Empirical Software Engineering manuscript No.
(will be inserted by the editor)

Contrasting Test Selection, Prioritization, and Batch Testing at Scale

Large-scale Empirical Study on 285 Million Test Results

Emad Fallahzadeh · Peter C. Rigby · Bram Adams

Received: July 31, 2023 / Accepted: November 6, 2024

Abstract The effectiveness of software testing is crucial for successful software releases, and various test optimization techniques aim to enhance this process by reducing the number of test executions or prioritizing potential test failures. Although different families of techniques exist, each with its own evaluation criteria, few studies have compared these different lines of research. This study addresses this gap by empirically comparing Yaraghi et al.'s test prioritization approach, Zhu et al.'s cross-build test prioritization and its equivalent test selection technique, and our *BatchAll* test batching algorithm. To evaluate these test optimization approaches, we empirically analyze millions of test results from Google Chrome, along with pre- and post-commit test outcomes for a Google project, as well as the JMRI Travis CI dataset.

Findings reveal that test selection can reduce actual median feedback time by up to 96% with the same number of machines but may miss up to 55% of failures. In contrast, batching achieves up to a 99% reduction in feedback time without missing any failures. Test selection cuts machine usage by up to 66%, while batching achieves up to an 88%

Grants: Natural Sciences and Engineering Research Council (NSERC) and Concordia University FRS.

Emad Fallahzadeh
School of Computing, Queen's University, Kingston, Ontario, Canada.
E-mail: emad.fallahzadeh@queensu.ca
https://orcid.org/0009-0005-5024-4868

Peter C. Rigby
Department of Computer Science and Software Engineering, Concordia University, Montreal, Quebec, Canada.
E-mail: peter.rigby@concordia.ca
https://orcid.org/0000-0003-1137-4297

Bram Adams
School of Computing, Queen's University, Kingston, Ontario, Canada.
E-mail: bram.adams@queensu.ca
https://orcid.org/0000-0001-7213-4006

reduction. For failure detection, the test selection is up to 62 minutes faster than the baseline, and the batching algorithm achieves up to a 63-minute median improvement without missing failures. Regarding test execution time, test selection saves up to 66%, whereas batching's saving can reach up to 98%, although its performance varies based on the machines used. The studied test prioritization algorithms significantly underperform compared to the test selection and batching algorithms.

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

Keywords Test selection · Test prioritization · Test batching · test optimization · Parallel testing · Chrome testing

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 (CI) 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 (Elbaum et al., 2014; Memon et al., 2017).

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 disperses tests across multiple execution units, aiming to minimize the time required to obtain the results for all tests associated with a change, i.e., feedback time. 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 typically investigated test optimization techniques individually, including test selection, test prioritization, and test batching. However, there is only limited research comparing these techniques, especially batching, which is often overlooked, despite its promise. It remains unclear how these techniques compare in terms of producing faster build feedback times. Does test selection or test batching provide faster feedback? Which technique saves more resources in terms of machine usage? Which one is more effective at detecting failures quickly? And which results in greater savings in test execution?

Moreover, as noted by Greca et al. (2023), the majority of prior studies on regression testing have primarily focused on small datasets. This emphasis has led to the development of solutions that may not be applicable to large-scale projects due to scalability limitations. For instance, although source code information has been extensively utilized for test optimization, its rapid obsolescence and high acquisition costs render it unsuitable for large-scale projects (Elbaum et al., 2014). Nevertheless, it is essential to recognize that test optimization challenges are particularly acute in large-scale contexts.

Another dimension often overlooked in the evaluation of test optimization techniques is parallelization. For instance, it remains unclear how test prioritization and test selection techniques perform when varying the number of machines used. What is the relative performance of different test optimization techniques in terms of feedback time, resource usage, failure detection, and savings in test executions when the number of machines is adjusted? Moreover, flaky tests are often mishandled in test executions, typically by treating flaky failures as regular failures or, less frequently, filtering them out. However, studies demonstrate that the way we handle flaky tests can significantly impact the outcomes of analyses involving test execution results (Peng et al., 2020; Fallahzadeh and Rigby, 2022).

To address these research gaps, this study compares 5 major families of test optimization techniques and evaluates their performance using various evaluation 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 from commit to receiving test results, resource usage, test execution time, and the speed of detecting failing tests.

While different families of test optimization techniques may sometimes be employed together as complementary approaches, comparing them within a parallelization environment provides valuable insights. This comparison helps practitioners understand the unique strengths and limitations of each technique, enabling them to tailor their testing strategies to specific project requirements and resource constraints.

We conduct our experiments using four large-scale datasets, providing us with a more realistic understanding of the testing scale and challenges. These datasets consist of 276 million test results from Google Chrome (Fallahzadeh and Rigby, 2022), 1.1 million test results from Google pre-commit, 1.5 million test results from Google Post-commit (Elbaum et al., 2014), and 6.5 million test results from the JMRI project which is one of the largest projects on Travis CI Yaraghi et al. (2023). Additionally, we take into account the presence of flaky tests for different test optimization algorithms, especially where the flaky labels are available.

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 address our research questions, we conduct comparative simulations using test results from the Chrome, Google pre-commit, Google post-commit, and *JMRI* datasets. Initially, we explore and describe the data to gain a better understanding of the test runs and changes in these projects. Following this, we replicate the state-of-the-art test prioritization technique proposed by Yaraghi et al. (2023). Subsequently, we simulate an optimized version of the test prioritization and its equivalent test selection

technique across builds proposed for a multi-machine environment, as suggested by Zhu et al. (2018).

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 (Fallahzadeh et al., 2023). 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 (Beheshtian et al., 2021; Bavand and Rigby, 2021).

This study makes the following contributions.

- **Comparison of test optimization algorithms:** We compare and contrast test selection, prioritization, and batching approaches 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 Chrome, Google pre-commit, Google post-commit, and *JMRI* large-scale datasets. Additionally, we provide details about these datasets and explore the data distributions for the number of concurrent builds in each of them.
- **Parallelization:** We run the different test optimization algorithms using various numbers of machines to evaluate their performance under different resource availabilities.
- **Flaky tests:** We treat flaky failures as non-blocking failures to simulate the algorithms under realistic conditions.
- **Approaches:** We replicate the state-of-the-art test prioritization technique proposed by Yaraghi et al. (2023). Additionally, we optimize and simulate the test prioritization technique suggested by Zhu et al. (2018) for cross-build and multi-machine environments, along with its test selection equivalent. Moreover, we replicate the *BatchAll* test batching algorithm, which we presented as an effective batching technique in our recent study (Fallahzadeh et al., 2023).
- **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: Test batching and test selection approaches offer the most significant reduction in commit feedback time. However, while the test selection algorithm may miss up to 55% of failures, the test batching approach ensures that no failures are overlooked.
  RQ2: *BatchAll* achieves the best reduction in resources while achieving the same feedback time as the baseline *TestAll* algorithm, which does not apply any test optimizations.
  RQ3: The test selection algorithm can deliver rapid failure detection, but it may overlook up to 55% of failures. Similarly, the test batching algorithm offers com-

parable performance, except in scenarios with few concurrent builds, and importantly, it does not miss any failures. Test prioritization approaches follow thereafter. RQ4: The savings in test execution time remain unchanged for the test selection algorithm regardless of the number of machines in use, while they vary for the batching algorithm depending on the number of machines in use.

The remainder of this work is structured as follows. In Section 2, we review the background and related work. Section 3 presents the design and methodology adopted to conduct this study. In Section 4, we provide explanations for each test optimization approach. Section 5 elaborates on the different evaluation criteria and metrics used in the simulation experiments. In Section 6, we present the results regarding our research questions and simulations. In Section 7, we discuss the threats to the validity of this research. Section 8 discusses the interpretations of the results and implications from this study for practitioners. Finally, in Section 9, we present the conclusion and potential future work.

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 (Poth et al., 2018; Hilton, 2016; Soni, 2015; Leppänen et al., 2015). However, testing each change individually becomes nearly impossible for large companies due to the sheer volume of tests involved (Herzig et al., 2016). Even Google, with its vast resources, struggles to keep up with the high code churn in its code base (Elbaum et al., 2014). 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.

2.1 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 (Li et al., 2007; Jiang et al., 2009; Zhang et al., 2013; Herzig et al., 2015; Henard et al., 2016; Lu et al., 2016; Luo et al., 2016; Hemmati and Sharifi, 2018; Liang et al., 2018; Najafi et al., 2019b; Jahan et al., 2020; Sharif et al., 2021; Bagherzadeh et al., 2022; Parthasarathy et al., 2022). In the past, researchers have primarily concentrated on selecting and prioritizing tests within the same build. The work of Kim and Porter (2002) 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. (2013) 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. (2014) 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 the most frequent failures was just as effective as the ARM model.

Herzig et al. (2015) 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. (2017) 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. (2014) concentrated on optimizing test selection and prioritization for Google's multi-machine and parallel testing environment. They designed test selection and prioritization approaches to apply to both the Google pre-submit and post-submit test data. For determining which tests should be selected and prioritized, they relied on test failure history and did not incorporate code information data in their dataset and their experiment. Additionally, they applied three distinct windows: failure, execution, and prioritization. The application of their algorithms on a sample of 3.5 million tests demonstrated a noteworthy enhancement in test performance.

Liang et al. (2018) 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 the Google Shared dataset and the Rails dataset from Travis CI. The results indicated that the Google dataset had a 12% improvement in $APFD_c$, whereas there were almost no noticeable improvements in Rails.

Zhu et al. (2018) 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. (2014).

Najafi et al. (2019b) 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.

Machine learning techniques have emerged as effective tools for enhancing test case prioritization. For instance, Lachmann et al. (2016) employed a supervised machine learning approach utilizing test case history information to bolster test failure detection. Similarly, Sharif et al. (2021) introduced a novel deep learning model that leverages factors such as test duration and execution status to refine test case prioritization, with evaluations focusing on time-effectiveness and fault detection. In a different vein, Bagherzadeh et al. (2022) applied Reinforcement Learning (RL) in test case prioritization, aiming to enhance the accuracy of identifying test failures promptly. Additionally, Yaraghi et al. (2023) utilized the random forest machine learning technique on TravisTorrent projects, evaluating their model in terms of data collection time and the efficacy of the ML-based TCP technique compared to heuristic-based approaches.

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. (2020) identified the effect of flaky tests on the performance of IR-based test prioritization algorithms on Travis CI projects. Similarly, in our previous work (Fallahzadeh and Rigby, 2022), we analyzed 276 million Chrome test results and found that flaky tests significantly affect history-based test prioritization algorithms.

With a few exceptions, prior research has generally overlooked the influence of flaky tests on test optimization algorithms. Our study addresses this gap by appropriately handling flaky tests, maintaining the assumption that flaky tests are non-blocking, as established in our previous work (Fallahzadeh and Rigby, 2022).

Previous studies have explored individual families of test selection and prioritization algorithms, yet there remains a significant gap in research comparing these families, particularly in multi-machine setups, to discern their relative performance and trade-offs. While our earlier research studied the impact of flaky tests on test prioritization specifically within the Chrome environment (Fallahzadeh and Rigby, 2022), our current endeavor compares the efficacy of various test optimization techniques across diverse dimensions using more extensive datasets. To achieve this goal, we replicate cutting-edge test prioritization, selection, and batching methodologies. Additionally, we employ a range of evaluation metrics, including Feedback time, GainedTime, machine usage, and reductions in test execution time. We also explore the implications of parallelization and the use of different machine configurations in executing these test optimization algorithms, adding another layer of evaluation. Our analysis encompasses four significant large-scale datasets, including Chrome (Fallahzadeh and Rigby, 2022), *GooglePre*, *GooglePost* (Elbaum et al., 2014), and JMRI (Yaraghi et al., 2023).

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 (Herzig et al., 2016; Elbaum et al.,

2014). 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 (Cho et al., 2017; Chang et al., 2009). 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 [1] 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. (2019a) 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. (2021) 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.

Most prior research has primarily concentrated on employing batching and culprit identification at the build level. In contrast, our approach in this study shifts the focus to batching and culprit identification at the test level, a methodology introduced in our recent work (Fallahzadeh et al., 2023). Additionally, prior studies typically utilize a single machine for evaluating test optimization algorithms, while our investigation delves into the impact of parallelism on the large-scale testing framework of the Chrome project.

Our recent research (Fallahzadeh et al., 2023) experimented with the performance analysis of various batching techniques, including the proposal of two new adaptive batching algorithms, alongside considering parallelization on Chrome. However, our current study evaluates the effectiveness of batching in comparison to other cutting-edge test optimization methods, including test selection and prioritization, employing more extensive datasets. To achieve this, we utilize diverse evaluation metrics, encompassing Feedback time, GainedTime, machine usage, and reductions in test execution duration. Furthermore, we incorporate three additional large-scale datasets beyond Chrome, namely *GooglePre* and *GooglePost* (Elbaum et al., 2014), and *JMRI* (Yaraghi et al., 2023).

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 (Jones et al., 2002; Misailovic et al., 2007; Bagies, 2020; Landing et al., 2020).

[1] https://git-scm.com/docs/git-bisect

One of these studies was carried out by Arabnejad et al. (2018), 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 (Shashban, 2022).

Candido et al. (2017) explored how test parallelization affects open-source software projects. They analyzed more than 450 Java projects and discovered that less than 20% of the 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. (2015) 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 et al. (2007) suggested a software behavior-oriented approach to parallelize testing, which aims to minimize conflicting behaviors.

While previous studies have largely overlooked the influence of parallelization on various test optimization algorithms, our recent research (Fallahzadeh et al., 2023) studied the impact of parallelization specifically on different test batching algorithms. However, our current study takes a broader approach by evaluating the ramifications of test parallelization at scale on batching, contrasting it with other test optimization methodologies like test selection and prioritization. We also evaluate parallelism from different dimensions using different evaluation metrics. Our objective is to investigate the relationship between the number of machines employed and the time required to acquire build and failing test outcomes, alongside the reduction in test execution duration. We also expand our analysis by incorporating more extensive datasets beyond Chrome, encompassing *GooglePre*, *GooglePost* (Elbaum et al., 2014), and *JMRI* (Yaraghi et al., 2023).

2.4 Datasets

Different datasets have been evaluated for test optimization. Elbaum et al. (2014) investigated test selection and prioritization using a dataset derived from Google's anonymized test results. This publicly available dataset encompasses 3.5 million pre- and post-commit test suites spanning a 30-day period.

Several studies have used test results from Travis CI, a continuous integration service that hosts many open-source projects, to conduct their evaluations (Bell et al., 2018; Hilton et al., 2018; Mattis et al., 2020; Shi et al., 2018; Labuschagne et al., 2017).

Mattis et al. (2020) 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. Moreover, Yaraghi et al. (2023) used 25 Travis CI projects for their test prioritization study, which they claim to be scalable. However, their criterion for defining a dataset as large-scale is based on the test duration exceeding 5 minutes, which may not be a universally accepted definition.

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 (Memon et al., 2017; Greca et al., 2023). Furthermore, according to Liang et al. (2018), 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 by Beller et al. (2017a).

In contrast, our analysis encompasses a vast dataset of 276 million test results from Google Chrome (Fallahzadeh and Rigby, 2022), along with 1,112,158 tests from the *GooglePre* dataset and 1,495,856 tests from the *GooglePost* dataset (Elbaum et al., 2014). Additionally, we incorporate the largest dataset from the study conducted by Yaraghi et al. (2023), enhancing the generalizability of our work. By incorporating these large-scale projects, we aim to gain deeper insights into the intricate challenges associated with optimizing tests for expansive projects. Moreover, the presence of massive concurrent builds in these datasets necessitates and facilitates the utilization of various test optimization techniques and parallelism.

3 Research Design and Methodology

This section discusses the design and methodology of the empirical study that we perform to address the research questions outlined in the introduction.

3.1 Datasets

In this study, we utilize four large-scale datasets, each comprising millions of test results, to conduct our experiments.

3.1.1 Chrome

The Chrome dataset is one of the largest datasets publicly available, consisting of 276 million Google Chrome test results, as previously published in our work (Fallahzadeh and Rigby, 2022). This dataset represents a large-scale system that executes millions of tests daily, underscoring the importance of optimizing testing for efficient release processes. The system encompasses multiple concurrent builds, offering an opportunity to implement batching techniques to further enhance the testing process.

One notable feature of this dataset is its inclusion of final outcomes for tests, including flaky flags. This provides a significant advantage over other studies, which may only identify flaky tests through thousands of test re-runs, as demonstrated in the research by Alshammari et al. (2021). The flaky test detection approach adopted in Chrome differs from academic methods, as it detects flaky tests through a few test re-runs initiated only after a failure occurs (Fallahzadeh and Rigby, 2022). For further details regarding the testing process and data types in this project, please refer to our prior work (Fallahzadeh and Rigby, 2022).

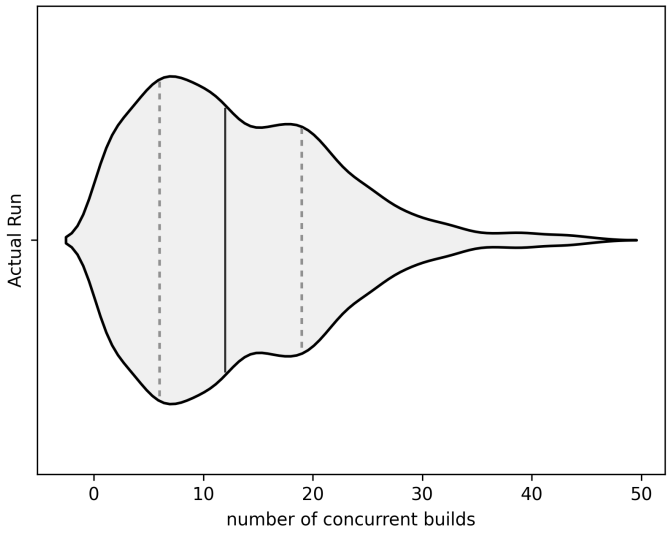

Fig. 1: Violin plot displaying the distribution of the actual number of concurrent builds in Chrome. The median number of concurrent builds is denoted by a solid vertical line, while the dashed vertical lines indicate the first and third quartiles of the distribution.

The Chrome dataset utilized in this study encompasses the change, build, and test results for Chrome from January 1, 2021, to January 31, 2021. During this timeframe, it includes the results for 9,524 change lists (commits), 19,045 builds, 49,932 test suites, and 276,550,812 test cases. On average, each test suite contains approximately 5,538 test cases. The median feedback time for the builds in this dataset is 32.95 minutes. Feedback time is the duration between a change being committed and the execution of all associated tests, measured across all builds during the studied period.

Figure 1 depicts the distribution of the number of concurrent builds in Chrome. This data is derived by comparing the start and end times of each build with those of other builds. In the actual Chrome build execution, each build has a concurrency with up to 47 other builds, with a median of 12 concurrent builds and the first and third quartiles at 6 and 19 concurrent builds, respectively. These concurrent builds offer an excellent opportunity to implement batching algorithms, facilitating the grouping and processing of builds together.

3.1.2 GooglePre Dataset

The *GooglePre* dataset encompasses pre-commit test suites data publicly accessible through Elbaum et al. (2014), encapsulating a 30-day period of test executions for a typical Google product. It encompasses anonymized test suite names, change requests,

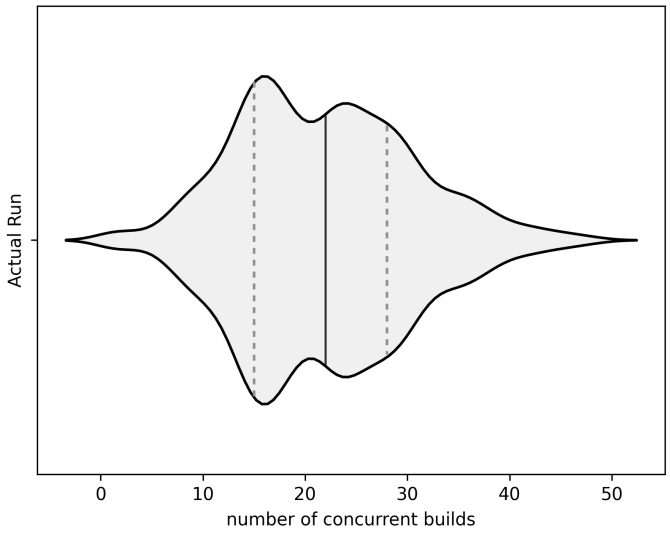

Fig. 2: Violin plot showing the distribution of the actual number of concurrent builds in *GooglePre*. The median number of concurrent builds is denoted by a solid vertical line, while the dashed vertical lines indicate the first and third quartiles of the distribution.

verdicts, start times, and execution durations. Our experimental setup replicates the dataset methodology employed in previous research, including studies by Elbaum et al. (2014) and Liang et al. (2018) as follows.

