The Impact of Parallel and Batch Testing in Continuous Integration Environments

Amir Hossein Bavand

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Master of Computer Science (Computer Science) at

Concordia University

Montréal, Québec, Canada

August 2021

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Amir Hossein Bavand**

Entitled: **The Impact of Parallel and Batch Testing in Continuous Integration Environments**

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

Master of Computer Science (Computer Science)

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

Signed by the Final Examining Committee:

_____ Chair
Dr. Joey Paquet

_____ Examiner
Dr. Joey Paquet

_____ Examiner
Dr. Weiyi Shang

_____ Supervisor
Dr. Peter C. Rigby

Approved by _____
Dr. Lata Narayanan, Chair
Department of Computer Science and Software Engineering

September 2021

_____
Dr. Mourad Debbabi, Dean
Gina Cody School of Engineering and Computer Science

Abstract

The Impact of Parallel and Batch Testing in Continuous Integration Environments

Amir Hossein Bavand

Testing is a costly, time-consuming, and challenging part of modern software development. During continuous integration, after submitting each change, it is tested automatically to ensure that it does not break the system's functionality. A common approach to reducing the number of test case executions is to batch changes together for testing. For example, given four changes to test, if we group them in a batch and they pass we use one execution to test all four changes. However, if they fail, additional executions are required to find the culprit change that is responsible for the failure.

In this study we first investigate the impact of batch testing in the level of the builds. We evaluate five batch culprit finding approaches: Dorfman, double pool testing, BatchBisect, BatchStop4, and our novel BatchDivide4.

All prior works on batching use a constant batch size. In this work, we propose a dynamic batch size technique based on the weighted historical failure rate of the project.

We simulate each of the batching strategies across 12 large projects on Travis with varying failures rate. We find that dynamic batching coupled with BatchDivide4 outperforms the other approaches. Compared to TestAll, this approach decreases the number of executions by 47.49% on average across the Travis projects. It outperforms the current state-of-the-art constant batch size approach, *i.e.* Batch4 by 5.17 percentage points.

Our historical weighting approach leads us to a metric that describes the number of consecutive build failures. We find that the correlation between batch savings and FailureSpread is $r = -0.97$

with a $p \ll 0.0001$. This metric easily allows developers to determine the potential of batching on their project.

However, we then show that in the case of failure of a batch, re-running all the test cases is inefficient. Also, for companies with notable resource constraints, *e.g.,* Ericsson, running all the tests in a single machine is not possible and realistic. To address this issues we extend our work to an industrial application at Ericsson.

We first evaluate the effect of parallel testing for a project at Ericsson. We find that the relationship between the number of available machines for parallelization and the *FeedbackTime* is nonlinear. For example, we can increase the number of machines by 25% and reduce the *FeedbackTime* by 53%.

We then examine three batching strategies in the test level: *ConstantBatching*, *TestDynamicBatching*, and *TestCaseBatching*. We evaluate their performance by varying the number of parallel machines. For *ConstantBatching*, we experiment with batch sizes from 2 to 32. The majority of the saving is achieved using batch sizes smaller than 8. However, *ConstantBatching* increases the feedback time if there are more than 6 parallel machines available. To solve this problem, we propose *TestDynamicBatching* which batches all of the queued changes whenever there are resources available. Compared to TestAll *TestDynamicBatching* reduces the *AvgFeedback* time and *AvgCPU* time between 15.78% and 80.38%, and 3.13% and 48.78% depending on the number of machines. Batching all the changes in the queue can increase the test scope. To address this issue we propose *TestCaseBatching* which performs batching at the test level instead of the change level. Using *TestCaseBatching* will reduce the *AvgFeedback* time and *AvgCPU* time between 19.84% and 84.20%, and 5.65% and 50.92% respectively, depending on the number of available machines for parallel testing. *TestCaseBatching* is highly effective and we hope other companies will adopt it.

Acknowledgments

I would like to take this opportunity to show my gratitude towards the people who have played an indispensable role in this memorable journey.

Foremost, I would like to express my sincere gratitude and respect towards my thesis supervisor, Dr. Peter Rigby. This work would not have been possible without his guidance, support and encouragement. His undying patience and guidance have helped me in all phases of this journey, from carrying out research to writing this thesis. Sincerely, I could not have asked for a better supervisor. I would like to thank Concordia University for providing me with an opportunity to do research.

Last but not the least, I would like to thank my parents for their love and constant support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Recently, there has been an increasing desire among developers to transfer their testing processes to a continuous integration (CI/CD) environment. To ensure each change does not break the software system, it is important to test each commit before merging it into the code repository [37]. Testing each change individually is costly, and in some cases, infeasible [70]. To reduce this cost, there are multiple techniques including test selection [83], test prioritization [100], and batch testing [70]. In this work, we focus on batch testing. A passing batch will save resources and allow the commits to be integrated quickly. However, if a batch fails a bisection must be performed to identify the failing commit, *i.e.* the culprit, potentially delaying and increasing the execution cost. In this work, we study the impact of batch testing with three case studies.

1.1 Software Batch Testing to Save BuildTest Resources and to Reduce FeedbackTime

Chapter 2 was accepted to the IEEE journal *Transaction on Software Engineering* [13]. I performed the major revisions for this TSE paper and added two additional research question. The first two research questions are included only as prior work, and my contribution is RQ3 and RQ4. In this chapter we examined how well the batching approaches, Batch4, BatchStop4, and RiskBatch, compared to prior work by Najafi *et al.* [70].

In RQ3, we examine the impact of failure rate on batching effectiveness. Prior works indicated

that batching is not effective with a failure rate at or above 25% [70, 12]. However, this assumes a normal distribution of failures. In this research question, we do an analysis on 152 Travis torrent projects. 85.5% of the projects see a savings in build test executions when using Batch4. All projects below a failure rate of 40% experience savings. With a failure rate between 40% and 60% the results vary by project, and developers would need to investigate the failure distribution to determine if batching is effective for their project. Only 5.2% of projects have a failure rate above 60% where batching results in a substantial increase in executions.

For RQ4, we examine the impact of batching on feedback time because prior works had only focussed on resource utilization [70, 12]. We find that on average all the proposed simple batching techniques provide feedback more quickly than TestAll. Compared to TestAll, Batch4 reduces the time for feedback by 32.22% on average and between 14.00% and 41.20% across projects. While BatchBisect and BatchStop4 can outperform Batch4 by a few percentage points, they require an optimal batch size and a variable amount of time to find the culprit. Even the minimum batch size of two can provide feedback savings with an average of 15.47%.

1.2 Mining Historical Test Failures to Dynamically Batch Tests to Save CI Resources

All prior works on batching use a constant batch size [70, 13]. In this chapter, we propose a dynamic batch size technique based on the weighted historical failure rate of the project.

We simulate each of the batching strategies across 12 large projects on Travis with varying failures rate. We find that dynamic batching coupled with BatchDivide4 outperforms the other approaches. Compared to TestAll, this approach decreases the number of executions by 47.49% on average across the Travis projects. It outperforms the current state-of-the-art Batch4 by 5.17 percentage points.

Our historical weighting approach leads us to a metric that describes the number of consecutive build failures. We find that the correlation between batch savings and FailureSpread is $r = -0.97$ with a $p \ll 0.0001$. This metric easily allows developers to determine the potential of batching on their project.

We also contribute a theoretical limit for the savings that can be achieved by batch testing. We show that using dynamic batching, we achieve an across project average of 58.91% of the theoretical limit. Although batching is highly effective, there is still substantial room for improving batching relative to the theoretical batch savings limit.

1.3 Parallel and Batch Testing in a Continuous Integration Environment

The previous chapters make three assumptions which are unrealistic on large software systems. First, they do not run tests in parallel and implicitly use a single machine. Second, after the failure of a batch, they rerun every test. However, this is unnecessary as we are aware of the failed test cases, and we do not need to re-run all the test cases that pass successfully. Third, previous researchers focus only on introducing new algorithms to reduce the resource costs of finding the culprit change in a batch [70, 13]. They did not perform an in depth investigation into the impact of the feedback time of batching.

In this chapter, we first evaluate the effect of parallel testing on a project at Ericsson. We find that the relationship between the number of available machines for parallelization and the *FeedbackTime* is nonlinear. For example, we can increase the number of machines by 25% and reduce the *FeedbackTime* by 53%.

We examine three batching strategies: *ConstantBatching*, *TestDynamicBatching*, and *TestCaseBatching*. We evaluate their performance by varying the number of parallel machines. For *ConstantBatching*, we experiment with batch sizes from 2 to 32. The majority of the saving is achieved using batch sizes smaller than 8. However, *ConstantBatching* increases the feedback time if there are more than 6 parallel machines available. To solve this problem, we propose *TestDynamicBatching* which batches all the queued changes whenever there are resources available. Compared to TestAll, *TestDynamicBatching* reduces the *AvgFeedback* time and *AvgCPU* time between 15.78% and 80.38%, and 3.13% and 48.78% depending on the number of machines. Batching all the changes in the queue can increase the test scope. To address this issue we propose *TestCaseBatching* which performs batching at the test level instead of the change level. Using *TestCaseBatching* will reduce

the *AvgFeedback* time and *AvgCPU* time between 19.84% and 84.20%, and 5.65% and 50.92% respectively, depending on the number of available machines for parallel testing. *TestCaseBatching* is highly effective, and we hope other companies will adopt it.

Chapter 2

Software Batch Testing to Save Build Test Resources and to Reduce Feedback Time

In this manuscript thesis, this Chapter is a verbatim copy of the paper published in Transaction on Software Engineering (TSE) [13]. I performed the major revisions for this TSE paper and added two additional research question. The first two research questions are included only as prior work and my contribution is RQ3 and RQ4.

2.1 Introduction

Testing is a critical but costly quality assurance practice [45]. Tests are run at multiple levels including unit, integration, and system tests [29]. The move to Continuous Integration and Delivery (CI/CD) emphasizes testing individual changes to ensure that problems are found immediately and before release [36]. In some development environments testing each change is infeasible and changes must be batched. For example, at Ericsson, expensive hardware simulation makes testing each change impossible [70]. To overcome these resource constraints Ericsson uses bisection to batch groups of builds and test them together. The bisection requires failing builds to be divided into

smaller sized batches until the culprit is found. At Google, individual integration tests can run for more than 45 minutes, requiring the batching of all recent changes. Test Feedback can be delayed by up to 9 hours [103, 68]. Google uses bisection in conjunction with static build dependencies to eliminate unlikely culprits. In this work, we examine the largest open source projects that use Travis CI to re-evaluate the existing batching approaches: BatchBisect and RiskTopN.

The open source projects that use Travis CI have different constraints from Ericsson and Google. Beller *et al.* [17] examine building and testing on Travis CI and finds that on most open source projects the time to run tests takes much longer and consumes more resources than the build and other aspects. This cost is amplified by the running of tests in different environments, *e.g.,* Python 2.7 and 3.7. Furthermore, the most common cause of CI build failure is a failed test which can further lengthen the integration and release cycle. In terms of resource consumption, Travis CI limits the number of builds that an open source project can run [3]. During busy times, the waiting time can more than doubles the build test cycle [18]. Resources are clearly an important consideration for open source projects. In contrast with prior work, we also examine how batching impacts the time to receive the test verdict, *i.e.* the change in feedback time.

We introduce three novel approaches, Batch4, BatchStop4, and RiskBatch, to improve the efficiency of testing CI. We conduct an evaluation on large open source projects that use Travis CI [18]. We release the `BatchBuilder` [15] tool that batches pull-request on GitHub for testing on Travis CI. Since the batching happens in the background and results are reported for each individual pull-request, the development process is unchanged.

2.1.1 Research Questions and Batching Approaches

RQ 1. Batching: How well does simple bisection and batching improve resource utilization?

RA 1. TestAll: Running tests on a single build containing a single pushed change immediately isolates any failing test to the changed code. This approach is simple, allowing developers to test each push as a single build in modern CI pipelines [35]. We use the TestAll as the baseline approach because it is in widespread use and does not require builds to be combined and does not introduce the complexity of bisection on test failure.

RA 2. BatchBisect: TestAll can be prohibitively expensive for large companies with many tests or expensive test hardware. For example, Google [103], Ericsson [70], and Shopify [66] combine commits into a single batch to reduce the total number of test executions. If all the tests pass on a build of size n, then there will be $n - 1$ saved build test executions. In the case of test failure, a bisection is performed until a single build is isolated as the culprit. The execution savings are dependent on the number of test failures that result in bisection. We run simulations to determine the best batch size for a project.

Compared to TestAll, we see a BatchBisect saves between 22.35% and 57.55% of the total build test executions with an average across projects of 46.05%. The best batch size per project ranges from 4 to 8.

RA 3. Batch4: Tooling exists to batch commits and perform bisection on test failure, for example, SandCastle from Facebook [44]. However, bisection adds additional batching and complexity to the CI process. To avoid bisection, we note that batches of size four have the special property: on failure a bisection will cost at least 4 additional executions, which is the same as testing each build individually. We propose the novel Batch4 approach, which groups builds into batches of four, saving $n - 1 = 3$ executions when all tests pass. On failure, we do not run a bisection, instead we revert to TestAll which costs 4 additional test executions or $n + 1 = 5$ executions in total.

This simple approach is also very effective at execution reduction. Compared to TestAll, we see that Batch4 saves between 29.51% and 55.84% with an average across projects of 47.63%. Compared to BatchBisect, Batch4 is not only simpler, requiring no bisections, but also outperforms BatchBisect with an average execution improvement of 1.58 percentage points.

RA 4. BatchStop4: On projects that have few failures, BatchBisect can still be more efficient than Batch4. For example, on the puppet project, the batch size is 8 and requires 5.09 percentage points fewer executions than Batch4. As a result, we introduce BatchStop4, which can make large batches and uses normal bisection until the batch size is four.

Compared to TestAll, we see that BatchStop4 saves between 29.51% and 60.83% with an average across projects of 50.31%. The majority of the savings are achieved with small batch sizes, batch 2, 4, and 8, realizing an average of 72%, 93%, and 99% of the total batch savings. Compared to BatchBisect and Batch4 the average execution improvement is 4.23 and 2.69 additional

percentage points.

RQ2. Risk Models: Can commit risk models improve the resource utilization during batching?

RA 5. RiskTopN: When a test fails on a batch, a bisection is required which costs additional executions. Not all builds are equally likely to fail, *i.e.* risky. Models of change risk have been widely used to identify bug-introducing changes [39]. Recent work by Najafi *et al.* [70] used risk models to identify commits that had likely failing tests at Ericsson, and we reproduce the RiskTopN approach on nine OSS projects. When a batch fails, the N builds with the highest risk are isolated and tested by themselves. The remaining builds that have a lower modelled risk are tested together in a single batch. The process is repeated until all culprits are found.

Compared to TestAll, RiskTopN reduces executions between 23.23% and 54.80% with an average across projects of 44.17%. However, Batch4 and BatchStop4 both outperform RiskTopN by 3 and 6 percentage points and do not require a statistical risk model.

RA 6. RiskBatch: In the previous approaches, the batch size is constant for all batches. We introduce the RiskBatch approach that uses a statistical model of risk to continue to add builds to a batch until a risk threshold is reached. Low-risk builds will be put into larger batches than high-risk builds, and a single high-risk build that is above the threshold will be tested individually. In our simulations, we vary the risk threshold.

Compared to TestAll, RiskBatch reduces executions by between 25.93% and 57.43% with an average across projects of 48.50%. RiskBatch outperforms previous risk-based approach, RiskTopN by 4.33 percentage points.

RQ3 FailureRate: How does the failure rate effect resource utilization during batching?

Prior work indicated that batching is not effective with a failure rate at or above 25% [70]. However, this assumes a normal distribution of failures. In this research question, we do an initial analysis of all the Travis torrent projects that have 1000 or more tested builds (152 projects). While the projects cover the full range of build failure rates from 2.80% to 96.03%, we find that the distribution is skewed towards lower failure rates, with a median failure rate of 23.23%. To simulate the impact of failure rate on batching, we use the Batch4 approach because it is effective and simple compared to the bisection and risk-based startegies.

85.5% of the projects see a savings in build test executions when using Batch4. The failure rate and savings have a strong negative correlation, Spearman $r = -0.97$ and $p \ll 0.001$. All projects below a failure rate of 40% experience savings. With a failure rate between 40% and 60% the results vary by project, and developers would need to investigate the failure distribution to determine if batching is effective for their project. Only 5.2% of projects have a failure rate above 60% where batching results in a substantial increase in executions.

RQ4 Feedback: What is the impact of batching on feedback time?

Batching is typically used when there are hard resource constraints such as hardware testing at Ericsson [70] or during massive integration testing at Google that can run for over 9 hours [68]. While we focus on resource savings, we provide an initial formulation of the change in feedback time on the large open source projects that use Travis CI.

We find that on average all the proposed simple batching techniques provide feedback more quickly than TestAll. Compared to TestAll, Batch4 reduces the time for feedback by 32.22% on average and between 14.00% and 41.20% across projects. While BatchBisect and BatchStop4 can outperform Batch4 by a few percentage points, they require an optimal batch size and a variable amount of time to find the culprit. Even the minimum batch size of two can provide feedback savings with an average of 15.47%.

2.2 Background on Batching and Definitions

In this section, we introduce the background on batching, bisection, and statistical risk models. We mathematically show the minimum and maximum number of build test executions required to find the culprit build on failing tests as well as the savings when builds pass. We note that these equations require the build sizes to be powers of two. These definitions are complimented by examples for each of our six research approaches. In subsequent simulations, we calculate the actual number of executions.

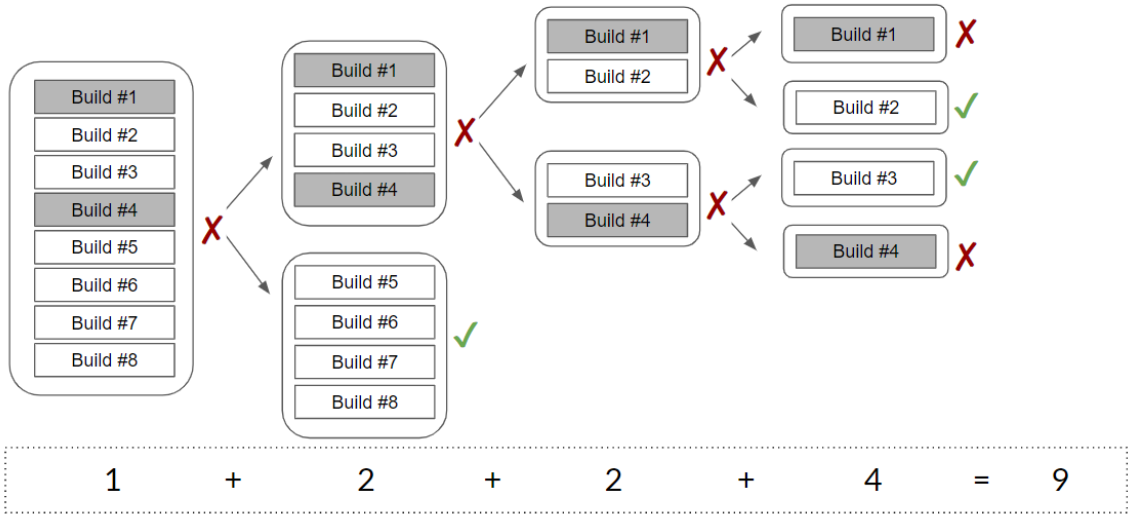

Figure 2.1: BatchBisect. In this example, a batch of size eight builds is tested. The batch fails. Bisection is used to isolate the culprits in two batches of size four. The bottom passes and can be integrated. The top contains two failures that are isolated. In total, we need nine executions to isolate the culprits.

