged. However, the batching process may bring this conflict to the developer's attention earlier as we batch different combinations of changes.

Construct Validity. To simplify the measurement of the feedback time we ignored the compile times of the changes. This is because we could not have any assumptions for the compile time of the batches. In the case that the batch contains no failing change, this can save on compile time. Otherwise, it might cause some extra compile time for the failing changes. Considering the failure rates in our study, we can suggest that calculating the compile time can even save more on the feedback time of the batching techniques. We leave this simulation parameter to future work.

Using the average value to compare the feedback time of the different batching algorithms can be a threat to the construct validity of this study. Although the feedback time distribution of each test batching experiment is a more accurate way of showing the results, for each batching algorithm and for each number of machines, we need a single value to compare the performance. This along with having an approximately normal distribution, lead us to determine the average feedback time to compare the outcomes.

3.7 Related Work

Continuous integration and delivery is an essential aspect of modern software development [46]. Using a CI infrastructure, developers can automatically build and test their changes to make sure it does not break software functionality [47–50]. In a CI pipeline, we need to test each commit before merging it to the main branch. In large software systems, testing every change individually is impossible [51]. Even Google with huge server farms is not able to test every single commit independently [52]. In the following, we explore various approaches employed to accelerate CI.

Test Selection and Prioritization. To reduce both resource consumption and provide earlier feedback, test selection has been widely adopted in the industry and extensively studied [11, 34–36, 53–57]. However, test selection's inherent trade-off is that not all tests are run, and some test failures may be missed. In contrast, test prioritization guarantees that all tests will be run, but those that are more likely to reveal faults will be run first, reducing feedback time on test failures, but not reducing the resource consumption in testing [37–39, 58–64]. Unfortunately, as Elbaum *et al.* [65] point out, even the largest open-source projects on Travis, such as rails, do not have a sufficient arrival rate of commits to justify the use of selection or prioritization because there are no changes waiting in the queue. Furthermore, the entire test suites run on the order of a few minutes, providing no justification for parallelization or batching (see Table 1 in Beller *et al.* [66]).

To provide a reasonable industrial comparison, we contrast our results with those found at Microsoft. Herzig *et al.* [51] report an improvement of 40.31%, 40.10%, and 47.45% in testing feedback time for Windows, Office, and Dynamics using association rule mining for selection. However, they note a slip-through rate of 71% and 91% at the first branch level, 21% and 3% at the second, and 8 and 0% at the three branch level. At Ericsson, the same approach saw a 42.78% reduction, but with 34.65% slipthroughs [67]. In contrast, *BatchAll* and *TestCaseBatching* reduce testing time by 64.43% and 69.37% respectively in Ericsson without allowing any slip-throughs. A direct comparison of selection with batching will be interesting for future work.

Test parallelization. Tests are distributed across machines using this technique to reduce feedback time. Previous works widely studied the impact of test parallelization on software testing and introduced algorithms to run tests in parallel [68–71]. For example, Arabnejad *et al.* [72] investigated using GPUs for running tests in parallel. The most popular algorithms for parallelizing tests are scheduling tests across the machines based on their IDs and their historical execution time [73].

Candido *et al.* [74] investigated the impact of test parallelization on open-source projects. By analyzing more than 450 Java projects, they found that less than 20% of major projects use test parallelization due to concerns with concurrency issues. They provide recommendations to practitioners to facilitate their parallel testing, such as refactoring tests for load balancing and grouping tests based on their dependencies, and running tests with dependencies on the same machine. Bell *et al.* [75] studied the impact of test dependency on test parallelization. They introduced the *ElectricTest* approach to detect dependencies before scheduling them across the machines. Ding *et al.* [76] proposed a software behavior-oriented test parallelization to reduce conflicting behaviors. Nagy *et al.* [77] devised a parallel architecture for test case dispatching to reduce idle nodes and improve execution time. At the Ericsson and Chrome, we only batch tests at the level that is already designed to be run in parallel. We note that none of the prior works studied the effect of test parallelization in the context of batch testing.

Build Prediction and Skip. Another line of work aimed at expediting Continuous Integration (CI) processes includes build prediction and build skip techniques. Build prediction techniques leverage machine learning methods to predict the result of a build, with a particular focus on reducing the cost of builds that are likely to pass [78–87]. For example, Chen *et al.* [86] developed a build prediction model based on build logs and changed files, and Saidani *et al.* [87] proposed a Multi-Objective Genetic Programming approach to address the imbalanced distribution of pass and fail builds. These techniques rely on the fact that the majority of builds pass, providing an opportunity to reduce their cost [86].

On the other hand, build skip techniques aim to identify builds that do not require execution, typically due to the absence of source code changes. There are different approaches for build skip, including manual configuration [88, 89] and rule-based or learning-based methods [90–96]. Jin *et al.* [92] proposed a build skip approach, leveraging the assumptions that passing builds are the majority and failing builds tend to occur consecutively. Abdalkareem *et al.* [90, 91] adopted rule-based and machine learning techniques to automate the process of recognizing changes suitable for CI skipping. Gallaba *et al.* [95] proposed a language-agnostic approach that infers data without relying on build specifications to skip unaffected build steps.

While build prediction and build skip techniques offer potential benefits for accelerating CI processes, they have limitations. Build prediction techniques struggle with accurately predicting failing builds, leading to costly misidentifications, and their practical use cases are often ill-defined [86]. On the other hand, build skip techniques risk skipping builds that may contain relevant changes or introduce failures. In contrast, batching combines multiple changes into a single build, eliminating the risk of skipping builds entirely. It enables efficient debugging and failure resolution within the batch, providing a reliable approach for managing complex software changes in continuous integration, as employed by major companies like Google.

Batch Testing. This technique is used to decrease the feedback time in resource-constrained environments [97, 98]. Instead of testing every single change in isolation, we batch them and run them all at once. If the batch fails, we have to find the culprit change(s) responsible for the failure. GitBisection [99] is the most well-known culprit-finding technique. It performs a binary search on the changes to find the culprit. GitBisection can find the culprit with $log(n)$ executions. However, in the condition that there is more than one culprit, GitBisection only finds the first one. To solve this problem, Najafi *et al.* [40] introduced a bisection that uses a divide-and-conquer algorithm to find all the culprits. It divides the failing batch into two sub-batches and tests both of them to find if they include a culprit change. As it performs a complete search, the total number of executions is between $2log(n)$ and $2n + 1$ when all the changes in the batch are culprits. Beheshtian *et al.* [41] showed that for batches with a size of 4 or fewer, bisection only increases the number of executions. They propose *BatchStop4* which tests every change individually in the batches with sizes 4 or fewer. They mathematically show that using *BatchStop4* always outperforms batch bisection. They also study the impact of failure rate on different batching algorithms.

Bavand *et al.* [42] implemented a complex dynamic batch size approach based on historical failure rates, resulting in a modest 5.17% reduction in test execution time compared to *Batch4* with a single machine. In the Ericsson context, *BatchAll* and *TestCaseBatching* achieved more significant EXECUTIONREDUCTION, with 11.87% and 13.92% reductions respectively compared to *Batch4*. For Chrome, *BatchAll* showed a substantial 23.88% improvement over *Batch4* with a single machine, while *TestCaseBatching* had negative results with less than 42 machines. Overall, *BatchAll* and *TestCaseBatching* consistently outperformed *ConstantBatching* in feedback time across different machine configurations.These previous works focused on batch testing at the change level and with a single machine, without exploring the effects of parallel testing. In our study, we investigate the combined impact of batching and parallel testing for the first time, examining how varying the number of parallel machines affects two extensive projects: Ericsson and Chrome.

3.8 Conclusions and Future Work

In this study, we aimed to investigate the impact of parallelization on different batching techniques, considering the limitations of previous studies that focused on single machine environments. Our evaluation yielded the following findings. The *TestAll* approach experienced compounded delays in subsequent builds, and the effect of changing the number of machines on feedback time was non-linear. *ConstantBatching* algorithms performed better in feedback time in high resource-constrained environments and reached a longer feedback time plateau. They provided a consistent EXECUTIONREDUCTION across different machine counts. The *BatchAll* approach is effective in feedback time in both high resource-constrained and highly available resource environments, being able to maintain actual average feedback time and attain plateaued feedback baseline with 90.78% and 54.14% fewer machines than the baseline respectively. The *TestCaseBatching* algorithm performed poorly in extreme resource constraint conditions but effectively in high resource available environments, utilizing 80.65% of baseline machines to maintain the actual average feedback time and 64.28% to reach the plateaued feedback baseline. Both *BatchAll* and *TestCaseBatching* algorithms exhibited variable EXECUTIONREDUCTION based on the number of machines used.

It should be noted that this study utilized simulated parallel test execution based on historical test results, and further validation with actual software builds is necessary to address real-world complexities such as resource contention and synchronization. Future research should aim to confirm and extend these findings through experiments conducted in practical settings.

Chapter 4

Assessing the Efficacy of Test Selection, Prioritization, and Batching Algorithms

This chapter is a verbatim copy of the manuscript that is submitted for review to the Empirical Software Engineer (EMSE) journal, preserving the original content without any modifications.

Abstract

The effectiveness of software testing is vital for successful software releases, and various test optimization techniques aim to enhance this process. However, few studies compare these algorithms using the same evaluation criteria. This study addresses this gap.

To evaluate different test optimization approaches, we analyze 276 million test results from the Chrome project. We assess Kim and Porter's, Elbaum *et al.*'s history-based test prioritization and selection techniques, and our *BatchAll* test batching algorithm.

Findings reveal that Elbaum *et al.*'s test selection reduces Chrome's baseline median feedback time by 96.12% from 33.55 to 1.28 minutes with the same machines but misses 31.25% of failures. In contrast, *BatchAll* achieves a 91.84% feedback time reduction to 2.69 minutes without missing failures. Elbaum *et al.*'s test selection cuts machine usage by 58.10% to 75 machines, while *BatchAll* achieves 88.27% reduction using only 21 machines. For failure detection, Elbaum

et al.'s test selection is 62 minutes faster than the baseline, and the *BatchAll* algorithm achieves a 59.4 minutes median improvement without missing failures. Regarding test execution time, Elbaum *et al.*'s test selection saves 66.31%, whereas *BatchAll*'s saving varies from 2% to 98.5% based on machines used. The studied test prioritization algorithms significantly underperform Elbaum *et al.*'s test selection and *BatchAll* algorithms.

In conclusion, this study provides practical recommendations for selecting appropriate test optimization algorithms based on the testing environment and failure loss tolerance.

4.1 Introduction

Software testing is an essential aspect of modern software development processes. With the growth of software systems and the adoption of Continuous Integration practices, it has become necessary to test each change individually, leading to a significant increase in the number of test runs, especially for large-scale systems. This presents a challenge for software development teams, as even companies like Google, with extensive resources, are unable to test every change individually [1,2].

To address the issue of ever-growing test runs, various test optimization techniques have been proposed. Test selection involves running only those test cases that are more likely to reveal failures. Test prioritization, on the other hand, prioritizes the order in which tests are executed so that failures are detected earlier. Test parallelization distributes tests across multiple execution units to reduce feedback time for changes. Finally, test batching groups multiple changes together and runs tests for the entire batch, saving time on both change feedback and test execution.

Previous research has investigated various test optimization techniques, but there is limited research on comparing these techniques, especially with batching, which is an efficient but often overlooked technique. Another limitation in the research literature is the lack of comparison between different test optimization techniques using standardized metrics. For instance, it would be interesting to determine how quickly test batching can detect failing tests, as well as to understand the performance of different test optimization techniques in various resource execution environments.

Prior studies on test optimization have mainly focused on small datasets, resulting in impractical solutions for projects at scale [2, 3], even though the problem of test optimization is much more relevant to large-scale projects. Additionally, the impact of flaky tests on test optimization techniques is often ignored, despite studies showing their significant impact [6, 12]. Some studies that did consider flaky tests relied on subjective re-running thousands of times [4], which is unrealistic [6]. Moreover, although parallelization has been studied in testing, limited research has investigated the impact of parallelization on different test optimization techniques.

To address the research gap, this study compares various test optimization techniques and evaluates their performance using the same metrics. Specifically, we evaluate the effectiveness of test selection, prioritization, and batching techniques with different numbers of machines under varying resource availability environments. Our evaluation criteria include the feedback time of changes, resource usage, test execution time, and the speed of detecting failing tests.

We conduct our experiments using the Chrome dataset that we provided in our previous work [6]. This dataset is associated with a large-scale Chrome project and provides us with a more realistic understanding of the testing scale and problem. It contains 276 million test results for the month of January 2021. One of the significant advantages of this dataset is that it includes labels for flaky tests identified by the Chrome testing infrastructure, enabling us to account for more realistic flaky results. In addition, we have taken into account the presence of flaky tests for different test optimization algorithms, which have been previously overlooked.

We provide answers to the following research questions in this study.

RQ1: How effective are different test optimization approaches in reducing feedback time when executed in parallel with varying numbers of machines?

RQ2: How effective are different test optimization approaches in reducing the number of machines in use when executed in parallel with varying numbers of machines?

RQ3: How quickly do different test optimization techniques detect failing tests when executed in parallel with varying numbers of machines?

RQ4: How effective are different test optimization techniques in saving test execution time when executed in parallel with varying numbers of machines?

To answer our research questions, we conduct comparative simulations using 276 million test results from the Chrome dataset [6]. We begin by exploring and describing the data to gain a better understanding of the test runs and changes in the Chrome project. Next, we simulate two history-based test prioritization algorithms, Kim and Porter's [8], and Elbaum *et al.*'s [1], using various numbers of machines. We also simulate Elbaum *et al.*'s test selection as a delegate for the history-based test selection algorithm with different numbers of machines.

Additionally, we simulate the *BatchAll* algorithm as one of the effective batching algorithms that uses adaptive batch sizes based on the number of changes in the queue and adopts an improved culprit-finding technique that we introduced and evaluated in our recent work [33]. We evaluate these test optimization algorithms using four metrics: feedback time, number of machines used, GAINEDTIME, and test execution time. These metrics allow us to evaluate the performance of the algorithms from different perspectives and under different resource availability scenarios, and have been used in multiple studies [42, 100].

This study makes the following contributions.

- **Comparison of test optimization algorithms:** We compare the efficiency of different test optimization algorithms to understand their strengths and weaknesses and provide insights for practitioners on the most effective techniques, considering varying resource availability.

- **Dataset:** We evaluate the algorithms using a dataset of 276 million test results from Chrome, which represents a large-scale software system. We also provide details and explore the data distributions of this dataset.

- **Parallelization:** We run the different test optimization algorithms using various numbers of machines to evaluate their performance under different resource availabilities.

- **Flaky tests:** We address flaky tests to simulate the algorithms under realistic conditions.

- **Approaches:** We simulate the *BatchAll* test batching algorithm that we presented in our recent study [33]. Additionally, we simulate test prioritization algorithms proposed by Kim and Porter [8] and Elbaum *et al.* [1]. We also revise the window sizes for Elbaum *et al.*'s algorithms following the resolution of flaky tests.

- **Outcome measures:** We evaluate various test optimization algorithms using the same metrics to assess their relative performance. These metrics include feedback time, the number of machines used, gained time to identify failing tests, and the level of test execution reduction.
- **Results summary:** We find the following outcomes.

  RQ1: Elbaum's test selection reduces the maximum in feedback at the cost of losing some failures, while batching achieves a comparable performance without losing any failures.

  RQ2: Batching achieves the best reduction in resources to achieve the same feedback time as the baseline.

  RQ3: Elbaum's test selection is the fastest to detect failures, but it misses some failures while batching achieves a similar performance without losing any failures.

  RQ4: The savings in test execution time are constant for Elbaum's test selection algorithm, while they vary for the batching algorithm based on the number of machines in use.

This remainder of this work is structured as follows. In Section 4.2, we review the background and related work. Section 4.3 presents the design and methodology adopted to conduct this study. In Section 4.4, we provide explanations for each test optimization approach. Section 4.5 elaborates on the different evaluation criteria and metrics used in the simulation experiments. In Section 4.7, we discuss the threats to the validity of this research. In Section 4.6, we present the results regarding our research questions and simulations. Section 4.8 discusses the interpretations of the results and implications from this study for practitioners. Finally, in Section 4.9, we present the conclusion and potential future work.

4.2 Background and Related Works

This section presents an overview of test optimization techniques relevant to the approaches studied in this research, and also the datasets adopted for test optimization. As software systems become more complex and Continuous Integration (CI) is increasingly adopted in modern software release processes, the number of tests required to verify these systems has significantly increased. With CI, every change made by developers must undergo a time-consuming process of separate building and testing [47–50]. However, testing each change individually becomes nearly impossible for large companies due to the sheer volume of tests involved [101]. Even Google, with its vast resources, struggles to keep up with the high code churn in their code base [1]. To address this challenge, researchers have proposed various test optimization techniques, such as test selection, prioritization, and batching, with the goal of reducing the time and cost of testing while maintaining the quality of the software product.

4.2.1 History-based test selection and prioritization

Test selection and prioritization approaches have been the primary solution to the problem of ever-increasing test runs. Different test selection and prioritization techniques have been proposed by researchers to improve feedback time [21–24, 26, 53, 58, 59, 102–107]. In the past, researchers have primarily concentrated on selecting and prioritizing tests within the same build. The work of Kim and Porter [8] was one of the initial studies in this field, which utilized the previous test records to prioritize tests. They established a metric that assigned a higher value and priority to the tests that had previously failed.

Marijan *et al.* [25] emphasized the importance of providing quick feedback in a continuous integration setup. They proposed a prioritization scheme for tests based on the time elapsed since the last failure, the test's execution time, and domain-specific knowledge.

Anderson *et al.* [108] proposed two models to predict future test failures in the Dynamics AX 2012 R2 project. The first model was based on the most frequent failures, while the second model utilized more expensive data such as code coverage through Associate Rule Mining (ARM) techniques. Interestingly, they found that the simpler model based on most frequent failures was just as effective as the ARM model.

Herzig *et al.* [104] aimed to reduce the cost of test execution on Microsoft products by implementing test selection at different test execution levels. They created a cost model based on historical test data, including test results and execution context. Their model was defined based on the failure probability of a test, and they were able to achieve significant reductions in test execution costs for Microsoft Office, Windows, and Dynamics.

Memon *et al.* [2] reduced feedback time in Google by using domain expertise and statistical analysis. Their model was built by taking into account failing tests, the relationship between tests and developers, the codebase, changes to the code, and the frequency of test execution. To conduct their experiment, they analyzed 5.5 million unique tests affected by 500k change lists for one month in 2016.

Elbaum *et al.* [1] focused on optimizing test selection and prioritization for Google's multi-machine and parallel testing environment. Their algorithms were designed to apply to both pre- and post-submit test data. To determine which tests should be selected and prioritized, they utilized a test failing history and three different windows: failure, execution, and prioritization. The results of running their algorithms on a sample of 3.5 million tests showed a significant improvement in test performance.

Liang *et al.* [106] found that traditional methods such as code analysis and coverage are not sufficient in the Continuous Integration (CI) environment. They also mentioned that some companies are hesitant to use test selection techniques because they could miss certain failures. As a solution, they proposed a commit-based prioritization approach, which they applied to both Google Shared dataset and Rails dataset from TravisCI. The results indicated that Google dataset had a 12% improvement in $APFD_c$, whereas there were almost no noticeable improvements in Rails.

Zhu *et al.* [10] suggested that co-failures can provide useful information for prioritizing tests, as they are likely to fail together in the future. They proposed a method that uses multiple queues to prevent test starvation and improve test prioritization. The authors evaluated their approach on the Google and Chrome datasets and compared the results with the study conducted by Elbaum *et al.* [1].

Najafi *et al.* [26] recognized that the duration of tests could be a significant factor in test prioritization. They proposed an approach that considers the historical frequency of test failures, the relationship between failures, and the cost of executing a test. Their method involves assigning lower priority to more time-consuming tests.

On the other hand, the increasing prevalence of flaky tests has led researchers to investigate their impact on test selection and prioritization algorithms. For instance, Peng *et al.* [12] identified the effect of flaky tests on the performance of IR-based test prioritization algorithms on Travis CI projects. Similarly, Fallahzadeh and Rigby [6] analyzed 276 million Chrome test results and found that flaky tests significantly affect history-based test prioritization algorithms.

With a few exceptions, most prior research has ignored the impact of flaky tests on test optimization algorithms. In our study, we take into account the presence of flaky tests in all test optimization algorithms experimented, while still considering flaky tests as non-blocking, as we assumed in our previous study [6].

4.2.2 Batch Testing

In Continuous Integration (CI), each change must be tested individually. However, for large software systems such as those found in Google products, even with extensive computational resources, this is often unattainable [1, 101]. Consequently, test batching has been proposed as a solution to this problem. By grouping multiple changes together and testing them simultaneously, computational resources and feedback time can be reduced in resource-constrained environments [109, 110]. However, the trade-off is that when a batch fails, there is a penalty to find the culprit change responsible for the breakage.

There have been several studies conducted by researchers on various methods for performing culprit-finding procedures. GitBisection [111] is a popular algorithm that employs binary search and requires $log(n)$ executions to detect a culprit change. However, this approach has a limitation: it can only identify the first culprit when multiple culprits are present. To address this limitation, Najafi *et al.* [112] proposed a divide-and-conquer bisection algorithm. This algorithm recursively divides each failing batch into two sub-batches to detect all culprits. The divide-and-conquer bisection algorithm requires between $2log(n)$ and $2n + 1$ batch tests to execute.

Beheshtian *et al.* [100] found that the effectiveness of batching decreases for batches of size four or smaller. To address this issue, they developed BatchStop4, which performs batching until the batch size reaches four, at which point it tests each change individually. Their approach outperformed batch bisection in terms of feedback time.

The prior research focused on batching at the build level, whereas in this study, we investigate batching at the test level and introduce advancements in this domain. Additionally, our research examines the impact of parallelism and multiple machines on the Chrome project's large-scale testing, while prior works only use a single machine for their experiments. Furthermore, we evaluate the effectiveness of batching in comparison to other test optimization methods such as test selection and prioritization.

4.2.3 Test Parallelization

The technique of test parallelization involves spreading out the testing process across multiple machines, with the aim of reducing the time it takes to obtain feedback on the tests. Various studies have explored the impact of test parallelization on software testing, and have put forward algorithms that allow for the efficient execution of tests in parallel [68–71]. One of these studies was carried out by Arabnejad *et al.* [72], who investigated the potential benefits of using GPUs to run tests in parallel. The most commonly used algorithms for parallelizing tests include those that schedule tests across machines based on their IDs and historical execution times [73].

Candido *et al.* [74] explored how test parallelization affects open-source software projects. They analyzed more than 450 Java projects and discovered that less than 20% of major projects employ test parallelization due to concerns about concurrency problems. To help practitioners with parallel testing, the authors provide various suggestions, such as reorganizing tests for load balancing, grouping tests based on dependencies, and executing tests with dependencies on the same machine.

Bell *et al.* [75] conducted research on the effect of test dependencies on parallelization. They presented the *ElectricTest* method to identify dependencies prior to scheduling tests on multiple machines. In a separate study, Ding and colleagues [76] suggested a software behavior-oriented approach to parallelize testing, which aims to minimize conflicting behaviors.

Our study applies parallel testing at a scale where parallel execution is essential. This differs from prior research as none have specifically examined how parallelization impacts test optimization. Furthermore, we aim to explore the correlation between the number of machines utilized and the time required to obtain build and failing test results, and the reduction in test execution time.

4.2.4 Datasets

Different datasets have been evaluated for test optimization. Elbaum *et al.* [1] studied test selection and prioritization on a sample of Google test results. This publicly available dataset includes 3.5 million tests over a period of 30 days.

Several studies have used test results from Travis CI, a continuous integration service that hosts many open-source projects, to conduct their evaluations [9, 14, 27, 28, 113].

Matiss *et al.* [14] collected test results from 20 open-source Java projects from Travis CI, and their dataset is publicly available for test optimization. However, the largest project in their dataset, spanning 9 years, contains only about 17 million test results, or approximately 158,000 tests per month. This pales in comparison to the vast number of test results present in the Chrome dataset.

Given that test optimization is primarily relevant to large-scale projects, the limited size of most existing datasets can hinder research efforts and lead to impractical and expensive approaches [2,3]. Furthermore, according to Liang *et al.* [106], the arrival rate of commits in the largest open-source projects on Travis, like rails, is not high enough to warrant the use of selection or prioritization algorithms, as there are usually no changes waiting in the queue. Additionally, the duration of the entire test suites is typically only a few minutes, which does not justify parallelization or batching techniques, as mentioned in Beller *et al.*'s work [66].

In contrast, we analyze 276 million test results from Google Chrome in our study [6]. This allows us to better understand the complex challenges involved in optimizing tests for large projects. Moreover, the availability of massive overlapping builds in the Chrome dataset makes the use of various test optimization techniques and parallelism feasible and even necessary.

Previous studies usually relied on subjective and impractical techniques to detect flaky tests, often labeling them as such by repeatedly executing them and identifying inconsistent results. For example, FlakeFlagger baseline [4] re-ran tests 1000 times, leading to an exaggerated rate of flaky tests. However, our study uses genuine flaky labels obtained from the Google Chrome testing infrastructure, which are usually the result of fewer than 10 re-runs of failing tests. This approach results in more precise and practical outcomes.

4.3 Research Design and Methodology

4.3.1 Study Context and Objectives

Software testing is a crucial part of software development, and various test optimization techniques have been used to improve the efficiency and effectiveness of the testing process. However, these techniques have not been compared to each other from different perspectives, especially under resource constraints. For instance, we do not know how the batching algorithms perform in comparison to Elbaum *et al.*'s test selection in terms of feedback time by using different numbers of machines. In this study, we aim to compare different test optimization approaches from different angels and under resource constraints.