Given the structure of the dataset, test suites are distributed across various shards, which are subsets of test suites. To address this fragmentation, we consolidate test suites with identical names but distributed across different shard numbers into unified test suites. This approach is the same as the methodology utilized by Liang et al. (2018). Following this consolidation process, the *GooglePre* dataset comprises 1693 change lists and 1,112,158 test suites, with an average of approximately 656 test suites per change list. The median feedback time per build within the *GooglePre* dataset is recorded at 1.6 minutes.

Figure 2 portrays the distribution of concurrent builds within the *GooglePre* dataset. Each build exhibits concurrency with up to 49 other builds, showcasing a median concurrency of 22 builds, with first and third quartiles at 15 and 28, respectively. This dataset's abundance of concurrent builds offers an extensive canvas for exploring diverse test optimization techniques, including parallelization, batching, selection, and prioritization.

3.1.3 GooglePost Dataset

The *GooglePost* dataset entails post-commit test suites data publicly disseminated by Elbaum et al. (2014). Analogous to the *GooglePre* dataset, it encapsulates a 30-day timeframe of test executions for a representative Google product, including anonymized test suite names, change requests, verdicts, start times, and execution durations. Our experimental approach aligns with previous dataset methodologies, such as those outlined by Elbaum et al. (2014) and Liang et al. (2018) as follows.

Similar to its precursor, the *GooglePost* dataset exhibits fragmentation among test suites across different shards. To mitigate this fragmentation, we merge test suites with identical names but varying shards into single entities, adopting the methodology used by Liang et al. (2018). Following this consolidation process, the *GooglePost* dataset comprises 4520 change lists and 1,495,856 test suites, averaging around 330 test suites per change list. The median feedback time per build within the *GooglePost* dataset is documented at 4.6 minutes.

Figures 3 illustrate the distribution of concurrent builds within the *GooglePost* dataset. Each build demonstrates concurrency with up to 86 other builds, with a median concurrency of 50 builds and first and third quartiles at 35 and 62, respectively. The wealth of concurrent builds within this dataset offers a fertile ground for investigating various test optimization techniques, including parallelization, batching, selection, and prioritization.

3.1.4 JMRI

The JMRI dataset is sourced from the study conducted by Yaraghi et al. (2023), focusing on scalable test prioritization. They examined a total of 25 projects, including 18 of the largest ones based on Source Lines of Code (SLOC) and build counts sourced from TravisTorrent (Beller et al., 2017b). Additionally, they included 7 major projects with a minimum of 5 minutes of average testing time from the RTPTorrent dataset (Mattis et al., 2020). In our experiment, we utilize the JMRI dataset, which represents the largest dataset within their study.

The JMRI dataset covers a span of 5 months of test executions, encompassing 69,300 commits, 1,481 builds, and a total of 6,469,640 test cases. This averages to approximately 4,368 test cases per build, with a median feedback time of 26.4 minutes for each build.

Figure 4 illustrates the distribution of concurrent builds in the JMRI dataset. This information is obtained by comparing the start and end times of each build with those of others. In actual JMRI build executions, each build has a concurrency with up to 4 other builds, with a median of 1 concurrent builds and the first and third quartiles at 0 and 1 concurrent build, respectively.

4 Test Optimization Approaches

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

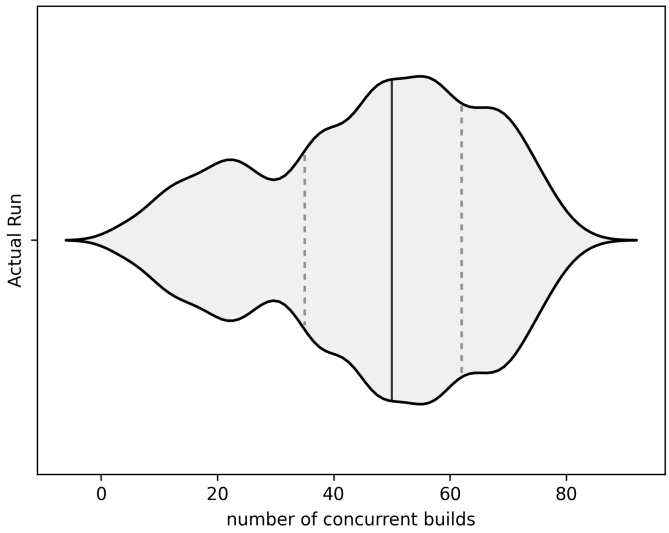

Fig. 3: Violin plot showing the distribution of the actual number of concurrent builds in *GooglePost*. The median number of concurrent builds is denoted by a solid vertical line, while the dashed vertical lines indicate the first and third quartiles of the distribution.

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.

4.2 *MLPrioritization*

One of the test optimization techniques widely used to improve failing test detection is test prioritization. This approach reorders tests within a build to detect failing tests faster. In this study, we compare the state-of-the-art test prioritization algorithm devised by Yaraghi et al. (2023) with other test optimization approaches to assess its effectiveness.

We replicated the approach outlined by Yaraghi et al. (2023) and applied features that were both available and feasible across our various datasets under study. These features were recognized as the most effective ones in their study and are aligned with their recommendations for large-scale projects. The features we incorporated include the test's Age, LastFailAge, LastTransitionAge, AvgExeTime, MaxExeTime, FailRate,

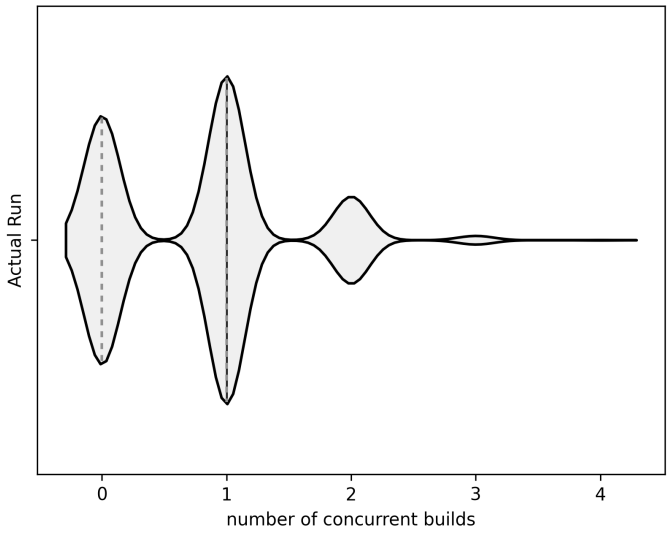

Fig. 4: Violin plot showing the distribution of the actual number of concurrent builds in JMRI. The median number of concurrent builds is denoted by a solid vertical line, while the dashed vertical lines indicate the first and third quartiles of the distribution.

ExcRate, TransitionRate, LastVerdict, and LastExeTime. For detailed explanations of these features, please refer to the main paper (Yaraghi et al., 2023).

In this study, we implement the approach proposed by Yaraghi et al. (2023) using the aforementioned features. We refer to this implementation as *MLPrioritization* in this paper, and it is based on the pseudocode shown in Algorithm 1. We use the same parameters and settings for executing the algorithm.

4.3 *CoDynaQPrioritization*

Elbaum et al. (2014) introduced the concept of utilizing previous failures to predict and prioritize future ones in a Continuous Integration (CI) environment across multiple builds. This approach reorders tests across queued builds to maximize the speed of failure detection. A more recent study by Zhu et al. (2018) extended this concept by incorporating co-failures, which are tests that fail in the same build. Zhu *et al.*'s test prioritization approach adjusts and reprioritizes test cases based on observed failures or non-failures. For example, if a test previously co-failed with another test, the failure of one in the current build increases the prioritization score of the other, based on the conditional probability of their co-failure.

In this study, we optimize and adopt Zhu *et al.*'s approach as a representative of test prioritization algorithms designed for CI systems across multiple builds. The pseudo-code for this method is shown in Algorithm 2, and we refer to this implementation as

Algorithm 1: *MLPrioritization*

```
Input  : Build queue Q_b
Output : Prioritized test queue Q_t
:
while Q_b is not empty do
    b ← next build from Q_b;
    Q_t ← tests from b;
    for t ∈ Q_t do
        Calculate ML features for t;
        Input the calculated ML features for t to the ML model to get P_k(t) as the failure
          probability of t;
    end
    Prioritize Q_t based on P_k(t);
    while Q_t is not empty do
        t ← 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
```

CoDynaQPrioritization throughout the study. The conditional probability $P(t_o|t_{fail})$ in this algorithm represents the probability of test t_o failing given that test t_{fail} has already failed in the corresponding build. This can be calculated using Formula 1. The $P(t_o \cap t_{\text{fail}})$ in Formula 1 is the probability that both tests t_o and t_{fail} fail together, and $P(t_{\text{fail}})$ represents the probability that test t_{fail} fails.

$$P(t_o|t_{\text{fail}}) = \frac{P(t_o \cap t_{\text{fail}})}{P(t_{\text{fail}})} \tag{1}$$

We utilize three time windows as defined by Elbaum et al. (2014): the failure window (w_f), execution window (w_e), and prioritization window (w_p). The failure window determines the timeframe used to consider previous failures. The execution window prevents low-priority tests from waiting too long, thereby avoiding starvation. Specifically, we set w_p to 2 builds, equivalent to the time window used in the main study by Elbaum et al. (2014). Additionally, we adopt an unlimited window size for w_f and a window size of 2 for w_e, to maximize the failure detection and optimize test executions.

We compare the performance of the *CoDynaQPrioritization* algorithm with other test optimization techniques. Notably, this algorithm relies solely on historical test results and does not require resource-intensive data such as source code information, which can be impractical for large-scale projects (Elbaum et al., 2014; Memon et al., 2017).

Algorithm 2: *CoDynaQPrioritization*

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 ← next builds from Q_b based on w_p;
    Q_t ← tests from b;
    for t ∈ Q_t do
        if t has recent failures (t ∈ w_f), is new, or has not been executed for a long time
          (t ∉ 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 ← next test from Q_t;
        Dispatch t to available machine;
        Execute t and record the results;
        Q_o ← other tests from Q_t;
        while Q_o is not empty do
            t_o ← next test from Q_o;
            if t fails then
                add P(t_o|t_fail) to the t_o score;
                Update failure history for t;
            end
        end
        Re-Prioritize Q_t based on new scores;
    end
end
```

4.4 *CoDynaQSelection*

To ensure a fairer comparison by applying consistent approaches and criteria between test prioritization and selection in continuous integration (CI) and across builds, we apply the same method used for *CoDynaQPrioritization* as an optimized version of Zhu et al. (2018) for test selection. This algorithm leverages the concept of previous failures and co-failures for test selection across builds, utilizing lightweight test result history information while avoiding the use of resource-intensive source code data, which may not be feasible for large-scale projects Memon et al. (2017).

Test selection based on the co-failures approach involves adjusting and reselecting test cases based on observed failures or non-failures. For instance, if a test co-failed with another test previously, the failure of one in the current build will increase the selection score of the other one, determined by the conditional probability of their co-failure. In this experiment, we implement the co-failure selection algorithm as described in the pseudocode shown in Algorithm 3. We refer to this implementation

as *CoDynaQSelection* in the remainder of this study. The conditional probability $P(t_o|t_{fail})$ in this algorithm is calculated based on Formula 1.

We utilize two time windows as defined by Elbaum et al. (2014): the failure window (w_f), and execution window (w_e). The failure window determines the timeframe used to consider previous failures. The execution window prevents low-priority tests from waiting too long, thereby avoiding starvation. Specifically, we set an unlimited window size for w_f and a window size of 2 for w_e, to maximize the failure detection and optimize test executions.

Algorithm 3: *CoDynaQSelection*