2.2.1 RA 1. TestAll

TestAll is the simplest and most common form of running tests in a CI flow. Every change will be tested individually before being merged to the main repository or master branch. The number of build test executions is equal to the number of changes made to the system, n. The number of executions is constant regardless of a pass or fail in a build because on failure there is only one possible culprit build. Formally, the number of executions for a pass, p, or fail, f, is defined below:

$$\text{TestAll}_p(n) = \text{TestAll}_f(n) = n \tag{1}$$

2.2.2 Batching

Instead of testing each build individually, we batch builds together and test them in groups. When the batches pass, we need only one test execution:

$$\text{Batch}_p(n) = 1 \tag{2}$$

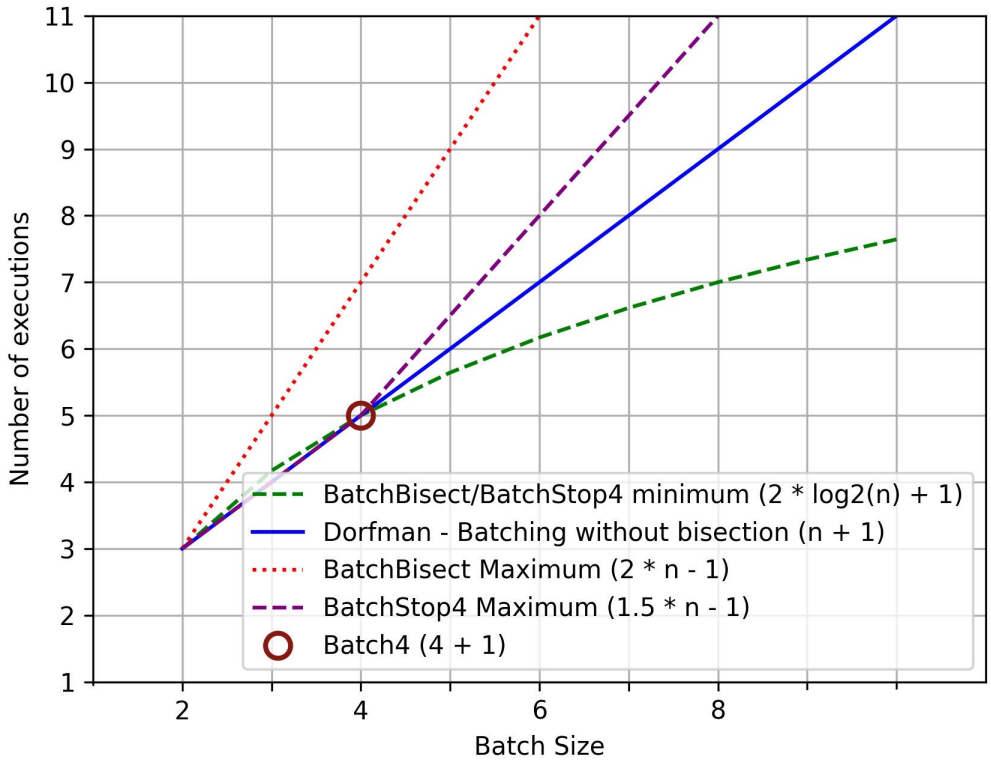

Figure 2.2: On batch failure, the minimum and maximum number of executions required to isolate the culprit build(s) for each approach.

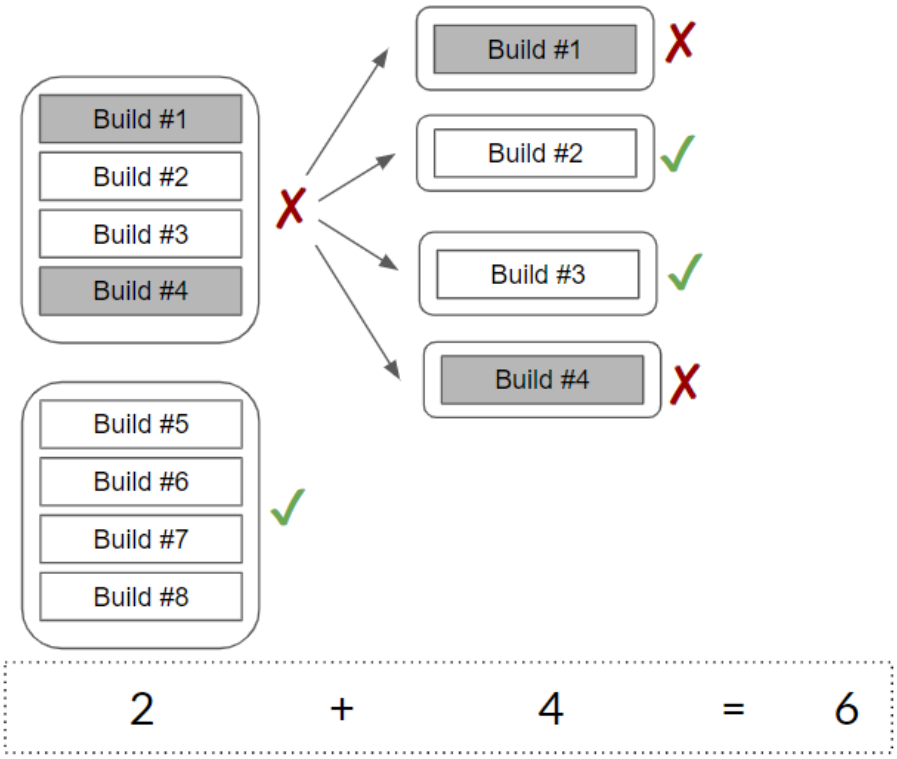

Figure 2.3: Batch4. If TestAll is run on failure the number of executions is constant and equal to the minimum executions for BatchBisect(4). In this example, the first four builds fail, and each is then tested individually for a total of five executions. The second batch passes requiring a single execution. We need a total of six executions, while the same builds required nine executions for BatchBisect in Figure 2.1

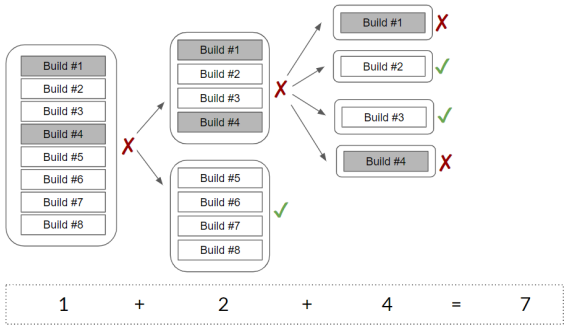

Figure 2.4: BatchStop4. We add a stopping condition for bisection when the batch is size four. For example, the first batch fails and a bisection is performed. In the second batch, Build 1 and 4 are culprits but batch size is four, so instead of bisection, all builds are tested individually. Build 5 to 8 have no failures and there is no need for further test executions. In total, we need 7 execution to find all culprits.

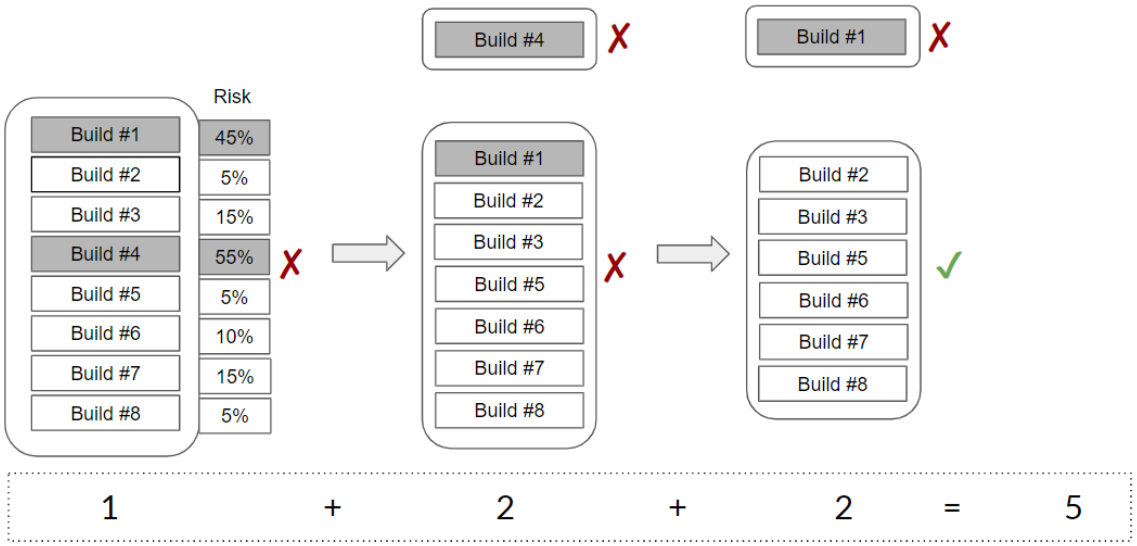

Figure 2.5: RiskTopN. A risk model is run to determine the likelihood of failure for each build [70]. The riskiest N builds are tested in isolation with the remainder tested as a batch. Using RiskTop1 in this example, Build 4 is the riskiest and is isolated for testing. When the remaining batch still fails, Build 1 is now the riskiest. The remaining batch passes. In total, we need five executions to isolate the culprits.

This savings can be substantial. In an extreme example, imagine a project that does a nightly test run on 100 builds, if the build passes the savings in execution will be $1 - n = -99$ or 99 build test executions.

$$\text{BatchSaving}_p(n) = \text{Batch}_p(n) - n = 1 - n = 1 - \text{TestAll}_p(n) \tag{3}$$

However, on failure the culprit must be identified and the number of executions varies depending on the approach used to identify the culprit failure.

2.2.3 RA 2. BatchBisect

When a batch passes, only one execution is required to merge the builds in the batch. However, if the batch fails, the build that has failing tests, *i.e.* the culprit(s), must be found using bisection. GitBisection uses a binary search and in so doing assumes ordered commits and that there is only one commit that introduces the failure (*i.e.* we search for the failing commit). However, if there are two commits that have failing tests, then GitBisection would only be able to find the oldest

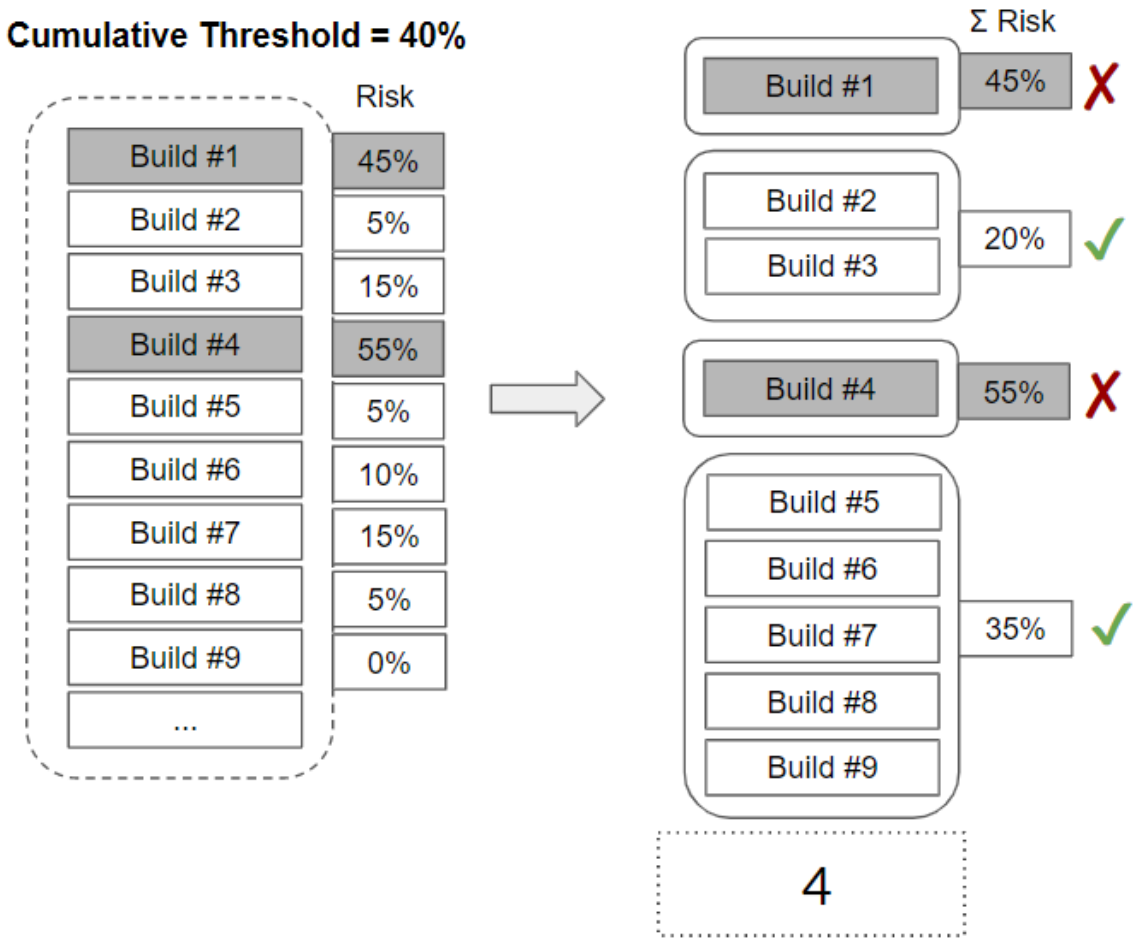

Figure 2.6: RiskBatch. Builds are added to the batch until a threshold is reached. With a threshold of 40%, Build 1 is isolated, while Builds 2 and 3 are tested together because their combined risk is 20%. Adding Build 4 would have a increased the cumulative risk to 75%, so Build 4 is tested individually. The remaining builds have a combined risk of 35%, so they are tested as a single batch. In total, we need four executions to isolate the culprits.

culprit commit. The remaining commits could not be integrated without further testing as a second culprit may also be present. Instead of a search for a single culprit, we need to traverse a binary tree identifying all the culprits. As a result, in this work we bisect by splitting the builds into equal batches for testing. Figure 2.1 illustrates the process.

Mathematically, we know that the number of test executions required to find a single culprit is the minimum cost on failure. The bottom line in Figure 2.2 shows the number of executions required to find a single culprit for batch sizes between 1 and 10.

$$min(\text{BatchBisect}_f(n)) = 2 * log_2(n) + 1 \tag{4}$$

If all builds in a batch contain a failing test, *i.e.* are culprits, then the number of required executions is equal to the number of nodes in a full binary tree, which is the maximum cost on batch failure. This maximum is shown as the top line in Figure 2.2.

$$max(\text{BatchBisect}_f(n)) = 2 * n - 1 \tag{5}$$

The greater the number of builds that have failing tests, the greater the number of test executions. For example, Figure 2.1, in a batch of 8 that contains two culprits we need 9 build test executions to find both culprits. This is actually larger than the TestAll scenario with one build test execution per build, *i.e.* eight. In Section 2.4.1, we run simulations to determine the optimal batch size and execution reduction attained by BatchBisect for the Travis projects.

2.2.4 RA 3. Batch4

When a batch fails, bisection requires test executions to find the culprits. Given that the bisection is performed using a binary tree, a batch of size four has special properties that we will discuss. The build test execution reduction when a batch of size four passes is constant at 3 build test executions. For completeness:

$$\begin{aligned} \text{Batch4}_p(4) - n &= 1 - 4 = -3 \\ &= \text{BatchBisect}_p(4)) - n \end{aligned} \tag{6}$$

However, one failure with BatchBisect(4) requires between 5 and 7 executions to identify the culprits. In contrast, the Batch4 approach, that on failure runs the tests on each individual build, *i.e.* TestAll, resulting in a constant number of executions described below.

$$\begin{aligned} min(\text{Batch4}_f(4)) &= max(\text{Batch4}_f(4)) \\ &= n + 1 = 4 + 1 = 5 \\ &= min(\text{BatchBisect}_f(4)) \\ &< max(\text{BatchBisect}_f(4) = 7 \end{aligned} \tag{7}$$

Figure 2.3 provides an example of the Batch4 approach. The first batch has two culprits and BatchBisect would require 9 executions. In contrast, Batch4 requires 6 test executions. The second batch has no culprits, so it requires one build test execution. When there is a single culprit, BatchBisect and Batch4 are the same (see Equation 7), however, when there are two or more culprits the Batch4 saves up to 2 executions.

Batch4 is a special case of the Dorfman [33] method introduced during World War II to batch medical tests of, for example, syphilis. The naive Dorfman algorithm combines n soldiers into a single batched test, on failure each individual solider is tested individually, *i.e.* TestAll. In Figure 2.2, we show that Dorfman requires additional executions beyond the minimum for BatchBisect after four builds. In our simulations in Section 2.2.4, we show that the simple Batch4 approach is highly effective.

2.2.5 RA 4. BatchStop4

In the previous section, we mathematically demonstrated that bisection with four builds should be replaced by Batch4. We build upon this idea with BatchStop4, which runs normal bisection until

the batch size is four, in which case it runs Batch4. For example, in Figure 2.4 the batch size is 8 and Builds 1 and 4 contain failures. After a bisection, the first batch contains the failures while the second batch passes. Since the batch size is four, bisection is no longer performed, instead each build is tested individually *i.e.* TestAll. The total number of execution is 7 for BatchStop4, while the total for BatchBisect is 9.

With the stopping condition at 4, then the number of executions required to find one culprit is the modified version of Equation 4:

$$\begin{aligned} min(\text{BatchStop4}_f(n)) &= 2 * log_2(n) + 1 - 4 + 4 \\ &= 2 * log_2(n) + 1 \\ &= min(\text{BatchBisect}_f(n)) \end{aligned} \tag{8}$$

While the maximum number of executions on failure is

$$\begin{aligned} max(\text{BatchStop4}_f(n)) &= 2 * n - 1 - (n/2 + n) + n \\ &= 2 * n - 1 - n/2 \\ &< max(\text{BatchBisect}_f(n)) \\ &= 2 * n - 1 \end{aligned} \tag{9}$$

Since we stop bisection when a batch contains 4 builds, the height of the tree is reduced by two with an execution reduction of $n/2 + n$. However, we still need to run TestAll on these batches of 4, so we need n additional executions. With one culprit BatchStop4 is equivalent to BatchBisect, however, with additional culprits we can save up to $n/2$ executions. Section 2.4.3 presents the simulation results, and we find that BatchStop4 has the second highest savings of our approaches.