The main objective of this study is to compare the effectiveness of different test optimization approaches in terms of reducing feedback time, decreasing the number of machines in use, accelerating the detection failing tests, and saving test execution time with resource constraints. To achieve this objective, we will evaluate four different test optimization approaches, including Kim and Porter's test prioritization, Elbaum *et al.*'s test prioritization and selection, *BatchAll*, and *TestAll* algorithm as the baseline.

The study will use a real-world, large-scale Chrome testing dataset to conduct the empirical study. The test selection and prioritization approaches used in this study are historical-based black box techniques. The effectiveness of the test optimization approaches will be measured based on four key metrics.

The findings of this study will contribute to the software engineering field by providing a comparative analysis of different test optimization approaches under resource constraints. The study will also help practitioners make informed decisions when selecting a test optimization approach for their projects, considering the specific resource constraints they face.

4.3.2 Dataset

Study Site

This empirical study analyzes 276 million Google Chrome test results, as published by Fallahzadeh and Rigby in their recent paper [6]. The project under examination represents a large-scale system that runs millions of tests per day, highlighting the importance of optimizing testing for efficient release processes. The system includes multiple overlapping builds, which provide an opportunity to adopt batching techniques to further optimize the testing process.

One of the notable features of this dataset is that it includes the final outcomes for tests, including flaky flags. This is a significant advantage over other studies that may only identify flaky tests through re-running tests thousands of times, as demonstrated in Alshammari *et al.*'s research [4]. Their approach differs from industrial methods for detecting flaky tests, where tests are re-run only a few times, and only after a failure occurs [6].

Testing Process and Data Types

Fallahzadeh and Rigby [6] provide a comprehensive overview of the Chrome release process and test result classification, including the following steps: developers upload their code changes (known as Change Lists or CLs) to the Gerrit code review tool, where reviewers can suggest modifications. If the reviewers are satisfied with the changes, they go to the Commit Queue (CQ) to be tested by try bots. Once all the try bots approve the changes, they are automatically committed to the Chrome main repository, followed by post-submit testing by the Chrome Continuous Integration (CI) system to ensure the overall stability and quality of the software.

A test result in Google Chrome is not a simple binary outcome, and it is determined in three phases.

Expected Results: Before running a test, Google Chrome's testing infrastructure and developers determine the expectations for the test. The primary expected results are pass, fail, crash, and timeout, which are determined based on the previous test results and the platform on which they are running. By default, all tests are expected to pass unless otherwise specified. A test may be expected to fail on a particular platform, such as Linux, because it is designed for another platform, like Windows.

Immediate Test Results: After a test run, there is an immediate outcome, and the primary immediate test results are pass, fail, crash, and timeout. However, we cannot make any complete interpretations based solely on the immediate results of a test because a test may be flaky.

Final Outcomes: Based on the expected results of a test and its immediate result, the final outcome is determined by the testing infrastructure as follows.

- **Expected:** If the immediate test result is included in the expected results for a test, it is categorized as an expected result. If the immediate test result is not included in the expected results for the test, the test will be re-run several times, typically 2 to 10 times.

- **Flaky:** If after multiple re-runs of a test the immediate test result changes to the expected result, the test is categorized as flaky, and the test rather than the change is flagged for later investigation.

- **Unexpected:** If the immediate test result of a test is still unexpected after multiple re-runs, it is classified as an unexpected outcome.

Both the immediate test results and the final outcomes are collected in the Chrome dataset.

Data Description and Exploration

The Chrome dataset used in this study comprises the change, build, and test results for Chrome from January 1, 2021, to January 31, 2021. During this period, it includes the results for 9,524 change lists, 19,045 builds, 49,932 test suites, and 276,550,812 test cases. To gain a better understanding of the actual performance in Chrome, we present the distribution of various measures in the Chrome dataset below.

Distribution of Feedback Time. Figure 4.1 illustrates the distribution of actual feedback time in Chrome. The feedback time is calculated as the time elapsed between a change being committed and all the tests for that change being run, and is measured for all builds during the entire period under study. The distribution ranges from a minimum of 10.1 minutes to a maximum of about 240.51 minutes, with a median feedback time of 32.95 minutes indicated by a solid line in the figure. The first and second quartiles of the feedback distribution are 28 and 41.06 minutes, respectively, as shown by the dashed lines. Notably, the distribution exhibits a long tail in the fourth quartile, which suggests the presence of a significant number of outliers in this quartile.

Distribution of the number of overlapping builds. Figure 4.2 displays the distribution of the number of overlapping builds in Chrome, which is obtained by comparing the start and end times of each build with those of other builds. In the actual Chrome build execution, each build overlaps with up to 47 other builds, with a median of 12 overlapping builds and the first and third quartiles at 6 and 19 overlapping builds, respectively. This presents an excellent opportunity to implement

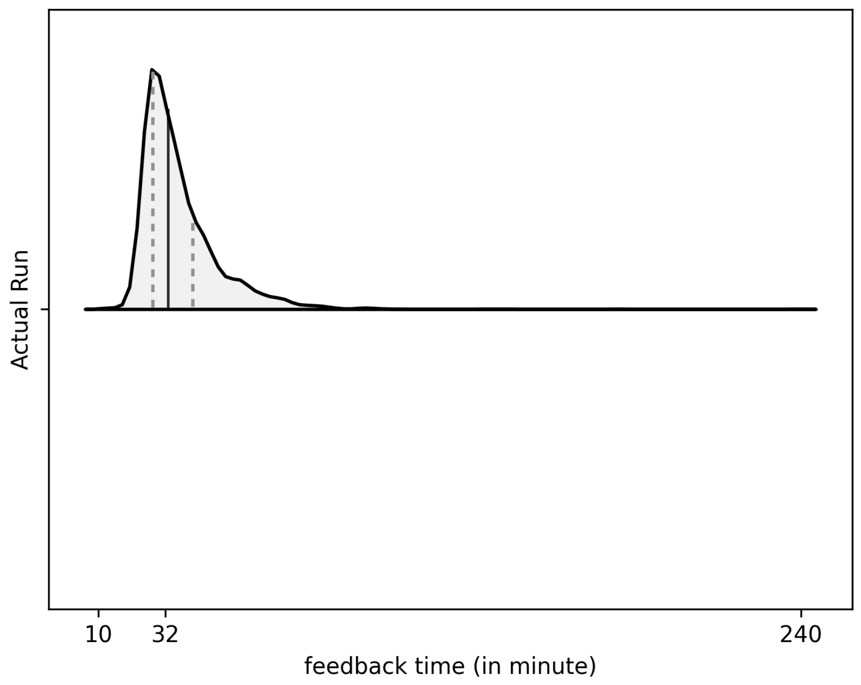


Figure 4.1: Violin plot displaying the distribution of actual feedback times, measured in minutes, for the builds in the Chrome dataset. The median feedback time is indicated by a solid vertical line, while the dashed vertical lines represent the first and third quartiles of the distribution.

batching algorithms as they require the existence of overlapping builds to group and operate on them together.

4.4 Test Optimization Approaches

In this section, we describe different test optimization approaches that are used in this study.

4.4.1 *TestAll*

As a general baseline for comparison, we use the *TestAll* algorithm, which does not involve any test selection, prioritization, or batching. In this approach, the test cases related to each change are processed separately based on their arrival time. By increasing the number of machines in this case, we can execute more test cases simultaneously, which leads to faster execution time.

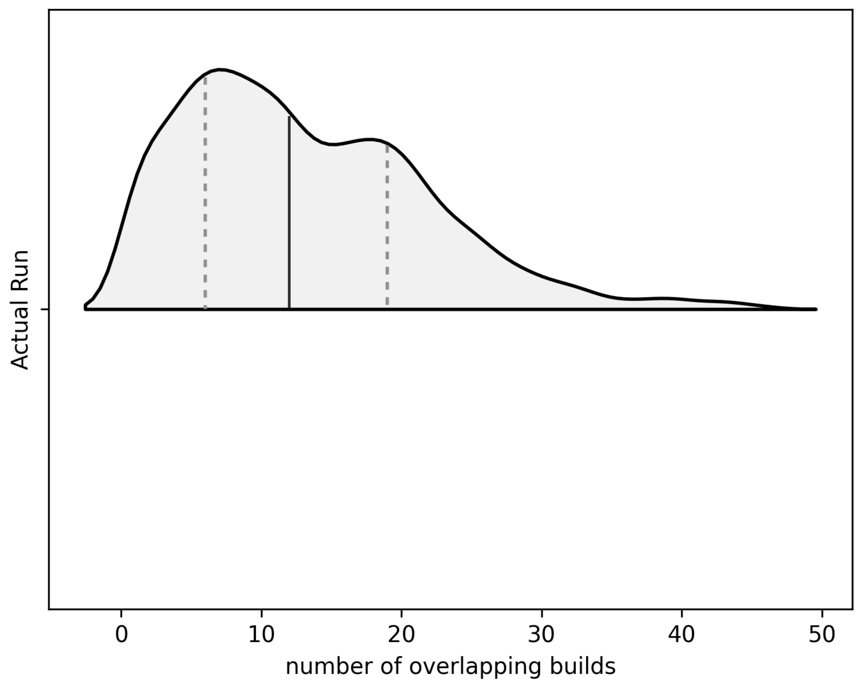


Figure 4.2: Violin plot showing the distribution of the actual number of overlapping builds in Chrome. The median feedback time is indicated by a solid vertical line, while the dashed vertical lines represent the first and third quartiles of the distribution.

4.4.2 *KimPorter*

One of the test optimization techniques widely used to improve failing test detection is test prioritization. In this study, we compare Kim and Porter's test prioritization algorithm [8] with other test optimization approaches to see its effectiveness. Unlike many other techniques that use information such as code coverage for prioritization, Kim and Porter's algorithm relies only on recent past failures to predict future ones. This approach is practical for large projects such as Chrome, which has high code churn and using source code for prioritization is impractical [2].

This algorithm assigns a score to each test, indicating the probability of that test failing, which is defined in formula 7.

$$\begin{aligned} &P_0 = 0 \\ &P_k = \alpha * h_k + (1-\alpha) * P_{k-1},\ 0 \leq \alpha \leq 1,\ k \geq 1 \end{aligned} \tag{7}$$

In this formula, P_k calculates the probability of a test failing at the kth observation based on past failures. The smoothing constant α weights historical observations, with larger α values giving more weight to recent observations. h_k indicates pass or fail, with 1 for failure and 0 for pass. For P_0, the correction made by Mattis *et al.* [14] was used as there is no observation at this phase. Kim and Porter's algorithm uses the prioritization score at each build to prioritize tests, aiming to detect failures faster. The advantage of using test prioritization techniques such as Kim and Porter's algorithm is that no failure is missed, which is an important feature in many software systems.

In this study, we implement Kim and Porter's approach based on the pseudo-code 3 and refer to it as *KimPorter* in this paper. In our simulation of the *KimPorter* algorithm, we set α to 0.8, giving 80% weight to recent failures.

Algorithm 3: *KimPorter*

```
Input  : Build queue $Q_b$
Output: Prioritized test queue $Q_t$
while $Q_b$ is not empty do
    $b \leftarrow$ next build from $Q_b$;
    $Q_t \leftarrow$ tests from $b$;
    for $t \in Q_t$ do
        Calculate $P_k(t)$, the failure probability of $t$;
    end
    Prioritize $Q_t$ based on $P_k(t)$;
    while $Q_t$ is not empty do
        $t \leftarrow$ next test from $Q_t$;
        Dispatch $t$ to available machine;
        Execute $t$ and record the results;
        if $t$ fails then
            Update failure history for $t$;
        end
    end
end
```

4.4.3 *ElbaumPrioritization*

Elbaum *et al.* [1] introduced the concept of using previous failures to predict and prioritize future ones in a Continuous Integration (CI) environment. We choose Elbaum *et al.*'s prioritization algorithm in this study as a representative of test prioritization algorithms designed for Continuous Integration systems to compare its performance with other test optimization techniques. One important reason for this selection is that the algorithm relies only on test results history and does not require expensive information such as source code data, which is impractical in large-scale projects [2].

Elbaum *et al.*'s test prioritization algorithm uses three time windows: failure window (w_f), execution window (w_e), and prioritization window (w_p). The failure window determines the time frame used to consider previous failures. The execution window is used to prevent low-priority tests from waiting too long, i.e., starvation. The prioritization window is used as a time window for prioritizing upcoming tests. Like the *KimPorter* algorithm, Elbaum *et al.*'s approach only considers historical test failures, not source code information, but across builds. It also does not miss any failures, like other test prioritization techniques.

We implement Elbaum *et al.*'s prioritization algorithm in this experiment as described in the pseudo-code algorithm 4, and we refer to it as the *ElbaumPrioritization* in the rest of this study. Windows used in the algorithm can be implemented based on time or the number of builds (Elbaum et al., 2014). In this study, we choose to implement windows based on the number of builds. Specifically, we set w_p to 2 builds, which is equivalent to the time window used in the main study. However, to determine the appropriate sizes for w_f and w_e as they significantly impact the results, we conducted an experiment. The details of this experiment will be further explained in the discussion section. Based on the results of the experiment, we adopt an unlimited window size for w_f and a window size of 2 for w_e for the remaining experiments in our study.

4.4.4 *ElbaumSelection*

Elbaum *et al.*'s test selection algorithm leverages past failures to select probable future ones in a Continuous Integration (CI) environment. In this study, we use Elbaum *et al.*'s test selection as a

Algorithm 4: *ElbaumPrioritization*

```
Input  : Build queue $Q_b$, Prioritization window $w_p$, Failure window $w_f$, and Execution
         window $w_e$
Output: Test prioritization queue $Q_t$
while $Q_b$ is not empty do
    $b \leftarrow$ next builds from $Q_b$ based on $w_p$; $Q_t \leftarrow$ tests from $b$;
    for $t \in Q_t$ do
        if $t$ has recent failures ($t \in w_f$), is new, or has not been executed for a long time
         ($t \notin w_e$) then
            Set the priority of $t$ to 1;
        end
        else
            Set the priority of $t$ to 0;
        end
    end
    Prioritize $Q_t$ based on priority;
    while $Q_t$ is not empty do
        $t \leftarrow$ next test from $Q_t$;
        Dispatch $t$ to available machine;
        Execute $t$ and record the results;
        if $t$ fails then
            Update failure history for $t$;
        end
    end
end
```

representative of test selection algorithms to compare its performance with other test optimization techniques. The reason for selecting this algorithm is that it only relies on a test's failure history to predict future failures and does not require any additional information such as source code, which is impractical in large-scale projects as mentioned by Memon *et al.* [2].

Elbaum *et al.*'s test selection algorithm utilizes two failure and execution windows. The failure window, w_f, determines a time window to look back at the previous failures. The execution window, w_e, defines an aging window that prevents a test from not being executed for longer than this time window. The important difference of this algorithm as a test selection approach is that we might lose some of the failures in contrast to other techniques that do not miss failures.

Elbaum *et al.*'s test selection algorithm uses two windows, failure window (w_f) and execution window (w_e). The failure window determines a time frame to look back at the previous failures while the execution window sets a threshold for test execution. Tests that have not been executed for a period longer than the execution window are given higher priority. This algorithm may miss some failures as any other test selection algorithm.

The pseudo-code implementation of this algorithm is as follows and we refer to it as *ElbaumSelection* in the rest of this paper. In this study, we implement *ElbaumSelection* windows based on the number of builds, and we experiment with different values for w_f and w_e to find the appropriate values for Chrome.

Algorithm 5: *ElbaumSelection*

```
Input  : Build queue $Q_b$, Failure window $w_f$, and Execution window $w_e$
Output: Test selection queue $Q_t$
while $Q_b$ is not empty do
    $b \leftarrow$ next build from $Q_b$;
    $Q_t \leftarrow$ tests from $b$;
    for $t \in Q_t$ do
        if $t$ has recent failures ($t \in w_f$), is new, or has not been executed for a long time
          ($t \notin w_e$) then
            Set the priority of $t$ to 1;
        end
        else
            Set the priority of $t$ to 0;
        end
    end
    Select tests in $Q_t$ with a score of 1;
    while $Q_t$ is not empty do
        $t \leftarrow$ next test from $Q_t$;
        Dispatch $t$ to available machine;
        Execute $t$ and record the results;
        if $t$ fails then
            Update failure history for $t$;
        end
    end
end
```

4.4.5 *BatchAll*

Batching is an effective yet often overlooked test optimization technique that has gained prominence as the number of overlapping changes and builds has increased. Therefore, it is important to compare the advantages and disadvantages of this approach in comparison to other test optimization techniques. In batching algorithms, multiple changes are batched together, and the union of all their tests is run. When the batch successfully runs all tests, there is a significant save in test runs. However, when the batch fails, there is a penalty to find the culprit change that is responsible for the failure. There are various batching algorithms and culprit-finding techniques [41].

In this study, we utilize the *BatchAll* algorithm, which demonstrated promising results in our recent work [33] comparing various batching techniques. Unlike waiting to receive a specific number of changes (builds) [41], *BatchAll* batches all the available changes waiting for the build. This approach avoids waiting for a specific number of changes when there are only a few available and increases the batch size when there are numerous changes. In case of a batch failure, the algorithm carries out the culprit-finding phase. We further improved this part by considering only the tests that have failed in the batch, rather than all the tests.

Assumptions for batching: Since the test results of batches are not available in the Chrome dataset, we make the following assumptions in our simulations:

(1) We assume that a test that shows an unexpected result in one of the builds included in the batch, also produces an unexpected outcome in the batch with the same number of re-runs. We consider these types of outcomes as a breakage for the batch that requires the culprit-finding process.

(2) We suppose that a test that displays a flaky result in one of the builds included in the batch produces a flaky result in the batch as well, and it is re-run the same number of times. However, flaky tests do not lead to a breakage for the batch and result the integration of the batch.

(3) Other types of tests in the batch are supposed to display expected results, which is considered as pass resulting the integration of the batch.

Figure 4.3 shows the summary of the decision tree for each type of test results in the builds included in the batch. We define the pseudo-code for the batching algorithm as the pseudo-code 6.

4.5 Evaluation Criteria

This section defines the different metrics used to evaluate test optimization algorithms.

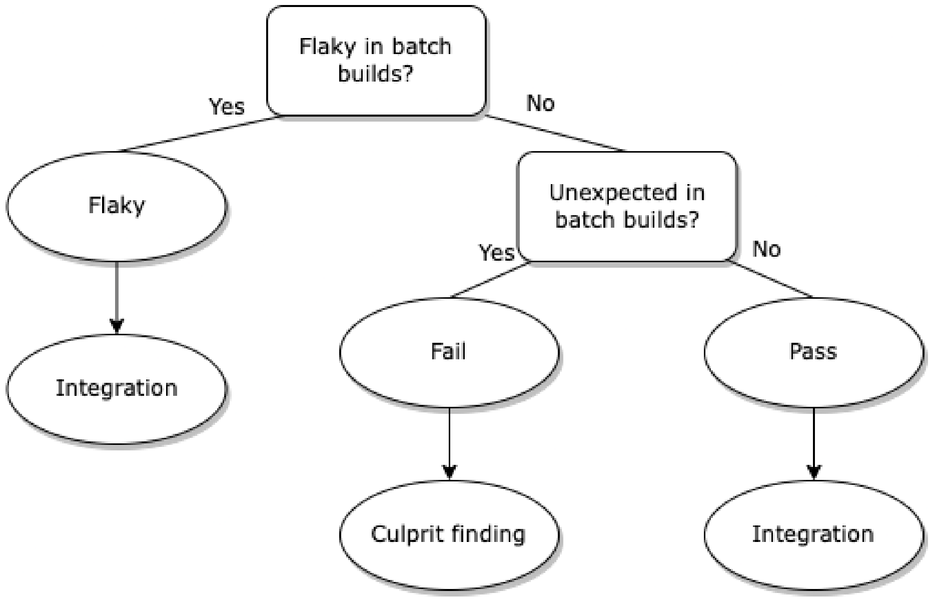


Figure 4.3: Decision tree for each type of test results in the builds included in the batch.

4.5.1 Feedback Time

One of the primary objectives of test optimization algorithms is to minimize feedback time, especially in Continuous Integration (CI) systems, to ensure prompt delivery of test results. Feedback time refers to the duration between the time a change is committed and the time all test results are received. It is a critical parameter that determines the effectiveness of testing.

We calculate feedback time using the following formula:

$$\text{FeedbackTime} = \text{Time}(\text{TestResults}) - \text{Time}(\text{ChangeCommit}) \tag{8}$$

Let's consider an example. Suppose a software developer commits a code change at 1 PM and wants to receive the test verdicts as soon as possible. If the feedback time is 1 hour and there is no waiting time before the tests can be executed, the developer will receive the test verdicts at 2 PM. However, if there is a waiting time of 1 hour before the tests can be executed, the feedback time will be 2 hours, and the developer will not receive the test verdicts until 3 PM.

To evaluate different test optimization algorithms, we use the median feedback time for each algorithm and a specific number of machines. This approach enables us to compare the performance of each algorithm under various machine configurations and identify the most efficient algorithm

Algorithm 6: *BatchAll*

```
Input  : Build queue $Q_b$
Output: Batched test execution
while $Q_b$ is not empty do
    $batch$ ← make a batch from available and arrived builds in $Q_b$;
    $testSet$ ← set of all unique tests across $batch$;
    $Q_t$ ← fill test queue with tests from $testSet$;
    while $Q_t$ is not empty do
        $t$ ← next test from $Q_t$;
        Dispatch $t$ to an available machine;
        Execute $t$ and record the results;
        if $t$ fails then
            Record the failing test in $batch$;
        end
    end
    if $batch$ failed then
        for each failed test $t_f$ in $batch$ do
            for each build $b$ in $batch$ do
                Execute $t_f$ in $b$ and record the result;
                if $t_f$ fails then
                    Update failure history for $t_f$;
                end
            end
        end
    end
end
```

for a given set of resources.

4.5.2 Number of Machines in Use

One of the metrics that we use to measure resource usage in this study is the number of machines in use. We measure the feedback time of different algorithms by using different numbers of machines. Then, we determine the number of machines required for different algorithms to reach the actual feedback time.

Baseline. To establish a baseline for comparison, we utilize the *TestAll* algorithm, which serves as a general reference without employing any optimization techniques. However, since we lack information about the precise number of machines used by Chrome to achieve the actual median feedback time of 32.95 minutes, we employ a specific configuration of *TestAll* with 179 machines as a baseline. This choice allows us to assess the performance of other test optimization algorithms in relation to the actual performance achieved. With this configuration, the feedback time amounts to 33.55 minutes, which is the closest approximation to the actual median feedback time of 32.95 minutes within our dataset. Consequently, this particular baseline serves as a reference point for evaluating the performance of other test optimization algorithms in this study.

4.5.3 GAINEDTIME

Another crucial metric for evaluating test optimization algorithms is their ability to quickly detect failing tests. This research aims to evaluate the change in test run time across multiple builds. Although the APFD metric is commonly used for this purpose, Elbaum *et al.* [1] discovered that it is not appropriate for use in large-scale continuous integration environments because it fails to take into account tests across builds. To address this challenge, Elbaum *et al.* [1] and our previous works [6, 33] introduced a different metric, GAINEDTIME, to measure the time saved by each approach A compared to *TestAll*. It is defined as:

$$\text{GAINEDTIME (A)} = \text{FailTime}(\textit{TestAll}) - \text{FailTime(A)} \tag{9}$$

We determine the time saved by each test optimization algorithm for every individual failed test by comparing their run times against *TestAll*, which runs tests in the order they arrive. By doing so, we can calculate the amount of time saved by each algorithm for every failed test. We then use the median GAINEDTIME to compare the performance of each algorithm that uses a specific number of machines with other algorithms.

4.5.4 Execution Reduction

To assess the efficiency of various algorithms in terms of saving execution time by utilizing different numbers of machines, we employ the EXECUTIONREDUCTION metric. This metric is calculated using Equation 10.

$$\text{EXECUTIONREDUCTION}_m(A) = 1 - \frac{\sum_{t=1}^{T} \text{Test Execution Time (A, m)}}{\sum_{t=1}^{N} \text{Test Execution Time } (\textit{TestAll})} \tag{10}$$

This equation quantifies the time saved in test execution by a test optimization algorithm (A) with m machines relative to executing all tests in their original order (*TestAll*). Here, T represents the number of tests executed by the test optimization algorithm (A) with m machines, and N is the total number of tests executed by *TestAll*.

The time saved is determined by dividing the sum of the test execution time of approach A running with m machines by the sum of the test execution time of the *TestAll* algorithm. This ratio represents the percentage of time required to run the tests using approach A compared to *TestAll*. Subtracting this percentage from 1 yields the percentage of time saved.

For instance, if approach A completes tests in 70% of the time taken by *TestAll*, the percentage of time saved by A is obtained by subtracting this percentage from 1, resulting in 30% of EXECUTIONREDUCTION. This implies that approach A achieves a 30% reduction in test execution time. By utilizing this formula as a metric, we can compare the performance of different test optimization algorithms and determine which ones are more effective in terms of saving time during test execution.

4.5.5 Experimental Setup

Simulated Hardware Environment

In this study, we simulate a range of different numbers of machines to evaluate parallelism on various test optimization algorithms. We divide the test executions algorithmically among the number of machines to ensure accurate results. Our range of machines in use ranges from 1 to 400, with a standard step size of 25. However, we use more precise steps to thoroughly examine the changes when we observe significant differences in results, such as between 1 to 25 machines. This provides a detailed analysis of the performance of various test optimization algorithms under a variety of machine configurations.