```
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 ← next builds from Q_b based on w_p;
    Q_t ← tests from b;
    for t ∈ Q_t do
        if t has recent failures (t ∈ w_f), is new, or has not been executed for a long time
          (t ∉ 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 ← next test from Q_t;
        Dispatch t to available machine;
        Execute t and record the results;
        Q_o ← other tests from Q_t;
        while Q_o is not empty do
            t_o ← next test from Q_o;
            if t fails then
                add P(t_o|t_fail) to the t_o score;
                Update failure history for t;
            end
        end
        Re-Select from Q_t based on new scores;
    end
end
```

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 (Beheshtian et al., 2021).

In this study, we utilize the *BatchAll* algorithm, which demonstrated promising results in our recent work (Fallahzadeh et al., 2023) comparing various batching techniques. Unlike waiting to receive a specific number of changes (builds) (Beheshtian et al., 2021), *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: Given that the datasets do not provide batch-level test results, we rely on the following assumptions in our simulations to extrapolate batch outcomes from individual build results:

1. We assume that if a test fails in any of the builds within a batch, it will cause the entire batch to fail. This test will also trigger the same number of re-runs enforced by the CI infrastructure to mitigate flakiness, just as it did when it ran in the individual build. Such outcomes are classified as batch breakages, necessitating the identification of the root cause through the culprit-finding process.
2. In the event of a test displaying a flaky result in any of the builds included in the batch, we assume that it will produce a flaky result in the batch as well. It undergoes the same number of re-runs in the batch as would be applied to the specific build included to identify the flaky test. However, in our simulations, flaky tests do not lead to the breakage of the batch and result in the integration of the batch.
3. Other types of tests in the batch are considered as pass, resulting in the integration of the batch.

Figure 5 shows the summary of the decision tree for each type of test result in the builds included in the batch. We define the pseudocode for the *BatchAll* algorithm as the pseudocode 4.

5 Evaluation Criteria

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

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

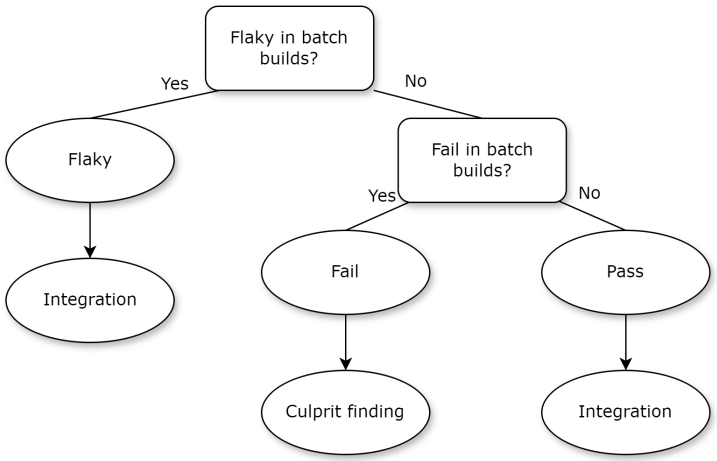

Fig. 5: Decision tree for each type of test results in the builds included in the batch.

Algorithm 4: *BatchAll*

Input : Build queue Q_b
Output Batched test execution
:

while Q_b *is not empty* **do**
  $batch \leftarrow$ make a batch from available and arrived builds in Q_b;
  $testSet \leftarrow$ set of all unique tests across $batch$;
  $Q_t \leftarrow$ fill test queue with tests from $testSet$;
  while Q_t *is not empty* **do**
    $t \leftarrow$ 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 commit* c *in* $batch$ **do**
        Execute t_f in c and record the result;
        if t_f *fails* **then**
          Update failure history for t_f;
        end
      end
    end
  end
end

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:

$$\text{FeedbackTime} = \text{Time(TestResults)} - \text{Time(Commit)} \quad (2)$$

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 for a given set of resources.

5.2 Number of Machines in Use

One significant aspect of this study is its examination of resource optimization concerning the number of build machines employed. We aim to determine how many machines we can save while achieving the same actual feedback time by employing different test optimization algorithms.

Baseline. To establish a benchmark, we consider the baseline number of machines utilized to achieve the actual feedback time in the reality of the projects under study. Since the actual number of machines used to achieve the feedback time for various projects is unavailable, we estimate this figure using the *TestAll* algorithm, which does not employ any test optimization techniques. By determining the number of machines *TestAll* requires to attain a feedback time similar to the actual value, we establish this count as the baseline for each dataset. This baseline serves as a reference point for evaluating the performance of different test optimization algorithms in terms of resource reduction.

For example, in the case of Chrome, the *TestAll* configuration with 179 machines closely approximates the actual median feedback time taken from historical test execution times and serves as our baseline. Similarly, for *GooglePre*, the *TestAll* configuration with 46 machines is chosen as the baseline due to its close alignment with the actual median feedback time. Likewise, for *GooglePost* and *JMRI*, our baselines are the *TestAll* configurations with 103 machines and 1 machine, respectively. These specific baselines provide a standardized basis for assessing the performance of other test optimization algorithms in this study.

5.3 GainedTime

Another essential aspect of evaluating test optimization algorithms is their efficiency in promptly detecting failing tests.

The Average Percentage of Faults Detected (APFD) metric is commonly used to assess failure detection within individual builds. It measures the average cumulative percentage of faults detected during the execution of test cases within a build when run in a given order (Rothermel et al., 1999). However, our study focuses on test optimization in a multi-machine, parallel environment and across multiple builds, rather than running builds individually and sequentially. Therefore, we need to evaluate failure detection performance across multiple builds. This involves determining when failures are detected across several builds rather than within a single build.

For example, consider a scenario where algorithm A identifies a failure as the first test in the 1000th build, while algorithm B identifies it as the 100th test in the same build. The timeframe for detecting this failure can vary significantly between scenarios. In one scenario, it might take 30 days to detect the failure due to the postponement of running tests and the inability of the algorithm to clear the queue efficiently. In another scenario, it might take only two days due to faster processing of the build queue.

Elbaum *et al.*, pioneers in utilizing the APFD metric for test prioritization, identified its limitations in measuring fail time across builds in large-scale continuous integration environments (Elbaum et al., 2014). Further insights on this matter are elaborated in the discussion section.

To address this challenge, Elbaum et al. (2014) and our previous work (Fallahzadeh and Rigby, 2022) introduced an alternative metric known as GainedHours. We extend this metric to GainedTime to accommodate various units such as hours or minutes. This metric quantifies the time saved by each approach (A) compared to the baseline (*TestAll*), and is defined as:

$$\text{GainedTime (A)} = \text{FailTime}(\textit{TestAll}) - \text{FailTime(A)} \tag{3}$$

Here, FailTime represents the time at which the corresponding algorithm detects the failing test. By comparing each algorithm's run times against *TestAll*, which processes tests in their arrival order, we calculate the time saved for each failed test. Subsequently, we use the median GainedTime to evaluate each algorithm's performance using a specific number of machines against other algorithms.

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 4.

$$\text{ExecutionReduction}_m(A) = 1 - \frac{\sum_{t=1}^{T} \text{Test Execution Time (A, m)}}{\sum_{t=1}^{N} \text{Test Execution Time } (\textit{TestAll})} \tag{4}$$

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.

5.5 Experimental Setup

5.5.1 Simulated Hardware Environment

In this study, we simulate the impact of varying numbers of machines on different test optimization algorithms to assess parallelism. We allocate test executions algorithmically across these machines to ensure the accuracy of our results. For Chrome, we span a range of machines from 1 to 400, with a standard step size of 25. However, in cases where we observe significant differences, such as from 1 to 25 machines, we employ more precise steps to thoroughly examine the outcomes. For the *GooglePre* dataset, our range extends from 1 to 100 machines, with a step size of 10. Similarly, we explore the range from 1 to 400 machines for the *GooglePost* dataset, using a step size of 25. In contrast, for the *JMRI* dataset, we explore the range from 1 to 19 machines with a step size of 2. The selection of these ranges and step sizes is meticulous, driven by the evaluation results. We choose ranges and steps where they make a meaningful difference in outcomes and where they plateau, ensuring a comprehensive analysis of the performance of various test optimization algorithms under diverse machine configurations.

5.5.2 Dealing with Test Results in Chrome

To conduct our simulation, we establish a set of assumptions regarding the treatment of various types of test results in Chrome, drawing from our previous studies (Fallahzadeh and Rigby, 2022; Fallahzadeh et al., 2023).

- **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.

6 Results

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

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 parallel environment with varying numbers of machines on different datasets. 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. One of the unique aspects of our study is the consideration of a multi-machine environment, allowing for a more realistic and comprehensive evaluation of test optimization strategies in modern CI environments where builds are typically distributed across several machines to improve efficiency and scalability.

To address this research question, we employ the *TestAll* algorithm as our baseline, alongside *MLPrioritization*, *CoDynaQPrioritization*, *CoDynaQSelection*, and *BatchAll* algorithms, analyzing their performance on various datasets. We vary the number of machines used to execute these algorithms and record the median feedback times across all commits for each algorithm and machine configuration.

We present the results of the median feedback time for the executions on Chrome in Figure 6 and 7. The results are shown as a continuous line by connecting the dots to enable better visualization of the outcomes. Figure 6 shows the median feedback time for each available number of machines and each test optimization algorithm in Chrome. Since the difference in feedback results is substantial when using different numbers of machines, we use a logarithmic scale to display the results in this figure.

The results depicted in Figure 6 indicate that the *BatchAll* algorithm outperforms other approaches, particularly in resource-constrained environments with substantially fewer machines than the baseline. However, when the number of machines exceeds approximately 70, the *CoDynaQSelection* algorithm demonstrates superior performance. Thus, in resource-abundant environments, *CoDynaQSelection* surpasses the other algorithms in effectiveness, though it does so at the cost of missing 31.25% of failures regardless of the number of machines used. The percentage of missed failures is calculated based on the total number of failures in the Chrome dataset and the number of failures detected by the *CoDynaQSelection* algorithm. Both the *MLPrioritization* and *CoDynaQPrioritization* algorithms exhibit performance comparable to the *TestAll* algorithm.

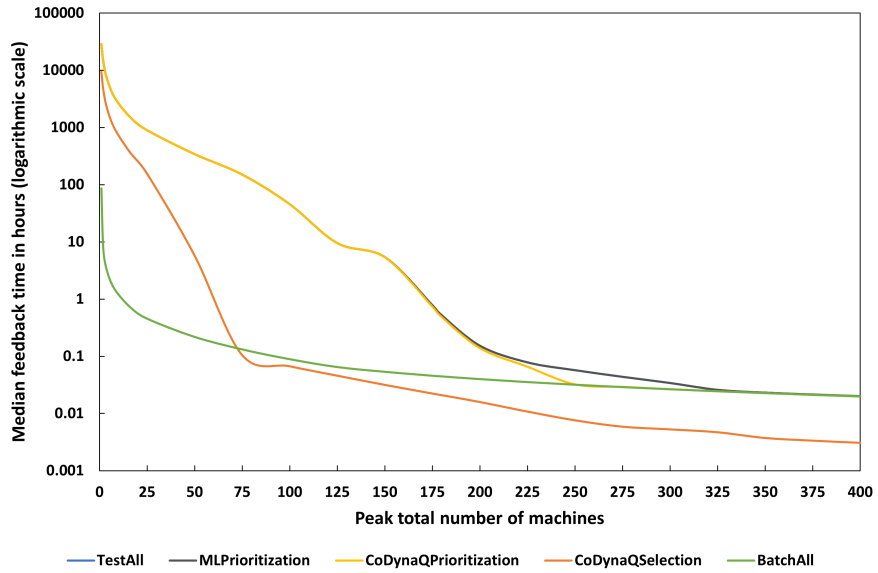

Fig. 6: Median feedback time for the *TestAll*, *CoDynaQSelection*, *CoDynaQPrioritization*, *MLPrioritization*, and *BatchAll* algorithms, displayed in logarithmic scale and expressed in hours, taking into account variations in peak machine numbers for Chrome.

In this research question, our primary focus is on the median feedback time and the improvements achieved by different algorithms using the actual baseline number of machines, which is 179 in the case of Chrome. While Figure 6 provides a holistic view of median feedback time performance, it may not adequately display the feedback time values and the improvements achieved by different approaches around the baseline actual feedback time due to its logarithmic scale. Therefore, we use Figure 7 for a clearer view. This figure presents the median feedback time on the left axis and the improvements relative to the actual median feedback time, showing deviations up to 100 percent, on the right axis.

As shown in Figure 7 for Chrome, with a baseline of 179 machines, the *BatchAll* algorithm achieves a median feedback time of 2.69 minutes. This represents a 91.84% improvement over the actual feedback time, while ensuring no failures are missed, as *BatchAll* executes all tests for the batch build. *CoDynaQSelection* results in a lower median feedback time of 1.28 minutes, showing a significant improvement of 96.12%, albeit missing 31.25% of failures because it avoids running all the tests. Neither the *TestAll* nor the *MLPrioritization* algorithms yield any improvements in feedback time, as expected. The median feedback time for *CoDynaQPrioritization* is 31.20 minutes, representing approximately a 5.30% improvement compared to the actual feedback time of 32.95 minutes.

Each data point in Figures 6 and 7 represents the median of the feedback time distribution across all commits in the Chrome dataset. To evaluate and compare the

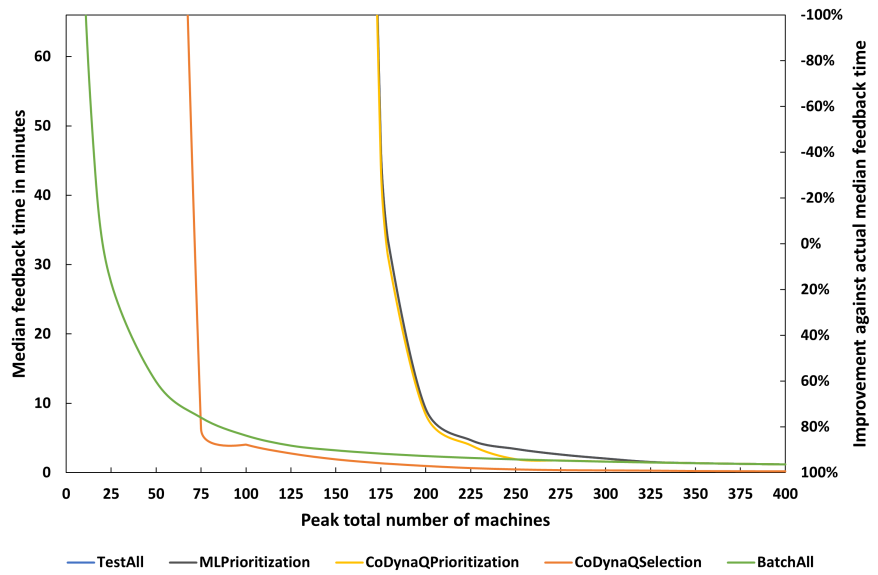

Fig. 7: Median feedback time in minutes and the improvement relative to the baseline median feedback time of 32.95 minutes for the *TestAll*, *CoDynaQSelection*, *CoDynaQPrioritization*, *MLPrioritization*, and *BatchAll* approaches, factoring in variations in peak machine numbers for Chrome.

feedback time distributions of various test optimization algorithms, 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 address 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. Consequently, each individual comparison is subject to a strict significance threshold, resulting in a p-value cutoff of 0.00024 for Chrome. This adjustment ensures that any observed differences between the algorithms are statistically significant.

To maintain content flow, we present the resulting table in Appendix A. Table 7 presents the calculated p-values for comparing the feedback time distribution between different test optimization algorithms for the Chrome dataset. These results are consistent with the observations of the median feedback time depicted in Figures 6 and 7. For instance, the findings reveal that there are no statistically significant differences in the distribution between *TestAll* and *MLPrioritization*, *TestAll* and *CoDynaQPrioritization* when the number of machines is below 175, as well as between *MLPrioritization* and *CoDynaQPrioritization* within the same machine range. However, for the remaining comparisons, the p-values indicate statistically significant differences, meeting the predetermined cutoff of 0.00024.

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 8. This additional data reinforces the observed patterns and underscores the significance of the differences depicted in the median feedback time plot.

For *GooglePre*, Figures 8 and 9 show the corresponding median feedback time results. Figure 8 depicts the median feedback time in logarithmic scale for each available number of machines and test optimization algorithm on *GooglePre*.

The results illustrated in Figure 8 indicate that the *BatchAll* algorithm consistently outperforms other algorithms in terms of feedback time on the *GooglePre* dataset, particularly in resource-constrained environments with fewer machines. Subsequently, *CoDynaQSelection* achieves the second-best median feedback time in most cases; however, it misses 6.24% of failures regardless of the number of machines used. This failure miss rate is due to *CoDynaQSelection* selectively executing tests, which leads to not detecting some failures. Both the *MLPrioritization* and *CoDynaQPrioritization* algorithms perform comparably to the *TestAll* algorithm, although *CoDynaQPrioritization* outperforms *TestAll* when the number of machines exceeds approximately 46, the baseline number.

Figure 9 showcases the improvements realized by each test optimization algorithm relative to the actual median feedback time of 15.21 minutes for *GooglePre*.

In the case of *GooglePre*, as illustrated in Figure 9, using 46 baseline machines, *BatchAll* achieves a median feedback time of 0.04 minutes, marking a significant improvement of 99.75% against the actual feedback time without missing any failures. *CoDynaQSelection* produces a longer median feedback time of 2.92 minutes, representing approximately an 80.83% improvement, while also missing 6.24% of failures due to executing tests selectively. Both *TestAll* and *MLPrioritization* do not result in any improvements, as expected. The *CoDynaQPrioritization* algorithm yields a median feedback time of 14.91 minutes, representing about a 1.94% improvement against the actual median feedback time of 15.21 minutes.

The Wilcoxon Rank-Sum test, conducted on the *GooglePre* dataset to compare the feedback time distribution across various test optimization approaches and commits, is presented in Table 9 in Appendix A. Employing the Bonferroni correction yields a p-value cutoff of 0.00045 for *GooglePre*. The findings of Table 9 closely align with the observations derived from the median feedback time depicted in feedback time figures 8 and 9. Additionally, the effect size measure, Cliff's delta, displayed in Table 10, reinforces the observed patterns and underscores the significance of the differences illustrated in the median feedback time plot.