2.2.6 RA 5. RiskTopN

When a batch fails, bisection requires expensive additional executions. Commit risk models have been used to alert developers to bug-introducing changes that may need additional testing or

review [8]. Najafi *et al.* [70] used risk models to isolate the top N riskiest commits to be tested individually while batching the remaining low-risk commits.

For example RiskTop1 is illustrated in in Figure 2.5. We see the modeled risk probabilities for each build with Build 4 has the highest risk, *i.e.* 55% chance of failure, so it is tested individually. The remaining builds are tested in a single batch. The process of testing risky builds in isolation is repeated until all failures are found and passing builds are integrated. Finding the 2 culprits in our examples take only 5 build test executions compared to the 9 and 7 required for BatchBisect and BatchStop4 respectively.

Najafi *et al.* [70] created a simple logistic regression model with seven features. In contrast, as we show in our data and methodology, Section 2.3, we create more sophisticated models, *e.g.,* Random Forest using 19 features. As we discuss in the result, the accuracy of the model dictates the degree of savings (see Section 2.5.1).

2.2.7 RA 6. RiskBatch

The approach in Najafi *et al.* [70] tests risky builds in isolation. In contrast, we introduce the RiskBatch approach that group builds into a batch up to a cumulative risk threshold. For example in Figure 2.6 we set the risk of failure threshold to 40%. Build 1, with a risk of 45%, is tested individually while Build 2 and build 3 are tested together because their combined risk is 20%. Build 4 could not be added to the previous batch because the combined risk of would be 75%, so build 4 is tested individually. The process is repeated for the remaining four builds that have a combined risk of 35%. In this example, we need four executions to isolate the culprits and integrate the passing builds, compared to the 9 and 5 for BatchBisect and RiskTop1, respectively. The savings are dependent on the accuracy of the risk model, and Section 2.5.2 presents our results and tuning with various thresholds.

2.3 Data and Methodology

In this section, we describe the projects and data used in the study. We then describe our statistical risk models. Finally, we describe our simulation method and define the outcome measures.

Table 2.1: Size of projects under study

Project	Failure Rate	Tested Builds	Years	Contributors
ruby	22.21%	15,382	5	192
metasploit	7.93%	8,836	4	703
graylog2	10.51%	5,194	4	98
owncloud	16.13%	4,452	2	71
vagrant	9.59%	4,402	4	914
gradle	8.96%	4,018	2	434
puppet	6.95%	3,223	4	532
opal	9.87%	2,980	4	99
rspec	19.36%	2,856	5	274

2.3.1 Travis Projects Under Study

Travis CI is a continuous integration system that is freely available for use by open source projects.[1] The data from the builds of 1,200 open source projects was made available by Travis Torrent [18]. We use the Travis Torrent dataset in this work. In the Travis Torrent dataset, a Travis build can have the following outcomes:

- "passed:" The code has been successfully tested and no failures have occurred.
- "failed:" The code has been successfully tested but some tests have failed.
- "errored:" There was an error while running the tests. For example, there is a bug in test code, an error in setup test environment, a timeout, or an error returned from git.
- "canceled:" The build has been canceled by the user.

We discard "canceled" builds because a developer manual stopped the test run and we cannot model the reason for this stoppage. We consider "errored" and "failed" builds as failures because environmental failures will also result in a bisection [70].

Following Najafi *et al.* [70], we only considered projects with a failure rate below 25%. We order projects by the number of builds and select the top nine active projects: Ruby, Metasploit, Graylog2, OwnCloud-android, Vagrant, Gradle, Puppet, Opal, and Rspec. We do not consider projects with a failure rate above 25% as batching is not effective with high failure rates [70]. Table 3.1

[1]Travis: https://travis-ci.com/

provides additional descriptive statistics on the projects. The projects have between 2.8k and 15k test builds, there is a wide range of failure rates from 7% to 22%, multiple years of development, and between 67 and 914 contributors per project.

The projects are from diverse software domains and we briefly describe each project. The Ruby project is a popular object oriented programming language that is often used for web development. The Metasploit project is a testing framework used for penetration testing with about 900 exploits for different operating systems. The Graylog2 project is an open source logging system capable of collecting, storing and analyzing logs in production. The OwnCloud-android project is an Android app to access cloud storage provided by an OwnCloud Server. The Opal project is a source-to-source compiler for converting Ruby code to JavaScript. The Rspec project is a testing framework for Ruby projects focusing on test driven development. The Vagrant project helps to build and manage portable virtual machines and containers such as AWS or Docker containers. The Gradle project is a build automation and dependency management software that supports many languages including Java, C++, and Python. The Puppet project is management software that controls distributed operating systems with a centralized configuration and facilities administrative tasks such as updating software and managing users.

Najafi *et al.* [70] argued that a failure rate of 25% should be the cutoff for batching because one in four builds will fail making even small batches sizes ineffective. However, this assumes a normal distribution of batch failures. For completeness, in RQ3, we run the Batch4 approach on all projects in the Travis torrent dataset with 1000 or more builds regardless of failure rate (a total of 152 projects). These projects cover the full range of failure rates, from 2.80% to 96.03% with a median of 23.11%, and allows us to understand the actual upper limit on failure rates and batch savings.

2.3.2 Statistical Risk Models

Two of our approaches require statistical models: RiskTopN (RA 5) and RiskBatch (RA 6). We develop risk models to identify the builds that are most likely to fail. We use scikit-learn[2] library

[2]https://scikit-learn.org/

for this purpose. Change risk modelling has been widely studied to identify faults [39] and bug-introducing changes [58]. Prior work by Najafi *et al.* [70] created a simple logistic regression using seven predictors. In this work, we use more sophisticated models and additional features. The dataset has 61 features for each Travis build. We exclude all features that are available only after the tests have been run, including number of failed tests, number of skipped tests, and test duration. We also exclude unique features including the commit hash, date, and project level features, such as the team size that would be constant across all project builds. In total, we have 19 features in total, which we describe briefly for completeness.

(1) gh_is_pr: true if this build is started by a pull request otherwise false.

(2) gh_num_commits_in_push: Number of commits in the push that started the build.

(3) git_prev_commit_resolution_status: String, ”merge found” if this build is a merge otherwise ”build found”.

(4) git_num_all_built_commits: Integer, Number of all commits in this build.

(5) gh_num_commit_comments: Number of comments on all commits in this build on GitHub.

(6) git_diff_src_churn: Number of modified lines of source code.

(7) git_diff_test_churn: Number of modified lines of test code.

(8) gh_diff_files_added: Number of files added.

(9) gh_diff_files_deleted: Number of files deleted.

(10) gh_diff_files_modified: Number of files modified.

(11) gh_diff_tests_added: Number of test cases added.

(12) gh_diff_tests_deleted: Number of test cases deleted.

(13) gh_diff_src_files: Number of source files changed.

(14) gh_diff_doc_files: Number of documentation files changed.

(15) gh_diff_other_files: Number of other files changed (other than source code and documentation).

(16) gh_num_commits_on_files_touched: Total number of commits on the files touched in this build in previous 3 months.

(17) gh_sloc: Total number of lines of source codes in the repository.

(18) gh_asserts_cases_per_kloc: Number of assertions per 1000 gh_sloc.

(19) gh_by_core_team_member: True if the triggering commit was by a core team member. (Someone who has committed code at least once in previous 3 months)[18]

The outcome of our risk model is the probability that a build will fail one or more tests. We evaluated five classifiers: random forest, Naive Bayes, MLP, logistic regression, and SGD.

2.3.3 Simulation and Evaluation

The Travis dataset provides the test verdict for each individual build. Failed builds must be investigated while passing builds are integrated. To simulate the impact of our batching approaches on the number of build test executions, we use the verdict of each build, and combine builds based on the approaches described in Section 3.2.

Our simulated batches contain only the builds that have been flagged as ready for integration with Travis CI. We do not introduce any new conflicts when we create batches because any conflict would have been dealt with when the developer ensures that the code can be merged in the pull request prior to submission to Travis CI.

We only combine builds that have the same Travis CI configuration, *e.g.,* that request the same dependencies and environment. If two builds have different configurations, we cannot combine them in a batch. For example, a build that requires postgres cannot be combined with one that requires MySQL. We only batch builds with identical configuration files.

For risk based approaches, we must train a risk model, and we use the first month of data for training. As we discuss in threats to validity, we experimented with larger training time periods,

but found that one month was equal or better than longer time periods. To compare with the other approaches, we also ignore the first month in non-risk approaches.

The goal of this work is to identify failing builds and integrate passing builds with a minimal number of build test executions. We report the percentage decrease in build test executions for each research approach, A, relative to the total number of builds that must be tested, *i.e.* the TestAll approach, according to the following equation:

$$\begin{aligned} \text{ExecutionReduction(A)} &= 1 - \frac{\text{Executions(A)}}{\text{TotalBuilds}} \\ &= 1 - \frac{\text{Executions(A)}}{\text{Executions(TestAll)}} \end{aligned} \tag{10}$$

We also report the additional savings for each approach relative to the total number of builds. This is equivalent to calculating the differences in percentages, *i.e.* percentage point difference, for each approach. We use the equation below to calculate the additional savings for approach, A2, given approach, A1, and the number of TotalBuilds:

$$\begin{aligned} &\text{AdditionalReduction(A2 — A1)} \\ &= (1 - \frac{\text{Executions}(A1)}{\text{TotalBuilds}}) - (1 - \frac{\text{Executions}(A2)}{\text{TotalBuilds}}) \\ &= \frac{\text{Executions}(A2) - \text{Executions}(A1)}{\text{TotalBuilds}} \\ &= \text{ExecutionReduction}(A1) - \text{ExecutionReduction}(A2) \\ &= \text{PercentagePointDifference}(A2, A1) \end{aligned} \tag{11}$$

Although resource savings is the primary goal of the work, we provide an initial investigation of the change in feedback time for batching relative to test all. The reduction in feedback time is calculated for approach A as follows:

$$\text{FeedbackReduction(A)} = 1 - \frac{\text{Duration(A)}}{\text{Duration(TestAll)}} \tag{12}$$

Table 2.2: Percentage savings in build test executions relative to TestAll. We see on average the techniques save slightly less than half the build test executions. The best performing approach is BatchStop4. However, Batch4, which does not require bisection or a risk model, performs well and is simple to implement.

Project	Batch Bisect	Batch4	Batch Stop4	Risk Top2	Risk Batch
ruby	22.87%	32.97%	32.97%	29.96%	33.77%
metasploit	52.37%	51.46%	54.64%	50.60%	53.54%
graylog2	52.05%	52.17%	55.69%	49.39%	55.20%
owncloud	53.82%	53.42%	57.98%	54.80%	57.43%
vagrant	57.55%	55.84%	60.83%	50.35%	55.27%
gradle	48.49%	49.21%	50.92%	41.91%	49.29%
puppet	57.16%	54.86%	59.34%	50.85%	56.26%
opal	47.81%	49.19%	50.91%	46.40%	49.84%
rspec	22.35%	29.51%	29.51%	23.23%	25.93%
Minimum	22.35%	29.51%	29.51%	23.23%	25.93%
Average	46.05%	47.63%	50.31%	44.17%	48.50%
Maximum	57.55%	55.84%	60.83%	54.80%	57.43%

2.4 RQ 1: Batching

How well does simple bisection and batching improve resource utilization?

2.4.1 Result: RA 1. BatchBisect

Batching commits is widely used for integration testing and when the tests are long-running or expensive [103]. Najafi *et al.* [70] empirically showed that batching commits and using a bisecting process to isolating the failing commit is effective at Ericsson with a savings in build test executions of 7%, 14%, and 41% depending on the project. We reproduce the result on nine large projects hosted on Travis CI. We run simulations with batch sizes between 1 and 20 builds and plot the saving in build test executions in Figure 2.7. From the execution saving curve in the figure, we note a logarithmic improvement with the majority of the savings coming from small batches sizes. At a batch size of 8, we see that at a minimum 97% of the total executions savings has been achieved. On the projects that Najafi *et al.* [70] studied, the improvements began to decrease with larger batch sizes. We see a similar trend on the rspec and ruby projects that have the highest failure rates. The remaining projects plateau with larger batch sizes resulting in little to no improvement in execution

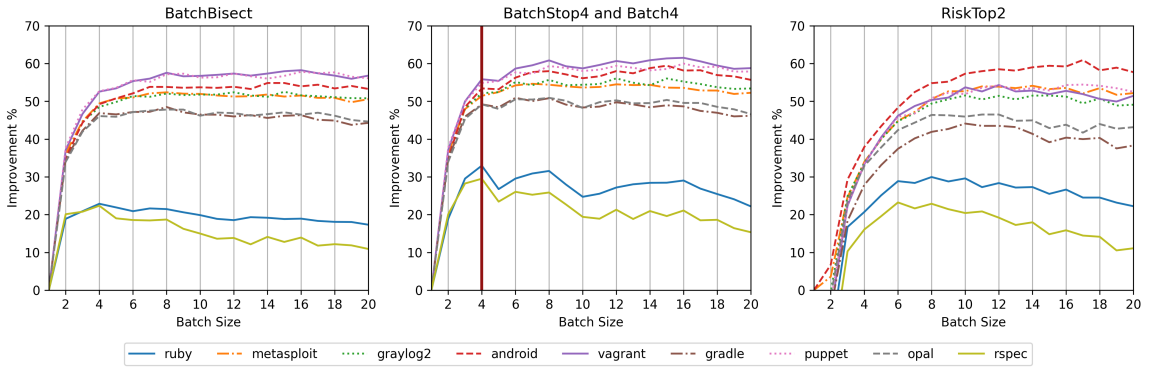

Figure 2.7: Savings in number of build test executions for each batch size. We see that much of the savings is achieved with small batch sizes. Batch4 is represented as a vertical line in the middle figure. Projects with higher failure rates see a decrease in savings with large batch sizes, while most projects plateau.

savings. As a result, we report the saving results at or below 8 for the remainder of the work. The best batch sizes are 4, 8, 8, 7, 8, 8, 8, 8, and 4 for each project respectively and the corresponding execution savings are 22.87%, 52.37%, 52.05%, 53.82%, 57.55%, 48.49%, 57.16%, 47.81%, and 22.35%, respectively, with an average of 46.05%.

> Compared to TestAll, we see a BatchBisect saves between 22.35% and 57.55% of the total build test executions with an average across projects of 46.05%. The best batch size per project ranges from 4 to 8, with the majority of the savings realized with small batch sizes.

2.4.2 Result: RA 2. Batch4

In Section 2.2.4, we mathematically showed that batches of four builds save three executions when they pass, but can require between five and seven executions on failure for BatchBisection. However, if we simply test all the builds individually on failure, we always need five executions which is the same as the minimum number of executions for BatchBisect. Furthermore, the Batch4 approach does not require bisection and the complexity of regrouping commits inherent in this process. In Figure 2.7, we see the vertical line represents the savings for Batch4 which are 32.97%, 51.46%, 52.17%, 53.42%, 55.84%, 49.21%, 54.86%, 49.19%, and 29.51% per project.

Compared to TestAll, we see that Batch4 saves between 29.51% and 55.84% with an average across projects of 47.63%. Compared to BatchBisect, Batch4 is not only simpler, requiring no bisections, but also outperforms BatchBisect with an average improvement of 1.58 percentage points.

2.4.3 Result: BatchStop4

For BatchStop4, we use bisection until there are only four builds in a batch at which point we revert to Batch4 and TestAll on failure as discussed in Section 2.2.3. Figure 2.7, shows the simulation results. The execution saving compared to TestAll are 32.97%, 54.64%, 55.69%, 57.98%, 60.83%, 50.92%, 59.34%, 50.91%, and 29.51% for each project respectively, with an average of 50.31. These savings are achieved by choosing batch sizes: 4, 7, 8, 8, 8, 6, 8, 8, and 4, respectively. Compared to BatchBisect and Batch4, we see a reduction of 4.26 and 2.69 percentage point in number of build test executions.

Again the figure shows a logarithmic improvement. In Table 2.3 we show the percentage of total savings for each batch size. With a batch size of 2 we have already realized an average of 72% of the total savings, by batch 4 we see an average of 93%, and by batch size 8 the average savings is 99%. It is clear that the largest gain in savings comes with small batch sizes and that larger batch sizes provide little further advantage and in some cases require extra executions.

Compared to TestAll, we see that BatchStop4 saves between 29.51% and 60.83% with an average across projects of 50.31%. The majority of the savings are achieved with small batch sizes, batch 2, 4, and 8, realizing an average of 72%, 93%, and 99% of the total batch savings. Compared to BatchBisect and Batch4 the average execution improvement is 4.23 and 2.69 additional percentage points.

2.5 RQ2: Risk Models

Can commit risk models improve the resource utilization during batching?

Table 2.3: Proportion of total build test execution savings with a given batch size for BatchStop4. We also include the optimal batch size in the final column. The added complexity of large batches are not worthwhile. Small batch sizes account for the vast majority of the savings, with at least 97% of the savings achieved with a batch size of 8 or less.

Project	Size = 2	Size = 4	Size = 6	Size = 8	Best Size
ruby	0.83	1.00	1.00	1.00	4
metasploit	0.71	0.93	0.99	1.00	7
graylog2	0.68	0.92	0.98	0.99	12
owncloud	0.62	0.87	0.93	0.97	15
vagrant	0.61	0.87	0.93	0.98	16
gradle	0.75	0.97	1.00	1.00	6
puppet	0.64	0.89	0.94	0.98	16
opal	0.74	0.97	0.99	1.00	8
rspec	0.88	1.00	1.00	1.00	4
Minimum	0.61	0.87	0.93	0.97	4
Average	0.72	0.93	0.97	0.99	9.77
Maximum	0.88	1.00	1.00	1.00	16

The RiskTopN and RiskBatch depend on a risk model of how likely a build is to fail. RiskTopN then tests the riskiest N builds in isolation, while RiskBatch groups builds into until a cumulative risk threshold is reached. We described the 19 features that we included in our risk model in Section 2.3.2. Najafi *et al.* [70] used 7 features and a simple logistic regression. In contrast, we evaluate five classifiers: Naive Bayes, Random Forest, Multilayer Perceptron (MLP), logistic regression and Stochastic Gradient Decent (SGD). We did not use decision trees because they are not designed to provide a probability for the prediction and would not be able to create risk thresholds need to create batches [94, 30]. Table 2.4 shows the F-score for each model. We see that Random Forest out performs the other predictors on all projects except gradle where it is 1 percentage point worse than SGD. As a result, we use Random Forest in the remainder of this work. For completeness we report the precision and recall for Random Forest. The precision is 0.51, 0.29, 0.40, 0.46, 0.30, 0.14, 0.23, 0.23, and 0.30 for each project respectively. The recall is 0.55, 0.18, 0.33, 0.37, 0.25, 0.09, 0.14, 0.16, and 0.27, respectively.

We tuned the parameters for random forest. For *number of trees* we experimented the values of 10, 50, 100, 200, and 400 and found a difference in F score between 2 and 4 percentage point. For *maximum depth* of the trees we evaluated the model with the values of 10, 20, 50, 100, 200 and no limit. The difference in F score was between 0 and 4 percentage point. For the *criterion* parameter,

Table 2.4: Comparison of F-scores for each model and project. Regardless of F-score all failing builds are found. Low F-scores result in more build test executions. Precision and recall for Random Forest are in the text.