Dealing with Test Results

To conduct our simulation, we determine a set of assumptions about the way to treat different types of test results as follows.

- **Expected:** These types of results in Chrome are treated as passing tests, which do not block the build. So, we consider these types of tests as passing tests that do not require any additional runs.

- **Flaky:** These outcomes in Chrome do not stop a build from being integrated into the main repository. As such, we also consider these types of test results as passing outcomes in our simulation. However, they cause re-runs of the tests that we take into account in the experiment.

- **Unexpected:** These types of results in Chrome are the main sources of blocking the builders. Because of this, we consider these types of tests as failing tests in our simulation. They usually require re-runs so that the testing infrastructure assures that the failure is consistent. Hence, we consider the re-runs from unexpected results in our simulation.

4.6 Results

This section presents the answer to each research question and displays the comparative results of running simulations.

4.6.1 RQ1: How effective are different test optimization approaches in reducing feedback time when executed in parallel with varying numbers of machines?

In this research question, we aim to understand the impact of using different test optimization approaches on feedback time in a in parallel environment with varying numbers of machines. This is important as the primary goal of most test optimization algorithms is to reduce the feedback time in getting the results from the builds.

To answer this research question, we execute *TestAll* as our baseline, *ElbaumSelection*, *Kim-Porter*, *ElbaumPrioritization*, and *BatchAll* algorithms on the Chrome test results. We use different numbers of machines to execute these algorithms and obtain the median feedback time results for each algorithm and each number of machines.

We present the results of the median feedback time for the executions in Figure 4.4 and 4.5. The results are shown as a continuous line by connecting the dots to enable better visualization of the outcomes. Figure 4.4 shows the median feedback time for each available number of machines and each test optimization algorithm. Since the difference between the results is significant, we use a logarithmic scale to show the results.

The results illustrated in Figure 4.4 indicate that both *KimPorter* and *ElbaumPrioritization* algorithms have a performance close to that of the *TestAll* algorithm. In resource-constrained environments with much fewer machines than the baseline, *BatchAll* algorithm shows a significantly better performance compared to the other algorithms. However, the *ElbaumSelection* algorithm shows better improvement in performance with an increase in the number of machines until it outperforms *BatchAll* when 75 machines are used. In resource-abundant environments, *ElbaumSelection* performs better than the other algorithms.

Figure 4.5 displays the median feedback time in minutes and the improvements achieved by each test optimization algorithm relative to the actual median feedback time of 32.95 minutes when using the baseline 179 number of machines. This figure allows us to compare the performance of different algorithms with the actual performance and shows the results when changing the median feedback time up to 100% away from the actual median feedback time.

Figure 4.5 also presents the 95% confidence intervals for the median feedback time values of all the algorithms analyzed in Chrome. The confidence interval range varies depending on the algorithm and the number of machines used.

For the *TestAll*, *KimPorter*, and *ElbaumPrioritization* algorithms, the confidence interval range narrows down as the number of machines increases from 175 to 250 and beyond. The reduction in the width of the confidence interval indicates a decrease in the range of the feedback time distribution. Specifically, the range decreases from about 16% to less than 1% of the actual median feedback time.

In the case of the *ElbaumSelection* algorithm, the confidence interval range is less than 1% when the number of machines is 75 and decreases to about 0% as the number of machines increases to 375 and beyond. Similarly, for the *BatchAll* algorithm, the confidence interval range becomes less than 1% when the number of machines is 25 and beyond. It approaches 0% as the number of machines reaches 150 and beyond.

To evaluate and compare the feedback time distributions of various test optimization algorithms for Chrome, we employ the Wilcoxon Rank-Sum test. This statistical test is specifically designed for comparing two independent samples without making assumptions about their underlying distributions. To account for the challenge of conducting multiple comparisons, we employ the Bonferroni correction method. This correction involves dividing the desired overall p-value of 0.05 by the number of comparisons being made. As a result, each individual comparison is subject to a strict significance threshold, with a p-value cutoff of 0.00028. This adjustment ensures that any observed differences between the algorithms are statistically significant.

Table 4.1 presents the calculated p-values for comparing the feedback time distribution between different test optimization algorithms. The findings reveal that there are no statistically significant differences in the distribution between *TestAll* and *KimPorter*, *TestAll* and *ElbaumPrioritization* when the number of machines is below 175, as well as between *KimPorter* and *ElbaumPrioritization* within the same machine range. However, for the remaining comparisons, the p-values indicate statistically significant differences, meeting the predetermined cutoff of 0.00028. These results align closely with the observed overlaps in the median feedback time depicted in Figure 4.4.

To quantify the magnitude and direction of differences between feedback time distributions, we employ Cliff's delta effect size measure. The resulting effect sizes between different algorithms at varying machine counts are presented in Table 4.2. This additional data reinforces the observed patterns and highlights the significance of the differences depicted in the median feedback time plot illustrated in Figures 4.4 and 4.5.

Table 4.3 presents the median feedback time achieved by each algorithm with 179 machines as the baseline and the corresponding improvements compared to the actual median feedback time. *KimPorter* achieves the same feedback time as the baseline with no improvements. *ElbaumPrioritization* achieves a median feedback time of 31.20 minutes with 5.31% improvement, while the *BatchAll* algorithm achieves a significantly better median feedback time of 2.69 minutes with 91.84% improvement. The *ElbaumSelection* approach outperforms all other algorithms with a median feedback time of 1.28 minutes and 96.12% improvement. However, using the *ElbaumSelection* algorithm comes with a tradeoff of missing some failing tests, i.e. 31.25% of failures in this experiment.

Another important aspect of Figure 4.5 is identifying where each test optimization algorithm reaches a plateau, as some algorithms may perform well initially but plateau too soon with no further room for improvement. We consider a plateau to occur when adding 25 machines does not lead to more than a 1% improvement in feedback time. Table 4.4 shows that the *ElbaumSelection* and *BatchAll* algorithms both plateau with 200 machines, but the former achieves a median feedback time of 0.96 minutes while the latter achieves a median feedback time of 2.4 minutes. The *ElbaumPrioritization* algorithm reaches a plateau with 250 machines and a median feedback time of 1.95 minutes, while the *TestAll* and *KimPorter* algorithms plateau with 325 machines and a median feedback time of 1.56 minutes.

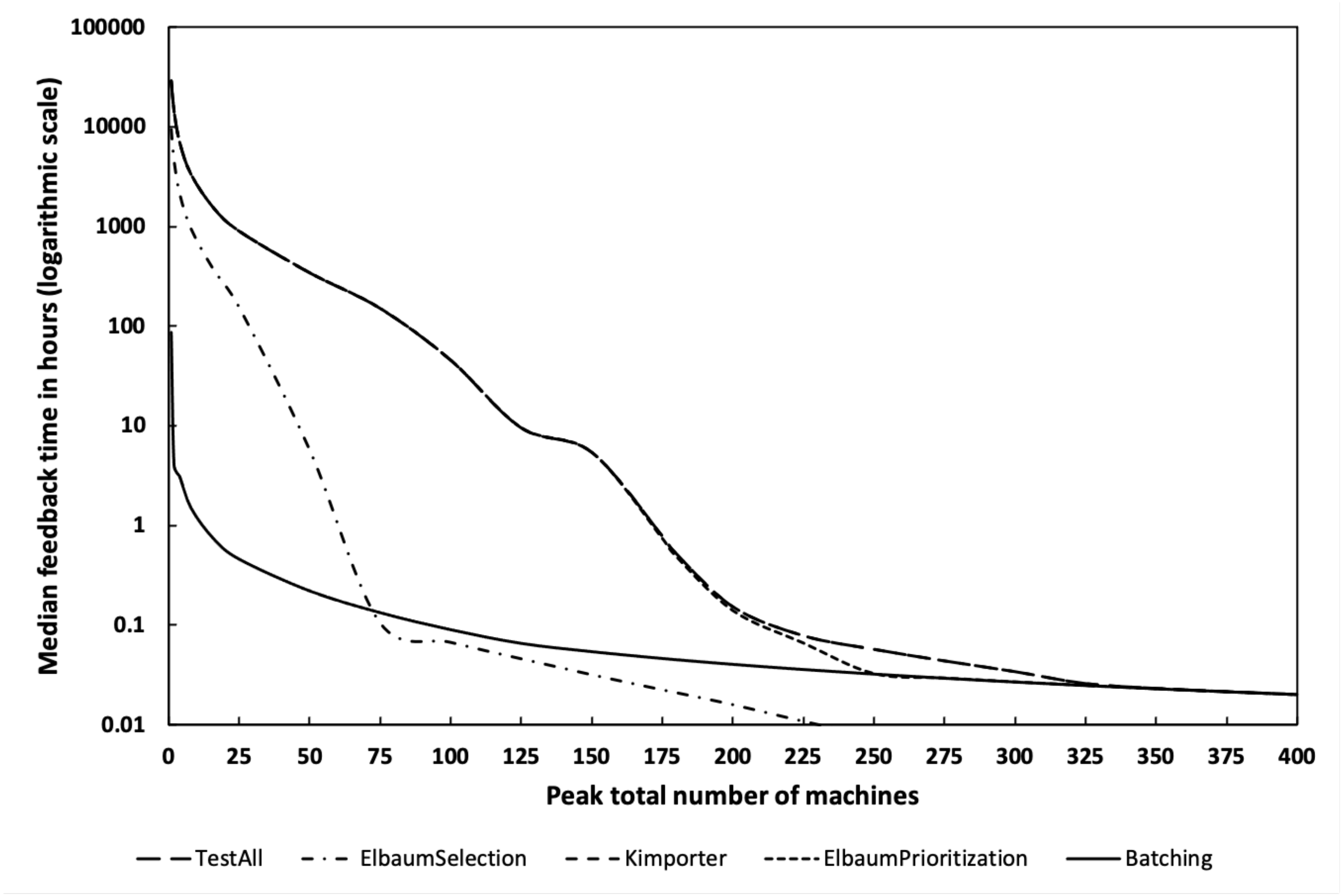


Figure 4.4: Median feedback time for *TestAll*, *ElbaumSelection*, *KimPorter*, *ElbaumPrioritization*, *BatchAll* algorithms displayed in logarithmic scale and in hours by considering the different peak numbers of machines.

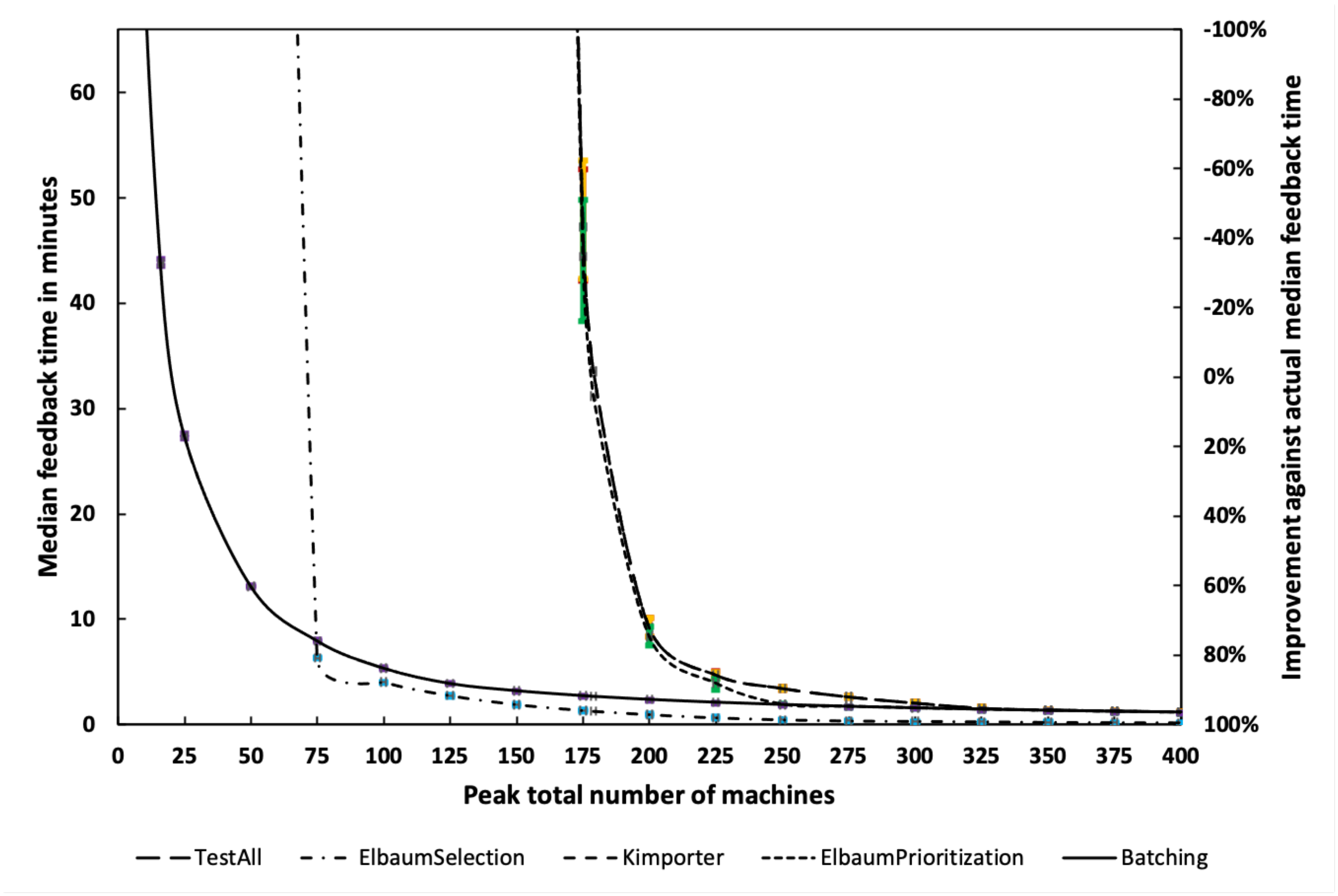


Figure 4.5: Median feedback time in minutes with 95% confidence intervals and the improvement against the actual median feedback time of 32.95 minutes for *TestAll*, *ElbaumSelection*, *KimPorter*, *ElbaumPrioritization*, *BatchAll* approaches by considering different peak numbers of machines.

RQ1: Kim and Porter's test prioritization algorithm shows almost no improvement in feedback time, while Elbaum's test prioritization approach reduces feedback time by 5.31% to 31.20 minutes using the same number of machines as baseline. Elbaum's test selection approach achieves a 96.12% reduction in feedback time to 1.28 minutes with the same number of machines as the baseline, but misses 31.25% of failures. The Batching algorithm technique achieves a 91.84% reduction in feedback time to 2.69 minutes using the same number of machines as the baseline, while being lossless in failure detection.

Table 4.1: P-values for pairwise comparison of the feedback time distribution of different test optimization algorithms for Chrome at different numbers of machines.

Machines	**TA vs KP**	**TA vs EP**	**TA vs ES**	**TA vs BA**	**KP vs EP**	**KP vs ES**	**KP vs BA**	**EP vs ES**	**EP vs BA**	**ES vs BA**
8	1	0.992355	0	0	0.992355	0	0	0	0	0
16	1	0.97506	0	0	0.97506	0	0	0	0	0
25	1	0.935696	0	0	0.935696	0	0	0	0	0
50	1	0.994567	0	0	0.994567	0	0	0	0	0
75	1	0.911593	0	0	0.911593	0	0	0	0	1E-20
100	1	0.842338	0	0	0.842338	0	0	0	0	0
125	1	0.323962	0	0	0.323962	0	0	0	0	0
150	1	0.064336	0	0	0.064336	0	0	0	0	0
175	1	0.00027	0	0	0.00027	0	0	0	0	0
200	1	5.69E-10	0	0	5.69E-10	0	0	0	0	0
225	1	6E-15	0	0	6E-15	0	0	0	0	0
250	1	1E-20	0	0	1E-20	0	0	0	0	0
275	1	0	0	0	0	0	0	0	0	0
300	1	0	0	0	0	0	0	0	0	0
325	1	0	0	0	0	0	0	0	0	0
350	1	0	0	0	0	0	0	0	0	0
375	1	0	0	0	0	0	0	0	0	0
400	1	0	0	0	0	0	0	0	1.88E-16	0

*Abbreviations: TA: *TestAll*, KP: *KimPorter*, EP: *ElbaumPrioritization*, ES: *ElbaumSelection*, BA: *BatchAll*.

4.6.2 RQ2: How effective are different test optimization approaches in reducing the number of machines in use when executed in parallel with varying numbers of machines?

In this research question, we investigate how different test optimization algorithms can help reduce the number of machines required for running tests. This is important as large companies may have to invest millions of dollars in machines, making it a critical cost factor [108].

To answer this research question, we use our baseline median feedback time of 33.55 minutes as the approximate actual median feedback and target time. We then find the number of machines each algorithm requires to achieve this target.

The results of our experiment are summarized in Table 4.5. Among the algorithms studied, *ElbaumSelection* requires the fewest machines to meet the target, using only 75 machines, which is a 58.10% reduction in machine usage compared to the baseline. However, the algorithm may miss some failing tests, which account for 31.25% of the failures in this experiment.

While *KimPorter* and *ElbaumPrioritization* algorithms can maintain the baseline feedback time, they do not significantly reduce machine usage, with the latter achieving only a negligible 0.56%

Table 4.2: Cliff's Delta effect size for pairwise comparison of the feedback time distribution of different test optimization algorithms for Chrome at different numbers of machines.

Machines	**TA vs KP**	**TA vs EP**	**TA vs ES**	**TA vs BA**	**KP vs EP**	**KP vs ES**	**KP vs BA**	**EP vs ES**	**EP vs BA**	**ES vs BA**
8	0	-9.2E-05	0.706822	0.99888	-9.2E-05	0.706822	0.99888	0.706818	0.998581	0.962966
16	0	0.000299	0.751347	0.989009	0.000299	0.751347	0.989009	0.75105	0.982282	0.943112
25	0	0.000771	0.816132	0.978437	0.000771	0.816132	0.978437	0.815469	0.968266	0.93275
50	0	6.51E-05	0.932567	0.962712	6.51E-05	0.932567	0.962712	0.931987	0.955843	0.498557
75	0	0.001061	0.962656	0.954267	0.001061	0.962656	0.954267	0.960958	0.945542	-0.0897
100	0	0.0019	0.95304	0.907298	0.0019	0.95304	0.907298	0.947478	0.88364	-0.46881
125	0	0.009422	0.934917	0.801704	0.009422	0.934917	0.801704	0.921641	0.741214	-0.68418
150	0	0.01767	0.936875	0.739204	0.01767	0.936875	0.739204	0.922936	0.665974	-0.76838
175	0	0.034798	0.928554	0.664818	0.034798	0.928554	0.664818	0.910099	0.555705	-0.79714
200	0	0.059211	0.921712	0.585619	0.059211	0.921712	0.585619	0.898798	0.438666	-0.81809
225	0	0.074544	0.917737	0.52128	0.074544	0.917737	0.52128	0.893508	0.360126	-0.83418
250	0	0.088959	0.915567	0.473919	0.088959	0.915567	0.473919	0.892155	0.303032	-0.84622
275	0	0.111049	0.911448	0.425083	0.111049	0.911448	0.425083	0.887479	0.240758	-0.85304
300	0	0.131509	0.908903	0.386175	0.131509	0.908903	0.386175	0.884227	0.190122	-0.85798
325	0	0.147072	0.906212	0.351274	0.147072	0.906212	0.351274	0.880944	0.146515	-0.86138
350	0	0.165802	0.905009	0.327571	0.165802	0.905009	0.327571	0.878711	0.111331	-0.86423
375	0	0.16668	0.903142	0.301935	0.16668	0.903142	0.301935	0.87832	0.092184	-0.86692
400	0	0.168428	0.902245	0.284956	0.168428	0.902245	0.284956	0.878464	0.078608	-0.86885

*Abbreviations: TA: *TestAll*, KP: *KimPorter*, EP: *ElbaumPrioritization*, ES: *ElbaumSelection*, BA: *BatchAll*.

Table 4.3: The median feedback time in minutes and the corresponding improvements achieved by different algorithms by using 179 baseline number of machines.

Algorithm	**TestAll**	**ElbaumSelection**	**KimPorter**	**ElbaumPrioritization**	**Batching**
Median Feedback time	33.55 (-1.81%)	1.28 (96.12%)	33.55 (-1.81%)	31.20 (5.31%)	2.69 (91.84%)

reduction. This is because test prioritization algorithms primarily focus on prioritizing tests and not on reducing machine usage.

In contrast, the *BatchAll* algorithm achieves a remarkable 88.27% reduction in machine usage compared to the baseline, requiring only 21 machines to hold the target feedback time.

RQ2: Kim and Porter's test prioritization approach shows no improvement in machine usage, while Elbaum's test prioritization approach reduces machine usage slightly by 0.56% with 178 machines to achieve the same feedback time. Elbaum's test selection approach reduces machine usage by 58.10% with 75 machines while maintaining the baseline feedback time of 33.55 minutes, but it loses 31.25% of failures. The Batching algorithm approach achieves an 88.27% reduction in machine usage by using only 21 machines while maintaining the baseline feedback time and not missing any failures.

Table 4.4: The number of machines and the median feedback time in minutes where each algorithm plateaus for Chrome. (improvement to the next 25 is less than 1%)

Algorithm	**ElbaumSelection**	**KimPorter**	**ElbaumPrioritization**	**Batching**	***TestAll***
Median feedback time	0.96	1.56	1.95	2.40	1.56
Number of machines	200	325	250	200	325

Table 4.5: The necessary quantity of machines for each algorithm to maintain the actual median feedback time of 32.95 minutes for Chrome, and the corresponding percentage of reduction in machine usage compared to the baseline number of machines.

Algorithm	**TestAll**	**ElbaumSelection**	**KimPorter**	**ElbaumPrioritization**	**Batching**
Number of machines	179 (0% baseline)	75 (58.10%)	179 (0%)	178 (0.56%)	21 (88.27%)

4.6.3 RQ3: How quickly do different test optimization techniques detect failing tests when executed in parallel with varying numbers of machines?

Aside from reducing the number of machines needed for running tests, another important aspect of test optimization is how quickly the algorithms can detect failing tests. While this measure is typically used for test prioritization algorithms, this study aims to investigate the impact of other test optimization approaches on this measure, providing a broader view of the implications of these algorithms.

To achieve this goal, we use the GAINEDTIME measure, which is similar to measures used in previous studies by Elbaum *et al.* [52] and Fallahzadeh and Rigby [6]. This measure represents the difference in time it takes to detect each individual failing test. We use the median GAINEDTIME obtained by each algorithm to compare them. To provide a fair comparison between algorithms, we focus on the failing tests detected by *ElbaumSelection*, which represents approximately 68.75% of all failing tests, as *ElbaumSelection* misses some failing tests.

The resulting median GAINEDTIME for different test optimization algorithms in this study using varying numbers of machines is presented in Figure 4.6. Each line in the figure represents the results by connecting the non-continuous result points for each approach. To better illustrate the differences between the median GAINEDTIME for each algorithm, we show the results in logarithmic scale and add 1 GAINEDTIME unit to the results.

As shown in Figure 4.6, the median GAINEDTIME achieved by all test optimization algorithms in this study is much better in highly resource-constrained environments. As the number of machines increases, the performance of these algorithms diminishes significantly. The *KimPorter* algorithm shows some improvement, while *ElbaumPrioritization* displays better results. *ElbaumSelection* and *BatchAll* algorithms achieve much better GAINEDTIME results than the prioritization algorithms, particularly in highly resource-constrained environments. The *BatchAll* algorithm achieves better results than *ElbaumSelection* in highly resource-constrained environments, while *ElbaumSelection* performs slightly better in highly resource-available environments. However, we should keep in mind that *ElbaumSelection* misses some failing tests, which represent 31.25% of all failures in this experiment.