For *GooglePost*, Figures 10 and 11 present the respective findings regarding the median feedback times. Figure 10 outlines the median feedback time in logarithmic scale concerning each available number of machines and test optimization algorithm on *GooglePre*.

The outcomes depicted in Figure 10 demonstrate that the *BatchAll* algorithm consistently outperforms other algorithms in terms of feedback time on the *GooglePost* dataset, with its superiority more pronounced in environments with limited resources and fewer machines. Following *BatchAll*, *CoDynaQSelection* achieves the second-best median feedback time in most instances, although it misses 0.86% of failures

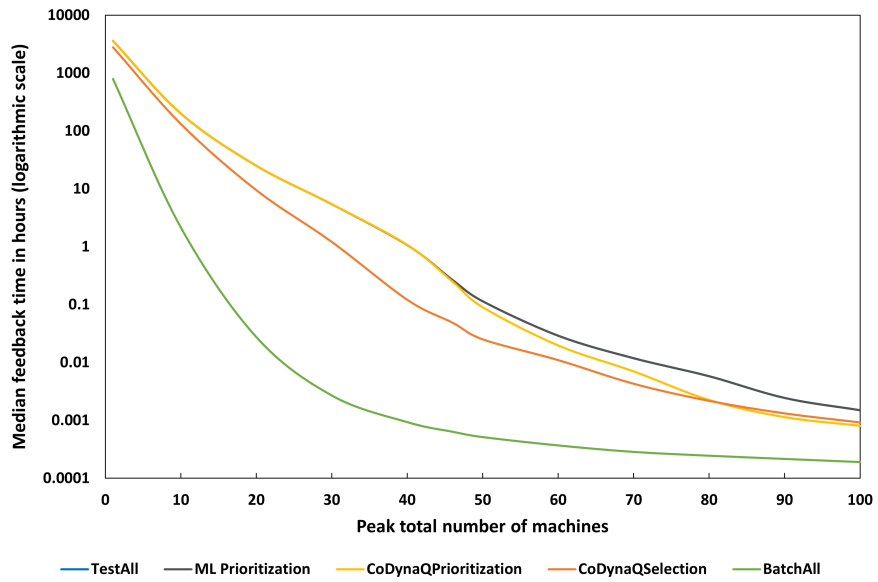

Fig. 8: Median feedback time for the *TestAll*, *CoDynaQSelection*, *CoDynaQPrioritization*, *MLPrioritization*, and *BatchAll* algorithms, displayed in logarithmic scale and expressed in hours, taking into account variations in peak machine numbers for *GooglePre*.

regardless of the number of machines used. Both the *MLPrioritization* and *CoDynaQPrioritization* algorithms perform comparably to the *TestAll* algorithm. However, *CoDynaQPrioritization* surpasses *TestAll* when the number of machines exceeds approximately 103, the baseline number.

For *GooglePost*, as depicted in Figure 11, with a baseline of 103 machines, the *BatchAll* algorithm showcases a median feedback time of 1.88 minutes, demonstrating a 60.21% improvement against the actual feedback time without missing any failures. *CoDynaQSelection* yields a median feedback time of 3.07 minutes, representing a 34.94% improvement, while missing 0.86% of failures. Both *TestAll* and *MLPrioritization* do not result in any improvements, as expected. The *CoDynaQPrioritization* algorithm results in a median feedback time of 4.24 minutes, marking a 10.25% improvement against the actual feedback time of 4.72 minutes.

The Wilcoxon Rank-Sum tests presented in Table 11 in Appendix A, considering the Bonferroni correction cutoff of 0.00028 for *GooglePre*, and Cliff's delta depicted in Table 12, closely align with the observations derived from the median feedback time depicted in Figures 10 and 11.

For *JMRI*, Figure 12 presents the findings on median feedback times, along with corresponding improvements relative to the actual feedback time of 26.39 minutes. Since there are no substantial variations in the feedback time results, we consolidate this information into the single 12 figure.

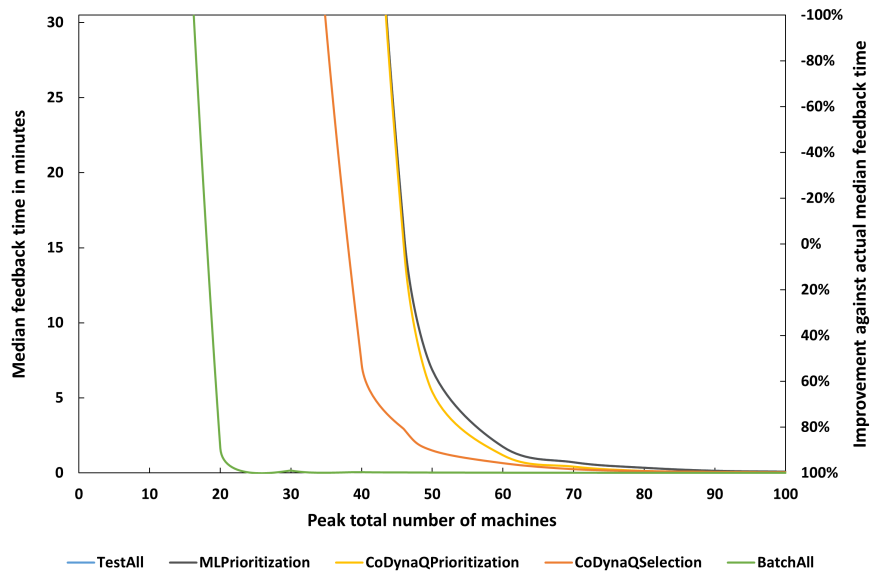

Fig. 9: Median feedback time in minutes and the improvement relative to the baseline median feedback time of 15.21 minutes for the *TestAll*, *CoDynaQSelection*, *CoDynaQPrioritization*, *MLPrioritization*, and *BatchAll* approaches, factoring in variations in peak machine numbers for *GooglePre*.

The outcomes depicted in Figure 12 demonstrate that the *CoDynaQSelection* algorithm consistently outperforms other algorithms in terms of feedback time on the *JMRI* dataset, particularly in environments with limited resources and fewer machines. However, this advantage comes at a cost, as it misses a substantial 55.27% of failures regardless of the number of machines used, since it selectively executes tests, which leads to missed failures. The *MLPrioritization* algorithm performs identically to the *TestAll* algorithm. In contrast, the *CoDynaQPrioritization* algorithm outperforms *TestAll* when using the baseline number of machines of 1. However, as the number of machines increases, its feedback time overlaps with *BatchAll* and eventually converges with *TestAll*. While the *BatchAll* algorithm initially outperforms the baseline *TestAll* algorithm in resource-constrained environments with fewer machines, its feedback time becomes comparable to the baseline as the number of machines increases.

For the *JMRI* dataset, employing a single baseline machine, *CoDynaQSelection* achieves a median feedback time of 7.85 minutes, marking a significant 70.27% improvement compared to the actual feedback time of 26.39 minutes, despite missing 55.27% of failures. While other algorithms failed to surpass the actual median feedback time, some managed to achieve improvements relative to the *TestAll* baseline algorithm. Specifically, *CoDynaQPrioritization* demonstrates an approximately 8% enhancement over the baseline algorithms, while the *BatchAll* algorithm outperforms the baseline by approximately 15

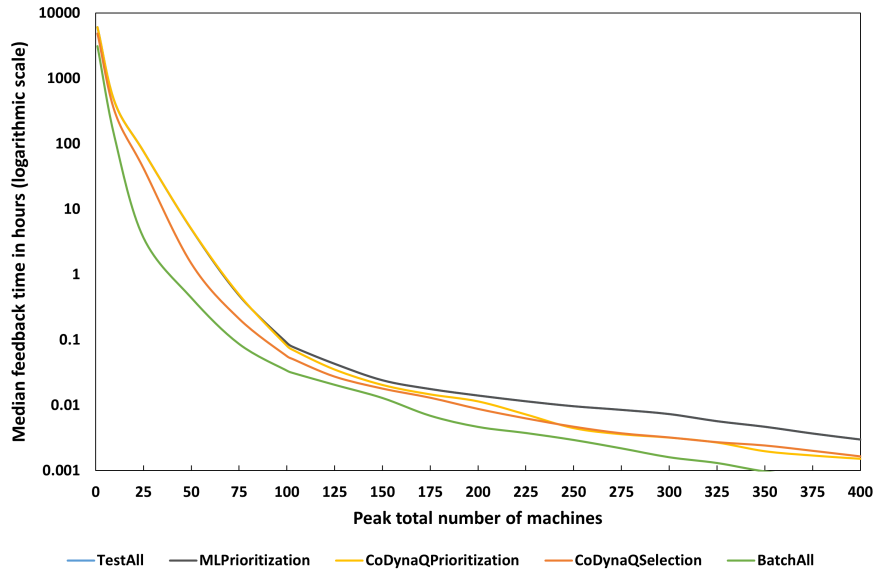

Fig. 10: Median feedback time for the *TestAll*, *CoDynaQSelection*, *CoDynaQPrioritization*, *MLPrioritization*, and *BatchAll* algorithms, displayed in logarithmic scale and expressed in hours, taking into account variations in peak machine numbers for *GooglePost*.

Table 1: The median feedback time in minutes and the corresponding improvements achieved by different algorithms for various datasets using the baseline number of machines: 179 for Chrome, 46 for *GooglePre*, 103 for *GooglePost*, and 1 for JMRI datasets.

Datasets	**TestAll**	**CoDynaQ Selection**	**CoDynaQ Prioritization**	**MLPrioritization**	**BatchAll**
Chrome	33.55 (-1.81%)	1.28 (96.12%)	31.20 (5.30%)	33.55 (-1.81%)	2.69 (91.84%)
GooglePre	15.96 (-4.91%)	2.92 (80.83%)	14.91 (1.94%)	15.96 (-4.91%)	0.04 (99.75%)
GooglePost	4.72 (0.01%)	3.07 (34.94%)	4.24 (10.25%)	4.72 (0.01%)	1.88 (60.21%)
JMRI	30.87 (-16.99%)	7.85 (70.27%)	28.81 (-9.17%)	30.87 (-16.99%)	26.84 (-1.71%)

The Wilcoxon Rank-Sum test, considering the Bonferroni p-value threshold of 0.0005 for *JMRI*, closely corresponds with the insights gleaned from the median feedback time portrayed in Figure 12. Moreover, Cliff's delta effect size measure, presented in Table 14, further confirms observed trends and accentuates the significance of differences illustrated in the median feedback time plot.

Table 1 provides a summary of the median feedback time achieved by each algorithm for each dataset using the baseline number of machines for each dataset, along with the corresponding improvements compared to the actual median feedback time.

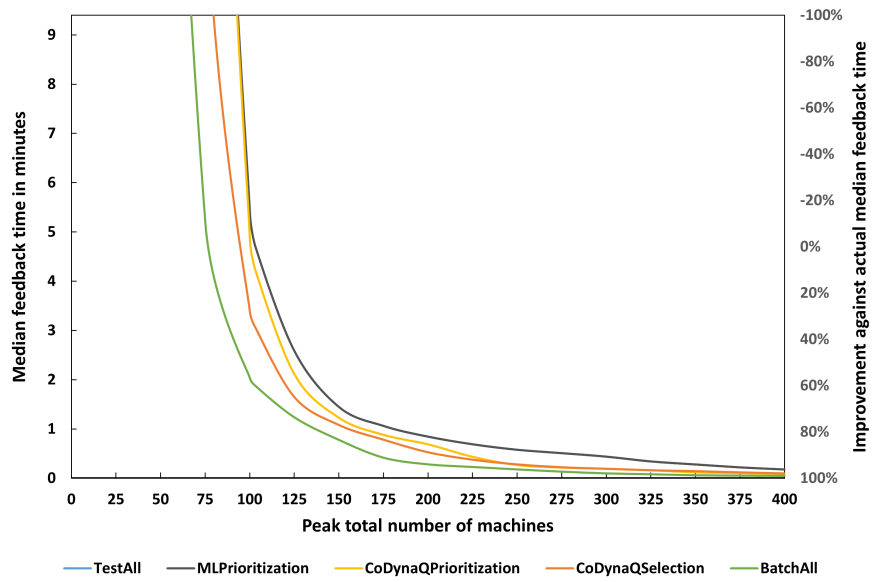

Fig. 11: Median feedback time in minutes and the improvement relative to the baseline median feedback time of 4.72 minutes for the *TestAll*, *CoDynaQSelection*, *CoDynaQPrioritization*, *MLPrioritization*, and *BatchAll* approaches, factoring in variations in peak machine numbers for *GooglePost*.

Observations spanning from Figure 6 to Figure 12 and detailed in Table 1 unveil the following insights. In larger-scale projects like Chrome, with numerous concurrent builds, the *BatchAll* algorithm consistently boasts superior feedback times across various test optimization algorithms. This advantage is particularly pronounced in resource-constrained environments with limited machine availability, all while ensuring no failures are missed. Following closely is the *CoDynaQSelection* algorithm. In contrast, in smaller-scale projects with fewer concurrent builds, such as *JMRI*, the *CoDynaQSelection* algorithm emerges as the top performer despite its propensity to miss up to 55% of failures. Interestingly, test prioritization algorithms, in general, do not yield substantial reductions in feedback time.

An additional insight from Figures 7, 9, 11, and 12 is identifying the point where each test optimization algorithm reaches a plateau. While some algorithms may initially perform well, they may plateau prematurely, with no further room for improvement. We define a plateau as occurring when increasing the number of machines, according to the dataset's unit, results in less than a 1% improvement in feedback time.

Table 2 presents the number of machines and the corresponding median feedback time in minutes at which each algorithm reaches a plateau for different datasets. As seen in both Table 2 and Figure 7, in the case of Chrome, both the *CoDynaQSelection* and *BatchAll* algorithms plateau with 200 machines. However, the former achieves a median feedback time of 0.96 minutes, while the latter achieves 2.4 minutes. The

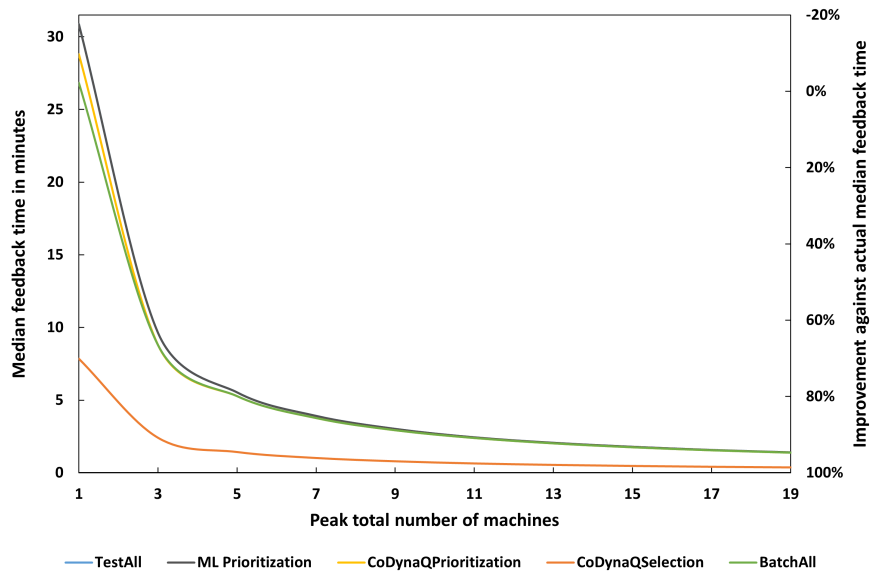

Fig. 12: Median feedback time in minutes and the improvement relative to the baseline median feedback time of 26.39 minutes for the *TestAll*, *CoDynaQSelection*, *CoDynaQPrioritization*, *MLPrioritization*, and *BatchAll* approaches, factoring in variations in peak machine numbers for JMRI.

CoDynaQPrioritization algorithm reaches a plateau with 250 machines, with a median feedback time of 1.94 minutes, while the *TestAll* and *MLPrioritization* algorithms plateau with 325 machines, with a median feedback time of 1.56 minutes.

As demonstrated by Table 2 and Figure 9, for the *GooglePre* dataset, the *BatchAll* algorithm reaches the quickest plateau, with 30 machines at 0.16 minutes, while the *CoDynaQSelection* algorithm plateaus with 70 machines at 0.26 minutes. The *CoDynaQPrioritization* algorithm plateaus with 80 machines at 0.13 minutes, and the *TestAll* and *MLPrioritization* algorithms approach their plateau with 90 machines, with a median feedback time of 0.15 minutes.

For the *GooglePost* dataset, as depicted in both Table 2 and Figure 11, the *BatchAll* algorithm reaches its plateau with the fewest machines, specifically 250, achieving a median feedback time of 0.18 minutes. In contrast, the other algorithms reach their plateau at a median feedback time of 0.22 minutes. Notably, the *CoDynaQSelection* and *CoDynaQPrioritization* algorithms require 275 machines to reach this point, while the *TestAll* and *MLPrioritization* algorithms require 375 machines.

As evidenced by Table 2 and Figure 12, for the *JMRI* dataset, the *CoDynaQSelection* algorithm plateaus with the lowest number of machines, 7, achieving the lowest median feedback time of 1.02 minutes. All the other algorithms plateau with 15 machines, with *BatchAll* and *CoDynaQPrioritization* at 1.76 minutes in median feedback time, and *TestAll* and *MLPrioritization* at 1.78 minutes.

Table 2: The number of machines and the corresponding median feedback time in minutes at which each algorithm reaches a plateau for different datasets.

Datasets	TestAll	CoDynaQ Selection	CoDynaQ Prioritization	MLPrioritization	BatchAll
Chrome	325 (1.56)	200 (0.96)	250 (1.94)	325 (1.56)	200 (2.40)
GooglePre	90 (0.15)	70 (0.26)	80 (0.13)	90 (0.15)	30 (0.16)
GooglePost	375 (0.22)	275 (0.22)	275 (0.22)	375 (0.22)	250 (0.18)
JMRI	15 (1.78)	7 (1.02)	15 (1.76)	15 (1.78)	15 (1.76)

RQ1: The *BatchAll* algorithm demonstrates a feedback time reduction of up to 99.75% with the baseline machine configuration, while ensuring no failures are missed. Meanwhile, the *CoDynaQSelection* approach achieves a reduction of up to 96.12% in feedback time under identical machine conditions, albeit with a drawback of missing between 1% to 55% of failures depending on the dataset. The *CoDynaQPrioritization* approach achieves up to a 10.25% reduction using the same number of machines as the baseline. In contrast, the *MLPrioritization* algorithm exhibits no improvement in feedback time.

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 (Anderson et al., 2014).

In this research question, we investigate the efficacy of various test optimization algorithms in minimizing the number of machines necessary for test execution. This inquiry holds significant relevance, particularly for large enterprises, where substantial investments in hardware infrastructure are commonplace and constitute a critical cost factor (Anderson et al., 2014).

Contrasting with our prior research question, which primarily revolved around comparing feedback time outcomes across various approaches and datasets, our current investigation pivots towards assessing how these approaches influence the reduction of overall machine utilization while upholding a specific median feedback time for each dataset. To achieve this, we quantify the number of machines required by each algorithm to attain the targeted median feedback time across the datasets. The results of our experiment are summarized in Table 3.

For Chrome, the *BatchAll* algorithm achieves a remarkable 88.27% reduction in machine usage, requiring only 21 machines to maintain the baseline feedback time compared to the baseline of 179 machines. *CoDynaQSelection* reduces the number of machines needed to maintain the actual median feedback of 32.95 minutes by 65.92%, needing only 61 machines. However, as seen in RQ1, this reduction

Table 3: The necessary quantity of machines for each algorithm to maintain the actual median feedback time for different datasets, and the corresponding percentage of reduction in machine usage compared to the baseline number of machines.

Datasets	TestAll	CoDynaQ Selection	CoDynaQ Prioritization	MLPrioritization	BatchAll
Chrome	179 (0% baseline)	61 (65.92%)	178 (0.56%)	179 (0%)	21 (88.27%)
GooglePre	46 (0% baseline)	37 (19.56%)	46 (0%)	46 (0%)	14 (69.56%)
GooglePost	103 (0% baseline)	92 (10.68%)	101 (1.94%)	103 (0%)	79 (23.30%)
JMRI	1 (0% baseline)	1 (0%)	1 (0%)	1 (0%)	1 (0%)

comes at the cost of missing 31.25% of the failures. Both the *MLPrioritization* and *CoDynaQPrioritization* algorithms maintain the baseline feedback time but do not significantly reduce machine usage, with the latter achieving only a negligible 0.56% reduction. This limited reduction is because test prioritization algorithms focus primarily on reordering tests rather than minimizing machine usage.

For the *GooglePre* dataset, the *BatchAll* algorithm reduces machine usage by 69.56%, requiring only 14 machines compared to the baseline of 46 machines, without missing any failures. *CoDynaQSelection* reduces the number of machines by 19.56% to 37 machines, while maintaining the median feedback time of 15.21 minutes, but at the cost of missing 6.24% of failures. Neither the *MLPrioritization* nor the *CoDynaQPrioritization* algorithms achieve any reduction in the number of machines used.

Moving to *GooglePost*, the *BatchAll* algorithm reduces machine usage by 23.30%, requiring 79 machines compared to the baseline of 103 machines, without missing any failures. *CoDynaQSelection* reduces the number of machines by 10.68% to 92, while maintaining the actual median feedback time of 4.72 minutes, albeit missing 0.86% of failures. While *MLPrioritization* does not reduce machine usage, *CoDynaQPrioritization* achieves a slight reduction of 1.94%, requiring 101 machines.

Finally, for *JMRI*, none of the algorithms could reduce machine usage since the baseline uses only 1 machine to maintain the actual median feedback time of 26.39 minutes.

Observations from Table 3 show that the *BatchAll* algorithm is the most effective approach for reducing the number of machines used without missing any failures. The *CoDynaQSelection* algorithm follows, but it misses up to 55% of failures. In contrast, test prioritization algorithms are generally not very effective in reducing machine usage.

RQ2: The *BatchAll* approach achieves up to an 88.27% reduction in machine usage while maintaining the baseline feedback time and not missing any failures. The *CoDynaQSelection* approach reduces machine usage by up to 65.92% while maintaining the baseline feedback time but misses up to 55.27% of failures. The *CoDynaQPrioritization* approach achieves a slight reduction in machine usage by up to 1.94%, maintaining the same feedback time, while the *MLPrioritization* approach shows no improvement in machine usage.

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

Apart from the feedback time required to obtain all the commit test results and reducing the number of machines needed for running tests, another crucial 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. (2014) and Fallahzadeh and Rigby (2022). This measure represents the difference in the time it takes to detect each individual failing test. We use the median GainedTime across failures detected by each algorithm to compare them. To ensure a balanced median GainedTime comparison among algorithms, we primarily concentrate on the failing tests detected by *CoDynaQSelection*, as it encompasses the common failing tests across all approaches.

The resulting median GainedTime for different test optimization algorithms in this study for Chrome using varying numbers of machines is presented in Figure 13. 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 a logarithmic scale and add 1 GainedTime unit to the results.

As illustrated in Figure 13, the median GainedTime across all test optimization algorithms on Chrome exhibits notable improvements in highly resource-constrained environments. However, as the number of machines increases, the efficacy of these algorithms decreases significantly. Notably, the *BatchAll* algorithm outperforms all others in highly resource-constrained settings. Conversely, *CoDynaQSelection* demonstrates slightly better performance than *BatchAll* in environments with ample resources. Nonetheless, it is essential to acknowledge that *CoDynaQSelection* overlooks certain failing tests, accounting for 31.25% of all failures on the Chrome dataset, as mentioned in the previous RQs. The test prioritization algorithms exhibit significantly lower GainedTime performance, with *CoDynaQPrioritization* following *BatchAll* and *CoDynaQSelection*, while *MLPrioritization* recording the lowest GainedTime results.

To evaluate and compare the distributions of GainedTime across various test optimization algorithms for Chrome, we employ statistical tests (see Tables 15 and

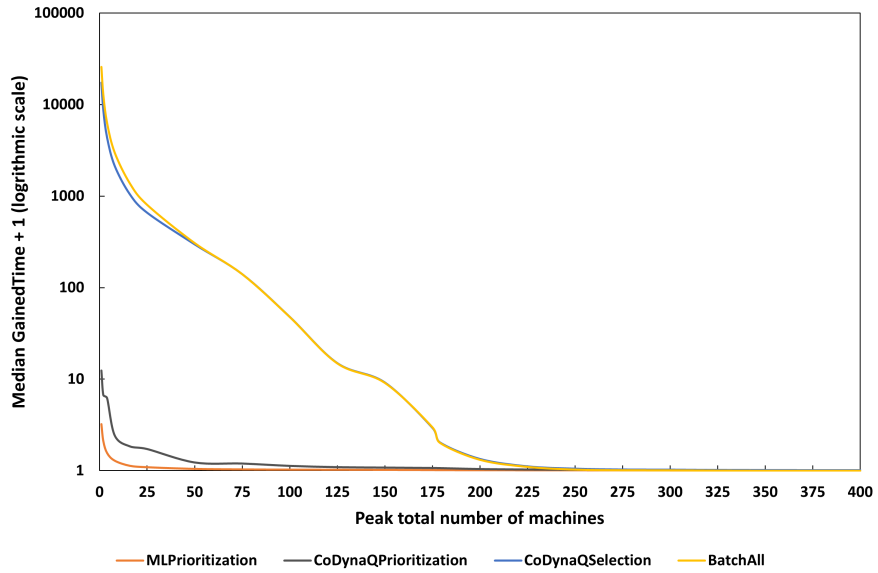

Fig. 13: Median GainedTime + 1 in hours for *MLPrioritization*, *CoDynaQPrioritization*, *CoDynaQSelection*, and *BatchAll* algorithms displayed in logarithmic scale and in hours by considering the different peak number of machines on Chrome.