Project	Random Forest	Naive Bayes	MLP	Logistic Regression	SGD
ruby	**0.53**	0.31	0.35	0.31	0.35
metasploit	**0.23**	0.03	0.08	0.03	0.08
graylog2	**0.36**	0.31	0.20	0.33	0.28
owncloud	**0.41**	0.32	0.28	0.14	0.11
vagrant	**0.28**	0.17	0.11	0.22	0.17
gradle	0.11	0.05	0.11	0.06	**0.12**
puppet	**0.18**	0.05	0.14	0.05	0.06
opal	**0.19**	0.09	0.03	0.18	0.13
rspec	**0.29**	0.22	0.19	0.25	0.19

we experimented gini and entropy and found the default gini function was the best choice in all of the projects. For *minimum samples split* we experimented the values of 2, 5, 10, 20, 50, 100. The default value of 2 had the best result in 8 of the projects. One of the projects had the best result with the value of 10 although the difference was 1 percentage point in F score. For *minimum samples leaf* we evaluated the values of 1, 2, 5, 10, 20, 50, 100 and found the default value of 1 generates the best result in all of the projects.

An accurate risk model will reduce the number of executions, while an inaccurate model can even increase the number of executions to find culprits. However, unlike bug prediction that can result in a developer investigating a commit that does not introduce a bug, *i.e.* a false positive, our risk models are used to automatically batch builds. The failing build will always be found and an inaccurate risk model will simply require more executions but will never change the final outcome, *i.e.* it will never add a false positive or negative.

2.5.1 Result: RA 4. RiskTopN

The RiskTopN approach isolates the riskiest builds to be tested in isolation, while testing the less risky builds in a batch. We reproduce Najafi *et al.*'s [70] Ericsson study on Travis CI projects using more predictors, a random forest, and removing the fixed batch size of four and top $N = 2$. We evaluate N = 1 to 10 and batch sizes from 1 to 20.

Figure 2.7 shows the execution savings for each batch size. We found that $N = 2$ produced the best results for all projects. Like the other approaches, we see that the majority of the savings are at batch 8, so for comparison purposes we report the results at batch size 8 in the work. The improvement over TestAll is 29.96%, 50.60%, 49.39%, 54.80%, 50.35%, 41.91%, 50.85%, 46.40%, and 23.23% respectively. On all projects, the savings in executions is lower than Batch4 and Batch-Stop4 which do not require a risk prediction model. Despite the use of more advanced models and predictors than Najafi *et al.* [70], the results do not justify the addition of a risk prediction model in the CI pipeline.

> Compared to TestAll, RiskTopN introduced by Najafi *et al.* [70] reduces executions between 23.23% and 54.80% with an average across projects of 44.17%. However, Batch4 and Batch-Stop4 both outperforms RiskTopN by 3 and 6 percentage points and do not require a statistical risk model.

2.5.2 Result: RA 6. RiskBatch

Instead of isolating risky builds, our RiskBatch approach adds builds to a batch until the sum of the builds added to the batch reaches a threshold. Section 2.2.7 and Figure 2.6 illustrate the process. We varied the cumulative risk threshold of failure from 10% to 200% in steps of 10 percentage point increases. We find that the best thresholds are 90%, 120%, 120%, 170%, 140%, 90%, 110%, 130%, and 80% for each project respectively. The cumulative risk is often over 100% indicating that although the model predicts a high cumulative risk of failure, the strategy of making large batches appears to outweigh this risk. However, Figure 2.8 plots the execution improvement for each threshold and shows that low-risk threshold are also reasonably effective.

Compared to TestAll, the reduction in number of executions are 33.77%, 53.54%, 55.20%, 57.43%, 55.27%, 49.29%, 56.26%, 49.84%, and 25.93% respectively. RiskBatch outperform previous risk based approach, RiskTopN by 4.33 percentage points.

> Compared to TestAll, RiskBatch reduces executions by between 25.93% and 57.43% with an average across projects of 48.50%. RiskBatch outperforms RiskTopN by 4.33 percentage points.

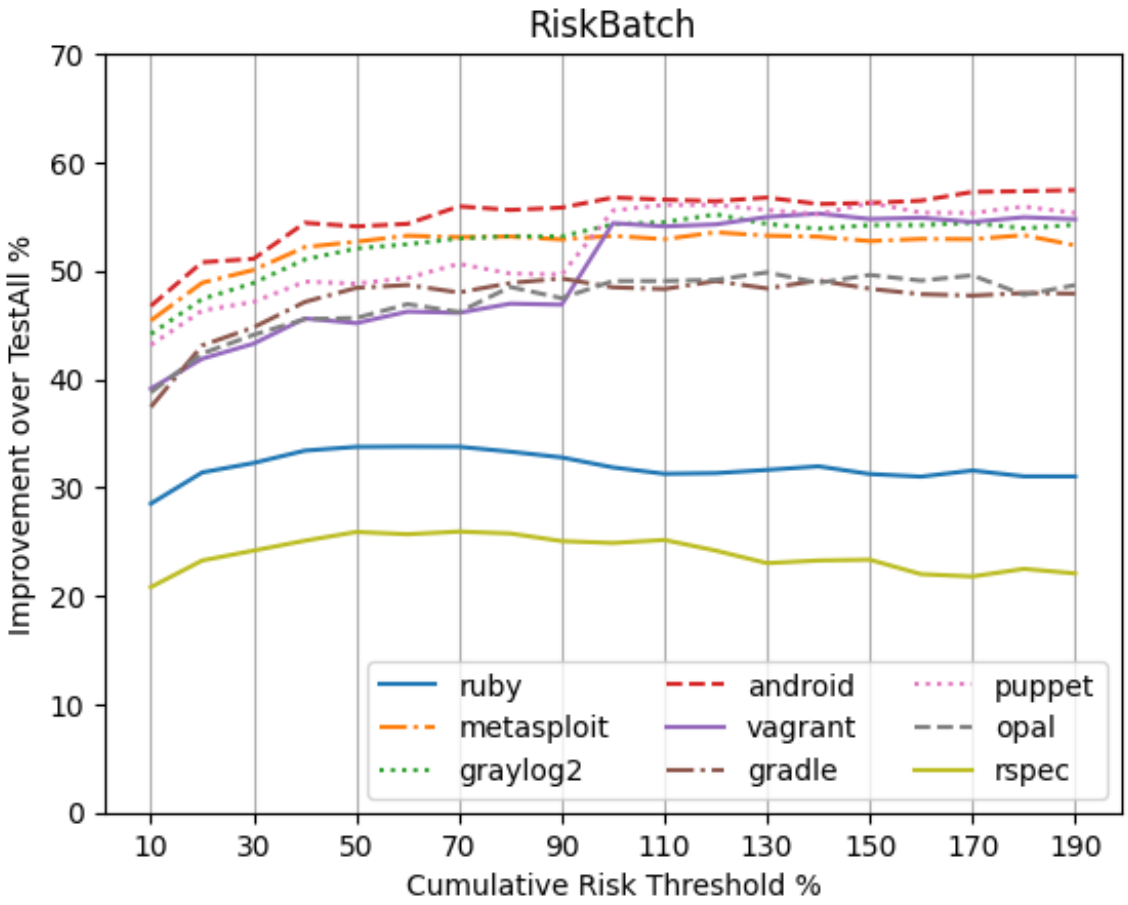

Figure 2.8: Experimenting with the RiskBatch cumulative risk threshold. We see that most projects have at or above 90%.

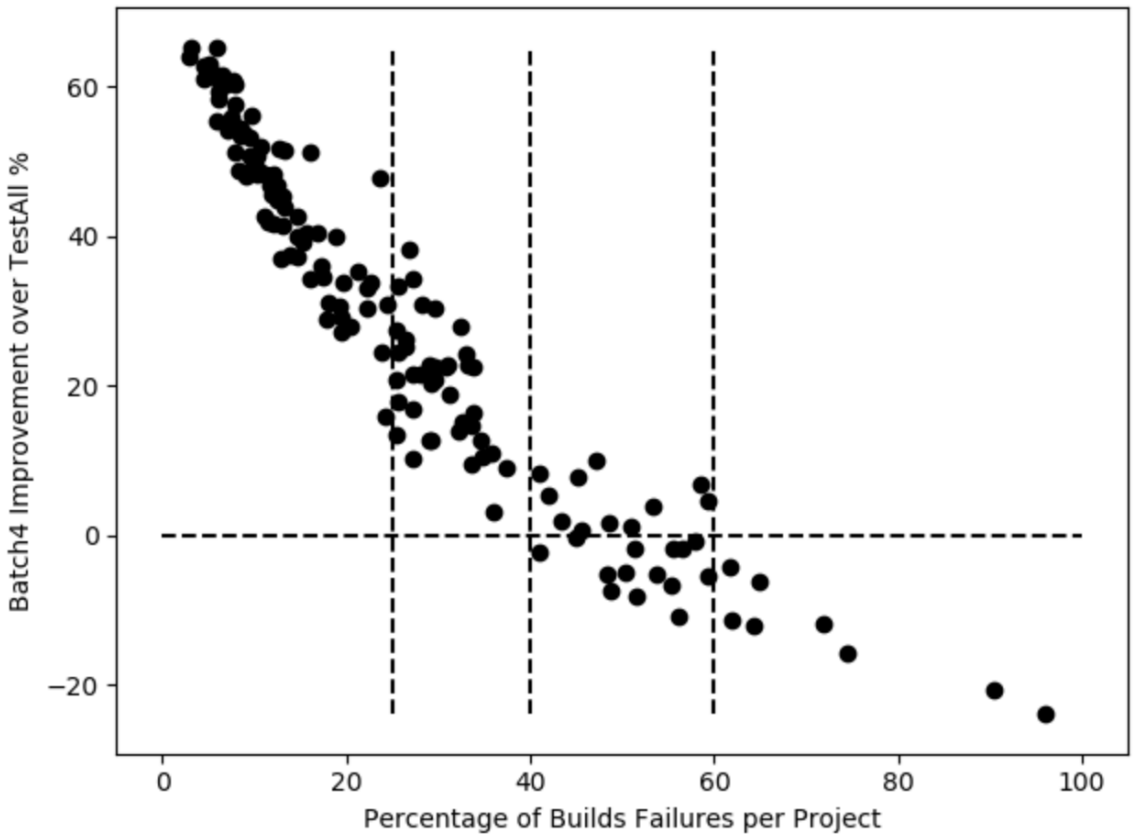

Figure 2.9: The scatter plot shows a strong negative relationship between failure rate and Batch4 executions savings. 85.5% of the projects are above the horizontal line representing zero execution savings. The data is skewed, with most projects seeing substantial savings. All projects with a failure rate below 40% save executions. Only 5.2% of projects have a failure rate above 60% where batching is ineffective.

2.6 RQ3: FailureRate

How does the failure rate effect resource utilization during batching?

Based on the Ericsson data of three projects, Najafi *et al.* [70] found that the failure rate limited the savings and they speculated that projects with a failure rate above 25% could not see savings. To further examine the relationship between failure rate and execution savings, we processed all of the large projects in the Travis torrent dataset that have 1000 or more builds, for a total of 152 projects. We only simulate the Batch4 approach as we have shown that it is as effective as the more complex bisection and risk approaches at reducing test executions. Section 2.3.1 fully describes the simulation and approach details.

Figure 2.9, plots the savings in build test executions on the y-axis relative to the project's failure rate on the x-axis. The Spearman correlation is strong and negative at $r = 0.97$ and $p \ll 0.001$, providing empirical evidence for Najafi *et al.*'s [70] observation that failure rate controls savings.

From the perspective of savings, 85.5% of a projects have a positive savings, *i.e.* are above the horizontal zero line in the figure. 76.97% see a savings of 10% or more, 66.45% see a savings of 20% or more.

From the perspective of failure rate (vertical lines in the figure), 21.05% of the projects have a failure rate under 10% and show the strength of batching with savings between 48.17% and 65.30%. These savings remain clear for projects with failure rates under 30% with savings between 10.34% and 47.73%. All projects under 40% see some savings, however, the savings diminish with projects with failure rates between 30% and 40% seeing savings between 3.16% and 28.05%. Between 40% and 60%, the savings vary between negative (-10.88%) and positive (10.04%). Only 5.2% of the projects have a failure rate above 60% and in all cases batching is ineffective.

> 85.5% of the projects see a savings in build test executions when using Batch4. The failure rate and savings have a strong negative correlation, Spearman $r = -0.97$ and $p \ll 0.001$. All projects below a failure rate of 40% experience savings. With a failure rate between 40% and 60% the results vary by project, and developers would need to investigate the failure distribution to determine if batching is effective for their project. With failure rates above 60%, batching is ineffective leading to additional build test executions.

2.7 RQ4: Feedback

What is the impact of batching on feedback time?

Batching has been most commonly used in strongly resource constrained environments such as with Ericsson's hardware simulation tests [70] and during Google's integration testing that can take upwards of 9 hours [68]. The focus of our work is on reproducing the resource savings of batching on the largest open source projects using Travis CI. However, we contribute an initial formulation of the change in feedback time from batching. We use a constant execution time, T,

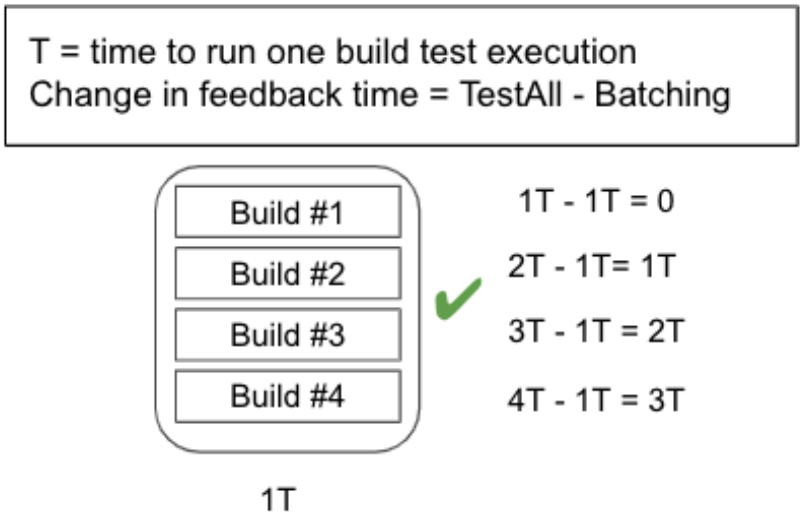

Figure 2.10: Feedback time for passing batch. For TestAll, each build is processed in order by Build number, for example, Build 1 would provide a verdict after time T, while Build 3 would have to wait and would provide a verdict only after 3T. On pass, with batching, Build 1 would still take time T, but Build 3 would be available 3T - 1T = 2T earlier.

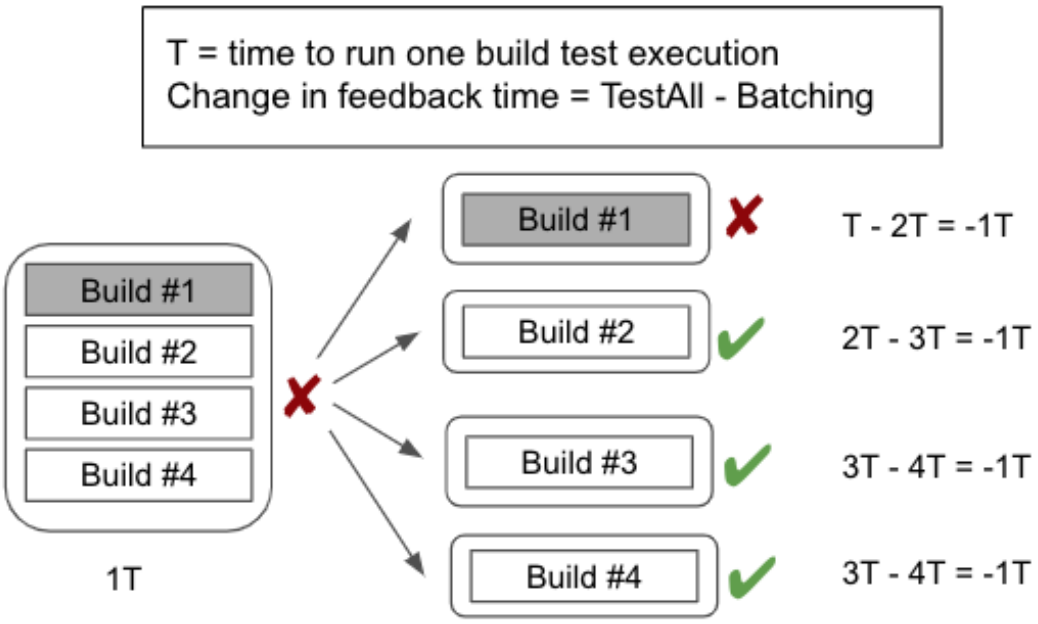

Figure 2.11: Feedback time for Batch4 on failure. Batch4 reverts to TestAll on failure. Since the original Batch test took 1T, each build will be delayed by -1T.

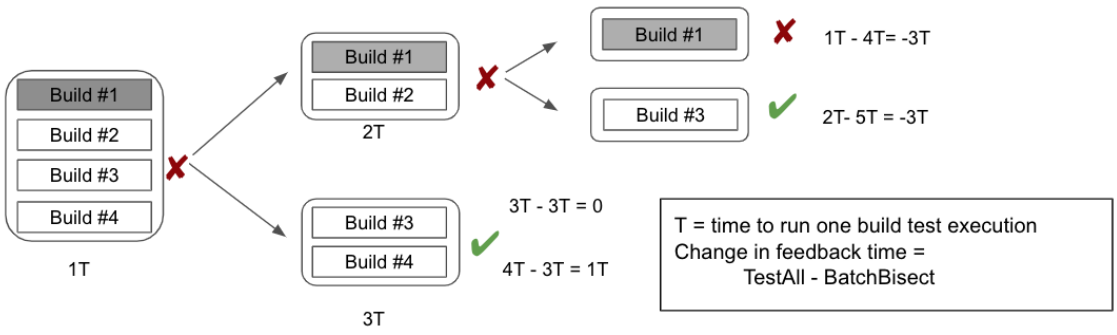

Figure 2.12: Feedback time for BatchBisect on failure. With bisection at each split, two new batches must be tested. The change in feedback time depends on where the culprit is in the build. In our example, Build 1 is delayed, -3T, while Build 4 sees a speedup of 1T.

for each build because we only batch builds that have the same configuration files. We calculate the change in feedback time relative to the TestAll scenario where each build in a batch is tested sequentially as we described in our simulation methodology in Section 2.3.3 and Equation 12. We run the simulation for Batch2, Batch4, BatchBisect, and BatchStop4. For the latter two we use the optimal batch size. We do not report feedback results for the risk based models because they do not outperform the simple batching and bisection strategies in terms of resource consumption.

When a batch passes, the savings for each build within the batch is proportional to its position. In Figure 2.10, we see that the first build in a passing batch will have no time savings, while the second will provide feedback one time unit earlier, the third will be two time units earlier, and the fourth will be three time units earlier.