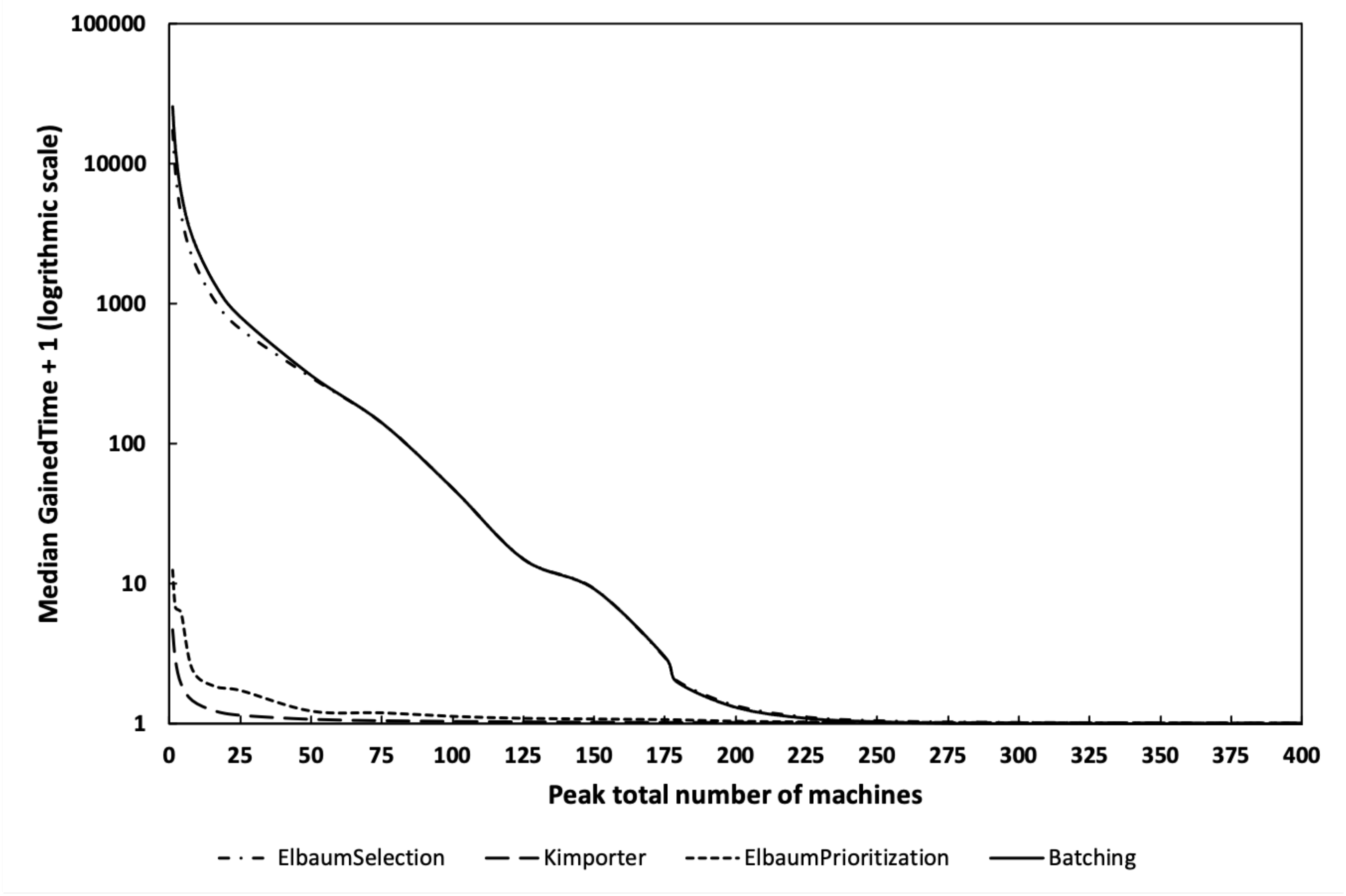


Figure 4.6: Median GAINEDTIME + 1 in hours for *ElbaumSelection*, *KimPorter*, *ElbaumPrioritization*, and *BatchAll* algorithms displayed in logarithmic scale and in hours by considering the different peak number of machines.

To assess and compare the distributions of GAINEDTIME across different test optimization algorithms for Chrome, we utilize statistical techniques such as the Wilcoxon Rank-Sum test and Cliff's delta effect size measure. To account for multiple comparisons, we apply the Bonferroni correction, dividing the desired overall p-value of 0.05 by the number of comparisons conducted. Consequently, each individual comparison is subject to a stringent significance threshold, with a p-value cutoff of 0.00046.

Table 4.6 presents the computed p-values and effect sizes for comparing the GAINEDTIME distributions between the various test optimization algorithms. The results indicate that most of the comparisons yield statistically significant differences, except for a few cases. Specifically, no significant differences are observed between *KimPorter* and *BatchAll* when using 250 and 275 machines. Similarly, there are no significant differences between *ElbaumPrioritization* and *ElbaumSelection* when the number of machines is 275 or higher. Moreover, no significant differences are found between *ElbaumPrioritization* and *BatchAll* when employing 225 and 250 machines, as well as between *ElbaumSelection* and *BatchAll* when utilizing 50 to 225 machines. These findings closely align with the observed overlaps in the median GAINEDTIME, as depicted in Figure 4.6.

Additionally, Table 4.6 includes the effect sizes between different algorithms at various machine counts. This supplementary information reinforces the observed patterns and emphasizes the significance of the differences illustrated in the GAINEDTIME plot shown in Figure 4.6.

To provide a more accurate and realistic comparison between the different test optimization algorithms based on GAINEDTIME, we analyzed the results at the baseline number of machines, which is 179 machines, and is closer to the actual number of machines used. The findings are presented in Table 4.7.

The table reveals that with the same number of machines as the baseline, *KimPorter* algorithm detects failing tests 1.2 minutes faster than *TestAll*. The *ElbaumPrioritization* approach achieves even better results by reducing the median time to detect failing tests by 3.9 minutes. Remarkably, the *BatchAll* algorithm outperforms all other algorithms by detecting failing tests earlier by a median of 59.4 minutes. Despite missing 31.25% of the failures in this experiment, the *ElbaumSelection* algorithm still achieves the best GAINEDTIME of 62.1 minutes.

Table 4.6: P-values and Cliff's Delta effect sizes for pairwise comparison of the GAINEDTIME distribution of different test optimization algorithms for Chrome at different numbers of machines.

	P-value						Effect Size					
Machines	**KP* vs EP**	**KP vs ES**	**KP vs BA**	**EP vs ES**	**EP vs BA**	**ES vs BA**	**KP vs EP**	**KP vs ES**	**KP vs BA**	**EP vs ES**	**EP vs BA**	**ES vs BA**
8	0	0	0	0	0	5.93E-09	-0.54259	-1	-1	-1	-1	-0.27554
16	0	0	0	0	0	1.47E-06	-0.98953	-1	-1	-0.99929	-0.99616	-0.22593
25	0	0	0	0	0	0.000439	-0.99403	-1	-0.99068	-0.98868	-0.98601	-0.16079
50	0	0	0	0	0	0.278022	-0.54259	-0.99152	-0.98601	-0.9868	-0.98376	-0.02846
75	0	0	0	0	0	0.463815	-0.79001	-0.99073	-0.9745	-0.97269	-0.96711	-0.0044
100	0	0	0	0	0	0.464619	-0.69656	-0.96425	-0.93899	-0.93679	-0.92533	-0.0043
125	0	0	0	0	0	0.357905	-0.5659	-0.91359	-0.78723	-0.81407	-0.75556	-0.0176
150	0	0	0	0	0	0.291761	-0.67676	-0.87424	-0.72776	-0.73092	-0.66952	-0.0265
175	0	0	0	0	0	0.139835	-0.68641	-0.84944	-0.61543	-0.6247	-0.52125	-0.05225
200	0	0	1.98E-17	0	7.95E-11	0.013678	-0.57934	-0.76237	-0.4066	-0.49655	-0.30911	-0.10663
225	0	0	7.14E-07	2.46E-16	0.001809	0.000307	-0.53998	-0.6871	-0.23297	-0.39207	-0.14062	-0.16553
250	0	0	0.012783	4.38E-12	0.385449	2.24E-06	-0.48594	-0.6397	-0.1079	-0.32984	-0.01408	-0.22172
275	0	0	0.037218	0.00059	8.4E-07	3.13E-12	-0.54196	-0.57385	-0.08621	-0.15676	-0.2314	-0.33217
300	0	0	1.98E-05	0.003117	8.25E-12	9.82E-17	-0.4665	-0.53154	-0.19862	-0.13218	-0.32542	-0.39743
325	0	0	1.48E-09	0.097034	1.04E-18	0	-0.45958	-0.49505	-0.28676	-0.06277	-0.42298	-0.45411
350	1E-20	2E-20	1.25E-15	0.447366	0	0	-0.44719	-0.44475	-0.38239	-0.00641	-0.51408	-0.51888
375	1.25E-18	8.55E-18	0	0.427828	0	0	-0.42195	-0.41132	-0.46193	-0.0088	-0.56868	-0.56646
400	1.03E-16	3.89E-15	0	0.269933	0	0	-0.39713	-0.37552	-0.49836	-0.02963	-0.60208	-0.587

*Abbreviations: KP: *KimPorter*, EP: *ElbaumPrioritization*, ES: *ElbaumSelection*, BA: *BatchAll*.

Table 4.7: The median GAINEDTIME in minutes achieved by different algorithms by using 179 machines as in the *TestAll* baseline for 68.75% of failures by ElbaumSelection.

Algorithm	**ElbaumSelection**	**KimPorter**	**ElbaumPrioritization**	**Batching**
Median GAINEDTIME	62.1	1.2	3.9	59.4

RQ3: Kim and Porter's test prioritization detects failing tests faster by a median of 1.2 minutes, and Elbaum's test prioritization detects failing tests faster by a median of 3.9 minutes. Elbaum's test selection detects failing tests faster by a median of 62 minutes in comparison to the baseline with 179 machines while missing 31.25% of failures. The Batching algorithm detects failing tests faster by a median of 59.4 minutes against the baseline without missing any failures.

4.6.4 RQ4: How effective are different test optimization techniques in saving test execution time when executed in parallel with varying numbers of machines?

Another important aspect of test optimization is the amount of time different algorithms can save in test executions (EXECUTIONREDUCTION). The more time they can save, the more resources and feedback time we can conserve.

To calculate the amount of time saved in test executions by different algorithms with varying numbers of machines, we utilize the EXECUTIONREDUCTION formula. For each algorithm with m machines, we divide the execution time of that algorithm by the execution time of the *TestAll* algorithm and subtract the result from 1.

For *TestAll*, *KimPorter*, and *ElbaumPrioritization* algorithms, there is no EXECUTIONREDUCTION as they run all tests, although they may reorder them. The results for the *ElbaumSelection* and *BatchAll* algorithms are shown in Figure 4.7. The figure demonstrates that *ElbaumSelection* always saves 66.31% of the time spent on test executions regardless of the number of machines used. On the other hand, the amount of time saved in test executions by *BatchAll* varies significantly depending on the number of machines used. With a single machine, *BatchAll* can save up to 98.69% in test execution time, whereas with 400 machines, it saves only 8.29%.

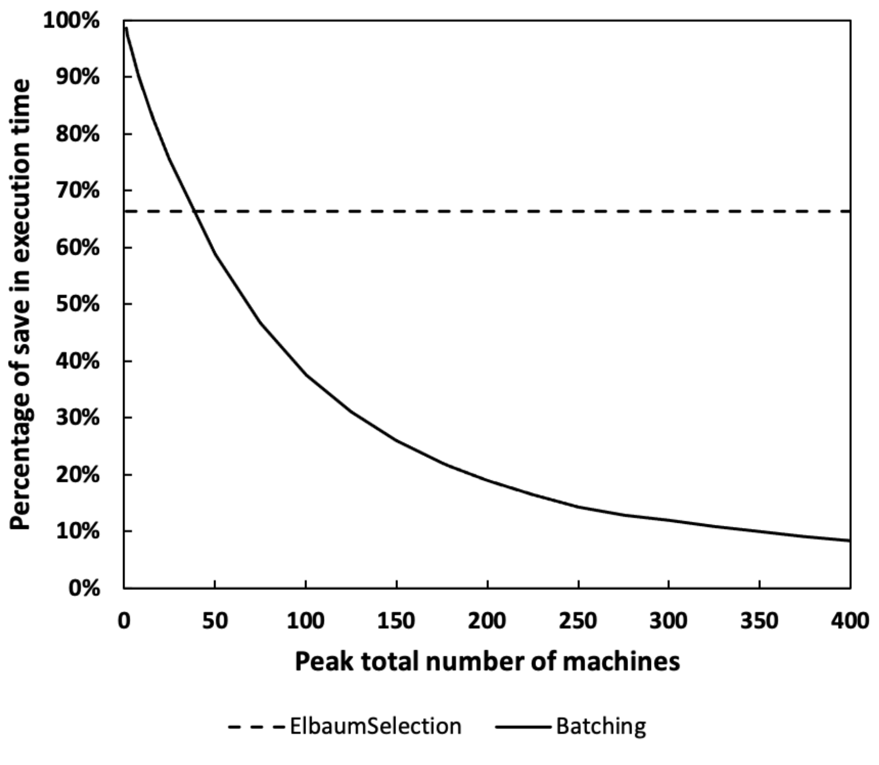


Figure 4.7: The percentage of time saved in test executions by *ElbaumSelection* and *BatchAll* algorithms, relative to the total execution time for all test cases, was calculated for different peak numbers of machines.

To assess the potential time savings achievable by different test optimization algorithms using

Table 4.8: The save in test execution time achieved by different algorithms using the baseline number of 179 machines.

Algorithm	**KimPorter**	**ElbaumPrioritization**	**ElbaumSelection**	**Batching**
Save in execution time	0%	0%	66.31%	21.45%

the baseline number of machines, which is more representative of the actual number of machines used in practice, we present the results of EXECUTIONREDUCTION for 179 machines in Table 4.8.

As expected, the table confirms that *KimPorter* and *ElbaumPrioritization* algorithms do not result in any time savings for tests execution. However, it shows that *ElbaumSelection* can achieve a considerable 66.31% reduction in test execution time. On the other hand, the *BatchAll* algorithm can save up to 21.45% in test execution time, although this figure depends on the specific number of machines used. Notably, the *ElbaumSelection* approach misses 31.25% of failing tests in this particular case.

RQ4: Elbaum's test selection saves 66.31% in test execution time, but the trade-off for this algorithm is losing failing tests, 31% in the given case. Kim and Porter and Elbaum's test prioritization approaches do not produce any savings in test executions. The saving in test executions for Batching algorithm varies between 2% and 98.5% depending on the number of machines in use.

4.7 Threats to Validity

This section discusses the threats to the validity of this research.

4.7.1 External Validity

In this study, our focus is solely on the test results obtained from Google Chrome. While this dataset provides valuable insights into large-scale testing infrastructures, it may not fully represent all other testing systems. However, it is worth noting that this dataset is the only publicly available dataset at such a scale, comprising 276 million tests and up to 47 overlapping builds. Other available datasets are not comparable in terms of scale, and they do not differentiate between flaky failures and true failures. Even the dataset provided by Elbaum *et al.* [1] on Google consists of 3.5 million test suites collected over a one-month period, without distinguishing flaky failures from true failures. Since the problem of test optimization is primarily relevant to large-scale projects, we believe that using the Chrome dataset allows us to better understand and address the challenges faced by practitioners in the field.

This study adopts history-based test selection and prioritization techniques, which may not represent every test selection and prioritization technique. However, when dealing with large-scale datasets like Chrome, the approaches that can be used for test optimization are limited. Due to high code churn in these projects, coverage-based techniques are not effective [2]. Additionally, previous studies have shown that these black box techniques are as effective as other complex and expensive test optimization techniques [21].

4.7.2 Construct Validity

To calculate the feedback time of a build, only the test execution time is considered, and other processing times such as compilation time are not taken into account. This data is not included in the dataset, and even if we had access to it, we cannot assume the compilation time of a batch. This could be particularly important for the batching algorithm. When a batch succeeds, it saves on compilation time, and when it fails, it has to compile included changes to find the culprit change. Given the 8.5% build failure rate in Chrome, batching is likely to be advantageous even when compile time is factored in.

4.7.3 Internal Validity

To carry out the test optimization techniques in this study, we had to change the order of tests, ignore some tests, or batch tests belonging to multiple changes. All of these assumptions require tests to be independent of each other. Otherwise, if there are any dependencies among tests, these manipulations could cause dependency flaky failures [45]. In this study, we only simulated test executions concurrently and autonomously if the Chrome developers had already been performing the same tests in parallel on multiple shards.

We also made a set of assumptions for the batching results. For instance, if a test shows an unexpected result in one of the builds included in the batch, it is assumed to be an unexpected result when combined with other builds in a batch as well. While this might be true in most cases, there could be instances where the unexpected outcome is resolved after applying the following changes in the batch, and the test subsequently gives an expected result. Similarly, we assume that if a test flakes in one of the builds included in the batch, it also flakes in the batch. While this scenario is highly probable given the repeated behavior of flaky tests [6], there could be instances where the test shows an unexpected or expected outcome in the batch. In the former case, an extra culprit-finding process would be necessary, while in the latter case, there would be no need for re-runs.

4.8 Discussion

This section discusses the implications of our findings for practitioners and key parameters that impact the study.

Implications of feedback time comparison in software testing. Based on the results of our study in RQ1, we recommend using test batching or test selection approaches to significantly improve feedback time for build results. The batching algorithm offers the best feedback time when resources are limited and practitioners want an efficient test optimization approach to reduce feedback time. On the other hand, if more resources are available and developers can overlook some missed failures, then test selection might be the best option for highly efficient test optimization. Interestingly, feedback time results for the test selection approach plateau faster than the others due to fewer tests to execute. If practitioners prioritize both accuracy and efficiency and cannot overlook losing failures, then the batching algorithm is the best option.

Implications of resource usage comparison in software testing. Based on the results of RQ2, we recommend the following suggestions to reduce resource usage in testing. Test selection and batching approaches are both effective in significantly reducing the number of machines and resources required for testing. While test selection may miss some failures, batching reduces resource consumption without losing any failures and saves more than the test selection algorithm. Additionally, the results show that the advantage of using the batching algorithm in terms of feedback time and resource reduction compared to other test optimization algorithms is more significant when resources are limited than in a resource-rich environment.

Implications of test optimization for failure detection speed. Our study in RQ3 highlights important recommendations for practitioners. Firstly, all of the test optimization algorithms improve the speed of failure detection compared to not using any test optimization techniques. Both test selection and batching approaches significantly boost the speed of failure detection. In highly resource-constrained environments, we recommend using the batching approach to detect failures faster. In highly resource-available environments, test selection may detect failures slightly faster, but at the cost of potentially missing some failures. Practitioners can choose between test selection and batching based on their tolerance for failure loss and required failure detection efficiency. Nevertheless, we recommend using the batching approach overall, even in highly resource-available environments.

Implications of the test execution time comparison Based on the results in RQ4, we offer the following recommendations for practitioners. Both test selection and batching can be highly effective in saving test execution time. However, the interesting finding from this research question is that the savings in test execution is always static for the test selection algorithm, while it varies significantly for the batching approach. This is because as the number of resources increases, the queue size for batching decreases, which forces batching to run more test executions, but in test selection, the test executions do not change by changing the number of machines. In a highly resource-constrained environment, we recommend using batching to reduce test execution time. In a highly resource-available environment, we recommend using test selection to reduce test execution time if missing some failures is acceptable. Otherwise, practitioners can use the batching algorithm, which saves test execution time depending on the number of machines in use and does not miss any failures.

Impact of window size on ElbaumSelection algorithm.

To determine the appropriate window sizes for the *ElbaumSelection* and *ElbaumPrioritization* algorithms in this study, we conduct an experiment similar to Elbaum *et al.*'s work [52]. In this experiment, we use the *ElbaumSelection* algorithm with 179 machines, which is the same number of machines used in our baseline.

For the purpose of this experiment, we use the percentages of selected tests, execution time, and detected failures in comparison to the baseline, as described in Elbaum *et al.*'s paper, to measure the performance of the algorithm.

In the first experiment, we choose a relatively small execution window size of $w_e = 4$ and vary w_f, displaying the results in Figure 4.8. As shown in Figure 4.8, increasing w_f improves the detection of failing tests. However, contrary to Elbaum *et al.*'s results [52], changing w_f does not significantly affect the percentage of executed tests or the percentage of test execution time.

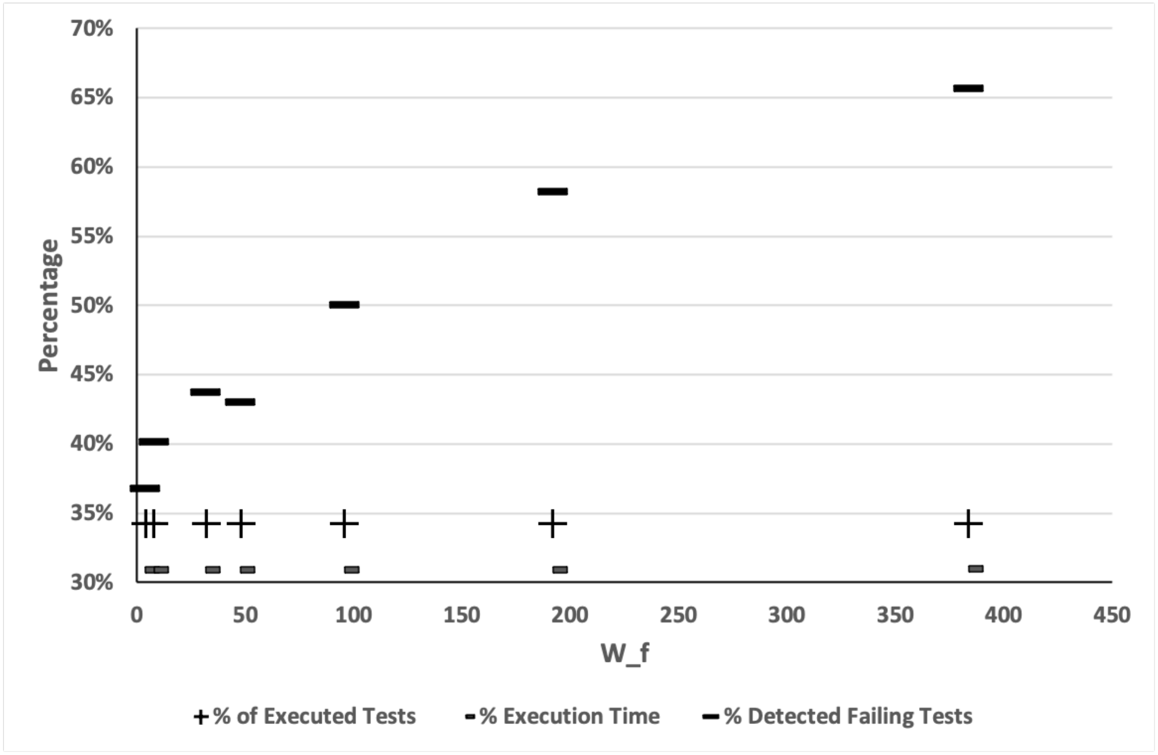


Figure 4.8: The Percentages of Executed Tests, Executed Time, and Detected Failing Tests for *ElbaumSelection* with $w_e = 4$ and varying w_f.

To validate this observation with a larger execution window, we increased w_e to 96 and varied w_f. The results, as shown in Figure 4.9, further confirm that changing w_f does not have a significant impact on the percentage of executed tests or the percentage of test execution time. This discrepancy can be attributed to the prevalence of flaky failures in Chrome, as highlighted by Fallahzadeh and Rigby (2022) [6], which is specifically addressed in this study.

However, both experiments show that increasing the size of w_f significantly improves the detection of failing tests without significantly impacting the time taken for test execution. Therefore, in this study, we adopt an unlimited size for w_f.

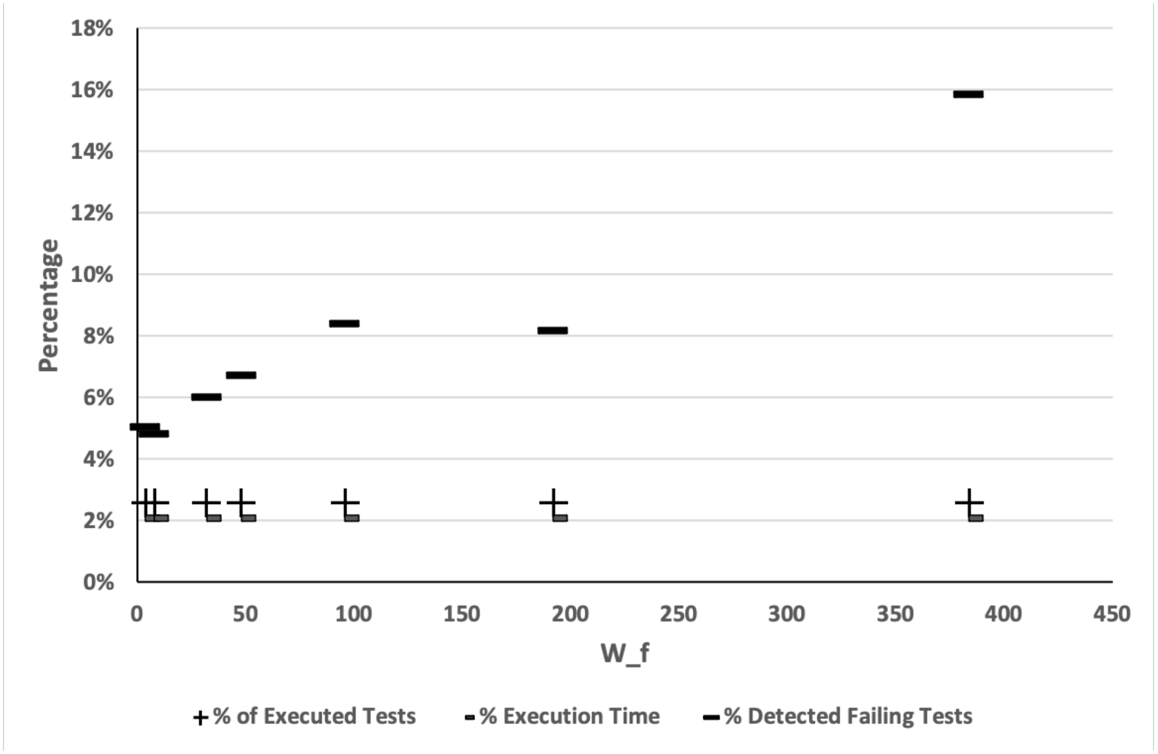


Figure 4.9: The Percentages of Executed Tests, Executed Time, and Detected Failing Tests for *ElbaumSelection* algorithm with $w_e = 96$ and varying w_f.

On the other hand, changing w_e significantly impacts the percentage of executed tests, the percentage of test execution time, and the number of detected failing tests, as shown in Figure 4.8, Figure 4.9, and Figure 4.10. To further investigate the effect of changing w_e, we conduct another experiment with an unlimited w_f size and vary w_e. The results are shown in Figure 4.10.