16 in Appendix A). These findings closely mirror the observed median GainedTime depicted in Figure 13. For instance, the results indicate statistically significant differences in most comparisons, except for a few instances. Specifically, there are no significant differences between *CoDynaQPrioritization* and *CoDynaQSelection* when the number of machines is 275 or higher. Moreover, there is no significant difference between the distributions of GainedTime for *CoDynaQSelection* and *BatchAll* when the number of machines is between 25 and 250. These differences are mostly apparent as their median GainedTime also overlaps, as shown in Figure 13.

For the *GooglePre* dataset, Figure 14 displays the median GainedTime results in logarithmic scale for all approaches studied, utilizing varying numbers of machines. Similar to Chrome, the GainedTime for all algorithms demonstrates more pronounced improvements in resource-constrained environments with fewer machines. However, as the number of machines increases, the GainedTime performance of all algorithms declines significantly. The *BatchAll* algorithm consistently achieves significantly better GainedTime performance compared to all other approaches, except when the number of machines increases substantially, resulting in similar and limited performance across all algorithms. Subsequently, the *CoDynaQSelection* algorithm follows the *BatchAll* algorithm, although it misses 6.24% of failures. Thereafter, with a notable margin, the test prioritization algorithms trail the *CoDynaQSelection* algorithm. While the performance of *CoDynaQPrioritization* and *MLPrioritization* is comparable, *CoDynaQPrioritization* exhibits slightly better performance.

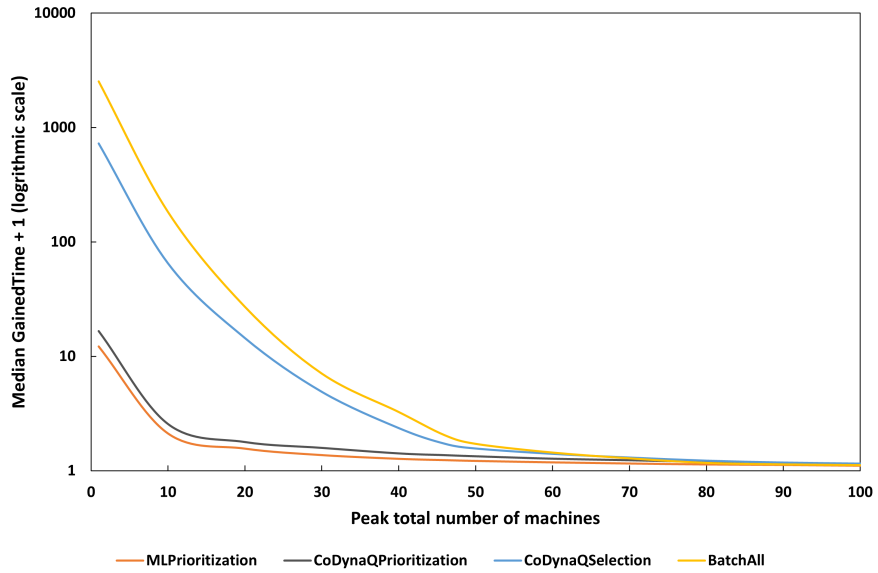

Fig. 14: Median GainedTime + 1 in hours for *MLPrioritization*, *CoDynaQPrioritization*, *CoDynaQSelection*, and *BatchAll* algorithms displayed in logarithmic scale and in hours by considering the different peak number of machines on *GooglePre*.

The statistical tests related to the *GooglePre* dataset, comparing GainedTime distributions across various test optimization approaches and commits, are detailed in Tables 17 and 18 within Appendix A. These results closely align with the insights derived from the median GainedTime depicted in Figure 14. In most cases, there are significant differences between the GainedTime distributions, except between *CoDynaQPrioritization* and *CoDynaQSelection* when the number of machines is more than 80, and between *MLPrioritization* and *CoDynaQPrioritization* when the number of machines is more than 90.

For the *GooglePost* dataset, Figure 15 illustrates the median GainedTime outcomes on a logarithmic scale for all analyzed algorithms, utilizing various numbers of machines. Similar to Chrome and *GooglePre*, the GainedTime for all algorithms displays more noticeable enhancements in settings with fewer resources. However, as the number of machines rises, the effectiveness of GainedTime for all algorithms decreases notably. The *BatchAll* algorithm achieves notably superior GainedTime performance compared to all other strategies, except in scenarios with a substantial increase in machine numbers, where performance across all algorithms becomes equally limited. Following the *BatchAll* algorithm is the *CoDynaQSelection* approach, despite missing 0.86% of failures. Subsequently, with a notable difference, *CoDynaQPrioritization* and *MLPrioritization* follow, with the *MLPrioritization* approach performing slightly better.

The statistical tests regarding the *GooglePost* dataset are shown in Tables 19 and 18 within Appendix A. The results indicate statistical significance in the Gained-

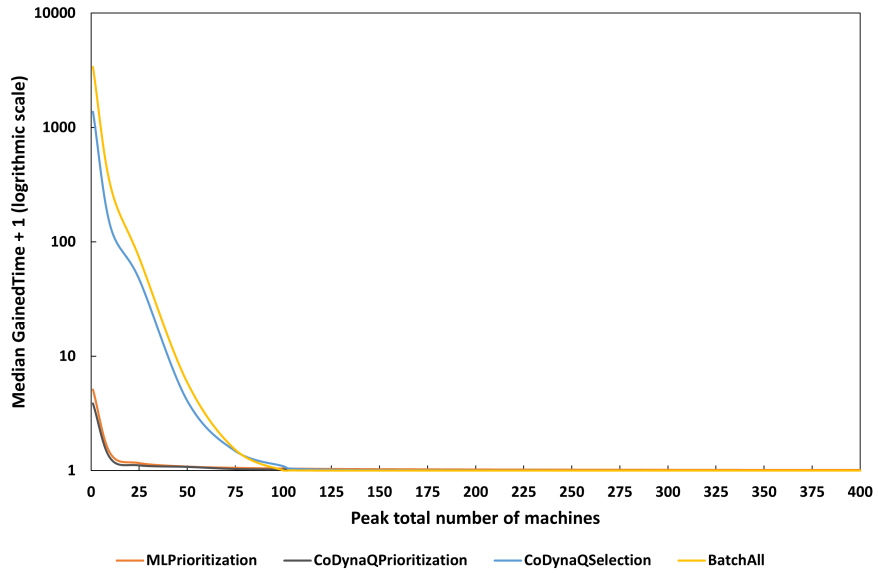

Fig. 15: Median GainedTime + 1 in hours for *MLPrioritization*, *CoDynaQPrioritization*, *CoDynaQSelection*, and *BatchAll* algorithms displayed in logarithmic scale and in hours by considering the different peak number of machines on *GooglePost*.

Time distributions, with some exceptions. Notably, there is no significant difference between the *CoDynaQPrioritization* and *CoDynaQSelection* algorithms when more than 125 machines are used. Similarly, there is no statistical significance between *CoDynaQSelection* and *BatchAll* when 75 machines are used. Additionally, between the *MLPrioritization* algorithm and *CoDynaQPrioritization*, there is no statistical significance when either 50 or more than 325 machines are used. Likewise, between *MLPrioritization* and *CoDynaQSelection*, there is no statistical significance when more than 275 machines are used.

For *JMRI*, Figure 16 presents the median GainedTime outcomes on a logarithmic scale for all analyzed algorithms, utilizing various quantities of machines. Unlike the Chrome, *GooglePre*, and *GooglePost* datasets, the effectiveness of the *BatchAll* algorithm in terms of GainedTime is negligible in the *JMRI* dataset due to the limited number of concurrent builds. However, other test optimization techniques demonstrate notable GainedTime improvements in resource-constrained environments with fewer machines, with their effectiveness diminishing as the number of machines increases. Remarkably, the *CoDynaQSelection* algorithm outperforms all other algorithms by detecting failures faster through selective test execution. However, it is noteworthy that this approach misses a significant portion, accounting for 55.27%, of failures. Additionally, the *CoDynaQPrioritization* algorithm consistently outperforms the *MLPrioritization* algorithm.

The statistical tests related to the *JMRI* dataset are displayed in Tables 21 and 14 within Appendix A. The results indicate statistical significance between the *CoDy-*

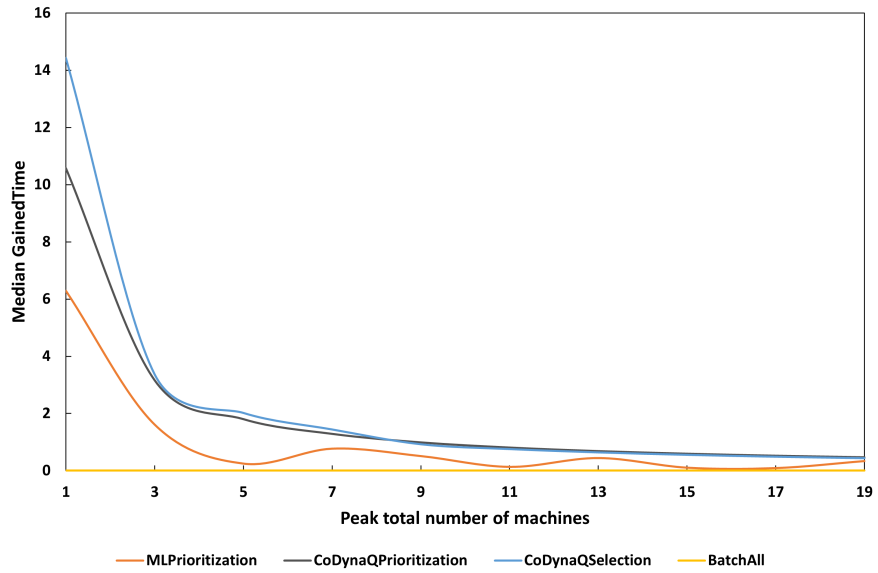

Fig. 16: Median GainedTime in minutes for *MLPrioritization*, *CoDynaQPrioritization*, *CoDynaQSelection*, and *BatchAll* algorithms displayed considering the different peak number of machines on JMRI.

naQPrioritization and *BatchAll* algorithms, as well as between *CoDynaQSelection* and *BatchAll*. Similarly, statistical significance is observed between *MLPrioritization* and *CoDynaQSelection*, and also between *MLPrioritization* and *BatchAll* in most cases. However, the stringent p-value threshold of 0.00083 does not confirm statistical significance between *CoDynaQPrioritization* and *CoDynaQSelection*, as well as between *MLPrioritization* and *CoDynaQPrioritization* algorithms.

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 closer to the actual number of machines used. The findings are presented in Table 4.

For Chrome, Table 4 demonstrates that when utilizing the baseline of 179 machines, both the *CoDynaQSelection* and *BatchAll* algorithms achieve similar median GainedTime values of 62.12 and 61.40 minutes, respectively. However, *CoDynaQSelection* misses 31.25% of failures, whereas *BatchAll* doesn't miss any. Subsequently, *CoDynaQPrioritization* with a significant difference, shows a median GainedTime of 3.92 minutes, while the *MLPrioritization* algorithm achieves the lowest median GainedTime of 0.71 minutes.

In the case of *GooglePre*, *BatchAll* outperforms other techniques, achieving a GainedTime of 61.40 minutes without missing any failures. Following this, *CoDynaQSelection* achieves a median GainedTime of 43.13 minutes, although it encounters a 6.24% failure rate. Next, *CoDynaQPrioritization* attains a GainedTime of

Table 4: The median *Gained Time* in minutes achieved by different algorithms using baseline number of machines as in the *TestAll* baseline for different datasets and for the failures detected by *CoDynaQSelection*.

Datasets	CoDynaQ Selection	CoDynaQ Prioritization	MLPrioritization	BatchAll
Chrome	62.12	3.92	0.71	61.40
GooglePre	43.13	22.09	14.40	62.84
GooglePost	2.79	2.93	2.37	0.24
JMRI	14.43	10.57	6.29	0

22.09 minutes. Finally, *MLPrioritization* shows the smallest GainedTime improvement, with 14.40 minutes.

For *GooglePost*, while *CoDynaQPrioritization* outperforms other approaches by achieving a median GainedTime of 2.93 minutes using the baseline of 103 machines, this may not represent the entire picture. Typically, *BatchAll* and *CoDynaQSelection* outperform other methods when fewer machines are used, as illustrated in Figure 15. At the baseline number of machines, *CoDynaQSelection* and *MLPrioritization* achieve median GainedTime of 2.79 and 2.37 minutes, respectively, which is close to the performance of *CoDynaQPrioritization*, with *CoDynaQSelection* missing 0.83% of failures. In contrast, *BatchAll* achieves a lower median GainedTime of 0.24 minutes with the baseline number of machines.

In the case of *JMRI*, *CoDynaQSelection* performs better than all other algorithms when using the baseline of one machine, achieving a median GainedTime of 14.43 minutes, though it misses a significant number of failures (55.27%). Following this, *CoDynaQPrioritization* achieves a median GainedTime of 10.57 minutes, and *MLPrioritization* attains a median GainedTime of 6.29 minutes. *BatchAll* is ineffective in terms of GainedTime due to the limited concurrent builds in the *JMRI* dataset.

Previous GainedTime results were derived from commonly detected failures across all approaches under study, ensuring comparability. However, to evaluate if the failures missed by *CoDynaQSelection* significantly affect the relative performance of these algorithms in terms of GainedTime, we compiled GainedTime outcomes for all algorithms except *CoDynaQSelection*, including all failures in Table 5. Across all datasets, we noticed a decline in GainedTime performance for all algorithms. This decline is particularly noticeable in datasets like *JMRI* and Chrome, where *CoDynaQSelection* missed more failures. Nevertheless, this observation does not change the overall relative performance of these algorithms.

The results from Figures 13 to 16 and Tables 4 and 5 demonstrate that in larger-scale projects characterized by substantial concurrent builds, such as Chrome, the *BatchAll* and *CoDynaQSelection* algorithms, which exhibit similar performance, excel in detecting failing tests faster, particularly in scenarios with fewer machines. Notably, *CoDynaQPrioritization* and *MLPrioritization* follow *BatchAll* and *CoDynaQSelection*, albeit with a considerable difference in performance. Conversely, in smaller-scale projects, such as *JMRI*, due to the lack of concurrent builds, the *BatchAll* algorithm becomes almost ineffective in terms of GainedTime, while the *CoDynaQSelection*

Table 5: The median *Gained Time* in minutes achieved by different algorithms except *CoDynaQSelection* using baseline number of machines as in the *TestAll* baseline for different datasets and all failures.

Datasets	CoDynaQ Prioritization	MLPrioritization	BatchAll
Chrome	2.68	0.67	48.32
GooglePre	19.09	13.89	56.32
GooglePost	1.44	2.34	0.23
JMRI	8.25	2.53	0

algorithm exhibits superior performance compared to other approaches, albeit with the caveat of missing up to 55% of failures in our experiment.

RQ3: *BatchAll* generally outperforms other algorithms in terms of GainedTime, achieving a maximum GainedTime of 62.84 minutes across different datasets, except in the *JMRI* dataset with limited concurrent builds. This advantage is notable as *BatchAll* does not miss any failures. The *CoDynaQSelection* algorithm demonstrates a significant improvement in median GainedTime, reaching up to 62.12 minutes, but it may miss as much as 55.27% of failures. Following, *CoDynaQPrioritization* achieves a maximum GainedTime of 22.09 minutes with a significant difference, while the *MLPrioritization* algorithm showcases a median GainedTime of up to 14.40 minutes.

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*, *MLPrioritization*, and *CoDynaQPrioritization* algorithms, there is no ExecutionReduction as they run all tests, although they may reorder them. The results for the *CoDynaQSelection* and *BatchAll* algorithms for Chrome are shown in Figure 17. The figure demonstrates that *CoDynaQSelection* 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 18 illustrates the ExecutionReduction for both the *CoDynaQSelection* and *BatchAll* algorithms on the *GooglePre* dataset. It is evident that the *CoDynaQS-*

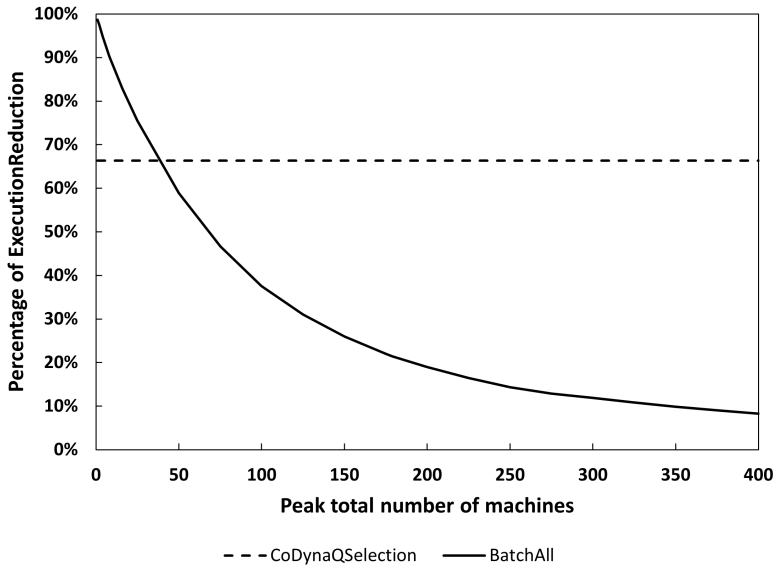

Fig. 17: The percentage of ExecutionReduction achieved by *CoDynaQSelection* and *BatchAll* algorithms, relative to the total execution time for all test cases, was calculated for different peak numbers of machines for Chrome.

election algorithm consistently achieves a 22.44% reduction in test execution time across different machine configurations. In contrast, the *BatchAll* algorithm demonstrates varying levels of ExecutionReduction, ranging from 76.13% with a single machine to 52.83% with 100 machines.

Figure 19 displays the ExecutionReduction achieved by the *CoDynaQSelection* and *BatchAll* algorithms on the *GooglePost* dataset. The *CoDynaQSelection* algorithm consistently reduces test execution time by 14.28% across different machine setups. Conversely, the *BatchAll* algorithm exhibits different levels of ExecutionReduction, ranging from 44.73% with one machine to 23.28% with 400 machines.

Figure 20 depicts the ExecutionReduction achieved by the *CoDynaQSelection* and *BatchAll* algorithms on the *JMRI* dataset. The *CoDynaQSelection* algorithm consistently reduces test execution time by 58.46% across different machine configurations. However, this improvement comes at the expense of missing 55.27% of failures. On the other hand, the batching algorithm only reduces test execution time by up to 2.67% when using a single machine, and its ExecutionReduction becomes almost negligible as the number of machines increases. This is due to the limited concurrent builds in the *JMRI* dataset.

To assess the potential time savings achievable by different test optimization algorithms using the baseline number of machines, which is more representative of the actual number of machines used in practice across different datasets, we present the results of ExecutionReduction for the baseline number of machines in Table 6.

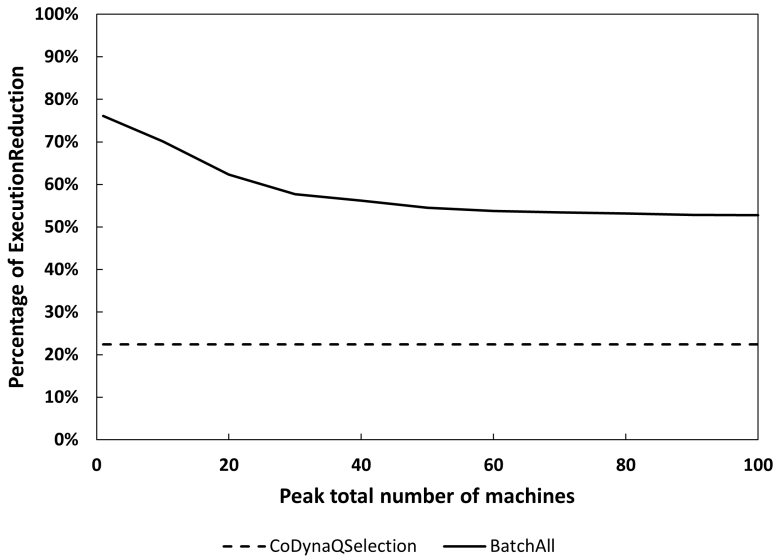

Fig. 18: The percentage of ExecutionReduction achieved by *CoDynaQSelection* and *BatchAll* algorithms, relative to the total execution time for all test cases, was calculated for different peak numbers of machines for *GooglePre*.

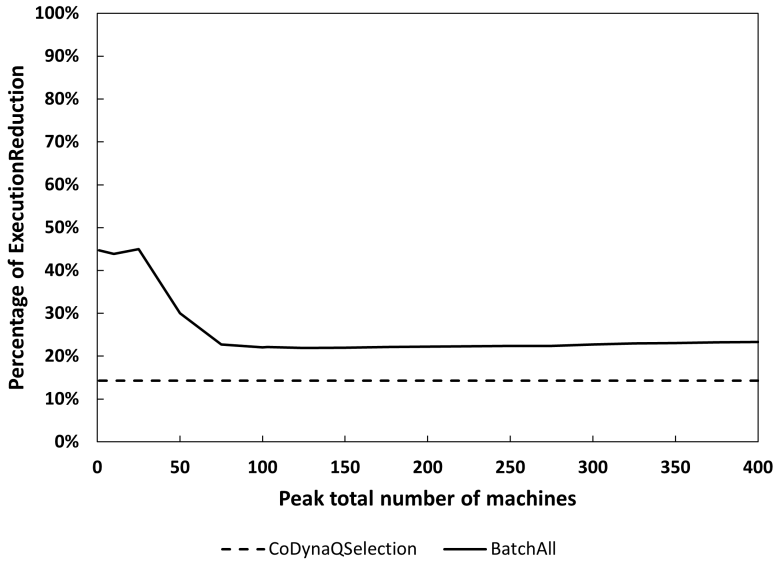

Fig. 19: The percentage of ExecutionReduction achieved by *CoDynaQSelection* and *BatchAll* algorithms, relative to the total execution time for all test cases, was calculated for different peak numbers of machines for *GooglePost*.

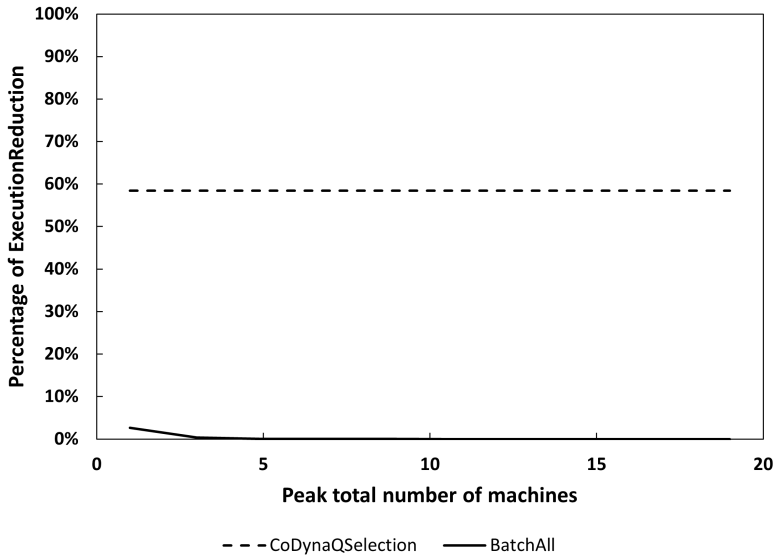

Fig. 20: The percentage of ExecutionReduction achieved by *CoDynaQSelection* and *BatchAll* algorithms, relative to the total execution time for all test cases, was calculated for different peak numbers of machines for JMRI.

Table 6: The *Execution Reduction* achieved by different algorithms using the baseline number of machines for different datasets.

Datasets	CoDynaQ Selection	CoDynaQ Prioritization	MLPrioritization	BatchAll
Chrome	66.31%	0%	0%	21.45%
GooglePre	22.44%	0%	0%	55.24%
GooglePost	14.28%	0%	0%	22.16%
JMRI	58.46%	0%	0%	2.67%

In the Chrome and *JMRI* datasets, where *CoDynaQSelection* misses 31.25% and 55.27% of failures respectively (see RQ1), it significantly outperforms other algorithms, demonstrating 66.31% and 58.46% reduction in execution time as illustrated in Table 6. However, in the *GooglePre* and *GooglePost* datasets, where *CoDynaQSelection* misses fewer failures at 6.24% and 0.86% respectively, the *BatchAll* algorithm surpasses all others, showing a significant execution time reduction of 55.24% and 22.16% respectively, without missing any failures. In the *JMRI* dataset, the execution time reduction achieved by *BatchAll* is only 2.67% when using a baseline of 1 machine, due to limited concurrent builds. The *MLPrioritization* and *CoDynaQPrioritization* algorithms do not result in any time savings for test execution, as expected.

RQ4: *CoDynaQSelection* achieves up to 66.31% reduction in test execution time, although it comes with the trade-off of potentially missing up to 55.27% of failing tests. Conversely, both the *MLPrioritization* and *CoDynaQPrioritization* approaches do not yield any savings in test executions. The *BatchAll* technique, on the other hand, demonstrates varied savings ranging from 0% to 98.69%, depending on the dataset and the number of machines being utilized.

7 Threats to Validity

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

7.1 External Validity

In this study, we applied our experiments to four available large-scale datasets, recognizing that they may not fully represent all large-scale projects. Nevertheless, we made every effort to include all publicly available large-scale datasets. The Google Chrome dataset (Fallahzadeh and Rigby, 2022) stands out as the largest publicly available dataset in terms of the number of test cases, boasting 49,932 test suites, 276 million test cases over one month, and up to 47 concurrent builds. Additionally, we analyzed the *GooglePre* dataset (Elbaum et al., 2014), comprising 1,112,158 test suites over one month with up to 49 concurrent builds. Furthermore, our experiments were conducted on the *GooglePost* dataset, which includes 1,495,856 test suites with up to 86 build concurrency. Finally, we incorporated the largest open-source dataset from the study by Yaraghi et al. (2023), consisting of 6,469,640 test cases spanning over 5 months, with up to 4 concurrent builds.