In Figure 2.11, we see that on failure, Batch4 reverts to TestAll, and will delay each build's feedback by one time unit. In contrast, the delay for a bisection process varies depending on where the culprit is in the batch. An example, is shown in Figure 2.12.

The simulation results are in Table 2.5, and all approaches provide feedback on average more quickly than TestAll. We can see that BatchBisect [70] has a wide range in feedback time savings from 2.89% to 49.33%. The projects with high failure rates incur feedback delays when the batch size is large and there are multiple culprits. Futhermore, Batch4, which does not require bisection is only 1.13 percentage points lower than BatchBisect.

Batch4 has an average speedup of 32.22% and the range across projects is from 14% to 41.20%. While BatchStop4 slightly outperforms Batch4 by 4.26 percentage points, the additional complexity

Table 2.5: Change in average feedback time for batching compared with TestAll. For all batching approaches, the feedback time on average is less than TestAll. We see that BatchBisect has a wide range of feedback times, while Batch4 has a simpler algorithm that leads to more consist feedback times and a constant delay on failure.

Project	Batch Bisect	Batch2	Batch4	Batch stop4
ruby	4.80%	2.66%	18.40%	18.40%
metasploit	40.88%	20.00%	36.40%	40.25%
graylog2	43.55%	18.66%	37.20%	43.55%
owncloud	39.50%	16.00%	36.40%	43.55%
vagrant	49.33%	21.33%	41.20%	49.33%
gradle	35.77%	18.00%	33.20%	35.70%
puppet	45.77%	21.33%	39.60%	45.77%
opal	37.77%	16.66%	33.60%	37.77%
rspec	2.79%	4.66%	14.00%	14.00%
Minimum	4.80%	2.66%	14.00%	14.00%
Average	33.35%	15.47%	32.22%	36.48%
Maximum	49.33%	21.33%	41.20%	49.33%

of bisection and the larger/optimal batch size make it unlikely that it is worth this minor speedup in feedback compared to Batch4.

Projects can also see advantages even when batching only two builds together, with feedback improving by 15.47% on average and a range of 2.66% to 21.33% depending on the project. Even the strategy of combining two waiting builds on projects with low failure rates can see both resource savings and an improvement in feedback time.

Future work on the feedback time related to batching has a huge potential. For example, algorithms could be designed to spend more resources on a failing batch to provide feedback on the culprit faster at the expense of additional parallel compute resources.

Compared to TestAll, Batch4 reduces the time for feedback by 32.22% on average and between 14.00% and 41.20% across projects. While BatchBisect and BatchStop4 can outperform Batch4 by a few percentage points, they require an optimal batch size and a variable amount of time to find the culprit. Batch2 is even simpler than Batch4 and can provide feedback savings with an average of 15.47%.

2.8 Tool implementation on GitHub: `BatchBuilder`

We created the `BatchBuilder` tool to implement the BatchStop4 approach for use by developers. The tool integrates with GitHub pull requests and runs Travis CI. We release the source code [15]. After configuring a batch size and a maximum waiting time, the development process will remain unchanged because each submitted change will still have its own test verdict regardless of how it was batched. If the 'batch size' is set to four the approach will be the Batch4 strategy. The highlevel pseudocode for `BatchBuilder` is show in Algorithm 1.

Algorithm 1: GitHub App: BatchBuilder

```
When there are 'batch size' changes or the 'wait' time has elapses create a batch
 branch to combine changes
Function TestBatch(batch):
    result = Travis(batch)
    if tests result is passed then
        set status of each change to "successful" on GitHub;
    else
        if batch length is equal to 1 then
            set status of the change to "failed" on GitHub;
        else
            if batch length is smaller than or equal to 4 then
                foreach change ∈ batch do
                    TestBatch(change);
                end
            else
                TestBatch(first half of batch);
                TestBatch(second half of batch);
            end
        end
    end
```

Merge Conflicts. Our approach does *not* introduce any new merge conflicts. With pre-merge testing, if two or more changes are combined in a testing batch and have a conflict, this conflict will also exist when the changes are added to the master or main branch and would need to be resolved regardless of batch testing. BatchStop4 preserves the testing order of changes, so the conflict can be assigned to the change that occurred later and the changes without conflict can still be tested and integrated with master. With post-merge testing, any conflicts related to integration with master will already have been dealt with before batch testing begins.

Performance. In our implementation, on failure, each batch is implemented as a git branch containing the changes that need to be tested. In TestAll, each commit must also be merged with master and tested. This same merge operation occurs with the branch. An additional branch operations must occur on failure. However, further optimizations could be performed because each branch has the same ancestor, *i.e.* the latest commit on master, meaning that we already know the last common ancestor and the branch creations simply involves a simple diff operation. In practice, we see that the new branch operation takes less than one second (about 700 milliseconds). In contrast, the testing time is on the order of minutes [17].

2.9 Threats to validity

External Validity. We selected large open source projects with at least 100 contributors and a failure rate at or below 25% from the Travis torrent dataset [18]. The projects covered a variety of software development contexts, from programming languages to cloud computing. In reproducing, Najafi *et al.* [70] work at Ericsson on OSS projects we increase the generalizability of batching and bisection. Our novel approaches will need to be evaluated in other development contexts. To this end we release our scripts, data, and our `BatchBisect` developer tool [15].

To further improve external validity, in RQ3, we simulated the Batch4 approach on all projects with more than 1000 builds regardless of failure rate. We provide quantitative evidence across 152 projects that the failure rate has a strong negative correlation with batch effectiveness. Future work could evaluate approaches that are robust against high failure rates as well as an investigation of failure distributions.

Construct Validity. We operationalize the overall resource savings and change in feedback time for builds. In our simulations, we only consider the order of the tests and test outcome. Future work may consider other constructs such as feedback time on individual tests or changes in test scope.

In this work, we use the Travis configuration file to ensure that the combined builds run the same tests in the same environment, *i.e.* have the same test scope. In simulation, it is only possible to batch builds that required the same test environment, *e.g.,* pull-requests that both requested python 2.7 can

be combined while these builds could not be combined with a request for python 3.7. On projects that select a subset of tests to be run, combining builds might increase the test scope and future studies of are necessary impact of test scope on batching.

Internal Validity. Our works involved simulation and assumed builds available for batching. On small projects, there may not always be multiple pull-request available for batching. Clearly, these projects require fewer resources and can either wait until there are enough changes, or run with a smaller batch size. As we show in Section 2.4.3, the majority of the savings happen with batch sizes of 4, *i.e.* 93%, and even the smallest batch size of two sees substantial savings, *i.e.* 72%. In our tool implementation, we provide a workaround that will test commits that have waited for longer than the "wait time" specified in the configuration file.

We created build failure risk models using five classifiers: Random Forest, Naive Bayes, MLP, logistic regression, and SGD. Random Forest was the best classifier, so we tuned five hyper parameters for Random Forest leading to a total of 27 configurations for each project. We found an average of one percentage point difference and did not find consistent configurations across projects, so we reported results with the default parameters. After tuning, the longest training time for the projects was reduced from 4.25 to 3.5 minutes (on a standard laptop).

We assessed the impact of the training period by using builds from the previous 30 days, two months, or six months of data. We found that the 30 day training period had the same or higher F-scores compared to the longer periods. As other researchers have reported, longer training periods tend to reduce the accuracy of the model by including stale data [52, 95].

2.10 Discussion and Future Work

We contrast the approaches and discuss the implications of our findings as well as future work. Table 2.6 shows the important variations for each approach. The first point of variation is the action to be taken on test failure. The original bisection algorithm continues recursively until the individual culprits have been identified [70]. In Section 2.2.4, we showed mathematically that it is more efficient to stop when the batch is of size four. On failure the Batch4 algorithm tests each commit individually, TestAll, for a constant of 5 executions on failure. The BatchStop4 algorithm

Table 2.6: Variations in Batching Technique. Stopping at batch size four is the most promising techniques. We do not report feedback time improvement for the risk models as they do not outperform the simpler models in terms of resource utilization.

Technique	In Case of Failure	Resource Improvement	Feedback Improvement	Stop At 4	Use Bug Model	Dynamic Batch Size	Preserve Order
TestAll	-	-	-	×	×	×	✓
BatchBisect	Repeat	46.05%	33.35%	×	×	×	✓
Batch4	TestAll	47.63%	32.22%	✓	×	×	✓
BatchStop4	Bisect until Batch4	50.31%	36.48%	✓	×	×	✓
RiskTopN	Repeat	44.17%	-	×	✓	×	×
RiskBatch	BatchStop4	48.50%	-	✓	✓	✓	✓

uses bisection on failure, but stops bisection when the failing batch contains only four builds using the Batch4 approach. RiskTopN uses a risk model to test the riskiest N builds in isolation and the remaining builds as a batch [70]. If the batch fails, RiskTopN recursively continues with the next N riskiest builds. RiskBatch uses a risk model to group builds until a cumulative risk threshold is reached. If the build fails RiskBatch, it cannot be repeated because the batch already reaches to threshold. Instead, BatchStop4 is used to isolate culprits.

Ranking of approaches. Compared to the standard practice of testing each change in an individual build, all approaches provide substantial improvements reducing the test executions by around half on average. The following is the ranking of approaches by average reduction in savings across projects from worst to best: 44.17% RiskTopN, 46.05% BatchBisect, 47.63% Batch4, 48.50% RiskBatch, and 50.31% BatchStop4.

Stop at 4 The worst two approaches do not use the stop at four condition. From the algorithmic analysis, BatchStop4 has requires the same number of executions as BatchBisect when there is one culprit, but when there are more culprits BatchStop4 requires less, see plot in Figure 2.7. In the empirical evaluation, we see that BatchStop4 is on average 4 percentage points better than BatchBisect. As we later discuss, most savings occurs with small batch sizes resulting in the simple Batch4 algorithm performing only 3 percentage points lower than BatchStop4.

Risk Model. RiskTopN and RiskBatch use a risk model. The model is created using traditional features such as SLOC and number of tests as well as change features, such as the number of changed files or added lines. The accuracy of the model affects the performance of batching approaches, however, we guarantee that all culprits are found and isolated in contrast to, for example, test selection methods that may allow failing tests to ”slip-through” to other QA stages.

Algorithmically, RiskTopN is substantially different from the other algorithms and it does *not preserve the test order* of builds providing results for the riskiest builds in isolation first. However, the approach appears to work poorly with the lowest reduction in executions of all techniques. The approach is highly dependent on the risk model and on projects with highly predictive risk the approach may be effective. In contrast, RiskBatch, also uses the risk model but allows for *variable batch sizes* and uses BatchStop4 on failure. This combination appears to allow for appropriate risk and batch sizes providing the second best average savings. For the ruby project, the F-score of 0.53 is the highest among the projects and RiskBatch outperform the other approaches. It is possible that a more accurate risk model may allow RiskBatch to the most effective approach.

Best Batch Size. In all batching approaches, most of the saving is found early with smaller batch sizes 2.7. For BatchStop4 the proportion of saving using different batch sizes is reported, see Figure 2.3. On average across projects, 93% of the saving is achieved with a batch size of 4. The saving achieved by batch size 8 is at least 97% and does not increase with batch size 10. As a result, we reported the savings with a maximum batch size of 8. However, on projects graylog2, owncloud, vagrant, and puppet we see that the true best batch size is actually 12, 15, 16, and 16. Table 2.3 shows the projects' best batch size and the additional percentage of savings for those batch sizes, 1, 3, 2, and 2, respectively. While developers from these projects would need to experiment with batch sizes, we feel that it is unlikely that these minor improvements would be beneficial given the additional need for bisection of large batches on failure.

Impact on Feedback Time. While prior work examined only resource savings [70], we conducted a initial examination of the change in feedback time of the non-risk based batching approaches relative to TestAll. With resource utilization, we are only interested in the total resources, however, with bisection, the feedback time for an individual build varies depending on the location of the culprit. With bisection, we see examples where some builds see huge delays, while other

builds in the same batch see speedups. The variability of bisection in feedback time makes it less attractive.

In contrast, Batch4 provides the same dramatic speedup when the batch passes, but has a *constant* delay of the time to run one additional build test cycle. While we see that in the average case all approaches improve feedback time Batch4 and even Batch2 provide a more stable average improvement of 32.22% and 15.47%, respectively. Future work is necessary to develop algorithms that optimize for feedback time potentially at the expense of parallel compute resources that focus on the failing batch.

Failure rate vs Savings. On the basis of three projects at Ericsson, Najafi *et al.* [70], concluded that the failure rate controls the batch savings. We examined 152 open source projects and found a strong negative correlation between failure rate and execution savings for the Batch4 approach ($r = -0.97$ with a $p \ll 0.001$). We see that projects with low failure rates can have substantial savings, *e.g.,* vagrant has a failure rate of 8.96% and a maximum savings of 60.83%. Projects with failure rates below 40% all see savings. The best batch size for bisection is also controlled by the failure rate, for example, on ruby and rspec that have the higher failure rates, 22.51% and 19.36%, we also see that the best batch size is the lowest at four with savings at 32.97% and 29.51%.

However, we see exceptions to the failure rate controlling the savings and batch size. For example, owncloud has the third highest failure rate but the second highest, 53.42%, savings and the best batch size of 15. Examining owncloud over time we see an uneven distribution of failures with some periods having multiple consecutive build failures followed by consecutive build passes.

Fixed Batch Size In our work, for BatchBisect, BatchStop4, and RiskTopN we have identified a single batch size for the entire period of study. Developers, will need to examine their project history to identify the best batch size. If the failure rate is not constant over time, then projects with an uneven distribution, would clearly benefit from a *variable batch size*. This uneven risk of failure, was the main motivation for introducing RiskBatch that dynamically adjust the batch size based on a risk model. We believe that dynamic batching strategies is the most promising direction for future work.

2.11 Related Work

Continuous integration and delivery (CI/CD) systems are beneficial in both industry and open-source projects because the deployment tasks are automated and developers receive feedback faster, tests are run automatically, and critical updates are delivered to customers more frequently [65, 88, 47, 78]. However, the goal of CI/CD is to release changes as quickly as possible which increases the already high computational requirements involved in regression testing [45]. Running a subset of tests can reduce the cost of testing. Regression testing research has three streams of research [98]. The first, *minimization*, involves eliminating tests that are redundant or of low value. Early work reduced the problem to one of code coverage, for example, tests become redundant as the system evolves and more than one test covers the same control flow. As a result, much of the work in this area is algorithmic, such as transforming it into a spanning set problem [67], using divide-and-conquer strategies [24], and greedy algorithms [91]. More recent approaches include ant colony optimization in a search space to find the optimum set of test cases [60]. The use genetic algorithms to optimize selected tests and evaluate by total code coverage has also received substantial attention, *e.g.,* [55, 59].

The second, *selection*, uses the same static analysis techniques such as coverage [90] and slicing [48], but selects tests that cover source files that are at higher risk because they have been changed recently [82]. Using specifications such as requirements defined by customer is also used in test selection [26]. A recent work have focused on using deep learning models to optimize test selection results [74].

Test case selection is also performed by choosing a subset of test cases, but in contrast to test minimization, test cases that verify risky or recent changes are chosen. Noor *et al.* [75] predict failed test based on similarity to previous failed tests. Wang *et al.* [93] first detect fault-prone source code and then identify related test cases by coverage. Nguyen *et al.* [73] select test cases based on change-sensitivity to external services. Laali *et al.* [61] dynamically identify failed test based on the location of previous failed tests.

The third, *prioritization*, orders tests such that expensive, low-value, or long-running tests are run after tests that find faults early. While early prioritization techniques continued to use coverage

measures to gauge priority [43], more recent approaches incorporate the faults found in past test runs [56, 38, 71] and change relationships among files [87] to identify high value tests. Zhu *et al.* [101] examine the tests that historically fail together prioritizing test runs. Just *et al.* [53] propose an approach based on mutation analysis. Qu *et al.* [79] suggest to prioritize risky configuration in testing. Wang *et al.* [93] utilizes the quality of source code before finding the relationships between tests and code based on coverage.

The savings from minimization and selection will have the cost of slip-throughs because not all tests are run [45]. In contrast, our reduction in test executions comes from grouping builds not from eliminate/selecting a subset of tests. As a result, we guarantee no slip-throughs because we run all the tests. In contrast, prioritization saves no resources but improves feedback time. With prioritization the assumption is that tests can be run in an arbitrary order. However, changing the test order can lead to new flaky failures. Lam *et al.* [62] found that flaky failures due to order dependencies account for 50.5% of flaky failures in the projects they examined. We do not introduce order dependency flaky failures, because the entire test suite is run in its original order. Batching saves not only resources but also reduces feedback time without introducing any slip-throughs and without changing the test run order.

2.11.1 Risk Models

Predicting software defects using statistical models is a research area which has been popular in recent years [42, 86, 72, 31]. Different learning models are used and evaluated to perform bug prediction, such as Support Vector Model (SVM) [57], Logistic Regression [54, 70], KNN [25], and Deep Learning [97, 76]. Bug prediction can be made on varying units, with early studies focusing on file level predictions while recent studies perform change level prediction [57, 54]

Radjenović *et al.*'s [80] survey of bug models categorized the metrics into 1) traditional source code metrics, such as SLOC, 2) object-oriented metrics, such number of children in a class and depth of inheritance [27], and 3) development process metrics such as code change frequency which uses historical data to predict failures. In our work, we use the first and third types of metrics.

Recent works have identified risky changes. Early work focused on regression models [54]. Chen *et al.* [25] use source code metrics such as number of methods, average method complexity,

and number of lines of code to build their model. Their learning model is created using k nearest neighbor (KNN). Yang *et al.* [97] study software defect prediction using deep learning at the change level. Their prediction has two stages: 1) feature selection which is done by extracting a set of features from a broader initial set of feature using Deep Belief Network. 2) building a logistic regression classifier using the selected attributes. Pandey *et al.* [76] introduce an approach to detect software defective modules using a deep ensemble learning model. Their approach allocates more testing resources to modules that are more likely bug-prone based on model prediction. In our work, we evaluated five classifiers and found that Random Forests performed the best.

A criticism of statistical bug prediction models is that they do not provide actionable outcomes [54], *e.g.,* what specific action can a developer take if a change is labeled 'risky' because it is in a recently changed file? A further problem is that predictions are often incorrect, which in practice reduces developer confidence [85]. In contrast, our work uses the risk to batch commits and requires no action from developers. If the prediction is inaccurate then additional build test executions are required. However, the saving achieved, even with relatively inaccurate models, is substantial compared to testing each change individually.

2.11.2 Batching and Bisection

Batching is an effective technique to deal with resource constraints, whether it is computational power, development costs, or time [4, 28]. When changes are batched together and there is a failure, bisection can be used to reduce the number of test execution. When commits are ordered, GitBisection [1] uses a binary search to identify the culprit in $O(log(n))$ time. The approach works well when finding a single regression, but is not designed to find multiple culprits in a batch of changes for integration. To ensure that all tests pass on all changes in a batch, GitBisect would need to run multiple searches, in the worst case n searches, $O(n * log(n))$. In contrast, the bisection approaches discussed in Section 2.2.5 are designed for integrating multiple commits in $O(log(n))$ time when there is a single culprit and in the worst case $0(n)$ time.