Figure 4.10 reveals that increasing w_e decreases the number and time for executing tests, but also decreases the number of detected failing tests. We choose the w_e size of 2 in our experiment because it decreases the number and time for test execution by 64.49% and 66.31%, respectively, while still maintaining a high failure detection rate of 68.75%. This high failure detection rate is necessary as we compare this algorithm with other test optimization algorithms that do not miss failures.

Impacts of failure rate on batching algorithm. The performance of the batching algorithm is significantly affected by the failure rate and the presence of flaky failures. A higher number of failures in test results can lead to an increase in the number of penalties for culprit finding in

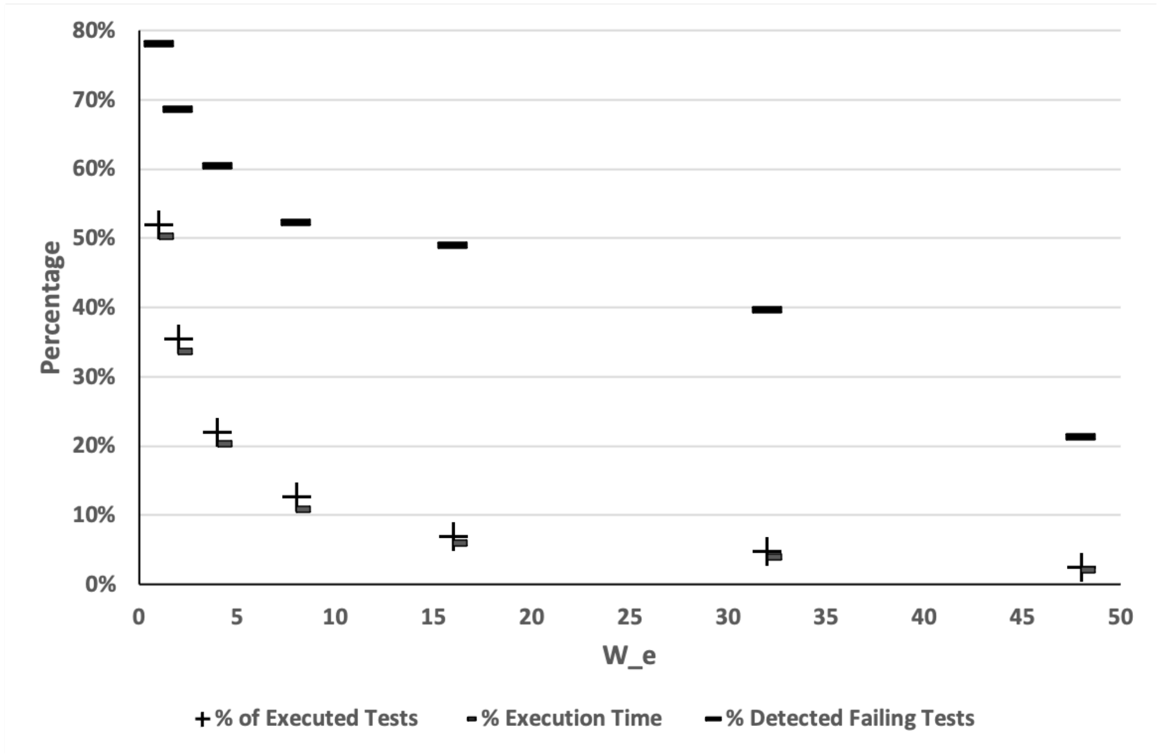


Figure 4.10: The Percentages of Executed Tests, Executed Time, and Detected Failing Tests for *ElbaumSelection* algorithm with the unlimited w_f and varying w_e.

batching. However, research by Beheshtian *et al.* [41] suggests that 85.5% of the Travis CI projects they studied could benefit from batching, as long as their failure ratio is below 40%. In this study, the build failure rate in Chrome is 8.5%, which is well below this threshold, making batching an effective approach for improving test execution time. Additionally, applying culprit finding at the test level can significantly reduce the cost of batching in this study.

Measuring failure detection. In this study, we aim to measure how quickly each algorithm detects failures on the Chrome dataset. To achieve this, we compare the time differences between the actual test runs in the *TestAll* order and the order recommended by other algorithms. While the APFD metric [20] is a popular measure in test prioritization, we found that it is not effective in a continuous integration environment where the focus is on detecting failures across builds. This observation is consistent with previous studies by Elbaum *et al.* [52] and Fallahzadeh and Rigby [6]. Therefore, we use a custom measure that better reflects the requirements of detecting failures in a continuous integration environment.

4.9 Conclusion and Future Works

This study aimed to compare and evaluate different test optimization algorithms for reducing feedback time, improving failure detection, and saving test execution time using varying numbers of execution resources. Our results show that the batching algorithm performs best in a highly resource-constrained environment, achieving significant reductions in feedback time, resource usage, and test execution time while also improving failure detection without missing any failures. Test selection performs slightly better in highly resource-available environments but at the cost of missing some failures. Test prioritization algorithms underperformed both batching and test selection. Practitioners can choose the appropriate test optimization algorithm based on the computational resource availability and the tolerance for missing failures. In resource-constrained environments, the batching algorithm would be the optimal choice, whereas in resource-rich environments, test selection could achieve slightly better results. The underperformance of test prioritization algorithms suggests that they should not be the first choice for test optimization. This study has limitations in terms of the type of software used (Chrome), the test optimization algorithms studied, and the specific metrics evaluated. Future research could investigate other software types, additional test optimization algorithms, and different evaluation metrics. Further research could focus on exploring the impact of combining different test optimization algorithms, investigating the effectiveness of other test optimization algorithms, and evaluating the impact of test optimization on software quality.

Chapter 5

Discussion and Conclusion

This thesis presents a comprehensive analysis of the efficiency of different test optimization algorithms in large continuous integration environments, taking into account the influence of flaky tests and parallel test execution.

In this study, we conducted an analysis of over 276 million test cases from the Google Chrome project and simulated various test optimization algorithms. The key findings and implications from the main three chapters of this thesis are outlined below.

Chapter 2: The impact of flaky tests on test prioritization at scale. Flaky failures constitute the majority of failures within the Chrome dataset, and they demonstrate a notably higher degree of recurrence when compared to non-flaky failures. Although the Chrome testing infrastructure employs analysis tools like Findit [114] to proactively identify and address the sources of flakiness in the long term, they permit changes containing flaky failures to be integrated into the main repository. Consequently, the presence of these flaky failures significantly influences the effectiveness of test prioritization algorithms, as they are accorded higher priority over non-flaky failures. These findings underscore the critical importance of factoring in the impact of flaky tests when devising test prioritization strategies and warrant a reevaluation of prior studies that may have overlooked this significant factor.

Chapter 3: The impact of parallel test execution on batch testing at scale. Our observations reveal that the impact of parallelism on test optimization algorithms is non-linear, leading to compounded delays in subsequent builds. Constant batching algorithms exhibit superior feedback time performance in resource-constrained environments but eventually reach a plateau in feedback time reduction as the number of machines increases. In contrast, batching algorithms with adaptive batch sizes consistently outperform other algorithms in terms of feedback time. These adaptive approaches offer the advantage of providing variable test execution reduction depending on available resources, making them suitable for both high and low resource-constrained environments. In particular, the *BatchAll* algorithm showcases promising performance across all resource availability environments, despite its simple and cost-effective approach. Therefore, we recommend practitioners to consider adopting adaptive batching techniques, as they have the potential to significantly enhance feedback time and test execution reduction while adapting to varying resource availability and test execution queues.

Chapter 4: Comparative assessment on test selection, prioritization, and batching algorithms at scale. Our findings indicate that the batching algorithm with adaptive batch size outperforms other test optimization algorithms in feedback time, particularly in highly resource-constrained environments, while maintaining a comparable performance to test selection algorithms in other scenarios without missing any failures. It also outperforms other algorithms in reducing the number of machines required to achieve the median feedback time and detecting failing tests faster. The performance of test batching algorithms varies based on the number of machines used, while test selection algorithms provide a consistent test execution reduction. Practitioners are advised to choose between test batching and test selection based on the number of available machines, tolerance for missing failures, and performance expectations. However, test batching algorithms generally outperform other algorithms or demonstrate comparable performance, with the added advantage of not missing any failures. Table 5.1 displays a summary of the results from the main chapters of this study.

In Table 5.2, we present a summary table detailing the data parameters from testing environments and their impact on the performance results of various test optimization algorithms. While most of these parameters remained unchanged due to the constraints of our Chrome testing environment, we offer our estimations based on a comprehensive survey of test optimization studies and the outcomes of our simulations.

In Table 5.3, we present a summary of the findings obtained for each outcome measure studied

Table 5.1: A summary of the results from main chapters.

Chapter	**Results**
Chapter 2: The impact of flaky tests on test prioritization at scale	Previous studies have been affected by a significant problem: they have treated flaky failures as genuine failures when evaluating history-based test prioritization algorithms. However, when we follow Chrome's approach and classify flaky failures as non-blocking, we observe a noticeable and detrimental drop in the performance of these algorithms. An in-depth examination of the dataset statistics further underscores this issue. It reveals that a substantial majority of failures in Chrome fall into the flaky category. Interestingly, flaky failures occur more frequently than their non-flaky counterparts. This fact clarifies why history-based test prioritization algorithms underperform when operating under the assumption that flaky failures are non-blocking.
Chapter 3: The impact of parallel test execution on batch testing at scale	Parallel test execution has a profound influence on batch testing at scale. Notably, the relationship between the increase in the number of machines and the feedback time performance of testing algorithms is non-linear. In resource-constrained environments where only a limited number of machines are available, employing larger batch sizes within constant batching algorithms yields improvements in feedback time, resource usage reduction. Among these algorithms, *BatchAll* stands out as the top performer, delivering the best feedback time and resource usage outcomes. Conversely, in highly resource-abundant environments, the feedback time performance of constant batching algorithms reaches a plateau. Here, the *TestCaseBatching* algorithm outperforms others. Remarkably, the *BatchAll* algorithm consistently delivers stable results across all environments, despite its simplicity.
Chapter 4: Comparative assessment on test selection, prioritization, and batching algorithms at scale	In a highly resource-constrained environment, both the *BatchAll* and *ElbaumSelection* algorithms exhibit superior performance in terms of feedback time, resource reduction, failure detection GainedTime, and execution reduction. However, it's important to note that the *ElbaumSelection* approaches miss approximately 30% of the failures. As we transition to a highly resource-available environment with an increased number of machines, the *ElbaumSelection* algorithm demonstrates exceptional feedback time and execution reduction performance, surpassing the *BatchAll* algorithm. Nevertheless, this enhanced performance comes at the expense of missing some failures. Overall, *BatchAll* delivers promising results across various metrics, including feedback time, resource usage, failure detection GainedTime, and execution reduction, all while ensuring that no failures are overlooked. In stark contrast, test prioritization algorithms fail to deliver significant improvements across these measures.

Table 5.2: A summary of the impact of each data parameter on the performance of different test optimization algorithms.

Parameter	**Impact**
Number of machines	In a resource-constrained environment with a limited number of available machines, batching algorithms, especially *BatchAll*, and test selection algorithms can considerably reduce feedback time, resource usage, time required for detecting failures, and test execution duration. Even in a highly resource-rich environment, these algorithms continue to provide performance enhancements, although the extent of improvement may be slightly diminished. Test selection achieves these improvements by potentially omitting some failures, whereas batching algorithms ensure that no failures are missed.
Failure rate	In an environment characterized by a low failure rate, batching algorithms can be highly effective, delivering improvements even when the failure rate reaches up to 40%. However, in a high-failure-rate environment, the performance of batching algorithms tends to diminish, and test selection algorithms tend to outperform them.
Flaky rate	In a low flake rate environment, the occurrence of test reruns due to false failures decreases, resulting in a reduction in overall test execution time. In a high flake rate environment, both batching and test selection algorithms excel in terms of test execution, as they effectively reduce the frequency of reruns needed for flaky tests. These observations hold true when flaky tests can be reliably distinguished from genuine failures.
Queue size	In scenarios where an extensive number of changes are queued up for processing, batching algorithms excel over other test optimization methods by significantly enhancing feedback time, resource utilization, expediting failure detection, and reducing test execution times. However, in cases where the queue contains only a limited number of changes, batching algorithms with adaptive batch sizes may adapt, but their advantages become less evident or non-existent.
Build time	In instances where longer build times are the norm, batching algorithms may outperform their counterparts, particularly in scenarios with a low failure rate. However, it's worth noting that in a high failure rate environment, this advantage might be offset by an increase in the time needed for the culprit-finding procedure. Conversely, when build times are shorter, the benefits provided by batching algorithms, stemming from the reduction in the number of builds, also diminish. Simultaneously, the time required for the culprit-finding process decreases.
Execution time	When the test execution time is extended, and multiple changes are simultaneously queued for processing, the benefits derived from the batching algorithm become notably more pronounced. However, when dealing with tests that have shorter execution times, the advantages stemming from reduced test execution time diminish to a lesser degree.

Table 5.3: A summary of the findings from different outcome measures.

Outcome Measure	**Findings**
Feedback Time	Batching algorithms can substantially enhance build feedback time when machine resources are scarce. However, as the number of available machines increases, test selection algorithms can surpass batching in performance, albeit with the trade-off of potentially missing some failures. In contrast, test prioritization algorithms typically do not yield significant improvements in build feedback time.
Resource Reduction	Resource reduction achieved by batching algorithms to attain the actual feedback time can be substantial, reaching approximately 88% and 58% for the test selection algorithm. Conversely, test prioritization algorithms do not result in any notable resource reductions.
GainedTime	In resource-constrained scenarios, the performance of both batching and test selection algorithms, as measured by failure detection GainedTime, is remarkably close and impressive, albeit with the trade-off of potentially missing some failures in test selection. Test prioritization algorithms also offer some improvements in such environments. However, in highly resource-abundant settings, the performance of all test optimization algorithms experiences a significant decline, approaching levels similar to the baseline.
Execution Reduction	The test selection algorithm consistently achieves a fixed test execution reduction of approximately 66%. In the case of test batching algorithms, the execution reduction is contingent on the batch size. Constant batching algorithms deliver a consistent execution reduction, whereas adaptive batching algorithms like *BatchAll* yield variable execution reductions. This variability can range from nearly 100% when machine resources are scarce to less than 10% in a highly resource-abundant environment.

in this thesis.

It is important to acknowledge the limitations of this study, which include its focus on the Chrome software, specific test optimization algorithms, and particular evaluation metrics. To further advance the field, future research should explore other software types, additional test optimization algorithms, and different evaluation metrics.

Future research endeavors can involve investigating the combination of different test optimization algorithms, exploring alternative algorithms, and evaluating the impact of test optimization on software quality. These avenues of research hold the potential to contribute to further advancements in test optimization strategies.

Appendix A

Practitioners Guideline

In this appendix, we provide a guideline for practitioners on how to use the results of this study.

A.1 Flaky Tests

Identifying and addressing flaky tests holds paramount importance in various testing environments. This significance is underscored by our findings in Chrome, where flaky tests comprise the majority of failures, and as demonstrated in Chapter 2, the treatment of flaky tests can substantially impact the results of test optimization techniques.

Following the approach adopted in Chrome, we offer recommendations on how to identify and manage flaky tests effectively. Firstly, unlike many studies that rerun tests thousands of times, flaky tests should be identified through only a few reruns following a failure. If these reruns consistently produce flaky results, the test can be classified as flaky.

Once flaky tests are identified, it is essential to handle them appropriately. Flaky failures are not indicative of genuine defects and should thus be considered as non-failures. This does not imply that practitioners should disregard these tests entirely. Instead, builds containing such tests should be allowed to integrate. In the long term, efforts should be directed toward resolving the underlying causes of this flakiness, a process akin to Chrome's FindIt infrastructure [114].

A.2 Test Prioritization

In our comprehensive evaluations of history-based test prioritization algorithms, specifically Kim and Porter's and Elbaum et al.'s algorithms, conducted from various perspectives and utilizing different evaluation metrics, we offer the following recommendations to practitioners.

Regrettably, these algorithms do not yield promising results, especially when flaky failures are appropriately treated as non-blocking. Evaluations encompassing feedback time, resource reduction, GainedTime failure detection, and execution reduction do not yield practical outcomes. Even concerning their primary objective, GainedTime failure detection, these algorithms do not perform admirably. In light of these findings, we strongly advise practitioners to consider the use of test selection and batching algorithms instead.

A.3 Test Selection

Within our study, we conducted an evaluation of Elbaum et al.'s test selection algorithm, which is regarded as one of the effective history-based test selection algorithms. Our assessments have revealed that this test selection algorithm exhibits exceptional efficacy, particularly in high resource-constrained environments. Based on these findings, we recommend the utilization of this test selection algorithm by practitioners, particularly in cases where the omission of bugs and failing tests does not pose critical consequences for the system.

In such scenarios, the adoption of the test selection algorithm can yield promising outcomes across various evaluation metrics, including feedback time, resource reduction, failure detection GainedTime, as well as delivering consistent and notable reductions in test execution time. However, if the system's integrity relies heavily on the detection of failing tests and bugs, we suggest prioritizing the use of batching algorithms instead.

A.4 Batching

This study conducted evaluations of several constant batching algorithms that had been previously examined in research. Additionally, we introduced two adaptive batching techniques. Drawing from our comprehensive assessments, we offer the following recommendations for practitioners.

In resource-constrained environments, batching algorithms demonstrate optimal performance. Specifically, utilizing larger batch sizes in these settings yields superior results, with the adaptive *BatchAll* techniques emerging as top performers across various metrics, including feedback time, GainedTime failure detection, execution reduction, and resource reduction. Notably, the *BatchAll* algorithm excels in achieving the most substantial resource reduction when compared to other algorithms.

Conversely, in resource-abundant environments, constant batching algorithms tend to reach a plateau, yielding less promising results. However, the *BatchAll* algorithm continues to exhibit promise, albeit with potentially diminished execution reduction when smaller batch sizes are employed. Nevertheless, its performance remains competitive with the test selection algorithm from various perspectives, offering the significant advantage of not missing any failures.

An important consideration in the utilization of batching techniques is the failure ratio. Higher failure rates can negatively impact the performance of these algorithms, as they may require extensive culprit finding efforts, resulting in a penalty. However, it's worth noting that the failure ratio for most systems typically falls below the 40% threshold recommended by Beheshtian et al. [100]. By employing effective culprit finding techniques, we believe that the culprit finding penalty can be significantly improved.

Furthermore, as the number of overlapping changes and builds increases, we strongly recommend the adoption of batching algorithms, especially the adaptive batching techniques. These techniques represent a promising yet often overlooked approach to optimize test execution and resource utilization.

A.5 Cost Savings

The cost savings, based on the results we achieved in Chapter 4 and information from the Google Cloud Pricing Calculator [115] updated as of September 6th, 2023, at the time of writing this report, are as follows:

In terms of resource reduction, we can achieve the same feedback time as the actual median feedback time using *BatchAll* with 21 machines, which costs USD 1,027.29 per month per particular builder. This is in contrast to using 179 baseline machines, which cost USD 8,756.38 per month per particular builder, resulting in a savings of USD 7,729.09 (88.27%) per month. For the *ElbaumSelection* algorithm, we will need 75 machines, which costs USD 3,668.88 per month per particular builder, translating to USD 5,087.50 (58.10%) savings per month.

Based on the Google Cloud Pricing Calculator [115], the cost of each machine per hour is USD 0.067. Regarding execution reduction, the total test execution time for the examined Chrome dataset is about 57,570.13 hours. By using the *BatchAll* algorithm, we can save up to 98.69% in test execution, which amounts to USD 3,807.26 instead of USD 3,857.80 for the entire executions without test optimizations, per one month of test executions and per a particular builder. For the *ElbaumSelection* algorithm, this saving is 66.31%, which is USD 2,558.11, per one month of test executions and per a particular builder.

Finally, it’s crucial to emphasize that these test optimization techniques may not be beneficial in scenarios where there are no overlapping changes, a small number of tests, or an environment with no resource constraints. This holds especially true when resources are abundant and inexpensive, although such a situation remains a distant reality.

Appendix B

Chrome Trybot Scraper Framework

In this appendix, we provide an overview of the framework developed for capturing data from Chrome. The implementation of this framework can be accessed from `https://github.com/CESEL/ChromiumTrybotScraper`. To develop this framework, we utilized scraping techniques and API calls.

Initially, we captured the list of changes from the Chromium Gerrit code review website. Figure B.1 illustrates the website used to retrieve the change lists. By applying filters such as ”after:2023-01-01,” we specified the range of change lists to be displayed.

The code snippet provided in Listing B.1 demonstrates the implementation of the API call to retrieve the change lists.

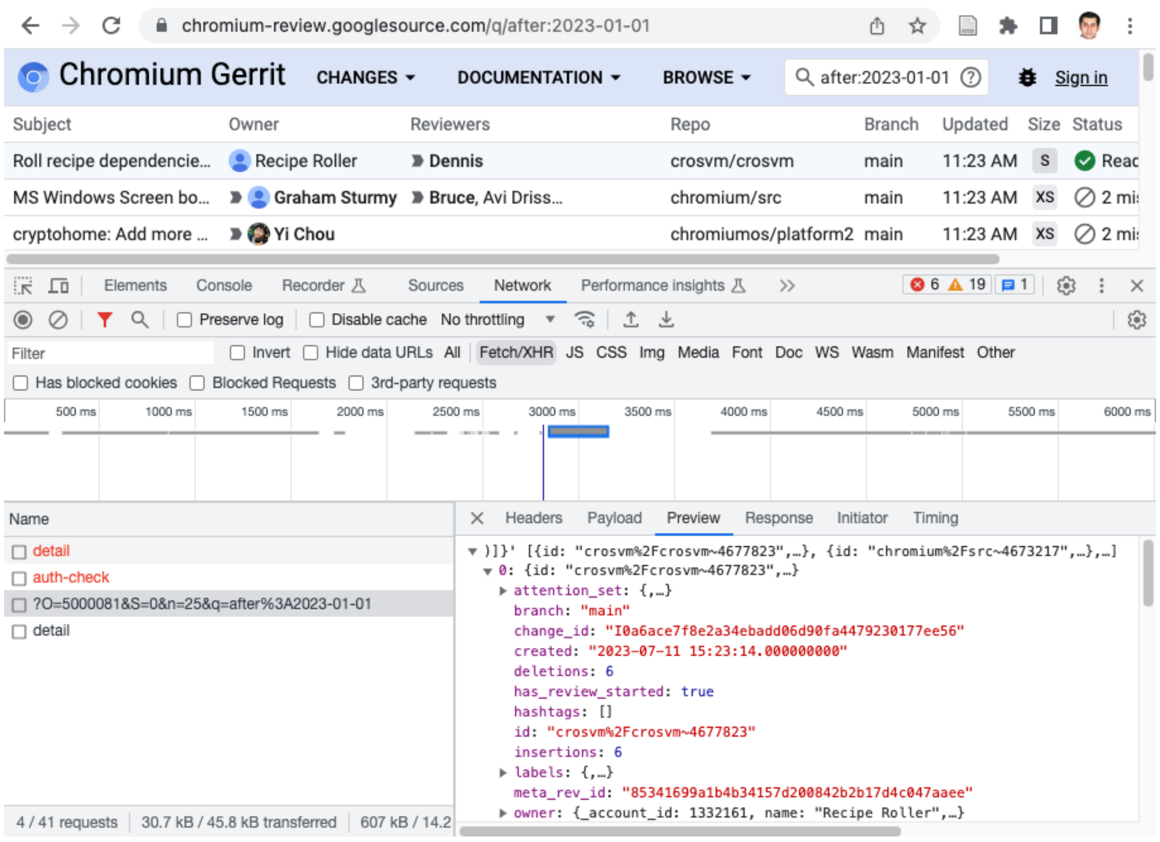
```python
def get_changelists_api_result(
    page, page_size, start_date='', end_date=''):
    params = (
        ('O', '81'),
        ('S', str(page)),
        ('n', str(page_size))
    )

    q = ('after:' +  start_date if start_date else '') + \
        (' before:' + end_date if end_date else'')
    if q:
        params += (('q', q),)

    response = requests.get(
        'https://chromium-review.googlesource.com/changes/', params=params)
    if response.status_code == 500:
        return Status.internal_server_error
    return response.content
```

Listing B.1: Get change list method

Figure B.2 illustrates a sample of change details extracted from the Chrome website.