The approaches adopted in this study may not encompass all test optimization techniques currently in use. However, we made every effort to replicate state-of-the-art techniques representing the main families of test optimization approaches relevant to large-scale projects. We used the *MLPrioritization* approach, which is the state-of-the-art test prioritization method for test prioritization (Yaraghi et al., 2023). We also used the *CoDynaQPrioritization* and *CoDynaQSelection* approaches as recent studies for test optimization in continuous integration environment and across builds (Zhu et al., 2018). As for the batching techniques, we applied the state-of-the-art batching technique, the effectiveness of which we recently demonstrated (Fallahzadeh et al., 2023).

This study primarily utilizes features related to historical test execution records for test selection and prioritization, recognizing that these may not encompass all possible features. The selection of these features is based on their demonstrated effectiveness at scale and in practical settings, as well as the limitations of available features across projects. As documented by Elbaum et al. (2014) and Memon et al. (2017), the utilization of features such as the ones collected by code instrumentation is infeasible for large-scale projects like those at Google. The impracticality arises from the high CI traffic, making data collection prohibitively expensive, and the rapid code churn,

rendering the collected data imprecise and quickly obsolete. In contrast, test execution records have been shown to be among the most lightweight and effective features for test optimization (Anderson et al., 2014; Yaraghi et al., 2023).

In this study, we conducted simulations based on historical test results rather than real-time execution or stopping of tests. Consequently, our simulations followed the settings of the projects under study, where test executions did not stop the build after detecting a single failure. Therefore, our findings may not be applicable to projects that halt build executions after the first detected failure. However, it is important to note that across all the projects we studied (Chrome, *GooglePre*, *GooglePost*, and *JMRI*), which reflect the Google and Travis CI settings, there were no instances where builds were stopped after a single failure. The occurrence of multiple failures within individual builds in these datasets indicates that the builds were not terminated after the first failure.

7.2 Construct Validity

To calculate the feedback time of a build, only the test execution time is considered, while 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 because multiple commits are batched together, making the compilation time unpredictable. 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.

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 (Lam et al., 2019). In this study, we simulated concurrent and autonomous test executions based on the test results for projects that already utilize parallel test executions.

In this study, we simulated varying numbers of machines using historical test execution times from different datasets. Instead of physically distributing test runs across multiple machines, we modeled parallel execution by dividing the test execution times by the number of machines used. In practice, distributing test execution across different machines requires additional time for test distribution and result collection. This extra time can potentially increase feedback time and reduce GainedTime and ExecutionReduction.

We also made a set of assumptions for the batching results. For instance, if a test fails in one of the builds included in the batch, it is assumed to fail when combined

with other builds in the batch as well. While this might be true in most cases, there could be instances where the failure is resolved after applying subsequent changes in the batch, and the test subsequently passes. 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 (Fallahzadeh and Rigby, 2022), there could be instances where the test either fails or passes in the batch. In the former case, an additional culprit-finding process would be necessary, while in the latter case, there would be no need for re-runs.

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. Building upon the findings of our study in RQ1, we recommend employing test batching techniques effectively in larger-scale projects with a considerable number of concurrent builds. These techniques exhibit the capability to significantly reduce feedback time without compromising the detection of failures. Furthermore, in resource-constrained environments where only a few machines are available, test batching demonstrates even greater advantages. Conversely, in smaller-scale projects with fewer concurrent builds, such as the *JMRI* project, test selection techniques emerge as a viable option for reducing feedback time. However, it is important to note that this approach may result in overlooking failures. If such loss of failure detection is deemed intolerable, we advocate for the adoption of test batching and test prioritization methods instead.

Implications of resource usage comparison in software testing. Based on the results of RQ2, we recommend the following practice to minimize resource usage in testing. Both test selection and batching approaches are effective in substantially reducing the number of machines and resources needed for testing. However, while test selection may overlook some failures, ranging from 1% to 55% in our experiment, batching reduces resource consumption without sacrificing any failure detection and yields greater savings compared to the test selection algorithm. Moreover, our findings indicate that the advantage of using the batching algorithm in terms of feedback time and resource reduction, compared to other test optimization algorithms, is more pronounced when resources are limited, as opposed to in a resource-rich environment.

Implications of test optimization for failure detection speed. Our study in RQ3 yields the following recommendations for practitioners. In larger-scale projects with substantial build concurrency, both test batching and test selection techniques are significantly more effective than test prioritization in accelerating failure detection. This discrepancy becomes even more pronounced in resource-constrained environments with limited machine availability. While test selection may miss some failures, this is not the case for test batching. Conversely, in smaller-scale projects with minimal build concurrency, test batching becomes less effective, with test selection emerging as the optimal choice, despite the risk of overlooking certain failures, which ranged from 1% to 55% in our experiment. Moreover, test prioritization techniques applied

across multiple builds demonstrate superior effectiveness compared to those applied within individual builds in terms of expediting failure detection.

Implications of the test execution time comparison. Drawing from the results obtained in RQ4, we propose the following recommendations for practitioners. Both test selection and batching techniques exhibit high effectiveness in Execution Reduction. However, a notable discovery from this investigation is the static nature of Execution Reduction in the test selection algorithm, contrasted with its significant variation in the batching approach. This discrepancy arises from the behavior observed as the number of resources increases. In batching, the queue size decreases, necessitating more test executions. Conversely, in test selection, the number of test executions remains constant regardless of resource changes. In resource-constrained environments, we advise the utilization of batching to minimize test execution time. Conversely, in environments with ample resources, practitioners may lean towards test selection, potentially achieving more efficient test execution, albeit with a risk of overlooking failures. Alternatively, practitioners can opt for the batching algorithm, which adjusts test execution time based on the number of machines in use, while ensuring comprehensive failure detection.

Implications of Comparing Test Optimization Techniques. Our comparative analysis reveals important insights into the practical application of test case prioritization, selection, batching, and parallelization. While these techniques may seen as complementary, understanding their individual performance helps practitioners prioritize their use based on specific project needs. For instance, in resource-constrained environments, we observed that effectively using test batching in large-scale projects might eliminate the need for test selection, which carries the risk of missing failures but uses similar resources. The use of batching in these environments improves build feedback time, machine usage, failure detection speed, and reduces test executions. Conversely, in scenarios with sufficient resources for parallelization, the benefits of other test optimization techniques may diminish.

Impacts of failure rate on batching algorithm. The performance of the batching algorithm is significantly influenced by the failure rate. A higher number of failures in test results can result in more penalties for identifying culprits in batching. However, research by Beheshtian et al. (2021) suggests that 85.5% of the Travis CI projects they studied could benefit from batching, provided their failure ratio remains below 40%. In our study, the build failure rates are as follows: 8.5% for Chrome, 16.42% for *GooglePre*, 23.12% for *GooglePost*, and 6.41% for *JMRI*. These rates are all well below the threshold, making batching an effective approach for improving test execution time. Nevertheless, the number of concurrent builds is another important factor that determines the effectiveness of batching. This justifies the lower batching performance observed for the *JMRI* dataset.

Measuring failure detection. In this study, one of our key objectives was to measure how quickly each algorithm detects failures across multiple builds. This is particularly relevant given that we utilized varying numbers of machines and employed different test optimization techniques, such as *CoDynaQPrioritization*, *CoDynaQSelection*, and *BatchAll*, all of which operate across builds. Specifically, we focused on assessing when failures are detected in parallel across several concurrent builds, as opposed to detecting failures within a single, sequentially executed build. Our

approach contrasts with studies that focus on running builds individually and sequentially, where test prioritization techniques are applied only within each build. Such studies typically compare performance by measuring the difference in repositioning of failures compared to the beginning of a single build.

For example, when applying test prioritization techniques on Chrome using only a few machines, a build can be delayed by hours or even days due to the inability of the algorithms to clear the queue quickly and the limited availability of resources. However, using the same resources but employing the *CoDynaQSelection* or *BatchAll* techniques would significantly accelerate the build executions by hours or days. As a result, the time required to detect the same test failure would be drastically reduced, and this speed of failure detection needs to be captured by the metric. Traditional test prioritization metrics, which only measure the repositioning of failures within a single build, fail to account for these differences. Instead, the metric should take a more holistic, cross-build view of failure detection to accurately reflect the improvements in speed.

To compare cross-build failure detection speed, we initially considered the APFD metric (Rothermel et al., 1999). However, we found that it falls short in differentiating the performance of various approaches in this context. This limitation of APFD was already suggested by Elbaum et al. (2014). While they were among the first to advocate for APFD in test case prioritization, they acknowledged its limitations in measuring the speed of failure detection for individual tests in continuous integration. Our work encountered more specific challenges when applying APFD across builds, which we discuss further below.

The primary limitation of APFD lies in its sensitivity to the total number of test cases being evaluated. APFD reflects changes in test case execution order relative to the total number of test cases. Figure 21 illustrates the impact of reordering on the APFD score for different test set sizes, assuming that each test case takes 1 minute to execute. In the left figure, with 10 test cases and a single failure, moving the failure from position 6 to position 5 results in a 10% improvement in the APFD score, reflecting a 1-minute gain. Conversely, the right figure shows the same reordering applied to 100 test cases, where moving the failing test from position 60 to position 59 leads to only a 1% increase in the APFD score—even though this still represents a 1-minute improvement. The area between the blue and orange lines in each figure indicates the extent of improvement due to reordering. Although the gain is 1 minute in both cases, APFD reflects a 9% smaller improvement in the 100-test case scenario simply because the total number of test cases is ten times larger. However, 1 minute remains 1 minute, allowing developers in both cases to start one minute earlier to debug and fix a test failure.

The sensitivity of the APFD metric to the total number of test cases makes its results incomparable across different test suites with varying numbers of test cases, as well as across different datasets. This limitation was highlighted by Wang et al. (2020) in their analysis of the metric (Section 3). However, a more critical issue we observed is that APFD becomes nearly ineffective in large-scale cross-build test optimization. For instance, when dealing with millions of test cases, even substantial changes in the order of test failures would have only a negligible impact on the APFD score, rendering the metric inadequate for evaluating large-scale testing scenarios.

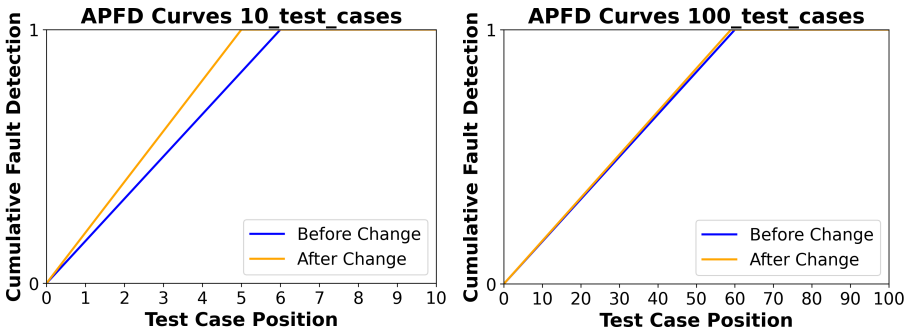

Fig. 21: Comparison of APFD results for 10 and 100 test cases with 1 failure and 1 order of improvement.

A further limitation of APFD is its lack of sensitivity to time. For instance, what does a 10% improvement in APFD actually mean when answering the research question of how quickly Algorithm A detects failures compared to Algorithm B? Does this improvement reflect a difference of 1 millisecond or 1 hour? Due to the relative nature of APFD, a 10% improvement could represent a millisecond in one dataset and an hour in another, making the results both unintuitive and difficult to compare across different datasets.

More critically, APFD becomes ineffective when dealing with parallel execution. Since APFD considers only the order of test case execution to evaluate algorithm performance, it neither accounts for the execution time nor handles cases where tests are run in parallel on multiple machines, each with a separate order. In practice, while test results can be aggregated using timestamps from different machines to create a unified order, the fundamental issue with APFD is its inability to consider the time taken for failure detection. This makes APFD inadequate for reflecting real-world performance improvements, especially when execution time is a crucial factor.

For example, in the earlier scenario with 10 test cases on a single machine, the total sequential execution time might be 10 minutes. Increasing the number of machines to 2 could reduce the execution time to 5 minutes, enabling failures to be detected twice as fast. However, if the overall order of test case execution remains the same when aggregated, the APFD results will not reflect this improvement.

Figures 22 to 25 show the cross-build APFD results for different datasets with varying numbers of machines. In these figures, the APFD curves for the *MLPrioritization* and *CoDynaQPrioritization* algorithms overlap, failing to confirm the significant performance differences in terms of median GainedTime results, which reveal that for Chrome, the *CoDynaQPrioritization* algorithm detects failures 9.24 hours faster than the *MLPrioritization* algorithm on a single machine, as shown in Figure 13. While APFD can indicate performance differences when substantial gaps exist—such as the significant differences observed between test prioritization algorithms and *CoDynaQSelection*, which can span hundreds of hours—it lacks the precision needed to capture more subtle distinctions.

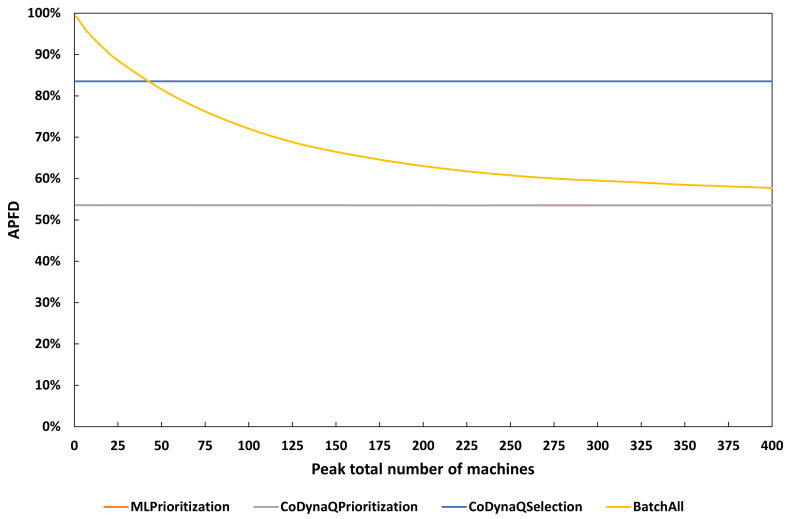

Fig. 22: APFD results for different algorithms on the Chrome dataset by varying the numbers of machines.

Moreover, APFD struggles to capture performance variations when the number of machines changes, as illustrated in Figures 22 to 25. The fluctuations observed in the *BatchAll* algorithm result from the varying number of test cases executed under different machine configurations, rather than APFD's ability to differentiate performance in parallel execution. While test optimization strategies generally detect failures faster as the number of machines increases, their relative advantage over the baseline with the same number of machines diminishes due to the reduced impact of test reordering when more machines are used.

This trend is clear in Figures 13 to 16, where beyond a certain machine count, the median GainedTime achieved by various test optimization algorithms becomes negligible and fails to outperform the baseline with the same number of machines. For example, in Chrome, with more than 75 machines, the median GainedTime for the *MLPrioritization* algorithm plateaus and becomes insignificant compared to the baseline. However, these conclusions are not evident through APFD alone, underscoring the limitations of relying on APFD to assess performance.