At Google, integration tests can run on the order of hours and can cover thousands of commits, making GitBisection too computationally expensive. Instead, Google developers use the static build dependencies to determine which tests must be run when a file is changed. When a group of changes

fails during integration testing, Google developers can immediately eliminate all changes that do not individually relate to the failing test. Since there can be thousands of changes in an integration test, Google also scores the remaining commits on the basis of the number of files in a change (more files, more likely to be the culprit) and the distance to the root of the build test dependency DAG (closer to the root, safer as more developers have assessed it by now) [103]. In our work, we do not have ordered commits and we do not have the static dependencies. As a result, we run the entire test suite on each build. Future work is necessary to determine which of the individual tests can be run independently. Breaking individual tests out of a test suite is often non-trivial and can lead to flaky, unexpected test order dependencies [62], but could increase the effectiveness of batching.

2.11.3 Pooling Medical Tests

In medical tests, pool testing, *i.e.* batching, is an effective way to reduce the number of required test kits and thus decreasing costs. Dorfman [33] proposed an approach to detect infected individuals in a large population during World War II. He suggests pooling tests to reduce the cost and the time. If the test is negative, it means all members of that group do not have the disease, otherwise each individual needs to be tested separately, *i.e.* the same strategy as TestAll. Gajpal *et al.* [41] propose an approach to partition people into groups and test each group with one kit. Only, groups with a positive result need to be divided into subgroups and tested further. To improve the pooling process, double and multiple pooling place samples into more than one pool [20, 92]. If a pool tests positive, the samples that are common among other negative tested pools can be removed from further testing. Aragón-Caqueo *et al.* [7] study the effectiveness of batching in COVID-19 tests and report that batching gains more saving when the infection rate is lower. The interest in pool testing has risen dramatically with COVID-19, with these works being submitted in early 2020. Medical pool testing and software batch testing have the same mathematical background and it will be interesting to use the approaches developed in the medical world, *e.g.,* double pool testing, in and SE context and vice versa, *e.g.,* BatchStop4 in medical pools.

2.12 Conclusion and Recommendations

In this work, we introduced a mathematical basis for the batching approaches and make the following research contributions and recommendations for development practices.

Najafi *et al.* [70] showed that BatchBisect was an effective strategy on three projects at Ericsson and could save 7%, 14%, and 41% of build test executions compared to TestAll. We reproduce this result on the Travis dataset and show that BatchBisect reduces the number of executions by between 22.35% and 57.55% with an average of 46.05%.

We introduce the Batch4 approach in Section 2.2.4, and we mathematically show that Batch4 requires a constant number of executions on failure, *i.e.* 5, which is the minimum for BatchBisect and saves up to two executions when there are multiple culprit builds in a batch. Batch4 reduces the number of execution required by 29.51% and 55.84% with an average of 47.63%. Batch4 is simpler and does not require bisection while saving an additional 1.57 percentage points on average relative to the total number of builds. We release our tool that integrates with GitHub and Travis CI tool to allow developers to seamlessly batch pull-requests [15].

We introduce BatchStop4 that uses bisection until a batch of four is reached in which case we use Batch4. With this stopping condition, we mathematically show that BatchStop4 is equivalent to BatchBisect when there is one culprit, but requires fewer executions when there is more than one culprit build. We see that BatchStop4 saves between 29.51% and 60.83% with an average of 50.31%, with an additional savings of 2.17 to 10.10 percentage points relative to BatchBisect. We recommend that any project already using BatchBisect should modify their algorithm to include a stopping condition for batches of size four. Our tool `BatchBuilder` allows developers to configure the batch size for their project.

We reproduce Najafi *et al.*'s [70] RiskTopN approach where the riskiest N changes in a batch are tested individually and the remaining builds in the batch are tested together. On the Travis projects under study, we find a reduction between 22.04% and 55.77% with an average of 44.04%. However, the simple Batch4 outperforms RiskTopN by 3.59 percentage points on average, and Batch4 does not require a risk model, so we do not recommend that developers adopt this approach.

We introduce the RiskBatch approach which adds builds to a batch until a risk threshold is

reached. RiskBatch reduces the number of executions by between 25.93% and 57.43% with an average of 48.50%. The approach is complex requiring both bisection and a risk model and does not perform better than BatchStop4 except in the projects that has the highest F-score in risk model. Projects that can build a highly accurate risk model may consider using this approach.

We examine the relationship between failure rate and build test execution savings. We examine Batch4 on 152 projects and find a strong negative correlation ($r = -0.97$ with a $p \ll 0.001$) with a skew towards lower failure rates and high execution savings. Furthermore, all projects with a failure rate below 40% see savings and some projects below 60% see savings. Batching is effective on 85.5% of projects.

We provide an initial formulation of feedback time and evaluate the non-risk based approaches. Compared to the bisection approaches that introduce variable delays on failure, Batch4 delays each batch by a constant single additional of the time to run one build test cycle. Batch4, and even Batch2, provide substantial average improvements in feedback time relative to TestAll, 32.22% and 15.47%. Batching saves not only resources but also reduces feedback time without introducing any slip-throughs and without changing the test run order.

Chapter 3

Mining Historical Test Failures to Dynamically Batch Tests to Save CI Resources

As a manuscript thesis, this chapter is a verbatim copy of the paper accepted to ICSME 2021: International Conference on Software Maintenance and Evolution.

3.1 Introduction

Recently there has been an increasing desire among developers to transfer their testing processes to a continuous integration (CI/CD) environment. To ensure each change does not break the software system, it is important to test each commit before merging it into the code repository [37]. Testing each small change is costly, and in some cases, infeasible [70].

To reduce this cost, there are multiple techniques including test selection [83], test prioritization [100], and batch testing [70]. In this work, we focus on batch testing. A passing batch will save resources and allow the commits to be integrated quickly. However, if a batch fails a bisection must be performed to identify the failing commit, *i.e.* the culprit, potentially delaying and increasing the execution cost.

We first introduce the origin of batching techniques in medical applications and explain two culprit finding techniques from the medical literature including Dorfman [34] testing and double pool testing [20]. We then build upon prior works that evaluated bisection at Ericsson [70] and Travis CI projects [14] and replicate their culprit finding techniques for a failed batch including BatchBisect, BatchStop4, as well as our new proposed technique, BatchDivide4.

These techniques combine a constant number of changes into a batch *e.g.,* a size of 4 or 8 [14]. However, the optimal batch size can vary over time as the number of test failures changes. For example, the tests may fail repeatedly as the developers work to fix a difficult bug. In contrast, most builds pass and on a stable branch there may be long periods of minor changes that do not result in build failures. Since culprit finding involves bisection and additional commits, when there are repeated failures, the batch size should be small. While during times of relative stability large batch sizes can be used to save test resources. Our novel contribution is to examine the history of test failures to dynamically adjust the batch size. We propose dynamic batch size adjustment approaches that complement the culprit finding techniques.

This paper is structured as follows. In Section 3.2 we present the background for the batching, and when a batch fails, the culprit finding techniques. In Section 3.3 we introduce dynamic batch sizes based on historical weighting of prior test runs. In Section 3.4 we describe our dataset and the simulation methodology. In Section 4.4 we present the result of our simulations. In Section 3.6, we introduce a theoretical upper limit on the savings we can achieve from batch testing. We also discuss the impact and importance of consecutive failures in limiting the effectiveness of batching. In Section 4.5 we discuss the threats to the validity of the paper. In Section 4.6 we position our work in the literature. In Section 4.7 we conclude the paper.

3.2 Background on Batching Approaches

We first explain TestAll which is our baseline and lower bound for test effectiveness. We then explain the background of the existing as well as our new culprit finding techniques: Dorfman, double pool testing, BatchBisect, BatchStop4, and BatchDivide4. We provide mathematical background and examples to create intuition for readers.

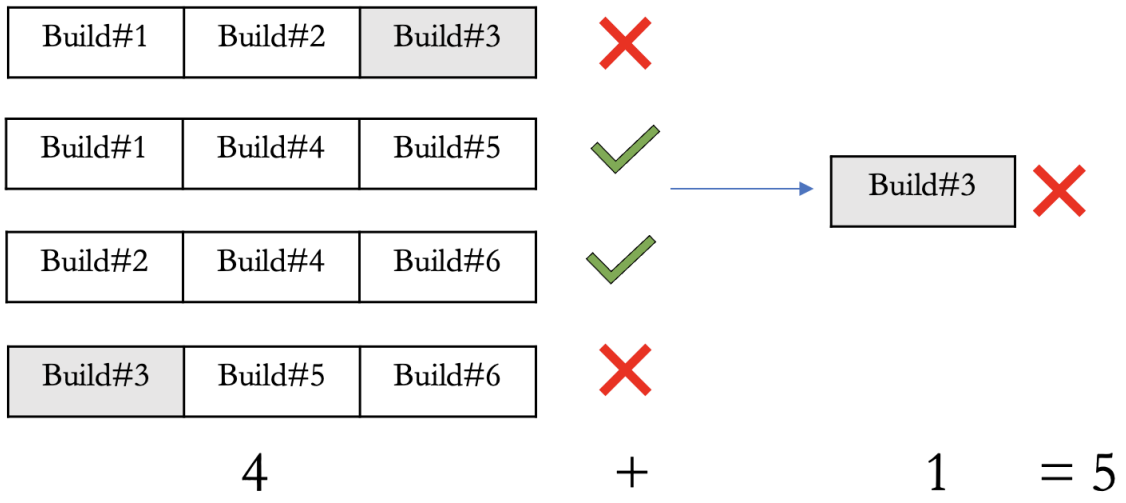

Figure 3.1: DoublePoolTesting: In this example, the total number of builds that are tested is 6. Builds 3 is broken. We first put each build in two batches, we see the first and the third batch are broken. We see only build3 is in two batches that both fail. So we run build3 in isolation to find the culprit. The total number of executions is 5 for the 6 builds.

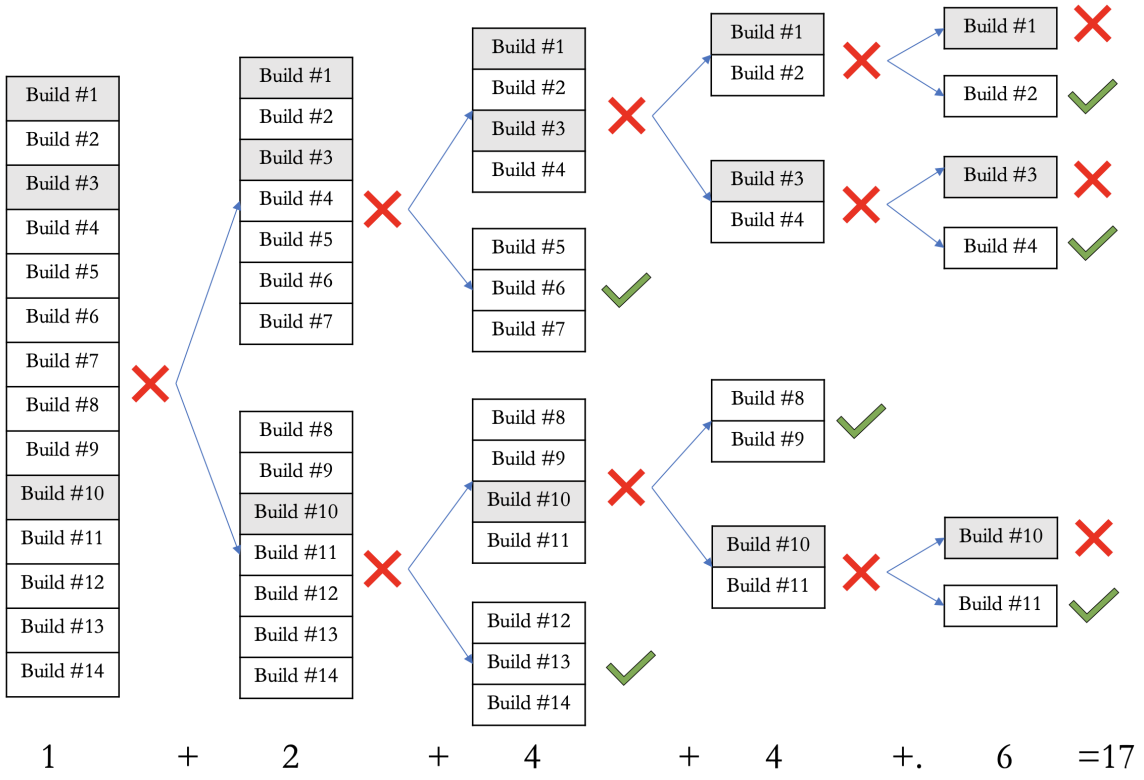

Figure 3.2: BatchBisect: In this example, the total number of builds that are tested is 14. Builds 1, 3, and 10 are broken. We first test the whole 14 builds in a batch and it fails. We then do bisection and traverse the binary tree for the next 4 levels. After 17 executions, we find the broken builds.

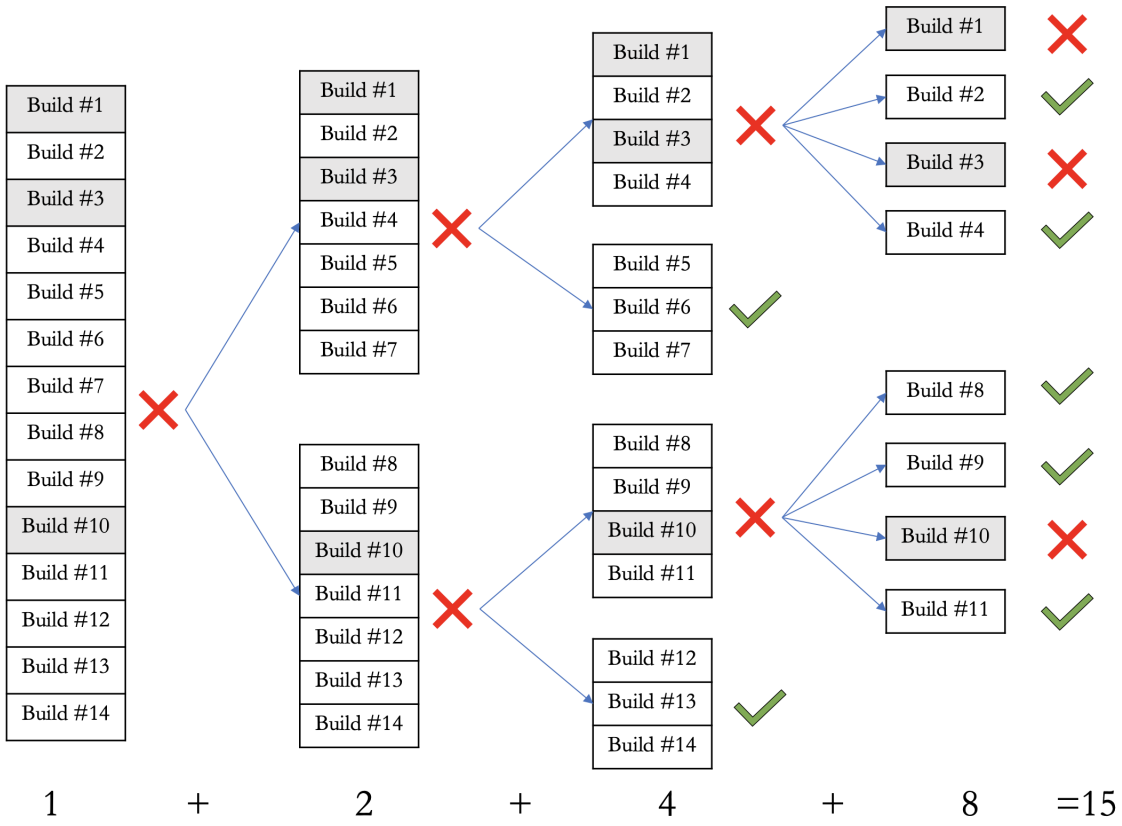

Figure 3.3: BatchStop4: Introduced by Beheshtian *et al.* uses a stopping condition for bisection when the size of a batch or sub batch is 4 or fewer. if the size of a batch is fewer than 4, instead of doing bisection, we test each build in isolation. In this example, at level 2 of the three, we approach this condition and execute each build in the broken batches in isolation. This reduces the total number of executions from 17 to 15.

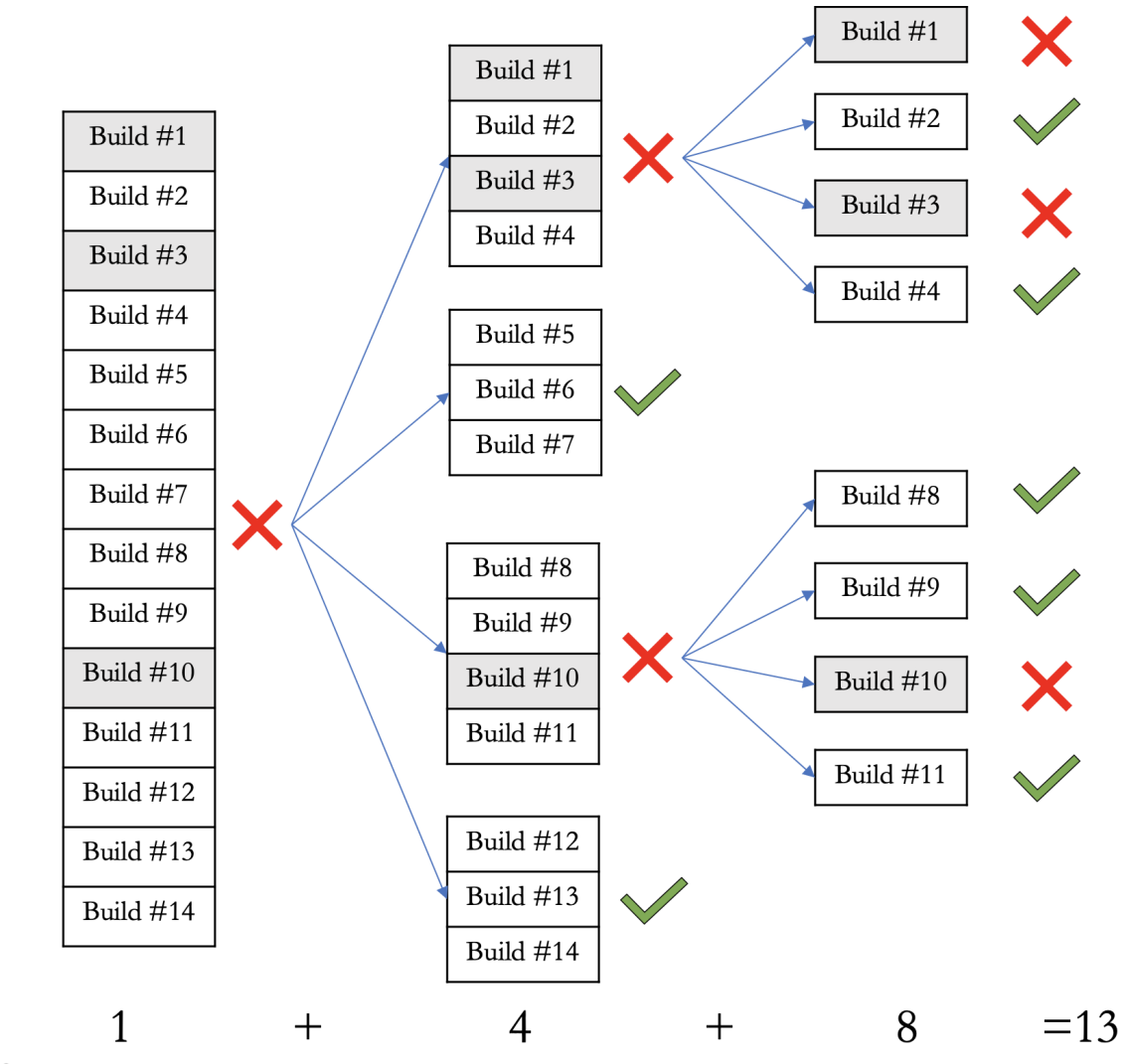

Figure 3.4: BatchDivide4: If the size of the batch is greater than 10, instead of doing bisection, we divide the batch into 4 sub batches. In this example, instead of doing bisection, after the failure of the original batch, we divide it into two batches with size 4 and two batches with size 3. This reduces the total number of executions to 13

3.2.1 TestAll

This is the naivest, yet the most common way of testing a software system in CI. In this technique, every single change in the code is tested individually. After a single pull request, the project is automatically built and test suites are run. The outcome shows whether the change is faulty. There is no batching or grouping builds together. The number of executions is the same as the total number of changes to the system. For example, if there are 100 changes in a repository, then there are exactly 100 build test executions. The failure rate of the builds does not change the build test execution numbers. We use TestAll as a reference point to allow easy comparison among techniques.