The code in Listing B.2 demonstrates the method used to capture a change detail using the Chrome API.

```python
def get_change_details_api_result(change_num):
    params = (
        ('O', '916314'),
```



Subject	Owner	Reviewers	Repo	Branch	Updated	Size	Status
Roll recipe dependencie...	Recipe Roller	Dennis	crosvm/crosvm	main	11:23 AM	S	Reac
MS Windows Screen bo...	Graham Sturmy	Bruce, Avi Driss...	chromium/src	main	11:23 AM	XS	2 mi
cryptohome: Add more ...	Yi Chou		chromiumos/platform2	main	11:23 AM	XS	2 mi

```
Name
detail
auth-check
?O=5000081&S=0&n=25&q=after%3A2023-01-01
detail

4 / 41 requests | 30.7 kB / 45.8 kB transferred | 607 kB / 14.2
```

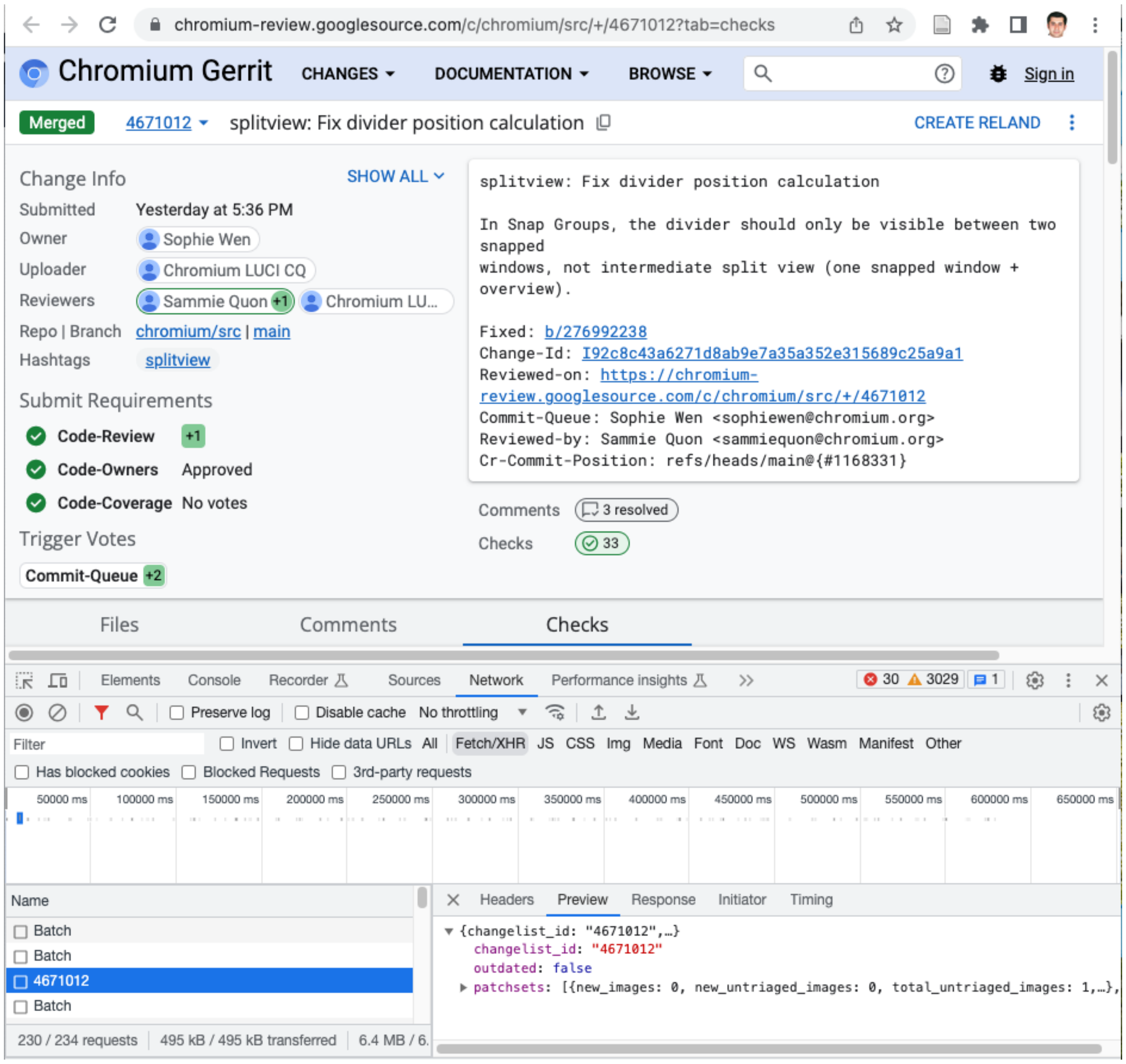
```
▼ )]}' [{id: "crosvm%2Fcrosvm~4677823",…}, {id: "chromium%2Fsrc~4673217",…},…]
  ▼ 0: {id: "crosvm%2Fcrosvm~4677823",…}
    ▶ attention_set: {,…}
      branch: "main"
      change_id: "I0a6ace7f8e2a34ebadd06d90fa4479230177ee56"
      created: "2023-07-11 15:23:14.000000000"
      deletions: 6
      has_review_started: true
      hashtags: []
      id: "crosvm%2Fcrosvm~4677823"
      insertions: 6
    ▶ labels: {,…}
      meta_rev_id: "85341699a1b4b34157d200842b2b17d4c047aaee"
    ▶ owner: {_account_id: 1332161, name: "Recipe Roller",…}
```

Figure B.1: A sample of change lists from Chrome Gerrit code review.

```
    )
    response = requests.get('https://chromium-review.googlesource.com/changes/'
   + str(change_num) + '/detail',
                                  params=params)
    return response.content
```

Listing B.2: Get change details method

Then, for each individual change list, we retrieve the corresponding builds. Figure B.4 showcases the builds associated with a change list on the Chrome website.



splitview: Fix divider position calculation

Change Info
Submitted Yesterday at 5:36 PM
Owner Sophie Wen
Uploader Chromium LUCI CQ
Reviewers Sammie Quon +1 Chromium LU...
Repo | Branch chromium/src | main
Hashtags splitview

Submit Requirements
✓ Code-Review +1
✓ Code-Owners Approved
✓ Code-Coverage No votes

Trigger Votes
Commit-Queue +2

```
splitview: Fix divider position calculation

In Snap Groups, the divider should only be visible between two
snapped
windows, not intermediate split view (one snapped window +
overview).

Fixed: b/276992238
Change-Id: I92c8c43a6271d8ab9e7a35a352e315689c25a9a1
Reviewed-on: https://chromium-
review.googlesource.com/c/chromium/src/+/4671012
Commit-Queue: Sophie Wen <sophiewen@chromium.org>
Reviewed-by: Sammie Quon <sammiequon@chromium.org>
Cr-Commit-Position: refs/heads/main@{#1168331}
```

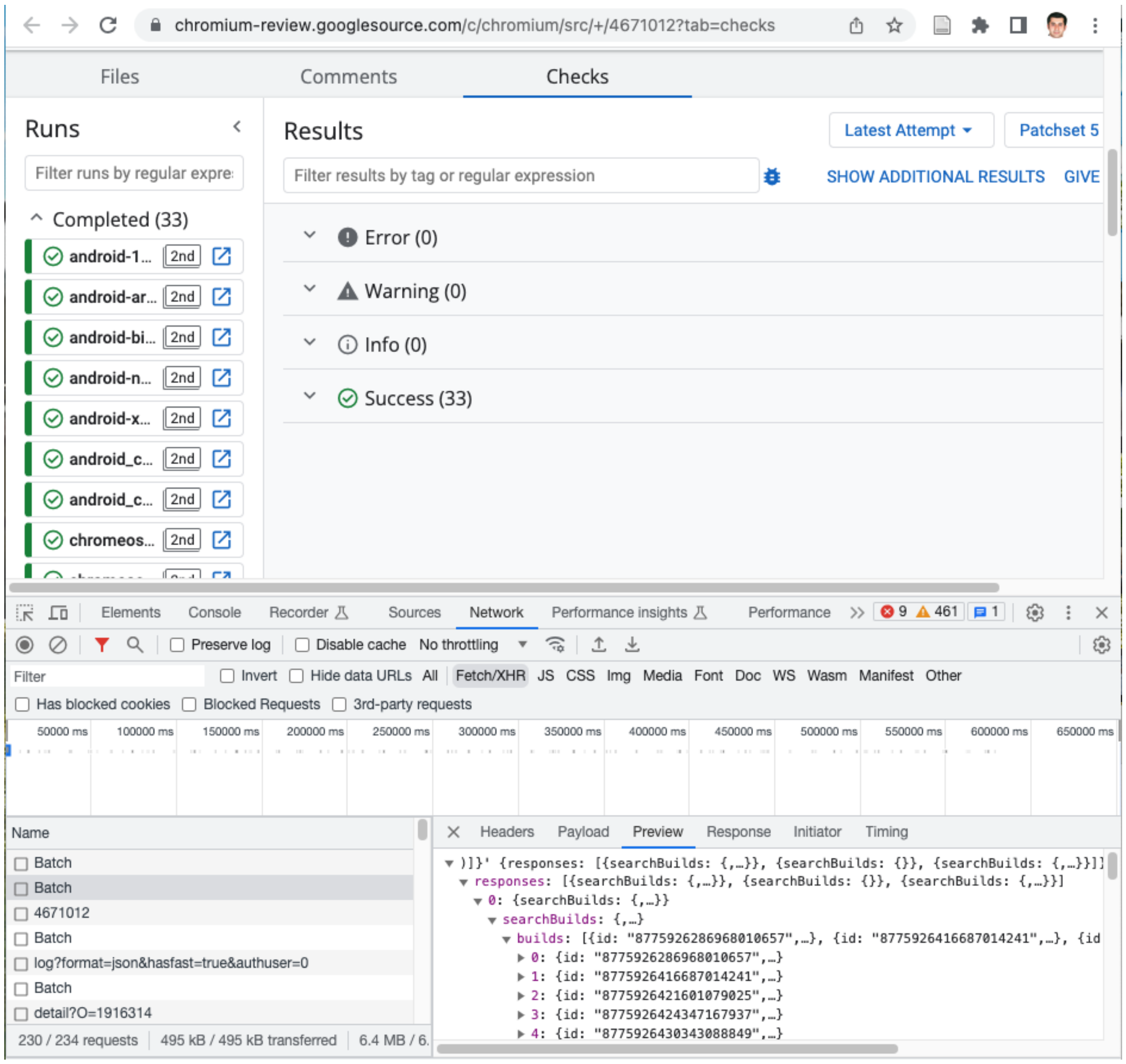
Comments 3 resolved
Checks ✓ 33

Name
Batch
Batch
4671012
Batch

230 / 234 requests | 495 kB / 495 kB transferred | 6.4 MB / 6.

```
▼ {changelist_id: "4671012",…}
    changelist_id: "4671012"
    outdated: false
  ▶ patchsets: [{new_images: 0, new_untriaged_images: 0, total_untriaged_images: 1,…},
```

Figure B.2: A sample of change details from Chrome Gerrit code review.



```
Batch
Batch
4671012
Batch
log?format=json&hasfast=true&authuser=0
Batch
detail?O=1916314
230 / 234 requests | 495 kB / 495 kB transferred | 6.4 MB / 6.
```

```
▼ )]}' {responses: [{searchBuilds: {,…}}, {searchBuilds: {}}, {searchBuilds: {,…}}]}
  ▼ responses: [{searchBuilds: {,…}}, {searchBuilds: {}}, {searchBuilds: {,…}}]
    ▼ 0: {searchBuilds: {,…}}
      ▼ searchBuilds: {,…}
        ▼ builds: [{id: "8775926286968010657",…}, {id: "8775926416687014241",…}, {id
          ▶ 0: {id: "8775926286968010657",…}
          ▶ 1: {id: "8775926416687014241",…}
          ▶ 2: {id: "8775926421601079025",…}
          ▶ 3: {id: "8775926424347167937",…}
          ▶ 4: {id: "8775926430343088849",…}
```

Figure B.3: A sample of builds for a change from Chrome Gerrit code review.

The code snippet B.3 demonstrates the method used to capture the builds data for each change list.

```
def get_tryjob_builds_api_result(change_num, patch_num):
url = "https://cr-buildbucket.appspot.com/prpc/buildbucket.v2.Builds/Batch"
headers = {
    'authority': 'cr-buildbucket.appspot.com',
    'sec-ch-ua': '"Chromium";v="86", "\\"Not\\\\A;Brand";v="99", "Google Chrome";v="86"',
    'accept': 'application/json',
```

```python
    # 'authorization': 'Bearer ya29.a0AfH6SMBqFDnz98YJB3n2DbB6I-oDKPxJgktdn3HeGAT7wgHvntGIw33oWwTLR8l3TgQ0b5xKpZtcEMf5nDenR5mHcG_S2oqrdJUEkusTT_6-7JL9GvmlcaDv7ZGikJWTwCzOtT9xtCtYA-gxa0_QE72dM5Kw-2THYk-NXu9_LZxXtKaVaqHSE6HZbgQ5rAF5DB84NoQRHElEUhTyRSO7ect_bHBvbmqAJbF8nVWLf0V8EtPEpHYf',
    'sec-ch-ua-mobile': '?0',
    'user-agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/86.0.4240.198 Safari/537.36',
    'content-type': 'application/json',
    'origin': 'https://chromium-review.googlesource.com',
    'sec-fetch-site': 'cross-site',
    'sec-fetch-mode': 'cors',
    'sec-fetch-dest': 'empty',
    'accept-language': 'en-US,en;q=0.9,fa-IR;q=0.8,fa;q=0.7',
}
data = '{"requests":[{"searchBuilds":{"pageSize":500,"predicate":{"gerritChanges":[{"host":"chromium-review.googlesource.com","project":"chromium/src","change":' + str(
    change_num) + ',"patchset":' + str(
    patch_num) + '}]},"fields":"builds.*.id,builds.*.builder,builds.*.tags,builds.*.status,builds.*.critical"}}]}'
response = requests.post(url=url, headers=headers, data=data)
return response.content
```

Listing B.3: Get change builds method

Figure B.4 presents a sample of build details obtained from the Chrome website.

The code in Listing B.4 demonstrates how we capture the details of a build.

```python
def get_build_details_api_result(build_id):
headers = {
    'authority': 'cr-buildbucket.appspot.com',
    'sec-ch-ua': '"Google Chrome";v="87", " Not;A Brand";v="99", "Chromium";v="87"',
    'accept': 'application/json',
    'sec-ch-ua-mobile': '?0',
    'user-agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_4)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/87.0.4280.88 Safari/537.36',
    'content-type': 'application/json',
    'origin': 'https://ci.chromium.org',
    'sec-fetch-site': 'cross-site',
    'sec-fetch-mode': 'cors',
    'sec-fetch-dest': 'empty',
    'referer': 'https://ci.chromium.org/',
    'accept-language': 'en-US,en;q=0.9,fa-IR;q=0.8,fa;q=0.7',
}

#data = '{"builder":{"project":"chromium","bucket":"try","builder":"
fuchsia_x64"},"buildNumber":713505,"fields":"*"}'
data = '{"id":"' + str(build_id) + '","fields":"*"}'

response = requests.post('https://cr-buildbucket.appspot.com/prpc/
buildbucket.v2.Builds/GetBuild', headers=headers,
                         data=data)
return response.content
```

Listing B.4: Get change builds method

Figure B.5 illustrates a sample of test results for a build in Chrome.

The code snippet B.5 demonstrates the process of retrieving the tests associated with a build by making an API call to Chrome.

```
def get_test_variants_api_result(build_id, nextPageToken=''):
    headers = {
        'authority': 'results.api.cr.dev',
        'sec-ch-ua': '"Chromium";v="86", "\\"Not\\\\A;Brand";v="99", "Google
    Chrome";v="86"',
        'accept': 'application/json',
        'sec-ch-ua-mobile': '?0',
        'user-agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_4)
    AppleWebKit/537.36 (KHTML, like Gecko) Chrome/86.0.4240.198 Safari/537.36',
        'content-type': 'application/json',
        'origin': 'https://ci.chromium.org',
        'sec-fetch-site': 'cross-site',
        'sec-fetch-mode': 'cors',
        'sec-fetch-dest': 'empty',
        'referer': 'https://ci.chromium.org/',
        'accept-language': 'en-US,en;q=0.9,fa-IR;q=0.8,fa;q=0.7',
    }

    data = '{"invocations":["invocations/build-' + build_id + '"],"pageToken":"'
    + nextPageToken +'"}'

    response = requests.post('https://results.api.cr.dev/prpc/luci.resultdb.
    internal.ui.UI/QueryTestVariants',
                                 headers=headers, data=data)
    if response.status_code == 404:
        return Status.not_found
    elif response.status_code == 500:
        return Status.internal_server_error
    return response.content
```

Listing B.5: Get change builds method

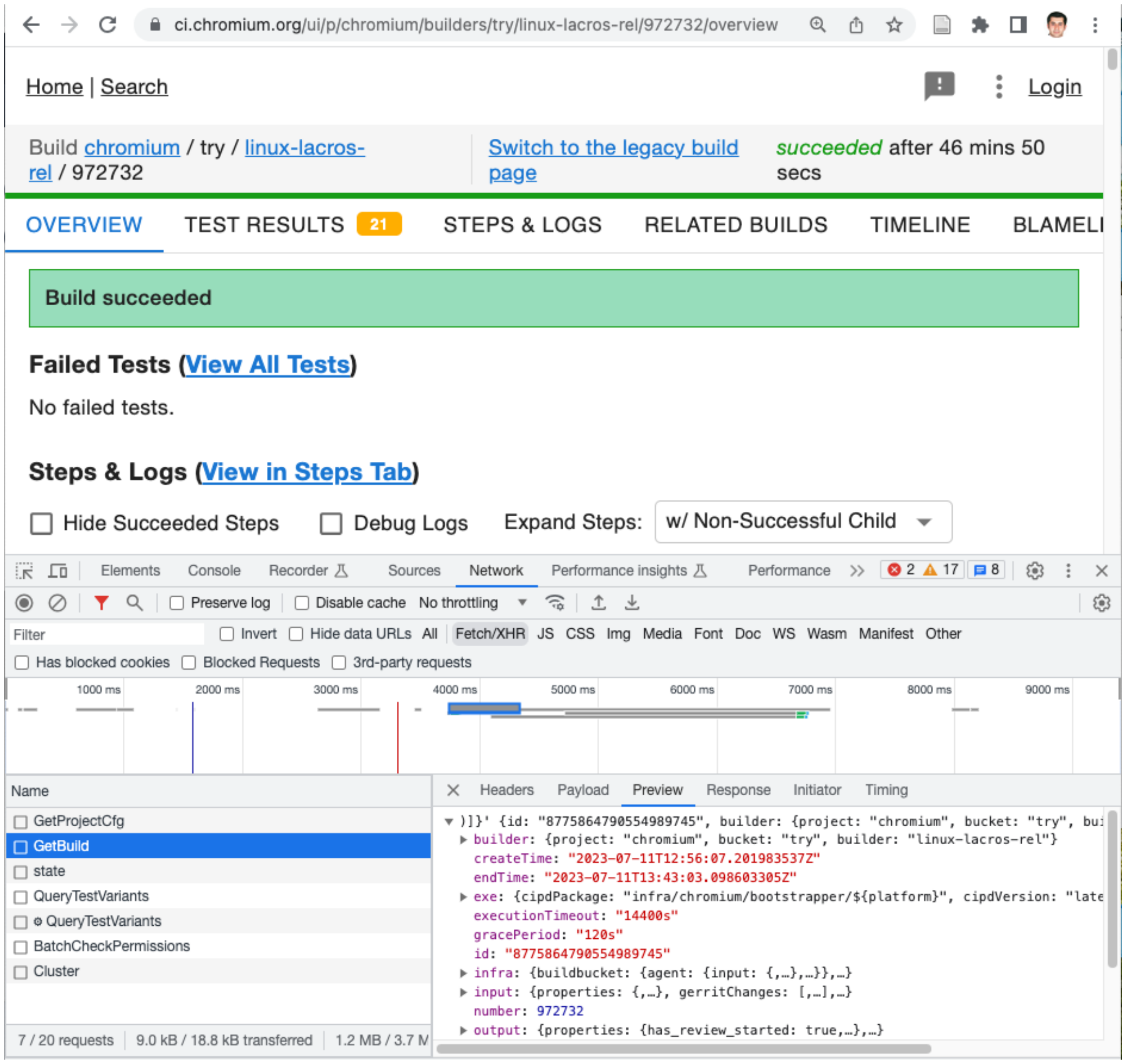
Using the previously mentioned APIs, we successfully captured the Chrome change lists, builds, and test results. The relevant data of interest for testing was then stored in three tables: Table B.1 for change lists, Table B.2 for builds and Table B.3 for test results.

Table B.1: Changes table

Fields	Types
change_id	text
change_num	int
subject	text
status	text
creat_t	timestamp
updated_t	timestamp
submitted_t	timestamp
has_review_started	boolean

Table B.2: Builds table

Fields	Types
build_id	text
builder	text
change_num	integer
patch_num	integer
build_start_t	timestamp
build_end_t	timestamp
status	text



Build chromium / try / linux-lacros-rel / 972732

Switch to the legacy build page

succeeded after 46 mins 50 secs

Build succeeded

Failed Tests (View All Tests)

No failed tests.

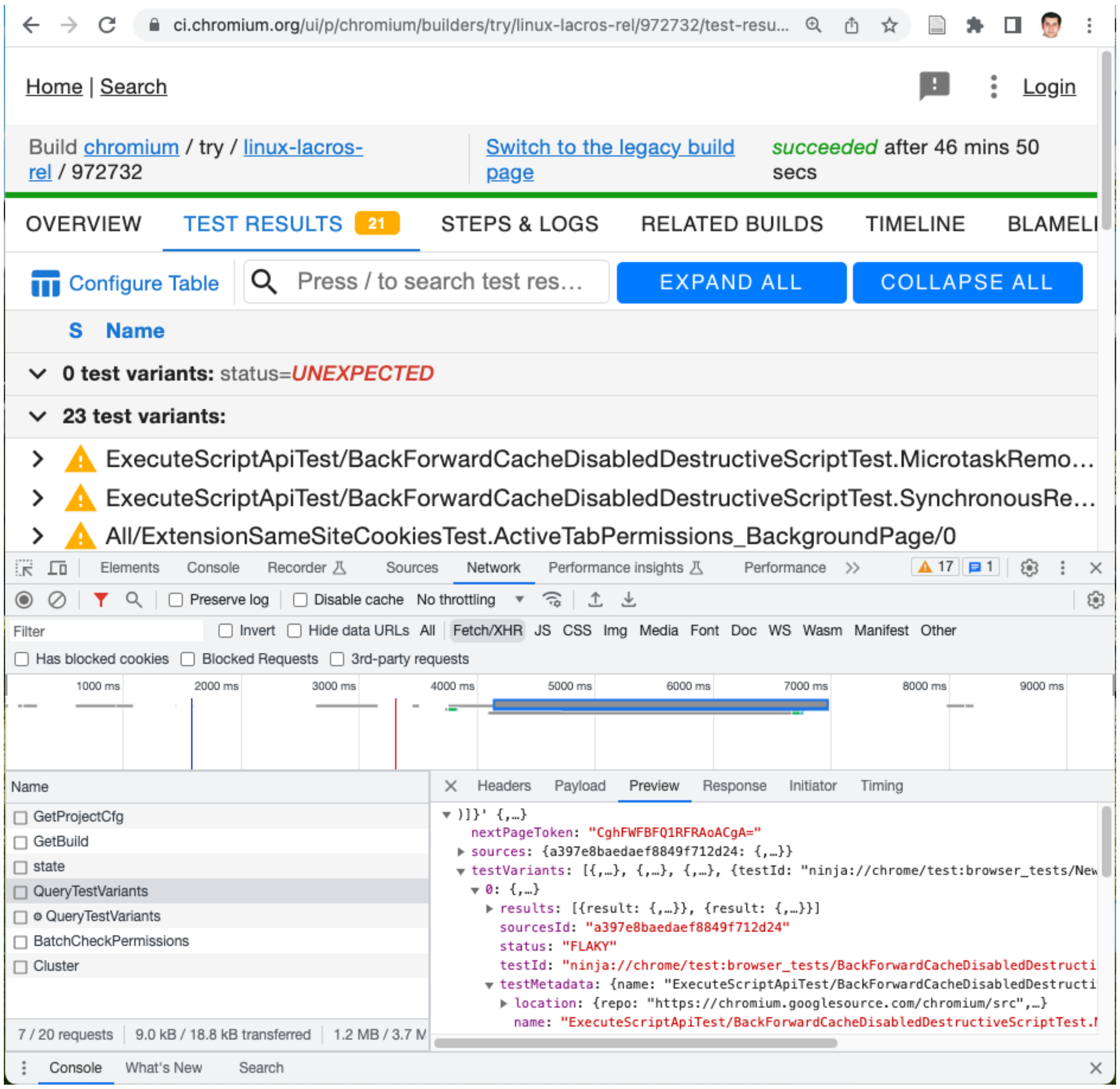
Steps & Logs (View in Steps Tab)

Hide Succeeded Steps   Debug Logs   Expand Steps: w/ Non-Successful Child

```
GetProjectCfg
GetBuild
state
QueryTestVariants
⚙ QueryTestVariants
BatchCheckPermissions
Cluster

7 / 20 requests   9.0 kB / 18.8 kB transferred   1.2 MB / 3.7 M
```

```
▼ )]}' {id: "8775864790554989745", builder: {project: "chromium", bucket: "try", bu
  ▶ builder: {project: "chromium", bucket: "try", builder: "linux-lacros-rel"}
    createTime: "2023-07-11T12:56:07.201983537Z"
    endTime: "2023-07-11T13:43:03.098603305Z"
  ▶ exe: {cipdPackage: "infra/chromium/bootstrapper/${platform}", cipdVersion: "late
    executionTimeout: "14400s"
    gracePeriod: "120s"
    id: "8775864790554989745"
  ▶ infra: {buildbucket: {agent: {input: {,…},…}},…}
  ▶ input: {properties: {,…}, gerritChanges: [,…],…}
    number: 972732
  ▶ output: {properties: {has_review_started: true,…},…}
```

Figure B.4: A sample of build details from Chrome Gerrit code review.



ci.chromium.org/ui/p/chromium/builders/try/linux-lacros-rel/972732/test-resu...