The GainedTime metric used in this study is straightforward yet effective for comparing the speed of failure detection among different test optimization algorithms in parallel environments. It avoids the issues associated with the APFD, as it quantifies the difference in failure detection times for each individual test between each test optimization algorithm and the baseline *TestAll* algorithm for a given number of machines. It is also the same metric employed by Elbaum et al. (2014) and in our previous study (Fallahzadeh and Rigby, 2022) for multi-build and multi-machine environments.

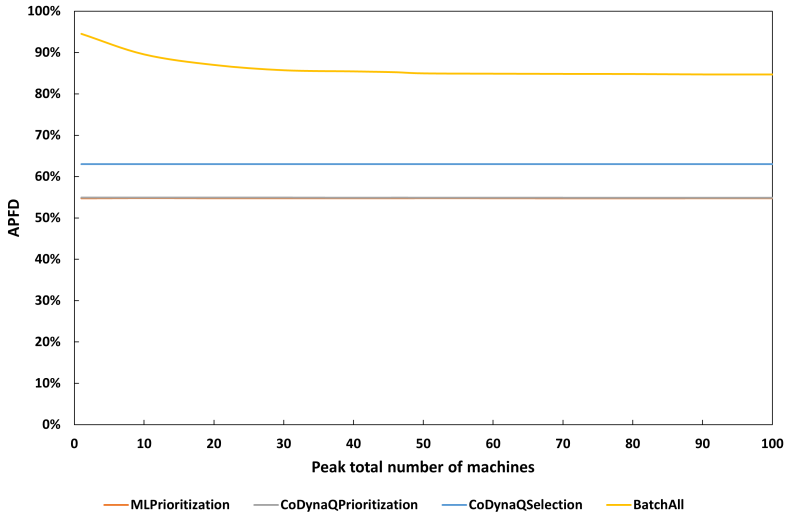

Fig. 23: APFD results for different algorithms on the *GooglePre* dataset by varying the numbers of machines.

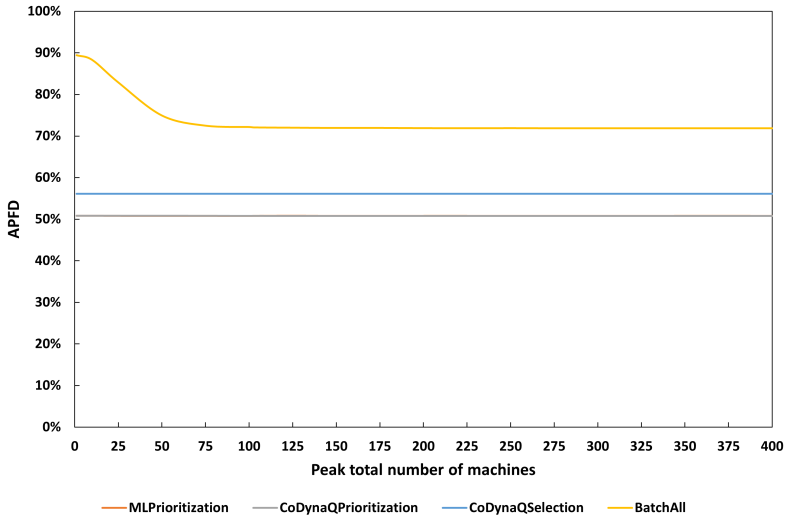

Fig. 24: APFD results for different algorithms on the *GooglePost* dataset by varying the numbers of machines.

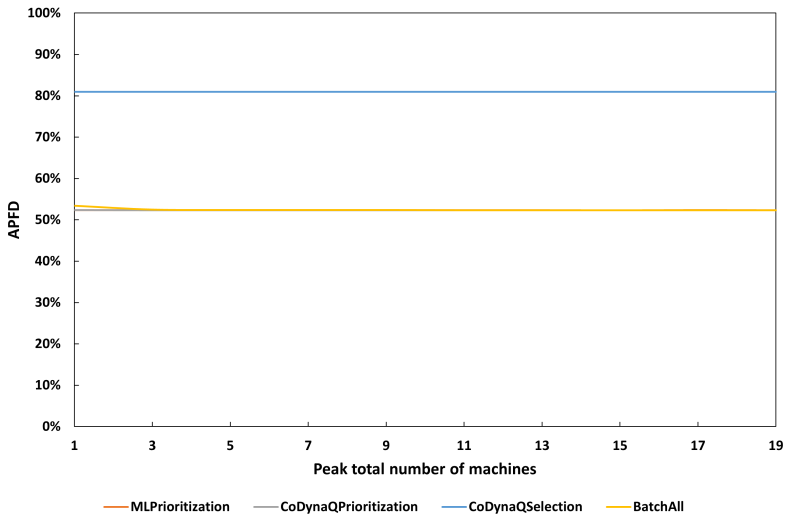

Fig. 25: APFD results for different algorithms on the JMRI dataset by varying the numbers of machines.

9 Conclusion and Future Works

This study aimed to compare and evaluate various test optimization algorithms with the goals of reducing feedback time, enhancing failure detection, and optimizing test execution time. We assessed these algorithms across diverse sets of execution resources and datasets. Through a detailed analysis of the strengths, limitations, and trade-offs of each technique, our study offers a comprehensive understanding that assists practitioners in crafting and refining their testing processes. This nuanced perspective enables the customization of testing strategies to suit specific project contexts, thereby maximizing efficiency and effectiveness. Our results indicate that the batching algorithm performs best in a highly resource-constrained environment with a substantial number of concurrent builds, achieving significant reductions in feedback time, resource usage, and test execution time while also improving failure detection without missing any failures. Test selection performs better in environments with fewer concurrent builds, but it comes at the cost of missing some failures. Test prioritization algorithms generally underperformed compared to both batching and test selection but should be considered in rare cases where batching is ineffective, and missing failures is not acceptable. Practitioners can choose the appropriate test optimization algorithm based on computational resource availability, tolerance for missing failures, and the number of concurrent builds. In resource-constrained environments with many concurrent builds, the batching algorithm is the optimal choice. In contrast, in resource-rich environments with fewer concurrent builds, test selection may yield slightly better results. Given the underperformance of test prioritization algorithms, they should not be the primary choice for test optimization. Although we made a concerted effort to include a variety of large-scale projects, different test op-

timization techniques, and various evaluation metrics, this study has its limitations in these areas. Future research could explore other types of software, additional test optimization algorithms, and other evaluation metrics to provide a more comprehensive analysis. 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.

Data Availability A replication package for this paper is available at https://github.com/emadfallahzadeh/ContrastTestOptimizations.

Conflict of interest All authors certify that they have no affiliations with or involvement in any organization or entity with any financial interest or non-financial interest in the subject or materials discussed in this manuscript.

Appendix A Statistical Tests

This appendix presents the statistical tests conducted to compare the distributions of various evaluation metrics across different datasets. Tables 7 to 14 provide the results of pairwise comparisons, including the P-values and Cliff's Delta effect sizes. These tables allow for the comparison of feedback time distributions among different approaches for each dataset and varying numbers of machines. Additionally, Tables 15 to 22 present similar pairwise comparisons for the GainedTime distributions across different approaches for each dataset and machine configuration.

Table 7: P-values for pairwise comparison of the feedback time distribution of different test optimization algorithms for Chrome at different numbers of machines.

CPUs	CP* vs CS	CP vs BA	CS vs BA	MP vs CP	MP vs CS	MP vs BA	TA vs CP	TA vs CS	TA vs BA	TA vs MP
1	0	0	0	0.98479	0	0	0.98479	0	0	1
2	0	0	0	0.984787	0	0	0.984787	0	0	1
4	0	0	0	0.941774	0	0	0.941774	0	0	1
8	0	0	0	0.992355	0	0	0.992355	0	0	1
16	0	0	0	0.97506	0	0	0.97506	0	0	1
25	0	0	0	0.935696	0	0	0.935696	0	0	1
50	0	0	0	0.994567	0	0	0.994567	0	0	1
75	0	0	1E-20	0.911593	0	0	0.911593	0	0	1
100	0	0	0	0.842338	0	0	0.842338	0	0	1
125	0	0	0	0.911593	0	0	0.323964	0	0	1
150	0	0	0	1.89E-12	0	0	0.064336	0	0	1
175	0	0	0	0.00027	0	0	0.00027	0	0	1
200	0	0	0	5.69E-10	0	0	5.69E-10	0	0	1
225	0	0	0	6E-15	0	0	6E-15	0	0	1
250	0	0	0	1E-20	0	0	1E-20	0	0	1
275	0	0	0	0	0	0	0	0	0	1
300	0	0	0	0	0	0	0	0	0	1
325	0	0	0	0	0	0	0	0	0	1
350	0	0	0	0	0	0	0	0	0	1
375	0	0	0	0	0	0	0	0	0	1
400	0	1.42E-16	0	0	0	0	0	0	0	1

*Abbreviations: TA: *TestAll* CP: *CoDynaQPrioritization*, CS: *CoDynaQSelection*, MP: *MLPrioritization*, BA: *BatchAll*.

Table 8: 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.

CPUs	CP* vs CS	CP vs BA	CS vs BA	MP vs CP	MP vs CS	MP vs BA	TA vs CP	TA vs CS	TA vs BA	TA vs MP
1	0.669023	0.996497	0.98772	0.000182	0.669296	0.996771	0.000182	0.669296	0.996771	0
2	0.675713	0.999174	0.997519	0.000182	0.675986	0.999448	0.000182	0.675986	0.999448	0
4	0.688125	0.998422	0.984402	0.000698	0.688739	0.999518	0.000698	0.688739	0.999518	0
8	0.706818	0.998581	0.962966	-9.2E-05	0.706822	0.99888	-9.2E-05	0.706822	0.99888	0
16	0.75105	0.982282	0.943112	0.000299	0.751347	0.98901	0.000299	0.751347	0.98901	0
25	0.815469	0.968267	0.93275	0.000771	0.816132	0.978438	0.000771	0.816132	0.978438	0
50	0.931987	0.955846	0.498563	6.51E-05	0.932567	0.962714	6.51E-05	0.932567	0.962714	0
75	0.960958	0.945547	-0.08969	0.001061	0.962656	0.95427	0.001061	0.962656	0.95427	0
100	0.947478	0.883652	-0.46879	0.0019	0.95304	0.907308	0.0019	0.95304	0.907308	0
125	0.921641	0.741251	-0.68416	0.009422	0.934917	0.801729	0.009422	0.934917	0.801729	0
150	0.922936	0.666033	-0.76835	-0.13343	0.910617	0.590958	0.01767	0.936875	0.739246	0
175	0.910099	0.555798	-0.79711	0.034798	0.928554	0.664883	0.034798	0.928554	0.664883	0
200	0.898798	0.438799	-0.81805	0.059211	0.921712	0.585709	0.059211	0.921712	0.585709	0
225	0.893508	0.36023	-0.83416	0.074544	0.917737	0.521319	0.074544	0.917737	0.521319	0
250	0.892155	0.303235	-0.84618	0.088959	0.915567	0.47406	0.088959	0.915567	0.47406	0
275	0.887479	0.240981	-0.853	0.111049	0.911448	0.42524	0.111049	0.911448	0.42524	0
300	0.884227	0.190371	-0.85794	0.131509	0.908903	0.386352	0.131509	0.908903	0.386352	0
325	0.880944	0.146785	-0.86134	0.147072	0.906212	0.351464	0.147072	0.906212	0.351464	0
350	0.878711	0.111637	-0.86418	0.165802	0.905009	0.327859	0.165802	0.905009	0.327859	0
375	0.87832	0.092492	-0.86688	0.16668	0.903142	0.302158	0.16668	0.903142	0.302158	0
400	0.878464	0.078927	-0.86881	0.168428	0.902245	0.285189	0.168428	0.902245	0.285189	0

*Abbreviations: TA: *TestAll* CP: *CoDynaQPrioritization*, CS: *CoDynaQSelection*, MP: *MLPrioritization*, BA: *BatchAll*.

Table 9: P-values for pairwise comparison of the feedback time distribution of different test optimization algorithms for *GooglePre* at different numbers of machines.

CPUs	CP* vs CS	CP vs BA	CS vs BA	MP vs CP	MP vs CS	MP vs BA	TA vs CP	TA vs CS	TA vs BA	TA vs MP
1	0	0	0	0.978318	0	0	0.978318	0	0	1
10	0	0	0	0.939368	0	0	0.939368	0	0	1
20	0	0	0	0.958698	0	0	0.958698	0	0	1
30	0	0	0	0.87921	0	0	0.87921	0	0	1
40	0	0	0	0.553443	0	0	0.553443	0	0	1
50	3.91E-09	0	0	0.448603	1.47E-11	0	0.448603	1.47E-11	0	1
60	2.4E-05	0	0	0.425441	3.32E-07	0	0.425441	3.32E-07	0	1
70	0.000877	0	0	0.320929	1.2E-05	0	0.320929	1.2E-05	0	1
80	0.022166	0	0	0.153844	0.000173	0	0.153844	0.000173	0	1
90	0.070165	0	0	0.12686	0.000753	0	0.12686	0.000753	0	1
100	0.148683	0	0	0.105089	0.002047	0	0.105089	0.002047	0	1

*Abbreviations: TA: *TestAll* CP: *CoDynaQPrioritization*, CS: *CoDynaQSelection*, MP: *MLPrioritization*, BA: *BatchAll*.

Table 10: Cliff's Delta effect size for pairwise comparison of the feedback time distribution of different test optimization algorithms for *GooglePre* at different numbers of machines.

CPUs	CP* vs CS	CP vs BA	CS vs BA	MP vs CP	MP vs CS	MP vs BA	TA vs CP	TA vs CS	TA vs BA	TA vs MP
1	0.243162	0.81174	0.775088	-0.00054	0.24292	0.811443	-0.00054	0.24292	0.811443	0
10	0.466472	0.965827	0.960636	-0.00151	0.465367	0.965704	-0.00151	0.465367	0.965704	0
20	0.358129	0.864186	0.731104	-0.00103	0.357912	0.866838	-0.00103	0.357912	0.866838	0
30	0.283577	0.685586	0.52928	0.003016	0.289093	0.698039	0.003016	0.289093	0.698039	0
40	0.223489	0.545468	0.392185	0.011762	0.237121	0.564023	0.011762	0.237121	0.564023	0
50	0.116853	0.406163	0.311175	0.015039	0.13398	0.427827	0.015039	0.13398	0.427827	0
60	0.083833	0.344001	0.271504	0.015819	0.101301	0.364924	0.015819	0.101301	0.364924	0
70	0.066036	0.307176	0.245965	0.019699	0.08689	0.331231	0.019699	0.08689	0.331231	0
80	0.0454	0.265889	0.220764	0.028304	0.074534	0.298126	0.028304	0.074534	0.298126	0
90	0.03594	0.241277	0.202614	0.030298	0.066876	0.274763	0.030298	0.066876	0.274763	0
100	0.028663	0.223258	0.192085	0.032165	0.061193	0.258677	0.032165	0.061193	0.258677	0

*Abbreviations: TA: *TestAll* CP: *CoDynaQPrioritization*, CS: *CoDynaQSelection*, MP: *MLPrioritization*, BA: *BatchAll*.

Table 11: P-values for pairwise comparison of the feedback time distribution of different test optimization algorithms for *GooglePost* at different numbers of machines.

CPUs	CP* vs CS	CP vs BA	CS vs BA	MP vs CP	MP vs CS	MP vs BA	TA vs CP	TA vs CS	TA vs BA	TA vs MP
1	0	0	0	0.982181	0	0	0.982181	0	0	1
10	0	0	0	0.998502	0	0	0.998502	0	0	1
25	0	0	0	0.992659	0	0	0.992659	0	0	1
50	0	0	0	0.950792	0	0	0.950792	0	0	1
75	0	0	0	0.730474	0	0	0.730474	0	0	1
100	0	0	6.57E-11	0.096969	0	0	0.096969	0	0	1
125	1.51E-08	2.3E-18	0.000114	0.000241	0	0	0.000241	0	0	1
150	3.84E-05	1.62E-10	0.005531	0.000364	6.01E-15	0	0.000364	6.01E-15	0	1
175	0.000444	2.82E-08	0.010487	0.000267	5.19E-13	2E-20	0.000267	5.19E-13	2E-20	1
200	0.001031	4.64E-06	0.058796	0.000199	2.43E-12	7.44E-17	0.000199	2.43E-12	7.44E-17	1
225	0.00095	4.89E-05	0.181293	0.000432	7.26E-12	3.17E-14	0.000432	7.26E-12	3.17E-14	1
250	0.001122	3.23E-05	0.19466	0.000403	8.91E-12	1.25E-14	0.000403	8.91E-12	1.25E-14	1
275	0.000728	2.53E-05	0.249855	0.000833	1.49E-11	4.21E-14	0.000833	1.49E-11	4.21E-14	1
300	0.000541	3.79E-06	0.189579	0.001063	1.29E-11	2.65E-15	0.001063	1.29E-11	2.65E-15	1
325	0.000516	2.21E-06	0.201284	0.00112	1.36E-11	1.28E-15	0.00112	1.36E-11	1.28E-15	1
350	0.000412	2.12E-06	0.243041	0.001155	9.43E-12	1.03E-15	0.001155	9.43E-12	1.03E-15	1
375	0.000148	9.34E-07	0.313979	0.002007	4.54E-12	1.08E-15	0.002007	4.54E-12	1.08E-15	1
400	5.54E-05	8.35E-07	0.463715	0.003027	1.84E-12	2.31E-15	0.003027	1.84E-12	2.31E-15	1

*Abbreviations: TA: *TestAll* CP: *CoDynaQPrioritization*, CS: *CoDynaQSelection*, MP: *MLPrioritization*, BA: *BatchAll*.

Table 12: Cliff's Delta effect size for pairwise comparison of the feedback time distribution of different test optimization algorithms for *GooglePost* at different numbers of machines.

CPUs	CP* vs CS	CP vs BA	CS vs BA	MP vs CP	MP vs CS	MP vs BA	TA vs CP	TA vs CS	TA vs BA	TA vs MP
1	0.169128	0.419346	0.303725	0.000271	0.16943	0.419698	0.000271	0.16943	0.419698	0
10	0.237222	0.586745	0.42845	-2.3E-05	0.237366	0.587062	-2.3E-05	0.237366	0.587062	0
25	0.42317	0.790574	0.563304	-0.00011	0.423251	0.790742	-0.00011	0.423251	0.790742	0
50	0.443186	0.671578	0.402074	0.00075	0.44366	0.672976	0.00075	0.44366	0.672976	0
75	0.267098	0.38625	0.19498	0.004184	0.274176	0.400561	0.004184	0.274176	0.400561	0
100	0.152299	0.195978	0.079312	0.020158	0.177467	0.223465	0.020158	0.177467	0.223465	0
125	0.068745	0.106166	0.04687	0.044596	0.116191	0.152521	0.044596	0.116191	0.152521	0
150	0.050002	0.077654	0.033696	0.043296	0.094778	0.122171	0.043296	0.094778	0.122171	0
175	0.042656	0.06743	0.031084	0.044275	0.087691	0.112375	0.044275	0.087691	0.112375	0
200	0.03986	0.055629	0.022951	0.045189	0.085105	0.101289	0.045189	0.085105	0.101289	0
225	0.040138	0.049321	0.016236	0.042748	0.083224	0.092199	0.042748	0.083224	0.092199	0
250	0.039568	0.050481	0.015751	0.042968	0.082869	0.093647	0.042968	0.082869	0.093647	0
275	0.041037	0.051161	0.013976	0.040583	0.081966	0.091752	0.040583	0.081966	0.091752	0
300	0.04202	0.056144	0.015932	0.039754	0.082224	0.096024	0.039754	0.082224	0.096024	0
325	0.042173	0.057487	0.01552	0.039575	0.082121	0.09712	0.039575	0.082121	0.09712	0
350	0.042902	0.057588	0.014179	0.03947	0.08277	0.097443	0.03947	0.08277	0.097443	0
375	0.046094	0.059572	0.012229	0.037519	0.084036	0.09737	0.037519	0.084036	0.09737	0
400	0.048963	0.059841	0.008899	0.036011	0.085577	0.096232	0.036011	0.085577	0.096232	0

*Abbreviations: TA: *TestAll* CP: *CoDynaQPrioritization*, CS: *CoDynaQSelection*, MP: *MLPrioritization*, BA: *BatchAll*.

Table 13: P-values for pairwise comparison of the feedback time distribution of different test optimization algorithms for *JMRI* at different numbers of machines.

CPUs	CP* vs CS	CP vs BA	CS vs BA	MP vs CP	MP vs CS	MP vs BA	TA vs CP	TA vs CS	TA vs BA	TA vs MP
1	0	0.155051	0	1.58E-13	0	6E-20	1.58E-13	0	6E-20	1
3	0	0.937408	0	0.004722	0	0.003538	0.004722	0	0.003538	1
5	0	0.97489	0	0.057673	0	0.053605	0.057673	0	0.053605	1
7	0	0.958345	0	0.125978	0	0.113723	0.125978	0	0.113723	1
9	0	0.956565	0	0.194514	0	0.17676	0.194514	0	0.17676	1
11	0	1	0	0.233076	0	0.233076	0.233076	0	0.233076	1
13	0	1	0	0.296421	0	0.296421	0.296421	0	0.296421	1
15	0	1	0	0.353246	0	0.353246	0.353246	0	0.353246	1
17	0	1	0	0.39718	0	0.39718	0.39718	0	0.39718	1
19	0	1	0	0.434494	0	0.434494	0.434494	0	0.434494	1

*Abbreviations: TA: *TestAll* CP: *CoDynaQPrioritization*, CS: *CoDynaQSelection*, MP: *MLPrioritization*, BA: *BatchAll*.

Table 14: Cliff's Delta effect size for pairwise comparison of the feedback time distribution of different test optimization algorithms for *JMRI* at different numbers of machines.

CPUs	CP* vs CS	CP vs BA	CS vs BA	MP vs CP	MP vs CS	MP vs BA	TA vs CP	TA vs CS	TA vs BA	TA vs MP
1	0.670594	0.030174	-0.66106	0.156606	0.730444	0.194189	0.156606	0.730444	0.194189	0
3	0.687933	0.001667	-0.68742	0.059956	0.706723	0.061894	0.059956	0.706723	0.061894	0
5	0.694967	0.000668	-0.69478	0.04028	0.707135	0.040956	0.04028	0.707135	0.040956	0
7	0.697596	0.001109	-0.69725	0.032471	0.707267	0.033564	0.032471	0.707267	0.033564	0
9	0.701175	0.001156	-0.7008	0.02753	0.709087	0.028665	0.02753	0.709087	0.028665	0
11	0.70181	0	-0.70181	0.025305	0.709253	0.025305	0.025305	0.709253	0.025305	0
13	0.703186	0	-0.70319	0.022157	0.709826	0.022157	0.022157	0.709826	0.022157	0
15	0.703818	0	-0.70382	0.019699	0.709673	0.019699	0.019699	0.709673	0.019699	0
17	0.704274	0	-0.70427	0.017967	0.709336	0.017967	0.017967	0.709336	0.017967	0
19	0.704784	0	-0.70478	0.016585	0.709402	0.016585	0.016585	0.709402	0.016585	0

*Abbreviations: TA: *TestAll* CP: *CoDynaQPrioritization*, CS: *CoDynaQSelection*, MP: *MLPrioritization*, BA: *BatchAll*.

Table 15: P-values for pairwise comparison of the GainedTime distribution of different test optimization algorithms for Chrome at different numbers of machines.

Machines	CP* vs CS	CP vs BA	CS vs BA	MP vs CP	MP vs CS	MP vs BA
1	0	0	2.78E-11	0	0	0
2	0	0	7.65E-11	0	0	0
4	0	0	6.19E-10	0	0	0
8	0	0	1.19E-08	0	0	0
16	0	0	2.94E-06	0	0	0
25	0	0	0.000877	0	0	0
50	0	0	0.554687	0	0	0
75	0	0	0.919794	0	0	0
100	0	0	0.943727	0	0	0
125	0	0	0.766059	0	0	0
150	0	0	0.878127	0	0	0
175	0	0	0.40057	0	0	0
200	0	3.49E-16	0.096645	0	0	0
225	4.92E-16	1.15E-07	0.010419	0	0	0
250	8.75E-12	0.001152	0.000803	0	0	0.00000454
275	0.001167	0.074664	3.25E-07	0	0	0.00006555
300	0.006187	0.000432	1.56E-10	0	0	0.00504651
325	0.186533	5.91E-07	8.27E-13	0	0	0.00785462
350	0.856054	1.14E-10	1.82E-15	0	0	0.00985162
375	0.908756	5.23E-14	2.02E-18	0	1.16E-16	0.02564895
400	0.584217	5.76E-16	2E-20	0	3.83E-14	0.04480623

*Abbreviations: CP: *CoDynaQPrioritization*, CS: *CoDynaQSelection*, MP: *MLPrioritization*, BA: *BatchAll*.

Table 16: Cliff's Delta effect sizes for pairwise comparison of the *Gained Time* distribution of different test optimization algorithms for Chrome at different numbers of machines.

Machines	CP* vs CS	CP vs BA	CS vs BA	MP vs CP	MP vs CS	MP vs BA
1	-1	-1	-0.32173	-0.60732	-1	-1
2	-1	-1	-0.31447	-0.61035	-1	-1
4	-1	-1	-0.29891	-1	-1	-1
8	-1	-1	-0.27554	-0.62201	-1	-1
16	-0.99929	-0.99616	-0.22593	-0.93599	-1	-1
25	-0.98868	-0.98601	-0.16079	-0.94743	-1	-0.99789721
50	-0.9868	-0.98636	-0.02856	-0.62727	-0.99359	-0.98882586
75	-0.97269	-0.9698	-0.00488	-0.65124	-0.98544	-0.97258456
100	-0.93679	-0.92904	0.003423	-0.69121	-0.96853	-0.95584136
125	-0.81407	-0.78344	0.014389	-0.71546	-0.95839	-0.94855123
150	-0.9027	-0.89098	0.008981	-0.74162	-0.95382	-0.94041545
175	-0.6247	-0.55687	0.040625	-0.70547	-0.81564	-7.8219E+11
200	-0.49655	-0.39388	0.080248	-0.63962	-0.75271	-0.56719155
225	-0.39207	-0.25586	0.123612	-0.61245	-0.69564	-0.42845129
250	-0.32984	-0.15678	0.161622	-0.60484	-0.61322	-0.36213848
275	-0.1569	0.085897	0.246039	-0.57232	-0.59232	-0.21584513
300	-0.1323	0.169397	0.307802	-0.53448	-0.52058	-0.13488435
325	-0.06384	0.240012	0.34366	-0.51841	-0.49232	-0.11894531
350	-0.00878	0.309575	0.381498	-0.50123	-0.45214	-0.10812385
375	0.00555	0.361069	0.419605	-0.49531	-0.39	-4.5646E-06
400	0.026456	0.387818	0.443176	-0.48506	-0.36565	0.096056042

*Abbreviations: CP: *CoDynaQPrioritization*, CS: *CoDynaQSelection*, MP: *MLPrioritization*, BA: *BatchAll*.

Table 17: P-values for pairwise comparison of the GainedTime distribution of different test optimization algorithms for *GooglePre* at different numbers of machines.