3.2.2 Dorfman Medical Pool Testing

Medical pool testing was introduced by Dorfman [34] during World War II to test soldiers for syphilis by mixing their blood samples in a batch. If the test is negative, none of the soldiers have syphilis. If the batch is positive, then each soldiers' sample is retested individually. In the case of software testing, we group builds together and test them in a batch. If the batch passes, all the builds in the batch are considered successful. However, if it fails, we run each of the builds in the batch in isolation to find the culprit builds.

3.2.3 Double Pool Testing

During the COVID-19 pandemic, due to the shortage of COVID test kits, double pool testing was proposed by Broder and Kumar [20] to reduce the number of required tests compared to Dorfman testing. They prove that double pool testing outperforms Dorfman pool testing for the populations that have a positivity rate less than 10%.

In the context of software testing for double pool testing, we place each build in two separate batches such that there is no overlap among batches. Figure 3.1, shows an example of double pool testing. We have 6 builds and Build3 is culprit. Build3 is in batches 1 and 3, which both fail. The only build that is in both these batches is Build3, so we know that Build3 is culprit and others are successful. However, to avoid complexity, Border and Kumar [20] always run the build fails in multiple batches in isolation. Using double pool testing, the total number of executions for testing

these 6 builds would be 5.

3.2.4 BatchBisect

Instead of testing commits individually on batch failure, we can use bisection to traverse the binary tree to find the culprits in a batch. In the medical world, this approach is called hierarchical group testing [77]. In the software engineering, batch bisection is widely used and we re-evaluate the prior works [70, 14] and propose a new BatchDivide4 approach.

With BatchBisect, we group builds together and test them in a batch. If the batch passes, all the builds in the batch are considered successful. However, if it fails, we perform bisection and traverse the binary tree to find the culprit builds in the batch. Figure 3.2 shows an example of BatchBisect. The original batch contains 14 builds. Builds 1, 3, and 10 are having failing tests. We first test this Batch of 14 builds all at once. After its failure, we divide it into two sub-batches with size 7. Both batches of size 7 fail. We perform another level of bisection and divide each batch into two sub batches with sizes 4 and 3. In both of the branches of the tree, the sub-batches with size 3 pass and sub-batches with size 4 fail. We then divide each of the failed sub batches into two sub-batches with size 2, we continue this process until finding the whole broken faulty builds. This process needs a total of 17 executions.

The maximum number of executions for BatchBisect happens when all the builds in the batch are the culprit. In this situation, the total number of executions will be $O(n)$. This means that all nodes in the binary tree should be visited. The minimum happens when only one build is culprit. For example, if in Figure 3.2 only Build 1 was culprit, then we would find it with the minimum number of executions. In this situation, the time complexity of the algorithm is $O(log(n))$.

As Najafi *et al.* [70] note, we cannot use a Git bisection because it assumes that the commits are ordered and that we only need to find the first failing commit for a failing test. In contrast, our goal is to integrate all commits. We do not have an order to the commits, *i.e.* none have already been tested, and we may have multiple culprits commits and multiple test failures, so even though a subbatch fails it does not imply that another subbatch will pass.

3.2.5 BatchStop4

Beheshtian *et al.* [14] proved that bisection is inefficient when there are 4 or fewer commits in a batch and introduced the BatchStop4 approach. BatchStop4 uses bisection when there are more than 4 commits to be batched, and uses TestAll when there are 4 or fewer. In the worst case, to find all the culprits in a batch of size 4, BatchBisect would need 7 executions. In contrast, if all four commits are tested individually after failure, we only need 5 executions.

Figure 3.3, shows an example of BatchStop4. Up to level two of the tree BatchStop4 uses BatchBisect. However, at level 3, instead of doing bisection, it tests each builds in a failed batch individually. It reduces the total number of executions from 17 to 15 in the example.

3.2.6 BatchDivide4

In BatchStop4, we do bisection until reaching batches with size 4 or fewer. We note that this is a general trend and can be applied to batches greater than 10. In our proposed algorithm Batch-Divide4, if the batch size is 10 or fewer, the algorithm is the same as the BatchStop4. However, in batch sizes higher than 10, instead of doing bisection, we divide each batch into 4 subbatches. In doing so, we combine two levels of search tree to prevent possible additional executions.

Figure 3.4 shows an example of BatchDivide4. In this example, instead of doing bisection in a batch with size 14, we directly divide it into four subbatches. It decreases the depth of the search tree by one level compared to BatchStop4 (contrast with Figure 3.3). In the example, BatchDivide4 reduces the total number of executions to 13 compared to 15 and 17 for BatchStop4 and BatchBisect, respectively.

3.3 Dynamic Batch Size Adjustment

Prior works used a constant batch size [34, 20, 70, 14]. A constant batch size fails to acknowledge the varying failure rates across time. For DynamicBatching we vary the batch size based on the historical failure rate of the project and dynamically update the batch size over the project lifetime. Intuitively, we know that the recent build outcomes of the project are more representative of the current state of the project. We weight recent failures more highly than older failures. The

WeightedFailureRate of the last C commits is

$$\text{WeightedFailureRate} = \frac{\sum_{c=1}^{C} \text{Smoothing(c)} * \text{IsFailure}_c}{\sum_{c=1}^{C} \text{Smoothingc(c)}} \tag{13}$$

where c shows the position of the commit relative to the current commit, and `IsFailure` is a function which returns 1 if the commit had at least one failing test and 0 otherwise.

For the `Smoothing` function we experimented with different weights including a constant weight of 1, $log(c)$, e^c, and $1/c$. With the exception of constant weight, there is marginal differences in effectiveness, and we report results with a $1/c$ weight.

We also varied the number of historical commits considered in the weighting function. We varied $C = 50, 100, 200$. We also used the last month and the entire history. We found that exact value of C has a marginal impact on the results with $C = 100$ being the best.

To calculate the batch size, we calculate the expected value for each batch size between $n = 1...20$ (Beheshtian *et al.* [14] show that batch sizes more than 20 are not effective), and choose the batch size that minimizes the expected value of the number of executions for testing a build in a batch. For each culprit finding approach, A, we calculate the following:

$$\text{BatchSize} = \arg\min_{n} \left(\text{ExpectedValue}(A, n, \text{WeightedFailureRate})\right) \tag{14}$$

To calculate the expected value, we perform a simulation. For each failure rate in $(0, 1)$ e.g. 0.01 and 0.02, we randomly generate passes and failures 100k times. We then vary the batch size to find the expected value of the number of executions for batch sizes between 1 and 20.

3.4 Data Sources and Evaluation Methodology

Our data comes from the Travis CI continuous integration service that is used to automatically build and test the projects on GitHub. To facilitate comparison we mine the build results from the same top 9 open-source projects from the TravisTorrent [18] as Beheshtian *et al.* [14]. The build outcomes on Travis CI are passed, failed, errored, and canceled. In this work, we divide the builds into successful, *i.e.* passed, or unsuccessful, *i.e.* all other build outcomes. In addition, since

Table 3.1: Size of projects under study and their failure rate

Project	Failure Rate	Tested Builds	Years
ruby	19.87%	14,180	5
metasploit	6.85%	7,736	4
graylog2	6.69%	2,105	4
owncloud	6.91%	882	2
vagrant	8.39%	4,036	4
gradle	9.23%	3,586	2
puppet	6.57%	2,311	4
opal	8.97%	2,551	4
rspec	17.65%	1,818	5
rails	32.78%	15,151	5
okhttp	40.30%	1,913	4
cloudify	24.09%	5,138	2

Beheshtian *et al.* [14] only picked projects with a failure rate below 20%, to generalize the result, we add the 3 largest project with a failure rate between 20% and 40% to understand the impact of high failure projects on batching. Table 3.1 shows the descriptive information for each project under study. The failure rate of the selected projects is between 6.57% and 40.30%. The largest project is Rails with 15,151 builds and the smallest one is Ownclowd with 882 builds.

We extract builds only from the master branch. The test suites run on master are constant and so any change to master can be combined because the same tests are requested. Furthermore, the commits to master have already been integrated, while other branches may have commits that would lead to a merge conflicts and cannot be easily batched. In this way, we do not introduce merge conflicts or change the test scope.

We use the first 100 commits from each project to train our dynamic algorithms. These commits are also excluded from the constant batch approaches to facilitate comparison of techniques. We run simulations for each culprit finding approaches with constant and dynamic bath sizes. We use an incremental simulation methodology [70, 19, 96, 49, 46]. We order the commits by time and batch them sequentially. If a batch fails we run the culprit finding technique. Our outcome measures is the percentage change in executions for each approach, A, relative to testing each commit individually [14]:

Table 3.2: Percentage savings in build test executions relative to TestAll. The batch size selection approaches are CB: ConstantBatching, and DB: DynamicBatching. The culprit finding approaches are BB: BatchBisect, BS4: BatchStop4, BD4:BatchDivide4, Dorfman: Drfman Medical Pool Testing, DPooling: Double Pool Testing, and Batch4. For example, with DB Pooling the batch size is selected using DynamicBatching and the culprits are found using Dorfman. Batch4 provides the same results as BatchDivide4 and BatchStop4, so we only show Batch4. We see that using DynamicBatching with BatchDivide4 (DB BD4) is the best performing combination.

Project	CB Batch4	CB DPooling(9)	CB BB(8)	CB Dorfman(4)	DB Pooling	DB DPooling	DB BB	DB BS4	DB BD4
ruby	37.20%	32.20%	28.17%	37.20%	38.90%	35.74%	36.74%	42.24%	42.89%
Metasploit	55.35%	58.44%	57.29%	55.35%	55.35%	54.03%	57.42%	60.41%	61.00%
graylog	56.15%	58.71%	57.10%	56.15%	56.65%	51.47%	60.04%	62.94%	63.29%
owncloud	57.70%	57.70%	55.62%	57.70%	59.33%	57.28%	62.14%	65.60%	65.72%
vagrant	56.96%	57.85%	58.51%	56.96%	60.28%	58.25%	62.04%	64.68%	64.81%
gradle	49.10%	58.54%	46.95%	49.10%	46.09%	39.73%	44.49%	49.05%	50.22%
puppet	55.60%	59.58%	56.98%	55.60%	56.67%	57.44%	58.07%	61.42%	61.73%
opal	50.68%	50.84%	48.63%	50.68%	49.12%	39.08%	48.83%	53.16%	53.89%
rspec	29.20%	20.35%	14.31%	29.20%	28.23%	22.58%	20.48%	29.51%	30.79%
rails	16.60%	02.40%	-04.75%	16.60%	17.63%	14.01%	11.09%	19.17%	19.93%
okhttp	06.79%	04.54%	-26.30%	06.79%	13.62%	08.93%	06.72%	14.83%	15.38%
cloudify	36.60%	28.94%	24.03%	36.60%	38.50%	28.58%	34.20%	39.34%	40.23%
Minimum	06.79%	04.54%	-26.30%	06.79%	13.62%	08.93%	06.72%	14.83%	15.38%
Average	42.32%	39.90%	37.04%	42.32%	43.36%	38.92%	41.85%	46.86%	47.49%
Maximum	57.70%	58.44%	57.29%	57.70%	60.28%	58.25%	62.14%	65.60%	65.72%

$$\begin{aligned} \text{ExecutionReduction(A)} &= 1 - \frac{\text{Executions(A)}}{\text{TotalCommits}} \\ &= 1 - \frac{\text{Executions(A)}}{\text{Executions(TestAll)}} \end{aligned} \tag{15}$$

We make our scripts and data available for replication [11].

3.5 Results

3.5.1 Result: ConstantBatching

ConstantBatching assumes a constant batch size for the entire lifetime of the project. We experimented with batch sizes from 1 to 20. Figure 3.5 shows the result for each culprit finding technique (we do not show BatchBisect because we mathematically show that BatchStop4 is always more

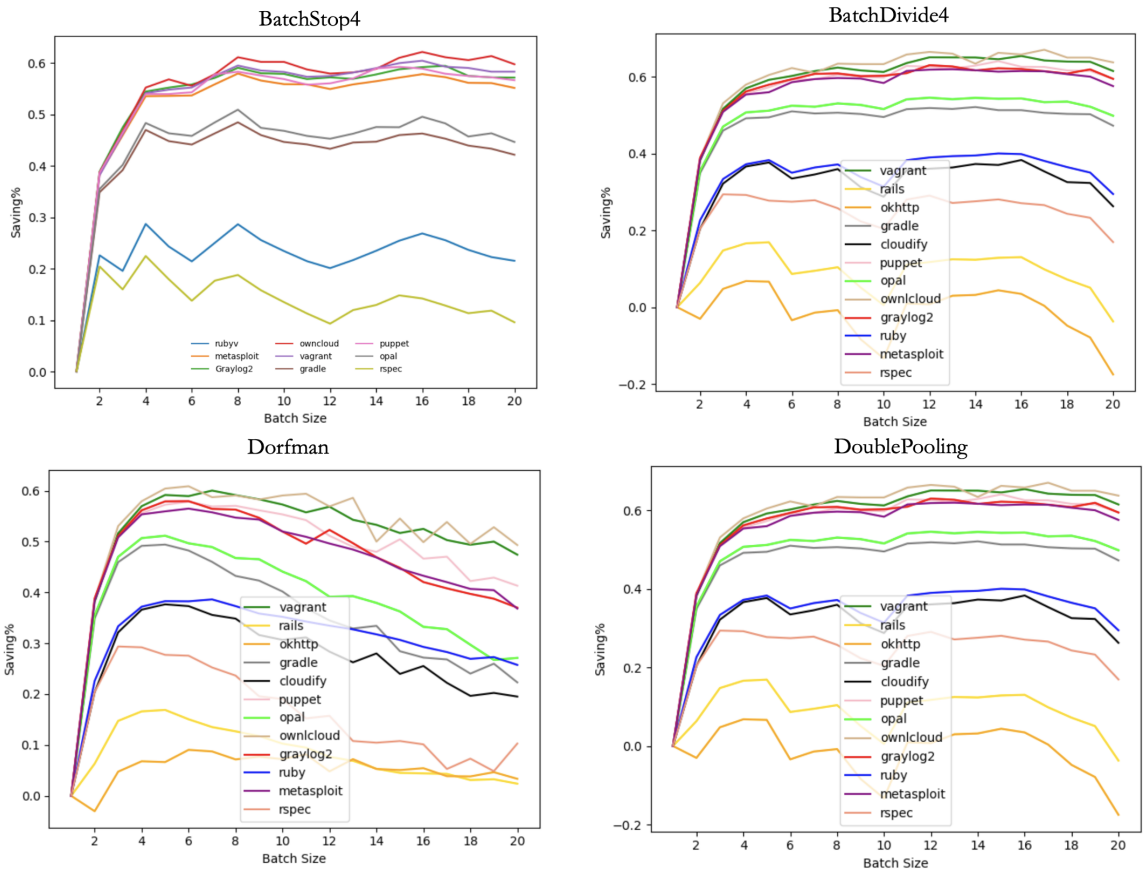

Figure 3.5: The result of ConstantBatching for different batch sizes for each of the projects under study. We see the majority of saving for all of the projects is achieved in a batch size of 4. After the batch size of 4, the number of required executions for some of the projects with higher failure rate increases e.g. Ruby. a few projects see an increase in resource saving. However, in average across the projects, using batches with size 4 can save more resources rather than other batch sizes. We also see that double pool testing outperforms other techniques in the projects with a failure rate below 10%.

efficient). We see curve is logarithmic with most of the savings achieved with small batch sizes. With a constant batch size, large batch sizes result in too many bisection making BatchBisect and BatchDivide4 ineffective.

The result is consistent with Beheshtian *et al.* [14], and few projects see a substantial increase in executions savings after a batch size of 4. The figure also shows that projects with higher failure rates actually see an increase in the required execution with larger batch sizes. For example, Okhttp see an increase in execution of -6.79% in batches of size 5, requiring more test executions than TestAll.

Double Pool Testing outperforms Dorfman testing in projects with a failure rate of below 10%. In Table 3.2 we see that Batch4, *i.e.* Dorfman(4), outperforms double pool testing by 2.42 percentage points across the projects. However, with a failure rate below 10%, double pool performs outperforms Batch4 by 2.87 percentage points on average.

The result shows that in comparison to TestAll, Batch4 can reduce the number of executions by an average of 42.32% across the projects. The savings are 37.20%, 55.35%, 56.15%, 57.70%, 56.96%, 49.10%, 55.60%, 50.68%, 29.20%, 16.60%, 6.79% and 36.60% respectively. The minimum saving is 6.79% for okhttp, and the maximum is achieved for Owncloud with 57.70% of saving.

> Batch4, *i.e.* Dorfman(4), is simple and effective with an a average of 42.32% executions across the projects. BatchBisect and BatchDivide4 are ineffective on these projects because failures require too many bisections. For projects with a failure rate below 10%, double pool testing outperforms Batch4 by 2.87 percentage points.

3.5.2 Result: DynamicBatching

DynamicBatching computes the next batch size based on the weighted failure rate of the previous builds. The weighting approach can be found in Section 3.3. Table 3.2 shows the result of DynamicBatching for each culprit finding approach. The average saving across the projects for Dorfman, double pool testing, BatchBisect, BatchStop4, and BatchDivide4 is 43.26%, 38.92, 41.85%, 46.86%, and 47.49% respectively.

Table 3.3: The theoretical limit as well as the CorrectedExecutions (CE) of each project for DynamicBatching(DB) and Batch4.