Home | Search   Login

Build chromium / try / linux-lacros-rel / 972732   Switch to the legacy build page   *succeeded* after 46 mins 50 secs

OVERVIEW TEST RESULTS 21 STEPS & LOGS RELATED BUILDS TIMELINE BLAMELI

Configure Table   Press / to search test res…   EXPAND ALL   COLLAPSE ALL

S Name

0 test variants: status=*UNEXPECTED*

23 test variants:

ExecuteScriptApiTest/BackForwardCacheDisabledDestructiveScriptTest.MicrotaskRemo…

ExecuteScriptApiTest/BackForwardCacheDisabledDestructiveScriptTest.SynchronousRe…

All/ExtensionSameSiteCookiesTest.ActiveTabPermissions_BackgroundPage/0

Name
GetProjectCfg
GetBuild
state
QueryTestVariants
QueryTestVariants
BatchCheckPermissions
Cluster

7 / 20 requests | 9.0 kB / 18.8 kB transferred | 1.2 MB / 3.7 M

```
▼ )]}' {,…}
    nextPageToken: "CghFWFBFQ1RFRAoACgA="
  ▶ sources: {a397e8baedaef8849f712d24: {,…}}
  ▼ testVariants: [{,…}, {,…}, {,…}, {testId: "ninja://chrome/test:browser_tests/New
    ▼ 0: {,…}
      ▶ results: [{result: {,…}}, {result: {,…}}]
        sourcesId: "a397e8baedaef8849f712d24"
        status: "FLAKY"
        testId: "ninja://chrome/test:browser_tests/BackForwardCacheDisabledDestructi
      ▼ testMetadata: {name: "ExecuteScriptApiTest/BackForwardCacheDisabledDestructi
        ▶ location: {repo: "https://chromium.googlesource.com/chromium/src",…}
          name: "ExecuteScriptApiTest/BackForwardCacheDisabledDestructiveScriptTest.M
```

Figure B.5: A sample of build test results from Chrome Gerrit code review.

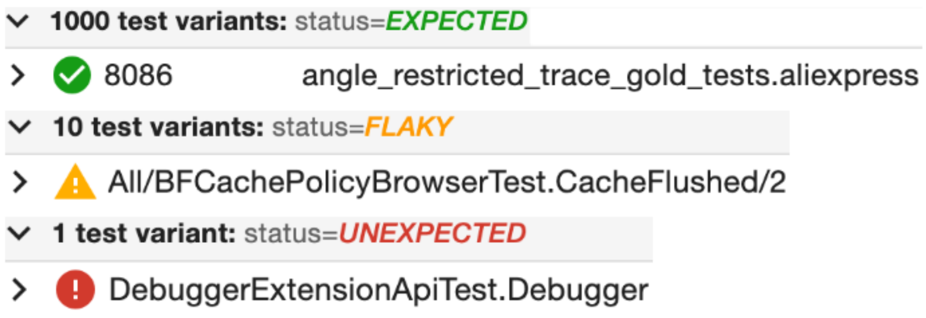
Table B.3: Test results

Fields	Types
test_id	text
test_name	text
test_suite	text
change_num	int
patch_num	int
build_id	text
build_start_t	timestamp
build_end_t	timestamp
builder	text
result_id	text
final_status	text
status	text
duration	interval
step_name	text

Appendix C

Samples of Different Test Results

In this appendix, we provide samples showcasing different types of test results. Figure C.1 illustrates examples of expected, unexpected, and flaky tests.



```
⌄ 1000 test variants: status=EXPECTED
> ✓ 8086          angle_restricted_trace_gold_tests.aliexpress
⌄ 10 test variants: status=FLAKY
> ⚠ All/BFCachePolicyBrowserTest.CacheFlushed/2
⌄ 1 test variant: status=UNEXPECTED
> ! DebuggerExtensionApiTest.Debugger
```

Figure C.1: A sample of different types of test results

Figure C.2 illustrates a sample of an expected result, where both the expected and actual results are ”pass”.

```
"inline-text-changes.html": {
  "actual": "PASS",
  "expected": "PASS",
  "time": 0.2
},
```

Figure C.2: A sample of expected pass result in Chrome logs

Figure C.3 displays a sample of an expected crash result. The expected list for the test includes ”fail”, ”crash”, and ”pass”, while the actual result is ”crash”.

```
"doubleclick-on-meter-in-shadow-crash.html": {
  "actual": "CRASH",
  "artifacts": {
    "crash_log": [
      "layout-test-results/editing/shadow/doubleclick-on-meter-in-shadow-crash-crash-log.txt"
    ],
    "stderr": [
      "layout-test-results/editing/shadow/doubleclick-on-meter-in-shadow-crash-stderr.txt"
    ]
  },
  "bugs": [
    "crbug.com/725470"
  ],
  "crash_site": "ephemeral_range.cc(45)",
  "expected": "FAIL CRASH PASS",
  "has_stderr": true,
  "time": 1.1
},
```

Figure C.3: A sample of expected crash result in Chrome logs

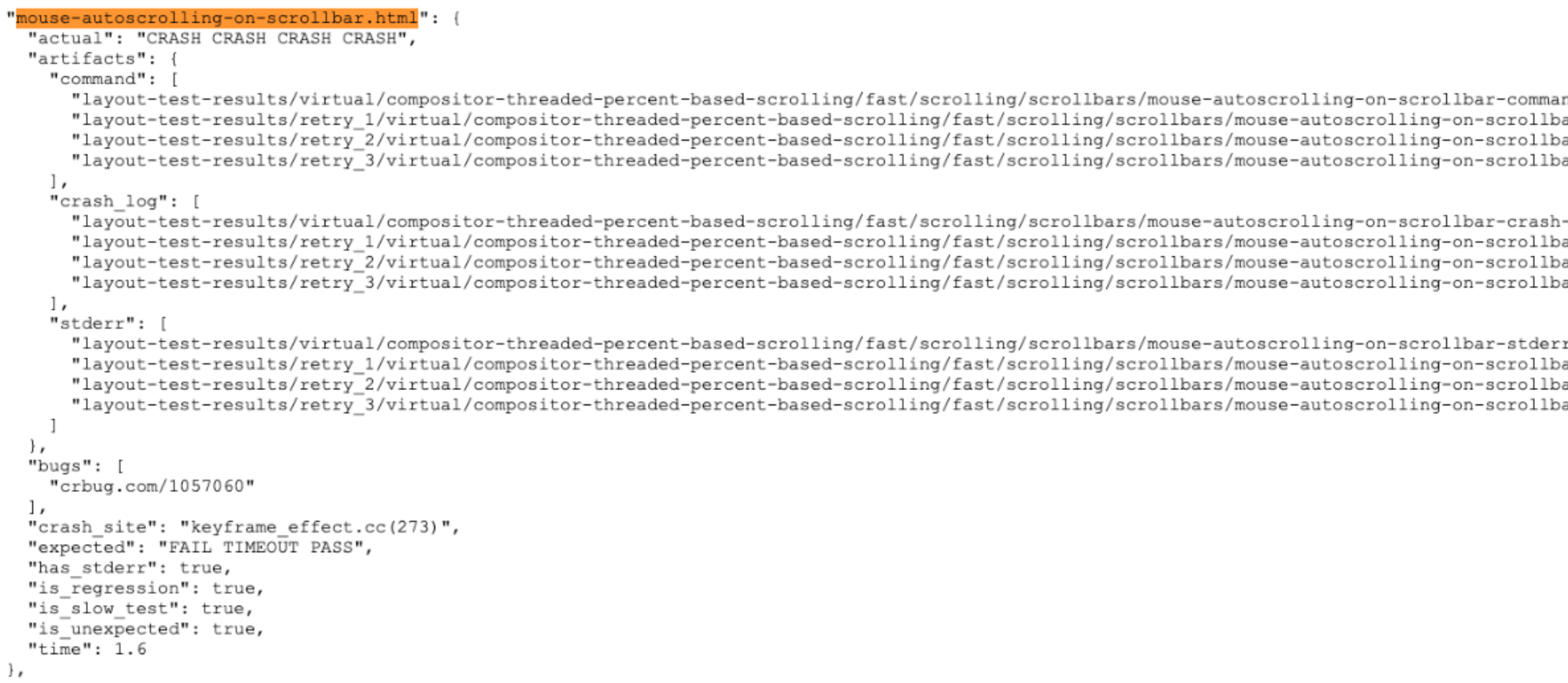
Figure C.4 depicts a sample of an expected failure result. The list of expected results includes ”fail”, ”timeout”, and ”pass”, while the actual result is ”fail”.

In Figure C.5, we can observe a sample of an unexpected crash test result. The list of expected results includes ”fail”, ”timeout”, and ”pass”. However, the test crashes, and since the result is unexpected, it is rerun three times with the same outcome. Therefore, the test is categorized as an unexpected failure. Figure C.6 provides further details on the reruns for this unexpected result.

Figure C.7 showcases a sample of a flaky test result. The test is expected to pass but produces a fail result. In the subsequent rerun, the result remains the same. However, in the third run, a pass result is generated. Consequently, this test is categorized as a flaky test result. Figure C.8 provides detailed information about the reruns for this flaky test.

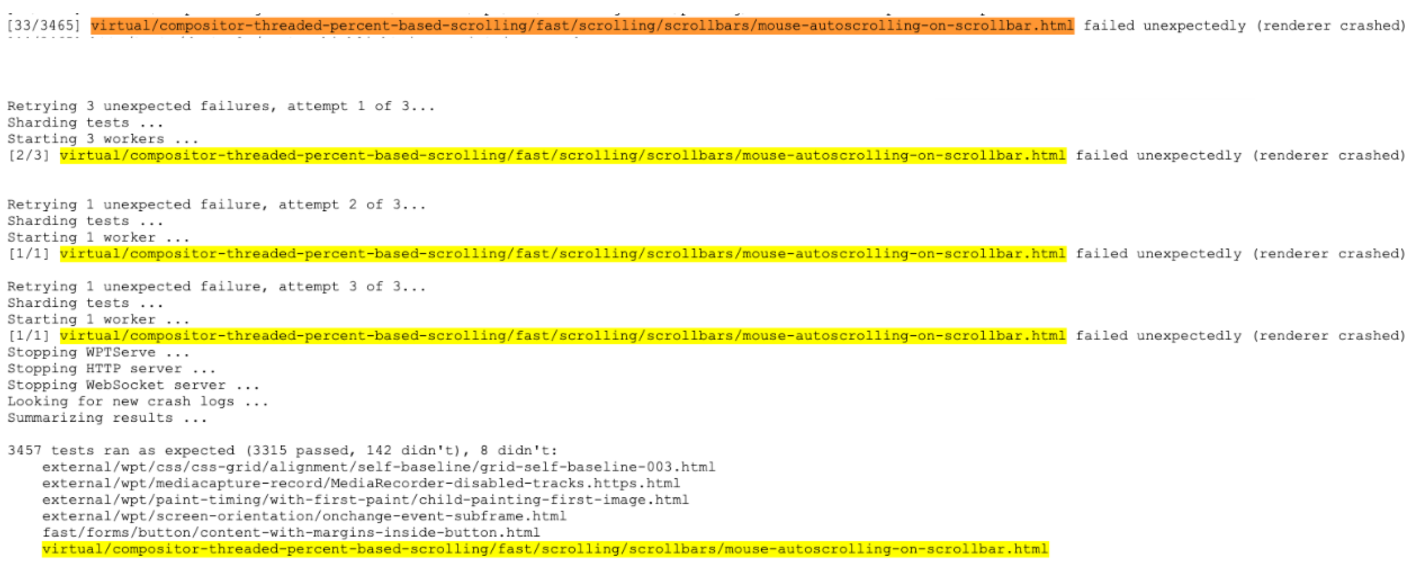
```
"playback-rate.https.html": {
  "actual": "FAIL",
  "artifacts": {
    "actual_text": [
      "layout-test-results/external/wpt/animation-worklet/playback-rate.https-actual.txt"
    ],
    "pretty_text_diff": [
      "layout-test-results/external/wpt/animation-worklet/playback-rate.https-pretty-diff.html"
    ],
    "stderr": [
      "layout-test-results/external/wpt/animation-worklet/playback-rate.https-stderr.txt"
    ],
    "text_diff": [
      "layout-test-results/external/wpt/animation-worklet/playback-rate.https-diff.txt"
    ]
  },
  "bugs": [
    "crbug.com/1014812"
  ],
  "expected": "FAIL TIMEOUT PASS",
  "has_stderr": true,
  "is_testharness_test": true,
  "time": 0.4
},
```

Figure C.4: A sample of expected failure result in Chrome logs



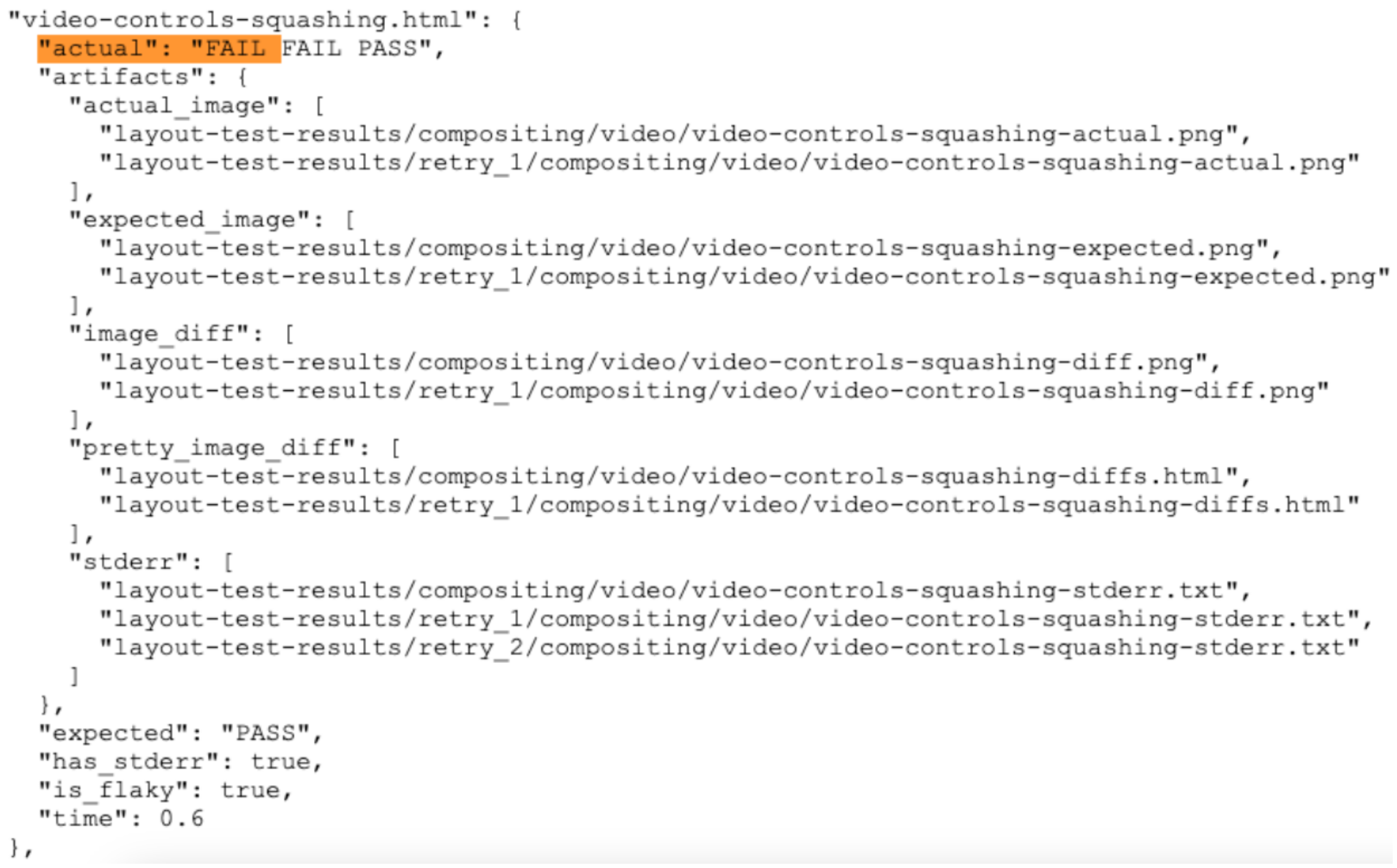
```
"mouse-autoscrolling-on-scrollbar.html": {
  "actual": "CRASH CRASH CRASH CRASH",
  "artifacts": {
    "command": [
      "layout-test-results/virtual/compositor-threaded-percent-based-scrolling/fast/scrolling/scrollbars/mouse-autoscrolling-on-scrollbar-comman
      "layout-test-results/retry_1/virtual/compositor-threaded-percent-based-scrolling/fast/scrolling/scrollbars/mouse-autoscrolling-on-scrollba
      "layout-test-results/retry_2/virtual/compositor-threaded-percent-based-scrolling/fast/scrolling/scrollbars/mouse-autoscrolling-on-scrollba
      "layout-test-results/retry_3/virtual/compositor-threaded-percent-based-scrolling/fast/scrolling/scrollbars/mouse-autoscrolling-on-scrollba
    ],
    "crash_log": [
      "layout-test-results/virtual/compositor-threaded-percent-based-scrolling/fast/scrolling/scrollbars/mouse-autoscrolling-on-scrollbar-crash-
      "layout-test-results/retry_1/virtual/compositor-threaded-percent-based-scrolling/fast/scrolling/scrollbars/mouse-autoscrolling-on-scrollba
      "layout-test-results/retry_2/virtual/compositor-threaded-percent-based-scrolling/fast/scrolling/scrollbars/mouse-autoscrolling-on-scrollba
      "layout-test-results/retry_3/virtual/compositor-threaded-percent-based-scrolling/fast/scrolling/scrollbars/mouse-autoscrolling-on-scrollba
    ],
    "stderr": [
      "layout-test-results/virtual/compositor-threaded-percent-based-scrolling/fast/scrolling/scrollbars/mouse-autoscrolling-on-scrollbar-stderr
      "layout-test-results/retry_1/virtual/compositor-threaded-percent-based-scrolling/fast/scrolling/scrollbars/mouse-autoscrolling-on-scrollba
      "layout-test-results/retry_2/virtual/compositor-threaded-percent-based-scrolling/fast/scrolling/scrollbars/mouse-autoscrolling-on-scrollba
      "layout-test-results/retry_3/virtual/compositor-threaded-percent-based-scrolling/fast/scrolling/scrollbars/mouse-autoscrolling-on-scrollba
    ]
  },
  "bugs": [
    "crbug.com/1057060"
  ],
  "crash_site": "keyframe_effect.cc(273)",
  "expected": "FAIL TIMEOUT PASS",
  "has_stderr": true,
  "is_regression": true,
  "is_slow_test": true,
  "is_unexpected": true,
  "time": 1.6
},
```

Figure C.5: A sample of unexpected crash result in Chrome logs



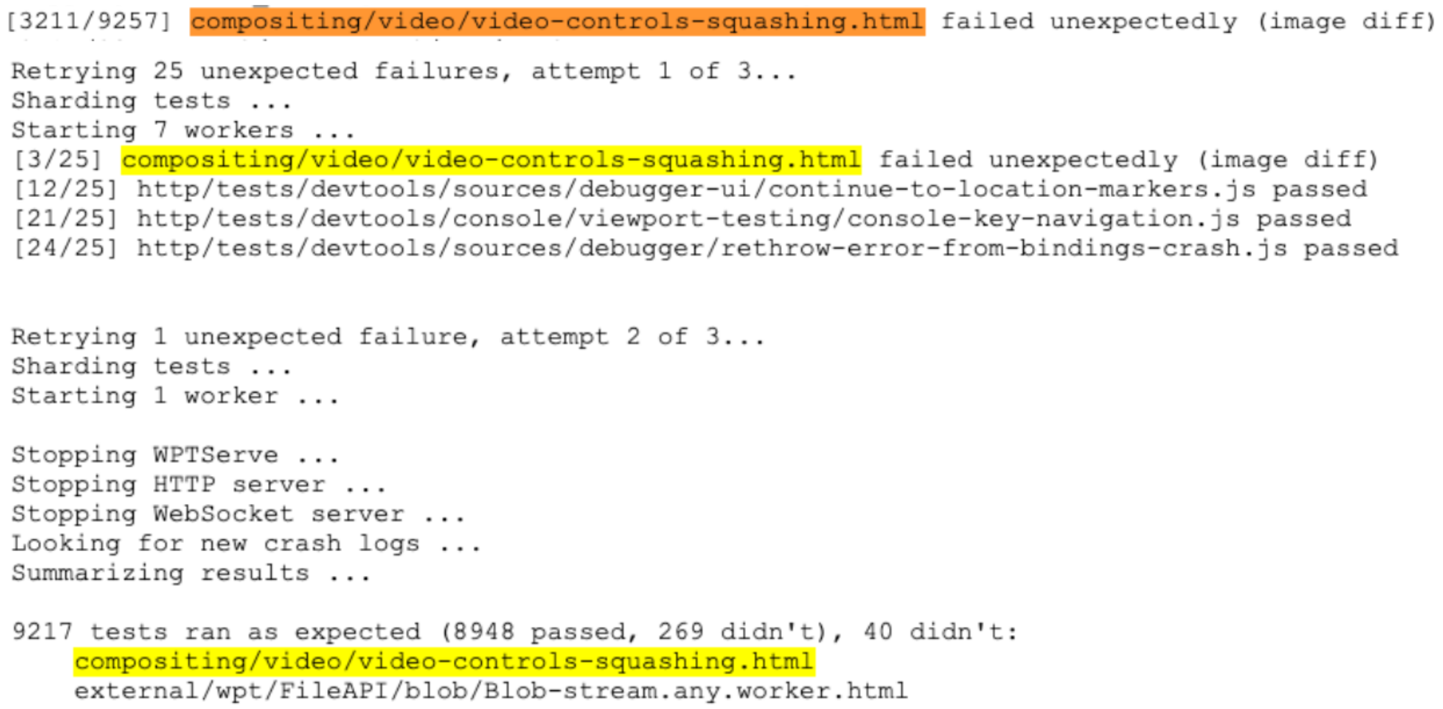
```
[33/3465] virtual/compositor-threaded-percent-based-scrolling/fast/scrolling/scrollbars/mouse-autoscrolling-on-scrollbar.html failed unexpectedly (renderer crashed)


Retrying 3 unexpected failures, attempt 1 of 3...
Sharding tests ...
Starting 3 workers ...
[2/3] virtual/compositor-threaded-percent-based-scrolling/fast/scrolling/scrollbars/mouse-autoscrolling-on-scrollbar.html failed unexpectedly (renderer crashed)


Retrying 1 unexpected failure, attempt 2 of 3...
Sharding tests ...
Starting 1 worker ...
[1/1] virtual/compositor-threaded-percent-based-scrolling/fast/scrolling/scrollbars/mouse-autoscrolling-on-scrollbar.html failed unexpectedly (renderer crashed)

Retrying 1 unexpected failure, attempt 3 of 3...
Sharding tests ...
Starting 1 worker ...
[1/1] virtual/compositor-threaded-percent-based-scrolling/fast/scrolling/scrollbars/mouse-autoscrolling-on-scrollbar.html failed unexpectedly (renderer crashed)
Stopping WPTServe ...
Stopping HTTP server ...
Stopping WebSocket server ...
Looking for new crash logs ...
Summarizing results ...

3457 tests ran as expected (3315 passed, 142 didn't), 8 didn't:
    external/wpt/css/css-grid/alignment/self-baseline/grid-self-baseline-003.html
    external/wpt/mediacapture-record/MediaRecorder-disabled-tracks.https.html
    external/wpt/paint-timing/with-first-paint/child-painting-first-image.html
    external/wpt/screen-orientation/onchange-event-subframe.html
    fast/forms/button/content-with-margins-inside-button.html
    virtual/compositor-threaded-percent-based-scrolling/fast/scrolling/scrollbars/mouse-autoscrolling-on-scrollbar.html
```

Figure C.6: A sample of unexpected crash reruns in Chrome logs



```
"video-controls-squashing.html": {
  "actual": "FAIL FAIL PASS",
  "artifacts": {
    "actual_image": [
      "layout-test-results/compositing/video/video-controls-squashing-actual.png",
      "layout-test-results/retry_1/compositing/video/video-controls-squashing-actual.png"
    ],
    "expected_image": [
      "layout-test-results/compositing/video/video-controls-squashing-expected.png",
      "layout-test-results/retry_1/compositing/video/video-controls-squashing-expected.png"
    ],
    "image_diff": [
      "layout-test-results/compositing/video/video-controls-squashing-diff.png",
      "layout-test-results/retry_1/compositing/video/video-controls-squashing-diff.png"
    ],
    "pretty_image_diff": [
      "layout-test-results/compositing/video/video-controls-squashing-diffs.html",
      "layout-test-results/retry_1/compositing/video/video-controls-squashing-diffs.html"
    ],
    "stderr": [
      "layout-test-results/compositing/video/video-controls-squashing-stderr.txt",
      "layout-test-results/retry_1/compositing/video/video-controls-squashing-stderr.txt",
      "layout-test-results/retry_2/compositing/video/video-controls-squashing-stderr.txt"
    ]
  },
  "expected": "PASS",
  "has_stderr": true,
  "is_flaky": true,
  "time": 0.6
},
```

Figure C.7: A sample of flaky result in Chrome logs



```
[3211/9257] compositing/video/video-controls-squashing.html failed unexpectedly (image diff)

Retrying 25 unexpected failures, attempt 1 of 3...
Sharding tests ...
Starting 7 workers ...
[3/25] compositing/video/video-controls-squashing.html failed unexpectedly (image diff)
[12/25] http/tests/devtools/sources/debugger-ui/continue-to-location-markers.js passed
[21/25] http/tests/devtools/console/viewport-testing/console-key-navigation.js passed
[24/25] http/tests/devtools/sources/debugger/rethrow-error-from-bindings-crash.js passed


Retrying 1 unexpected failure, attempt 2 of 3...
Sharding tests ...
Starting 1 worker ...

Stopping WPTServe ...
Stopping HTTP server ...
Stopping WebSocket server ...
Looking for new crash logs ...
Summarizing results ...

9217 tests ran as expected (8948 passed, 269 didn't), 40 didn't:
    compositing/video/video-controls-squashing.html
    external/wpt/FileAPI/blob/Blob-stream.any.worker.html
```

Figure C.8: A sample of flaky result reruns in Chrome logs

Bibliography

[1] S. Elbaum, G. Rothermel, and J. Penix, "Techniques for Improving Regression Testing in Continuous Integration Development Environments," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 235–245, event-place: Hong Kong, China. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635910

[2] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, "Taming Google-scale continuous testing," in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, May 2017, pp. 233–242.