Machines	CP* vs CS	CP vs BA	CS vs BA	MP vs CP	MP vs CS	MP vs BA
1	0	0	0	1.7E-09	0	0
10	0	0	0	4.71E-09	0	0
20	0	0	0	3.31E-09	0	0
30	0	0	0	1.11E-08	0	0
40	0	0	3.63E-17	5.81E-09	0	0
50	3E-20	0	7.93E-10	1.81E-09	0	0
60	3.06E-13	0	5.88E-07	2.28E-08	0	0
70	2.28E-08	1E-20	2.87E-05	1.47E-07	0	0
80	0.001753	9.63E-12	6.51E-05	5.16E-06	5.33E-15	0
90	0.027968	2.43E-09	5.85E-05	0.001173	2.24E-08	1E-20
100	0.229486	9.69E-07	0.000125	0.007699	5.08E-05	3.63E-16

*Abbreviations: CP: *CoDynaQPrioritization*, CS: *CoDynaQSelection*, MP: *MLPrioritization*, BA: *BatchAll*.

Table 18: Cliff's Delta effect sizes for pairwise comparison of the GainedTime distribution of different test optimization algorithms for *GooglePre* at different numbers of machines.

Machines	CP* vs CS	CP vs BA	CS vs BA	MP vs CP	MP vs CS	MP vs BA
1	-0.91579	-0.97832	-0.68068	-0.12501	-0.93576	-0.98438
10	-0.89411	-0.96718	-0.64091	-0.12155	-0.91949	-0.9763
20	-0.78198	-0.85856	-0.40362	-0.12276	-0.8346	-0.89235
30	-0.56891	-0.66081	-0.22002	-0.11857	-0.65294	-0.7257
40	-0.36988	-0.47099	-0.1749	-0.12082	-0.47891	-0.567
50	-0.1917	-0.29174	-0.1276	-0.12481	-0.31078	-0.40559
60	-0.15135	-0.2374	-0.10369	-0.11599	-0.26375	-0.34783
70	-0.116	-0.19318	-0.08684	-0.10908	-0.22595	-0.30286
80	-0.06494	-0.1414	-0.08289	-0.09459	-0.16225	-0.23817
90	-0.04561	-0.12383	-0.08341	-0.06735	-0.11604	-0.19465
100	-0.02494	-0.10166	-0.07963	-0.0553	-0.08407	-0.16915

*Abbreviations: CP: *CoDynaQPrioritization*, CS: *CoDynaQSelection*, MP: *MLPrioritization*, BA: *BatchAll*.

Table 19: P-values for pairwise comparison of the GainedTime distribution of different test optimization algorithms for *GooglePost* at different numbers of machines.

Machines	CP* vs CS	CP vs BA	CS vs BA	MP vs CP	MP vs CS	MP vs BA
1	0	0	0	1.87E-17	0	0
10	0	0	0	3.82E-18	0	0
25	0	0	0	5.41E-17	0	0
50	0	0	0	0.604562	0	0
75	0	0	0.013733	0	0	0
100	0	0	0	0	0	0
125	0.00089	0	0	0	0	0
150	0.747586	0	0	0	0	0
175	0.185964	0	0	0	0	0
200	0.372207	0	0	0	0	0
225	0.296089	0	0	0	0	0
250	0.278252	0	0	0	0	0
275	0.427549	0	0	0	0.005415	0
300	0.300333	0	0	0	0.000874	0
325	0.267384	0	0	0.015647	0.023485	0
350	0.269513	0	1E-20	0.125464	0.218456	0
375	0.254679	1.55E-16	5.68E-12	0.214896	0.235649	1.62E-16
400	0.22929	9.1E-16	4.05E-11	0.601545	0.215648	7.22E-16

*Abbreviations: CP: *CoDynaQPrioritization*, CS: *CoDynaQSelection*, MP: *MLPrioritization*, BA: *BatchAll*.

Table 20: Cliff's Delta effect sizes for pairwise comparison of the GainedTime distribution of different test optimization algorithms for *GooglePost* at different numbers of machines.

Machines	CP* vs CS	CP vs BA	CS vs BA	MP vs CP	MP vs CS	MP vs BA
1	-0.99691	-0.98085	-0.78185	0.098486	-0.99673	-0.98076
10	-0.99691	-0.98426	-0.78967	0.1006	-0.99672	-0.98415
25	-0.98825	-0.9819	-0.64638	0.09705	-0.98672	-0.9812
50	-0.89651	-0.82749	-0.23656	-0.006	-0.89878	-0.82967
75	-0.65157	-0.49314	0.028546	0.123743	-0.60169	-0.45192
100	-0.34127	0.15727	0.403375	0.125687	-0.24618	0.241087
125	-0.03848	0.286044	0.317881	0.17572	0.145608	0.44234
150	0.003726	0.267496	0.266816	0.202174	0.210681	0.463439
175	-0.01531	0.186382	0.203761	0.23475	0.225491	0.450819
200	0.01033	0.172321	0.163358	0.224763	0.235122	0.447595
225	0.012094	0.162849	0.151446	0.235457	0.238741	0.417849
250	0.01255	0.153788	0.141439	0.235465	0.242165	0.392316
275	0.009183	0.15393	0.143404	0.245456	0.258451	0.382156
300	0.011987	0.142502	0.128535	0.213132	0.282136	0.405641
325	0.012836	0.13206	0.1177	0.255133	0.242315	0.385125
350	0.012779	0.12204	0.108103	0.261223	0.228456	0.375412
375	0.013182	0.095573	0.079755	0.172131	0.238749	0.395431
400	0.013913	0.09309	0.076451	0.151875	0.258743	0.382156

*Abbreviations: CP: *CoDynaQPrioritization*, CS: *CoDynaQSelection*, MP: *MLPrioritization*, BA: *BatchAll*.

Table 21: P-values for pairwise comparison of the GainedTime distribution of different test optimization algorithms for *JMRI* at different numbers of machines.

Machines	CP* vs CS	CP vs BA	CS vs BA	MP vs CP	MP vs CS	MP vs BA
1	0.003006	4.73E-07	1E-20	0.004542	1.11E-07	0.000116
3	0.042738	4.47E-08	0	0.152402	0.00673	7.43E-06
5	0.028089	2.41E-08	0	0.000804	4.23E-06	0.003307
7	0.035722	3.37E-09	0	0.604891	0.075351	2.77E-11
9	0.04798	7.69E-10	0	0.40303	0.021144	2.35E-06
11	0.048226	2.88E-10	0	0.206654	0.00504	0.000376
13	0.048226	2.88E-10	0	0.908921	0.208151	5.53E-12
15	0.048226	2.88E-10	0	0.002448	4.83E-05	6.31E-05
17	0.048226	2.88E-10	0	0.000524	1.43E-05	0.000376
19	0.048226	2.88E-10	0	0.912432	0.200779	2.58E-11

*Abbreviations: CP: *CoDynaQPrioritization*, CS: *CoDynaQSelection*, MP: *MLPrioritization*, BA: *BatchAll*.

Table 22: Cliff's Delta effect sizes for pairwise comparison of the GainedTime distribution of different test optimization algorithms for *JMRI* at different numbers of machines.

Machines	CP* vs CS	CP vs BA	CS vs BA	MP vs CP	MP vs CS	MP vs BA
1	-0.20515	0.342806	0.634286	-0.19622	-0.36694	0.261939
3	-0.1401	0.357908	0.737449	-0.09898	-0.18735	0.291939
5	-0.15184	0.36301	0.75898	-0.23173	-0.31796	0.190306
7	-0.1452	0.384082	0.783061	-0.03582	-0.12296	0.430612
9	-0.13673	0.399592	0.79449	-0.05786	-0.15939	0.305306
11	-0.13658	0.407143	0.807143	-0.08735	-0.19388	0.228571
13	-0.13658	0.407143	0.807143	-0.00796	-0.08704	0.442857
15	-0.13658	0.407143	0.807143	-0.20949	-0.28092	0.257143
17	-0.13658	0.407143	0.807143	-0.2398	-0.2999	0.228571
19	-0.13658	0.407143	0.807143	-0.00765	-0.08847	0.428571

*Abbreviations: CP: *CoDynaQPrioritization*, CS: *CoDynaQSelection*, MP: *MLPrioritization*, BA: *BatchAll*.

References

Alshammari A, Morris C, Hilton M, Bell J (2021) FlakeFlagger: Predicting Flakiness Without Rerunning Tests. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp 1572–1584, DOI 10.1109/ICSE43902.2021.00140, iSSN: 1558-1225

Anderson J, Salem S, Do H (2014) Improving the effectiveness of test suite through mining historical data. In: Proceedings of the 11th Working Conference on Mining Software Repositories, Association for Computing Machinery, New York, NY, USA, MSR 2014, pp 142–151, DOI 10.1145/2597073.2597084, URL https://doi.org/10.1145/2597073.2597084

Arabnejad H, Bispo J, Barbosa JG, Cardoso JM (2018) 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, pp 13–19

Bagherzadeh M, Kahani N, Briand L (2022) Reinforcement Learning for Test Case Prioritization. IEEE Transactions on Software Engineering 48(8):2836–2856, DOI 10.1109/TSE.2021.3070549, conference Name: IEEE Transactions on Software Engineering

Bagies TOS (2020) Parallelizing unit test execution on gpu. PhD thesis, Iowa State University

Bavand AH, Rigby PC (2021) Mining Historical Test Failures to Dynamically Batch Tests to Save CI Resources. In: 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp 217–226, DOI 10.1109/ICSME52107.2021.00026, iSSN: 2576-3148

Beheshtian MJ, Bavand A, Rigby P (2021) Software batch testing to save build test resources and to reduce feedback time. IEEE Transactions on Software Engineering pp 1–1, DOI 10.1109/TSE.2021.3070269

Bell J, Kaiser G, Melski E, Dattatreya M (2015) Efficient dependency detection for safe java test acceleration. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2015, p 770–781, DOI 10.1145/2786805.2786823, URL https://doi.org/10.1145/2786805.2786823

Bell J, Legunsen O, Hilton M, Eloussi L, Yung T, Marinov D (2018) DeFlaker: Automatically Detecting Flaky Tests. In: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), pp 433–444, DOI 10.1145/3180155.3180164, iSSN: 1558-1225

Beller M, Gousios G, Zaidman A (2017a) 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), pp 356–367, DOI 10.1109/MSR.2017.62

Beller M, Gousios G, Zaidman A (2017b) TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), pp 447–450, DOI 10.1109/MSR.2017.24

Candido J, Melo L, d'Amorim M (2017) 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, pp 838–848

Chang F, Ren J, Viswanathan R (2009) Optimal Resource Allocation for Batch Testing. In: 2009 International Conference on Software Testing Verification and Validation, pp 91–100, DOI 10.1109/ICST.2009.25, iSSN: 2159-4848

Cho C, Chun B, Seo J (2017) Adaptive Batching Scheme for Real-Time Data Transfers in IoT Environment. In: Proceedings of the 2017 International Conference on Cloud and Big Data Computing, Association for Computing Machinery, New York, NY, USA, ICCBDC 2017, pp 55–59, DOI 10.1145/3141128.3141145, URL https://doi.org/10.1145/3141128.3141145

Ding C, Shen X, Kelsey K, Tice C, Huang R, Zhang C (2007) Software behavior oriented parallelization. ACM SIGPlan Notices 42(6):223–234

Elbaum S, Rothermel G, Penix J (2014) Techniques for Improving Regression Testing in Continuous Integration Development Environments. In: Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, New York, NY, USA, FSE 2014, pp 235–245, DOI 10.1145/2635868.2635910, URL http://doi.acm.org/10.1145/2635868.2635910, event-place: Hong Kong, China

Fallahzadeh E, Rigby PC (2022) The Impact of Flaky Tests on Historical Test Prioritization on Chrome. In: 2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), pp 273–282, DOI 10.1109/ICSE-SEIP55303.2022.9793941

Fallahzadeh E, Bavand AH, Rigby PC (2023) 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), ACM, New York, NY, USA, p 13, accepted, to be presented at ACM ESEC/FSE 2023

Greca R, Miranda B, Bertolino A (2023) State of Practical Applicability of Regression Testing Research: A Live Systematic Literature Review. ACM Comput Surv DOI 10.1145/3579851, URL https://dl.acm.org/doi/10.1145/3579851

Hemmati H, Sharifi F (2018) Investigating NLP-Based Approaches for Predicting Manual Test Case Failure. In: 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST), pp 309–319, DOI 10.1109/ICST.2018.00038

Henard C, Papadakis M, Harman M, Jia Y, Le Traon Y (2016) Comparing White-Box and Black-Box Test Prioritization. In: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), pp 523–534, DOI 10.1145/2884781.2884791, iSSN: 1558-1225

Herzig K, Greiler M, Czerwonka J, Murphy B (2015) The Art of Testing Less without Sacrificing Quality. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol 1, pp 483–493, DOI 10.1109/ICSE.2015.66, iSSN: 1558-1225

Herzig K, Czerwonka J, Murphy B, Greiler M (2016) Selecting tests for execution on a software product. URL https://patents.google.com/patent/US20160321586A1/en

Hilton M (2016) Understanding and improving continuous integration. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp 1066–1067

Hilton M, Bell J, Marinov D (2018) A large-scale study of test coverage evolution. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, Association for Computing Machinery, New York, NY, USA, ASE 2018, pp 53–63, DOI 10.1145/3238147.3238183, URL https://doi.org/10.1145/3238147.3238183

Jahan H, Feng Z, Mahmud SMH (2020) Risk-Based Test Case Prioritization by Correlating System Methods and Their Associated Risks. Arab J Sci Eng 45(8):6125–6138, DOI 10.1007/s13369-020-04472-z, URL https://doi.org/10.1007/s13369-020-04472-z

Jiang B, Zhang Z, Chan WK, Tse TH (2009) Adaptive Random Test Case Prioritization. In: 2009 IEEE/ACM International Conference on Automated Software Engineering, pp 233–244, DOI 10.1109/ASE.2009.77, iSSN: 1938-4300

Jones JA, Harrold MJ, Stasko J (2002) Visualization of test information to assist fault localization. In: Proceedings of the 24th International Conference on Software Engineering, Association for Computing Machinery, New York, NY, USA, ICSE '02, p 467–477, DOI 10.1145/581339.581397, URL https://doi.org/10.1145/581339.581397

Kim JM, Porter A (2002) 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, pp 119–129, DOI 10.1109/ICSE.2002.1007961

Labuschagne A, Inozemtseva L, Holmes R (2017) 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, Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2017, p 821–830, DOI 10.1145/3106237.3106288, URL https://doi.org/10.1145/3106237.3106288

Lachmann R, Schulze S, Nieke M, Seidl C, Schaefer I (2016) System-Level Test Case Prioritization Using Machine Learning. In: 2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA), pp 361–368, DOI 10.1109/ICMLA.2016.0065, URL https://ieeexplore.ieee.org/abstract/document/7838169

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

Landing C, Tahvili S, Haggren H, Langkvis M, Muhammad A, Loufi A (2020) Cluster-based parallel testing using semantic analysis. In: 2020 IEEE International Conference On Artificial Intelligence Testing (AITest), pp 99–106, DOI 10.1109/AITEST49225.2020.00022

Leppänen M, Mäkinen S, Pagels M, Eloranta VP, Itkonen J, Mäntylä MV, Männistö T (2015) The highways and country roads to continuous deployment. Ieee software

32(2):64–72

Li Z, Harman M, Hierons RM (2007) Search Algorithms for Regression Test Case Prioritization. IEEE Transactions on Software Engineering 33(4):225–237, DOI 10.1109/TSE.2007.38, conference Name: IEEE Transactions on Software Engineering

Liang J, Elbaum S, Rothermel G (2018) Redefining Prioritization: Continuous Prioritization for Continuous Integration. In: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), DOI 10.1145/3180155.3180213, iSSN: 1558-1225

Lu Y, Lou Y, Cheng S, Zhang L, Hao D, Zhou Y, Zhang L (2016) How does regression test prioritization perform in real-world software evolution? In: Proceedings of the 38th International Conference on Software Engineering, Association for Computing Machinery, New York, NY, USA, ICSE '16, pp 535–546, DOI 10.1145/2884781.2884874, URL https://doi.org/10.1145/2884781.2884874

Luo Q, Moran K, Poshyvanyk D (2016) 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, Association for Computing Machinery, New York, NY, USA, FSE 2016, pp 559–570, DOI 10.1145/2950290.2950344, URL https://doi.org/10.1145/2950290.2950344

Marijan D, Gotlieb A, Sen S (2013) Test Case Prioritization for Continuous Regression Testing: An Industrial Case Study. In: 2013 IEEE International Conference on Software Maintenance, pp 540–543, DOI 10.1109/ICSM.2013.91, iSSN: 1063-6773

Mattis T, Rein P, Dürsch F, Hirschfeld R (2020) RTPTorrent: An Open-source Dataset for Evaluating Regression Test Prioritization. In: Proceedings of the 17th International Conference on Mining Software Repositories, Association for Computing Machinery, New York, NY, USA, MSR '20, pp 385–396, DOI 10.1145/3379597.3387458, URL https://doi.org/10.1145/3379597.3387458

Memon A, Gao Z, Nguyen B, Dhanda S, Nickell E, Siemborski R, Micco J (2017) Taming Google-scale continuous testing. In: 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), pp 233–242, DOI 10.1109/ICSE-SEIP.2017.16

Misailovic S, Milicevic A, Petrovic N, Khurshid S, Marinov D (2007) 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, Association for Computing Machinery, New York, NY, USA, ESEC-FSE '07, p 135–144, DOI 10.1145/1287624.1287645, URL https://doi-org.lib-ezproxy.concordia.ca/10.1145/1287624.1287645

Najafi A, Rigby PC, Shang W (2019a) 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, Association for Computing Machinery, Tallinn, Estonia, ESEC/FSE 2019, pp 279–289, DOI 10.1145/3338906.3338944, URL https://doi.org/10.1145/3338906.3338944

Najafi A, Shang W, Rigby PC (2019b) 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), pp 213–222, DOI 10.1109/ICSE-SEIP.2019.00031

Parthasarathy G, Rushdi A, Choudhary P, Nanda S, Evans M, Gunasekara H, Rajakumar S (2022) RTL Regression Test Selection using Machine Learning. In: 2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC), pp 281–287, DOI 10.1109/ASP-DAC52403.2022.9712550, iSSN: 2153-697X

Peng Q, Shi A, Zhang L (2020) Empirically revisiting and enhancing IR-based test-case prioritization. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Association for Computing Machinery, New York, NY, USA, ISSTA 2020, pp 324–336, DOI 10.1145/3395363.3397383, URL http://doi.org/10.1145/3395363.3397383

Poth A, Werner M, Lei X (2018) How to deliver faster with ci/cd integrated testing services? In: European Conference on Software Process Improvement, Springer, pp 401–409

Rothermel G, Untch R, Chu C, Harrold M (1999) 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), pp 179–188, DOI 10.1109/ICSM.1999.792604, iSSN: 1063-6773

Sharif A, Marijan D, Liaaen M (2021) DeepOrder: Deep Learning for Test Case Prioritization in Continuous Integration Testing. In: 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp 525–534, DOI 10.1109/ICSME52107.2021.00053, iSSN: 2576-3148

Shashban (2022) Run VSTest tests in parallel - Azure Pipelines. URL https://learn.microsoft.com/en-us/azure/devops/pipelines/test/parallel-testing-vstest?view=azure-devops

Shi A, Gyori A, Mahmood S, Zhao P, Marinov D (2018) Evaluating test-suite reduction in real software evolution. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, Association for Computing Machinery, New York, NY, USA, ISSTA 2018, pp 84–94, DOI 10.1145/3213846.3213875, URL https://doi.org/10.1145/3213846.3213875

Soni M (2015) 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, pp 85–89

Wang Z, Fang C, Chen L, Zhang Z (2020) A Revisit of Metrics for Test Case Prioritization Problems. Int J Soft Eng Knowl Eng 30(08):1139–1167, DOI 10.1142/S0218194020500291, URL https://www.worldscientific.com/doi/abs/10.1142/S0218194020500291

Yaraghi AS, Bagherzadeh M, Kahani N, Briand LC (2023) Scalable and Accurate Test Case Prioritization in Continuous Integration Contexts. IEEE Transactions on Software Engineering 49(4):1615–1639, DOI 10.1109/TSE.2022.3184842, conference Name: IEEE Transactions on Software Engineering

Zhang L, Hao D, Zhang L, Rothermel G, Mei H (2013) Bridging the gap between the total and additional test-case prioritization strategies. In: 2013 35th International

Conference on Software Engineering (ICSE), pp 192–201, DOI 10.1109/ICSE.2013.6606565, iSSN: 1558-1225

Zhu Y, Shihab E, Rigby PC (2018) Test Re-Prioritization in Continuous Testing Environments. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp 69–79, DOI 10.1109/ICSME.2018.00016, iSSN: 1063-6773