[3] R. Greca, B. Miranda, and A. Bertolino, "State of Practical Applicability of Regression Testing Research: A Live Systematic Literature Review," *ACM Comput. Surv.*, Jan. 2023. [Online]. Available: https://dl.acm.org/doi/10.1145/3579851

[4] A. Alshammari, C. Morris, M. Hilton, and J. Bell, "FlakeFlagger: Predicting Flakiness Without Rerunning Tests," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, May 2021, pp. 1572–1584, iSSN: 1558-1225.

[5] E. Fallahzadeh and P. Rigby, "Replication Package," Apr. 2021. [Online]. Available: https://github.com/CESEL/FlakyTestsInPrioritization

[6] E. Fallahzadeh and P. C. Rigby, "The impact of flaky tests on historical test prioritization on chrome," in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '22. New York, NY,

USA: Association for Computing Machinery, Oct. 2022, pp. 273–282. [Online]. Available: https://dl.acm.org/doi/10.1145/3510457.3513038

[7] P. Gupta, M. Ivey, and J. Penix, "Testing at the speed and scale of google," *Google Engineering Tools Blog*, 2011.

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

[9] A. Labuschagne, L. Inozemtseva, and R. Holmes, "Measuring the cost of regression testing in practice: A study of java projects using continuous integration," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 821–830. [Online]. Available: https://doi.org/10.1145/3106237.3106288

[10] Y. Zhu, E. Shihab, and P. C. Rigby, "Test Re-Prioritization in Continuous Testing Environments," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2018, pp. 69–79, iSSN: 1063-6773.

[11] M. Machalica, A. Samylkin, M. Porth, and S. Chandra, "Predictive Test Selection," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. Montreal, QC, Canada: IEEE, May 2019, pp. 91–100.

[12] Q. Peng, A. Shi, and L. Zhang, "Empirically revisiting and enhancing IR-based test-case prioritization," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, Jul. 2020, pp. 324–336. [Online]. Available: http://doi.org/10.1145/3395363.3397383

[13] "U.S. and global browser market share 2021." [Online]. Available: https://www.statista.com/statistics/276738/worldwide-and-us-market-share-of-leading-internet-browsers/

[14] T. Mattis, P. Rein, F. Dürsch, and R. Hirschfeld, "RTPTorrent: An Open-source Dataset for Evaluating Regression Test Prioritization," in *Proceedings of the 17th International Conference on Mining Software Repositories*, ser. MSR '20. New York, NY, USA: Association for Computing Machinery, Jun. 2020, pp. 385–396. [Online]. Available: https://doi.org/10.1145/3379597.3387458

[15] "Gerrit Code Review." [Online]. Available: https://chromium-review.googlesource.com

[16] 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*, ser. ICSE '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 329–338, event-place: Toronto, Ontario, Canada. [Online]. Available: http://dl.acm.org/citation.cfm?id=381473.381508

[17] A. Srivastava and J. Thiagarajan, "Effectively Prioritizing Tests in Development Environment," in *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '02. New York, NY, USA: ACM, 2002, pp. 97–106, event-place: Roma, Italy. [Online]. Available: http://doi.acm.org/10.1145/566172.566187

[18] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Prioritizing test cases for regression testing," in *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, ser. ISSTA '00. Portland, Oregon, USA: Association for Computing Machinery, Aug. 2000, pp. 102–112. [Online]. Available: https://doi.org/10.1145/347324.348910

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

[20] ——, "Test case prioritization: an empirical study," in *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*, Aug. 1999, pp. 179–188, iSSN: 1063-6773.

[21] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y. Le Traon, “Comparing White-Box and Black-Box Test Prioritization,” in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, May 2016, pp. 523–534, iSSN: 1558-1225.

[22] Y. Lu, Y. Lou, S. Cheng, L. Zhang, D. Hao, Y. Zhou, and L. Zhang, “How does regression test prioritization perform in real-world software evolution?” in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE ’16. New York, NY, USA: Association for Computing Machinery, May 2016, pp. 535–546. [Online]. Available: https://doi.org/10.1145/2884781.2884874

[23] Q. Luo, K. Moran, and D. Poshyvanyk, “A large-scale empirical comparison of static and dynamic test case prioritization techniques,” in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, Nov. 2016, pp. 559–570. [Online]. Available: https://doi.org/10.1145/2950290.2950344

[24] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei, “Bridging the gap between the total and additional test-case prioritization strategies,” in *2013 35th International Conference on Software Engineering (ICSE)*, May 2013, pp. 192–201, iSSN: 1558-1225.

[25] D. Marijan, A. Gotlieb, and S. Sen, “Test Case Prioritization for Continuous Regression Testing: An Industrial Case Study,” in *2013 IEEE International Conference on Software Maintenance*, Sep. 2013, pp. 540–543, iSSN: 1063-6773.

[26] A. Najafi, W. Shang, and P. C. Rigby, “Improving Test Effectiveness Using Test Executions History: An Industrial Experience Report,” in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, May 2019, pp. 213–222.

[27] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, “DeFlaker: Automatically Detecting Flaky Tests,” in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, May 2018, pp. 433–444, iSSN: 1558-1225.

[28] M. Hilton, J. Bell, and D. Marinov, "A large-scale study of test coverage evolution," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: Association for Computing Machinery, Sep. 2018, pp. 53–63. [Online]. Available: https://doi.org/10.1145/3238147.3238183

[29] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta, "Root causing flaky tests in a large-scale industrial setting," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. New York, NY, USA: Association for Computing Machinery, Jul. 2019, pp. 101–111. [Online]. Available: https://doi.org/10.1145/3293882.3330570

[30] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov, "iFixFlakies: a framework for automatically fixing order-dependent flaky tests," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, Aug. 2019, pp. 545–555. [Online]. Available: https://doi.org/10.1145/3338906.3338925

[31] M. T. Rahman and P. C. Rigby, "The impact of failing, flaky, and high failure tests on the number of crash reports associated with Firefox builds," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, Oct. 2018, pp. 857–862. [Online]. Available: https://doi.org/10.1145/3236024.3275529

[32] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, "Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset," *Empir Software Eng*, vol. 22, no. 4, pp. 1936–1964, Aug. 2017. [Online]. Available: https://doi.org/10.1007/s10664-016-9470-4

[33] E. Fallahzadeh, A. H. Bavand, and P. C. Rigby, "Accelerating Continuous Integration with

Parallel Batch Testing," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2023)(ESEC/FSE2023)*. New York, NY, USA: ACM, 2023, p. 13, accepted, to be presented at ACM ESEC/FSE 2023.

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

[35] E. Engström, P. Runeson, and M. Skoglund, "A systematic review on regression test selection techniques," *Information and Software Technology*, vol. 52, no. 1, pp. 14–30, 2010. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584909001219

[36] S. Chen, Z. Chen, Z. Zhao, B. Xu, and Y. Feng, "Using semi-supervised clustering to improve regression test selection techniques," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, 2011, pp. 1–10.

[37] M. Bagherzadeh, N. Kahani, and L. Briand, "Reinforcement learning for test case prioritization," *IEEE Transactions on Software Engineering*, pp. 1–1, 2021.

[38] Y. Zhu, E. Shihab, and P. C. Rigby, "Test re-prioritization in continuous testing environments," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2018, pp. 69–79.

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

[40] A. Najafi, P. C. Rigby, and W. Shang, "Bisecting commits and modeling commit risk during testing," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. Tallinn, Estonia: Association for Computing Machinery, Aug. 2019, pp. 279–289. [Online]. Available: https://doi.org/10.1145/3338906.3338944

[41] M. J. Beheshtian, A. Bavand, and P. Rigby, "Software batch testing to save build test resources and to reduce feedback time," *IEEE Transactions on Software Engineering*, pp. 1–1, 2021.

[42] A. H. Bavand and P. C. Rigby, "Mining Historical Test Failures to Dynamically Batch Tests to Save CI Resources," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Luxembourg: IEEE, Sep. 2021, pp. 217–226, iSSN: 2576-3148.

[43] C. Ziftci and J. Reardon, "Who broke the build? automatically identifying changes that induce test failures in continuous integration at google scale," in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, 2017, pp. 113–122.

[44] R. Dorfman, "The detection of defective members of large populations," *Ann. Math. Statist.*, vol. 14, no. 4, pp. 436–440, 12 1943. [Online]. Available: https://doi.org/10.1214/aoms/1177731363

[45] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "idflakies: A framework for detecting and partially classifying flaky tests," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. Los Alamitos, CA, USA: IEEE Computer Society, apr 2019, pp. 312–322. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICST.2019.00038

[46] X. Jin and F. Servant, "What helped, and what did not? an evaluation of the strategies to improve continuous integration," 2021.

[47] A. Poth, M. Werner, and X. Lei, "How to deliver faster with ci/cd integrated testing services?" in *European Conference on Software Process Improvement*. Springer, 2018, pp. 401–409.

[48] M. Hilton, "Understanding and improving continuous integration," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 1066–1067.

[49] M. Soni, "End to end automation on cloud with build pipeline: the case for devops in insurance industry, continuous integration, continuous testing, and continuous delivery," in *2015 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*. IEEE, 2015, pp. 85–89.

[50] M. Leppänen, S. Mäkinen, M. Pagels, V.-P. Eloranta, J. Itkonen, M. V. Mäntylä, and T. Männistö, "The highways and country roads to continuous deployment," *Ieee software*, vol. 32, no. 2, pp. 64–72, 2015.

[51] 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 - Volume 1*, ser. ICSE '15. IEEE Press, 2015, p. 483–493.

[52] S. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 235–245.

[53] G. Parthasarathy, A. Rushdi, P. Choudhary, S. Nanda, M. Evans, H. Gunasekara, and S. Rajakumar, "RTL Regression Test Selection using Machine Learning," in *2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC)*. Taipei, Taiwan: IEEE, Jan. 2022, pp. 281–287, iSSN: 2153-697X.

[54] A. Arrieta, "Multi-objective metamorphic follow-up test case selection for deep learning systems," in *Proceedings of the Genetic and Evolutionary Computation Conference*, ser. GECCO '22. New York, NY, USA: Association for Computing Machinery, Jul. 2022, pp. 1327–1335. [Online]. Available: https://dl.acm.org/doi/10.1145/3512290.3528697

[55] D. Elsner, R. Wuersching, M. Schnappinger, A. Pretschner, M. Graber, R. Dammer, and S. Reimer, "Build system aware multi-language regression test selection in continuous integration," in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '22. New York, NY, USA: Association for Computing Machinery, Oct. 2022, pp. 87–96. [Online]. Available: https://doi.org/10.1145/3510457.3513078

[56] A. S. Verma, A. Choudhary, and S. Tiwari, “A novel chaotic archimedes optimization algorithm and its application for efficient selection of regression test cases,” *Int. j. inf. tecnol.*, vol. 15, no. 2, pp. 1055–1068, Feb. 2023. [Online]. Available: https://doi.org/10.1007/s41870-022-01031-7

[57] C. Birchler, N. Ganz, S. Khatiri, A. Gambi, and S. Panichella, “Cost-effective simulation-based test selection in self-driving cars softwareImage 1,” *Science of Computer Programming*, vol. 226, p. 102926, Mar. 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167642323000084

[58] H. Jahan, Z. Feng, and S. M. H. Mahmud, “Risk-Based Test Case Prioritization by Correlating System Methods and Their Associated Risks,” *Arab J Sci Eng*, vol. 45, no. 8, pp. 6125–6138, Aug. 2020. [Online]. Available: https://doi.org/10.1007/s13369-020-04472-z

[59] A. Sharif, D. Marijan, and M. Liaaen, “DeepOrder: Deep Learning for Test Case Prioritization in Continuous Integration Testing,” in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Luxembourg: IEEE, Sep. 2021, pp. 525–534, iSSN: 2576-3148.

[60] R. Huang, D. Towey, Y. Xu, Y. Zhou, and N. Yang, “Dissimilarity-based test case prioritization through data fusion,” *Software: Practice and Experience*, vol. 52, no. 6, pp. 1352–1377, 2022, _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.3068. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.3068

[61] M. Mahdieh, S.-H. Mirian-Hosseinabadi, and M. Mahdieh, “Test case prioritization using test case diversification and fault-proneness estimations,” *Autom Softw Eng*, vol. 29, no. 2, p. 50, Aug. 2022. [Online]. Available: https://doi.org/10.1007/s10515-022-00344-y

[62] S. Nayak, C. Kumar, and S. Tripathi, “Analytic hierarchy process-based regression test case prioritization technique enhancing the fault detection rate,” *Soft Comput*, vol. 26, no. 15, pp. 6953–6968, Aug. 2022. [Online]. Available: https://doi.org/10.1007/s00500-022-07174-w

[63] A. Bajaj, O. P. Sangwan, and A. Abraham, “Improved novel bat algorithm for test case prioritization and minimization,” *Soft Comput*, vol. 26, no. 22, pp. 12 393–12 419, Nov. 2022. [Online]. Available: https://doi.org/10.1007/s00500-022-07121-9

[64] A. S. Yaraghi, M. Bagherzadeh, N. Kahani, and L. C. Briand, “Scalable and Accurate Test Case Prioritization in Continuous Integration Contexts,” *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1615–1639, Apr. 2023, conference Name: IEEE Transactions on Software Engineering.

[65] J. Liang, S. Elbaum, and G. Rothermel, “Redefining prioritization: Continuous prioritization for continuous integration,” in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 688–698. [Online]. Available: https://doi.org/10.1145/3180155.3180213

[66] M. Beller, G. Gousios, and A. Zaidman, “Oops, my tests broke the build: An explorative analysis of travis ci with github,” in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, 2017, pp. 356–367.

[67] A. Najafi, W. Shang, and P. C. Rigby, “Improving test effectiveness using test executions history: An industrial experience report,” in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2019, pp. 213–222.

[68] J. A. Jones, M. J. Harrold, and J. Stasko, “Visualization of test information to assist fault localization,” in *Proceedings of the 24th International Conference on Software Engineering*, ser. ICSE ’02. New York, NY, USA: Association for Computing Machinery, 2002, p. 467–477. [Online]. Available: https://doi.org/10.1145/581339.581397

[69] S. Misailovic, A. Milicevic, N. Petrovic, S. Khurshid, and D. Marinov, “Parallel test generation and execution with korat,” in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE ’07. New York, NY,

USA: Association for Computing Machinery, 2007, p. 135–144. [Online]. Available: https://doi-org.lib-ezproxy.concordia.ca/10.1145/1287624.1287645

[70] T. O. S. Bagies, "Parallelizing unit test execution on gpu," Ph.D. dissertation, Iowa State University, 2020.

[71] C. Landing, S. Tahvili, H. Haggren, M. Langkvis, A. Muhammad, and A. Loufi, "Cluster-based parallel testing using semantic analysis," in *2020 IEEE International Conference On Artificial Intelligence Testing (AITest)*, 2020, pp. 99–106.

[72] H. Arabnejad, J. Bispo, J. G. Barbosa, and J. M. Cardoso, "Autopar-clava: An automatic parallelization source-to-source tool for c code applications," in *Proceedings of the 9th Workshop and 7th Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*, 2018, pp. 13–19.

[73] shashban, "Run VSTest tests in parallel - Azure Pipelines," Nov. 2022. [Online]. Available: https://learn.microsoft.com/en-us/azure/devops/pipelines/test/parallel-testing-vstest?view=azure-devops

[74] J. Candido, L. Melo, and M. d'Amorim, "Test suite parallelization in open-source projects: a study on its usage and impact," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 838–848.

[75] J. Bell, G. Kaiser, E. Melski, and M. Dattatreya, "Efficient dependency detection for safe java test acceleration," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 770–781. [Online]. Available: https://doi.org/10.1145/2786805.2786823

[76] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang, "Software behavior oriented parallelization," *ACM SIGPlan Notices*, vol. 42, no. 6, pp. 223–234, 2007.

[77] S. M. Nagy, H. A. Maghawry, and N. L. Badr, “An Enhanced Parallel Automation Testing Architecture for Test Case Execution,” in *2022 5th International Conference on Computing and Informatics (ICCI)*. New Cairo, Cairo, Egypt: IEEE, Mar. 2022, pp. 369–373.

[78] J. Finlay, R. Pears, and A. M. Connor, “Data stream mining for predicting software build outcomes using source code metrics,” *Information and Software Technology*, vol. 56, no. 2, pp. 183–198, Feb. 2014. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584913001766

[79] A. E. Hassan and K. Zhang, “Using Decision Trees to Predict the Certification Result of a Build,” in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE’06)*. Tokyo, Japan: IEEE, Sep. 2006, pp. 189–198, iSSN: 1938-4300.

[80] F. Hassan and X. Wang, “Change-Aware Build Prediction Model for Stall Avoidance in Continuous Integration,” in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. Toronto, ON, Canada: IEEE, Nov. 2017, pp. 157–162.

[81] I. Kwan, A. Schroter, and D. Damian, “Does Socio-Technical Congruence Have an Effect on Software Build Success? A Study of Coordination in a Software Project,” *IEEE Transactions on Software Engineering*, vol. 37, no. 3, pp. 307–324, May 2011, conference Name: IEEE Transactions on Software Engineering.

[82] A. Ni and M. Li, “Cost-Effective Build Outcome Prediction Using Cascaded Classifiers,” in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. Buenos Aires, Argentina: IEEE, May 2017, pp. 455–458.

[83] A. Schröter, “Predicting build outcome with developer interaction in Jazz,” in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ser. ICSE ’10. New York, NY, USA: Association for Computing Machinery, May 2010, pp. 511–512. [Online]. Available: https://dl.acm.org/doi/10.1145/1810295.1810456

[84] T. Wolf, A. Schroter, D. Damian, and T. Nguyen, “Predicting build failures using social network analysis on developer communication,” in *2009 IEEE 31st International Conference on Software Engineering*, May 2009, pp. 1–11, iSSN: 1558-1225.

[85] Z. Xie and M. Li, “Cutting the software building efforts in continuous integration by semi-supervised online AUC optimization,” in *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, ser. IJCAI’18. Stockholm, Sweden: AAAI Press, Jul. 2018, pp. 2875–2881.

[86] B. Chen, L. Chen, C. Zhang, and X. Peng, “BUILDFAST: History-Aware Build Outcome Prediction for Fast Feedback and Reduced Cost in Continuous Integration,” in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Sep. 2020, pp. 42–53, iSSN: 2643-1572.

[87] I. Saidani, A. Ouni, M. Chouchen, and M. W. Mkaouer, “On the prediction of continuous integration build failures using search-based software engineering,” in *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, ser. GECCO ’20. New York, NY, USA: Association for Computing Machinery, Jul. 2020, pp. 313–314. [Online]. Available: https://dl.acm.org/doi/10.1145/3377929.3390050

[88] “Travis CI.” [Online]. Available: https://travis-ci.org/

[89] “Using the Skip Next Build plugin.” [Online]. Available: https://docs.huihoo.com/jenkins/enterprise/14/user-guide-14.5/skip-sect-using.html

[90] R. Abdalkareem, S. Mujahid, and E. Shihab, “A Machine Learning Approach to Improve the Detection of CI Skip Commits,” *IEEE Transactions on Software Engineering*, vol. 47, no. 12, pp. 2740–2754, Dec. 2021, conference Name: IEEE Transactions on Software Engineering.

[91] R. Abdalkareem, S. Mujahid, E. Shihab, and J. Rilling, “Which Commits Can Be CI Skipped?” *IEEE Transactions on Software Engineering*, vol. 47, no. 3, pp. 448–463, Mar. 2021, conference Name: IEEE Transactions on Software Engineering.

[92] X. Jin and F. Servant, "A cost-efficient approach to building in continuous integration," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, Oct. 2020, pp. 13–25. [Online]. Available: https://doi.org/10.1145/3377811.3380437

[93] ——, "Which builds are really safe to skip? Maximizing failure observation for build selection in continuous integration," *Journal of Systems and Software*, vol. 188, p. 111292, Jun. 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121222000486

[94] ——, "HybridCISave: A Combined Build and Test Selection Approach in Continuous Integration," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 4, pp. 93:1–93:39, May 2023. [Online]. Available: https://dl.acm.org/doi/10.1145/3576038

[95] K. Gallaba, J. Ewart, Y. Junqueira, and S. McIntosh, "Accelerating Continuous Integration by Caching Environments and Inferring Dependencies," *IEEE Transactions on Software Engineering*, vol. 48, no. 6, pp. 2040–2052, Jun. 2022, conference Name: IEEE Transactions on Software Engineering.

[96] I. Saidani, A. Ouni, and M. W. Mkaouer, "Detecting Continuous Integration Skip Commits Using Multi-Objective Evolutionary Search," *IEEE Transactions on Software Engineering*, vol. 48, no. 12, pp. 4873–4891, Dec. 2022, conference Name: IEEE Transactions on Software Engineering.

[97] C. Cho, B. Chun, and J. Seo, "Adaptive batching scheme for real-time data transfers in iot environment," in *Proceedings of the 2017 International Conference on Cloud and Big Data Computing*, 2017, pp. 55–59.

[98] F. Chang, J. Ren, and R. Viswanathan, "Optimal resource allocation for batch testing," in *2009 International Conference on Software Testing Verification and Validation*, 2009, pp. 91–100.

[99] "git-bisect manual page," 2015. [Online]. Available: https://git-scm.com/docs/git-bisect

[100] M. J. Beheshtian, A. Bavand, and P. Rigby, “Software Batch Testing to Save Build Test Resources and to Reduce Feedback Time,” *IEEE Transactions on Software Engineering*, vol. 48, no. 8, pp. 1–1, 2021, conference Name: IEEE Transactions on Software Engineering.

[101] K. Herzig, J. Czerwonka, B. Murphy, and M. Greiler, “Selecting tests for execution on a software product,” US Patent US20 160 321 586A1, Nov., 2016. [Online]. Available: https://patents.google.com/patent/US20160321586A1/en

[102] 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, Apr. 2007, conference Name: IEEE Transactions on Software Engineering.

[103] B. Jiang, Z. Zhang, W. K. Chan, and T. H. Tse, “Adaptive Random Test Case Prioritization,” in *2009 IEEE/ACM International Conference on Automated Software Engineering*, Nov. 2009, pp. 233–244, iSSN: 1938-4300.

[104] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy, “The Art of Testing Less without Sacrificing Quality,” in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. Florence, Italy: IEEE Press, May 2015, pp. 483–493, iSSN: 1558-1225.

[105] H. Hemmati and F. Sharifi, “Investigating NLP-Based Approaches for Predicting Manual Test Case Failure,” in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, Apr. 2018, pp. 309–319.

[106] J. Liang, S. Elbaum, and G. Rothermel, “Redefining Prioritization: Continuous Prioritization for Continuous Integration,” in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, May 2018, iSSN: 1558-1225.

[107] M. Bagherzadeh, N. Kahani, and L. Briand, “Reinforcement Learning for Test Case Prioritization,” *IEEE Transactions on Software Engineering*, vol. 48, no. 8, pp. 2836–2856, Aug. 2022, conference Name: IEEE Transactions on Software Engineering.

[108] J. Anderson, S. Salem, and H. Do, "Improving the effectiveness of test suite through mining historical data," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: Association for Computing Machinery, May 2014, pp. 142–151. [Online]. Available: https://doi.org/10.1145/2597073.2597084

[109] C. Cho, B. Chun, and J. Seo, "Adaptive Batching Scheme for Real-Time Data Transfers in IoT Environment," in *Proceedings of the 2017 International Conference on Cloud and Big Data Computing*, ser. ICCBDC 2017. New York, NY, USA: Association for Computing Machinery, Sep. 2017, pp. 55–59. [Online]. Available: https://doi.org/10.1145/3141128.3141145

[110] F. Chang, J. Ren, and R. Viswanathan, "Optimal Resource Allocation for Batch Testing," in *2009 International Conference on Software Testing Verification and Validation*, Apr. 2009, pp. 91–100, iSSN: 2159-4848.

[111] "Git - git-bisect Documentation." [Online]. Available: https://git-scm.com/docs/git-bisect

[112] A. Najafi, P. C. Rigby, and W. Shang, "Bisecting commits and modeling commit risk during testing," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. Tallinn, Estonia: Association for Computing Machinery, Aug. 2019, pp. 279–289. [Online]. Available: https://doi.org/10.1145/3338906.3338944

[113] A. Shi, A. Gyori, S. Mahmood, P. Zhao, and D. Marinov, "Evaluating test-suite reduction in real software evolution," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. New York, NY, USA: Association for Computing Machinery, Jul. 2018, pp. 84–94. [Online]. Available: https://doi.org/10.1145/3213846.3213875

[114] "Chrome Analysis Tooling - Findit." [Online]. Available: https://sites.google.com/chromium.org/cat/findit

[115] "Google Cloud Pricing Calculator." [Online]. Available: https://cloud.google.com/products/calculator