Project	Batch 4	DB BatchDivide4	Theoretical Limit	CE Batch4	CE DB
ruby	37.20%	42.89%	72.46%	51.33%	59.19%
metasploit	55.35%	61.00%	88.32%	62.66%	69.06%
graylog	56.15%	63.29%	88.74%	63.27%	71.32%
owncloud	57.70%	65.72%	89.11%	64.75%	73.75%
vagrant	56.96%	64.81%	88.13%.	64.63%	73.53%
gradle	49.10%	50.22%	84.69%	57.97%	59.29%
puppet	55.60%	61.73%	88.61%	62.74%	69.66%
opal	50.68%	53.89%	85.22%	59.46%	63.23%
rspec	29.20%	30.79%	71.17%	41.02%	43.26%
rails	16.60%	19.93%	56.48%	29.39%	35.28%
okhttp	06.79%	15.38%	46.88%	14.48%	32.08%
cloudify	36.60%	40.23%	70.39%	51.99%	57.15%
Minimum	06.79%	15.38%	46.88%	14.48	32.08%
Average	42.32%	47.49%	77.51%	51.97%	58.91%
Maximum	57.70%	65.72%	89.11%	64.75%	73.75%

Double pool testing is ineffective even on projects with a failure rate below 10% because even on these projects, there will be periods where the failure rate is above 10%

BatchDivide4 outperforms the other approaches and achieves per project savings of 42.89%, 61.00%, 63.29%, 65.72%, 64.81%, 50.22%, 61.73%, 53.89%, 30.79%, 19.93%, 15.38%, and 40.23% respectively. With low failure rates and large batches, it traverses the search tree efficiently using fewer executions to find the culprit. When the failure rate of the project is high and the batch size is small, it uses Batch4 and tests failing builds in isolation similar to Dorfman. This allows it to perform well in both high and low failure rate periods.

Compared to TestAll, DynamicBatching with BatchDivide4 reduces the number of executions by 47.49%. Compared to Batch4, which is the best ConstantBatching technique, the improvement is 5.17 percentage points.

Table 3.4: The ConsecutiveFailureRate of the projects under study

Project	ConsecutiveFailureRate
ruby	61.46%
metasploit	29.62%
graylog2	31.91%
owncloud	41.66%
vagrant	58.70%
gradle	33.93%
puppet	26.97%
opal	35.37%
rspec	36.70%
rails	67.24%
okhttp	67.92%
cloudify	77.14%

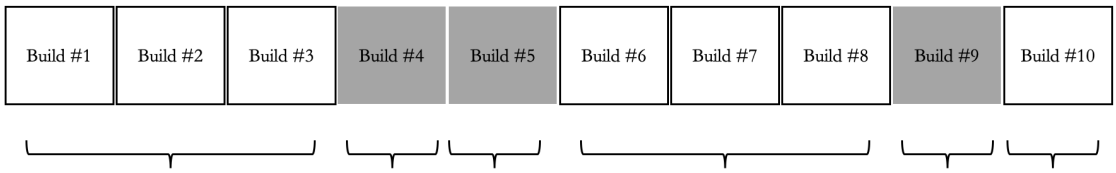

Figure 3.6: Theoretical Limit. In this example, Builds 4, 5 and 9 are broken. Without changing the order of the builds, the minimum number of executions occur when builds 1, 2, and 3 are tested together; builds 4 and 5 are tested in isolation; builds 6, 7, and 8 are tested together; and build 9 and 10 are tested in isolation. We test all 10 builds with the minimum number of 6 executions.

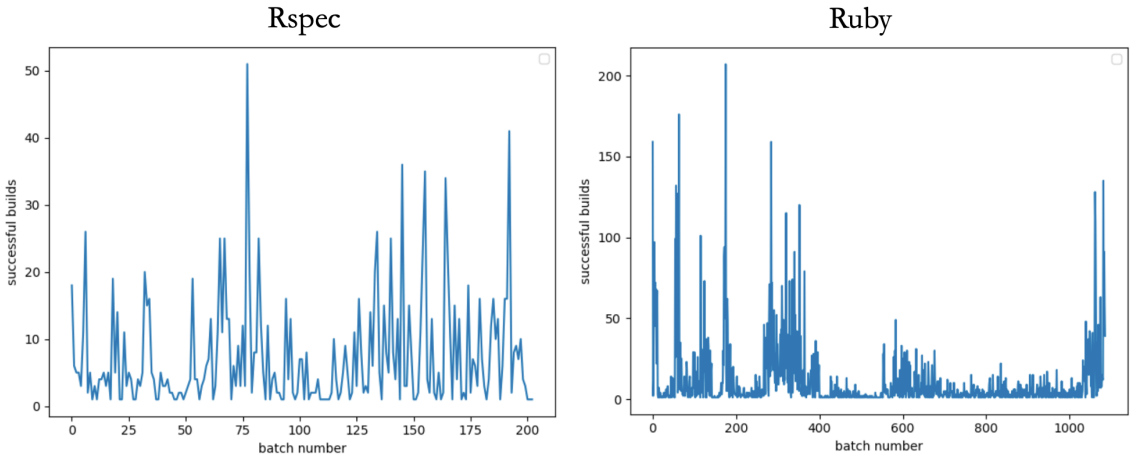

Figure 3.7: The number of successful builds between each two failures for Ruby and Rspec. We see the distribution of the failures are more even in Rspec rather than Ruby.

3.6 Discussion

3.6.1 Theoretical upper limit

We calculate the theoretical limit on the minimum number of executions that can be saved with batch testing. The optimal batch size is attained when we test each failing commit in isolation and group all the passing commits between failures. Figure 3.6 shows an example of the theoretical limit. Builds 4, 5, and 9 are failures, while the other builds are successful. If we group build 1, 2, and 3 together, we can test all three of them in one execution. Builds 4 and 5 fail and grouping a failing build requires culprit finding, so we test each of them in isolation instead. We then group builds 6, 7, and 8 and test them in a batch. Build 9 is tested in isolation and fails, so build 10 is also tested by itself. This process for testing 10 builds using 6 executions in total in contrast to the 10 required by TestAll and the 13 for Batch4.

Table 3.3 shows the theoretical limit for each project. We can see that it is between 46.88% for Okhttp and 89.11% for Owncloud. Based on this theoretical limit, we correct the reduction percentage by dividing the actual savings by the savings limit for each project:

$$\text{CorrectedReduction} = \frac{\text{ExecutionReduction}}{\text{TheoreticalLimit}} * 100 \tag{16}$$

where ExecutionReduction is equal to the saving percentage of our selected batching technique. If CorrectedReduction is equal to 100%, we achieve the theoretical limit.

We calculate CorrectedReduction for the current state of the art technique Batch4 and our best suggested technique, DynamicBatching with BatchDivide4. We show the result in Table 3.3. We can see that the average CorrectedReduction across the projects is 51.97% and 58.91% for Batch4 and DynamicBatching respectively. We can see that the difference between DynamicBatching and ConstantBatching is 6.94 percentage points.

On owncloud we have achieved 73.75% of the theoretical limit, while on Okhttp we can achieve only 32.08%. In both cases, we drastically reduce the number of execution required for testing, but the theoretical limit reveals a large potential for future work. We note that this theoretical limit may be difficult to achieve as we have evaluated state-of-the-art techniques from both medical

pool testing and software testing. It is possible that a classifier that predicts build failures with high accuracy could perform better. However, existing attempts with machine learning are more complicated and less successful than simpler batching [14]. We look forward to innovative batching strategies.

3.6.2 Relationship between the Failure Rate, Consecutive Failure, and Savings

Prior works assumed an even or normal distribution of failures [70, 14] and stated that the failure rate controls the possible batch savings. Although the failure rate of a project is a critical metric for the saving achieved by batching, it is not the only controlling metric for execution reduction. We find that most projects have times of consecutive failures followed by periods with few failing builds. To measure these consecutive failures, we define a new metric called ConsecutiveFailureRate for each project. It is the number of consecutive failing builds divided by the the total number of failing builds for the project. For example, in figure 3.6, the total number of culprit builds is 3, *i.e.* builds 4, 5, and 9. After build 4 that is a culprit, the next build *i.e.* build 5 is also a culprit. But the next build after builds 5 and 9 are successful. In this example, the ConsecutiveFailureRate is 33.33%

Table 3.4 shows the ConsecutiveFailureRate for each of the projects. The values are between 26.97% for Puppet and 77.14% for Cloudify. All projects see repetitive failures and do not see an even or normal distribution of failures as would be necessary for a constant batch size. To quantify the significance of consecutive failures to the failure rate, we define the spread of failures with following metric:

$$\text{FailureSpread} = \frac{\text{FailureRate}}{\text{ConsecutiveFailure}} \tag{17}$$

If all of the failures are consecutive and in a row, the FailureSpread would be equal to the FailureRate. In an extreme case, where the failure rate is 1 in 4 and the spread is perfectly even, a batch of size 4 will always fail and result in an additional execution. In this "perfect storm" batching will be ineffective. In contrast, if the failure rate is low, and the failures are consecutive, then large batch size will be effective.

To understand the effect of FailureSpread on saving in batch testing, we calculate the Spearman

correlation between ExecutionSavings vs FailureRate and FailureSpread. We find that the correlation between FailureRate and ExecutionSavings for our 12 projects is -0.86 with a $p \ll 0.0001$. This shows a strong correlation between failure rate and saving for a project. However, we also calculate the correlation between FailureSpread and ExecutionSavings and find an $r = -0.97$ with a $p \ll 0.0001$. This means that in addition to the overall FailureRate, ConsecutiveFailure is also a critical metric for execution reduction in a batch testing.

To illustrate the importance of considering the FailureSpread we examine Ruby and Rspec, that despite having similar failure rates have disparate executions saving: 43.42% and 31.54% respectively. Figure 3.7 shows the number of consecutive passing commits for both projects. For Ruby the failure rate varies dramatically, but remains consistent for periods of time. For example, between commit 400 and 600 and 700 and 1100, the failure rate is much higher than the earlier development on Ruby. Ruby is an ideal project for dynamic weighted batch sizes. In contrast, Rspec also has substantial variation, but there are no periods of constant failure rates making it difficult to do dynamic weighted batch sizing as the failure clusters are nearly random. We believe a promising area of further study will be using time series analysis to better model this variation in failure rates.

3.7 Threats to validity

External Validity. We choose our projects from the largest projects available on TravisTorrent dataset [18]. The projects are from a variety of development contexts from cloud computing to development tools. Unlike Najafi *et al.* [70], we choose projects with a wide range of failure rates. This range allowed us to understand that failure rate was not the only controlling factor on savings.

Internal Validity. In our simulations, we assume that there are always some builds waiting for testing. In other words, batching is only effective when there are builds in the queue waiting for resources. To accelerate testing these builds, we can run then in parallel machines or batch them. However, in some projects, especially small projects, there might not always be available some builds for batching. In this situation batching can lead to an increase in the time between submitting a change and receiving feedback. However, even small batch sizes lead to large savings, so if resources are available all waiting commits should be batched regardless of the batch size.

Also, we simulated the result using only a single machine. Having multiple machines to parallelize testing the queued builds can affect the result.

We have only batched commits to master that require the same test suites. In future work, researchers may want to examine how batching after test selection on large project effects the test scope, resource consumption, and feedback time.

3.8 Related Work

3.8.1 Test Selection and CI/CD

Continuous integration and delivery is an important part of developing modern software systems; and is beneficial for both industrial and open-source projects. It automatically runs the test cases after committing a change and integrates the changes to the code. This helps to provide faster feedback to the developers about their changes [78, 47, 88, 65]. In a Continuous integration pipeline, each commit needs to be tested to make sure it does not break the software functionality. However, testing every change is very costly and time-consuming [45]. Even companies with farms of servers *e.g.,* Google [37] cannot run all tests for every commit. To solve this problem, there are multiple techniques including test prioritization [9, 102, 84], test selection, test minimization [67, 91, 24], and batch testing.

Test selection is a technique to reduce the cost of regression testing [81, 40]. It reduces the number of tests and chooses only the most important tests that can reveal bugs in the code. Some studies investigate using static analysis approaches including code coverage [90] or slicing [48] for test selection purposes. They select and run only the tests that cover files with a higher probability of being faulty. Zhang [99] proposes a hybrid test selection strategy and instead of using a test selection in a just method level or file level, they use a combination. They show their hybrid approach outperforms other pure method level or file level strategies. Recently, there have been multiple strategies based on mining historical data and machine learning algorithms for implementing test selection. Spieker *et al.* [89] have proposed a reinforcement test case selection strategy. Chen *et al.* [23] propose a semi-supervised approach based on clustering for test case selection. However, the computational complexity of this approach is high and may not be feasible in very large industrial

software systems. Anderson *et al.* [5] use association rule mining. They extract and use various types of information in software systems and by conducting an empirical study on multiple industrial projects show that their approach can find the tests that are likely to reveal failures.

The saving for test selection is achieved by eliminating some of the tests being run. This selection, may result in slipthoughs as tests that fail may not be run. In contrast, with batch testing all tests are run and the savings comes from grouping changes not eliminating tests.

3.8.2 Batch Testing and Bisection

Batching is a technique for performing regression testing in a resource constraint environment [28, 22]. In this technique, we run the entire commits in the batch all at once. However, in the case of a failure of a batch, there is a need to find the culprit commits. GitBisection [1] can be seen as a form of batch testing where a binary search is used to find the culprit commit on a given failing test. The number of executions for GitBisection is $log(n)$ in which n is the number of commits in the batch. However, GitBisection has been designed for a situation that there is only one culprit commit in the batch. If the number of culprits in the batch is more than one, GitBisection would fail to find other culprits except the first one. To solve this problem, we can use a simple bisection technique. In simple bisection [70], we divide batches into sub-batches in every branch of the search tree to make sure that we find all of the culprits commits. In this scenario, however, the number of executions will be at least $2 * log(n)$ when there is only one culprit and $2 * n + 1$ in the worst-case scenario that all of the commits in the batch are culprits. To improve the speed of simple bisection, Beheshtian *et al.* [14] proposed BatchStop4 that has a stopping condition at the batches with size 4 or fewer. There are also risk-based techniques to prioritize testing the commits in a failed batch [14, 70]. However, Beheshtian *et al.* show that none of them can outperform simple BatchStop4. In this study, we introduce a new culprit finding technique BatchDivide4 and we show that it outperforms all of the previous approaches.

At companies with an extremely large codebase e.g. Google [103] even running a GitBisection could be extremely expensive. Google developers determine the dependency of builds using static analysis tools and find the tests that need to be run based on the files under change. In the case of failures, Google developers can find the files that are not related to the failed test cases and do not

investigate them. They also rank commits based on the distance between commits in a batch and the root of the build test dependency DAG. However, in this work, we are not able to perform such a process because we do not have the build dependencies or the ability to select tests within a test suite.

3.8.3 Medical Pool Testing

The batch testing idea originally comes from the medical research field. Dorfman introduced the idea of medical batch testing during world war II to reduce the number of kits for testing infected people [34]. By using one kit to test all the samples and having a negative result, we can make sure that no one in the population is infected. However, in the case of having a positive result, we have to test each of the samples individually using a separate kit. There are also other studies that use a similar technique to perform batch testing for medical analysis [41]. Recently due to the Covid-19 pandemic and lack of test kits in large populations, there has been an increasing interest in medical pool testing again. Aragón-Caqueo et al. [7] investigate the effect of batch testing on Covid-19 testing. Their conclusion is that the lower the infection rate, the higher the efficiency of pool testing. This is similar to the observations of our experiments in the context of software testing. Broader and Kumar [20] proposed the idea of double pool testing for Covid-19. In this approach, they put each sample in two pools. So in the case of being positive of a sample, the result of both of the pools that contain that sample must be positive. If one of the pools positive and another one is negative, this means that we know the sample which is both the pools is negative. Double pool testing was effective on software projects with low failure rates. The idea behind medical pool testing and software testing is similar. This means future researches in the area of software batch testing can be taken from the ideas of medical pool testing or vice versa.

3.9 Concluding Remarks

Testing every change is expensive and sometimes impossible for large companies *e.g.,* Ericsson[70] and Google[37]. In this work, we build upon previous research by Beheshtian *et al.* [14] and Najafi

et al. [70]. We also examine medical batch pool testing. For culprit finding, we examined the existing algorithms: Dorfman [34], double pool testing [20], BatchBisect [70], BatchStop4 [14], and we then propose our new culprit finding algorithm, BatchDivide4.

Batch testing requires a certain number of builds to be batched. All prior works used constant batch sizes, in this work we suggest a dynamic batch size adjustment solution. We calculate the WeightedFailureRate of the recent builds of the project. We prioritize the most recent builds over the older ones. We use this WeightedFailureRate to calculate the batch size which minimizes the expected number of executions for testing each build.

Our simulation on 12 large open-source projects that use Travis CI shows that DynamicBatching with BatchDivide4 outperforms the other approaches. It decreases the number of executions by 47.85% over TestAll with an improvement over Beheshtian's [14] Batch4 of 5.17 percentage points. We note that all the batching approaches are much more effective than testing each build individually.

We describe a theoretical limit for the savings that can be achieved in batch testing. We show that using DynamicBatching, we achieve an across project average of 58.91% of the theoretical limit. Although batching is highly effective, there is still substantial room for improving batching relative to the theoretical batch savings limit.

Najafi *et al.* [70] suggested that the failure rate dictates the potential savings for batching. We provide a more nuanced view showing that the failure rate varies overtime and that there are periods of consecutive passes and failures. We develop the FailureSpread metric that measures consecutive build failures and find that the correlation between batch savings and FailureSpread is $r = -0.97$ with a $p \ll 0.0001$. This metric easily allows developers to determine the potential of batching on their project. We make our scripts and data available for replication [11].

Chapter 4

Parallel and Batch Testing in a Continuous Integration Environment

As a manuscript thesis, this chapter is a verbatim copy of the paper *submitted* to ICSE 2022: the International Conference on Software Engineering.

4.1 Introduction

Testing is costly, time-consuming, and one of the most challenging parts of modern software development. In the past few years, many large companies transferred their testing process to a continuous integration pipeline. In a CI environment, to merge the changes to the main repository, we need to test every single change to ensure it does not introduce a new fault to the system [37]. However, testing each individual change is a costly process that can be prohibitively expensive on large software systems [70].

Batch testing groups changes and tests to reduce the number of redundant test runs. If the batch passes the tests, we save resources and merge the changes to the code base. If the batch contains a failing test, we have to perform a bisection to identify the culprit change(s). Identifying the culprit can add additional build test executions.

There have been recent works that study the impact of software batch testing. Najafi *et al.* [70]

proposed a new batching technique using machine learning to reduce the cost of testing at Ericsson [70]. Beheshtian *et al.* [13] propose BatchStop4 which currently is the state-of-the-art algorithm for culprit finding in a batch. However, previous works make three assumptions which are unrealistic on large software systems. First, they do not run tests in parallel and implicitly use a single machine. Second, after the failure of a batch, they rerun every test suites. However, this is unnecessary as we are aware of the failed test cases, and we do not need to run all the test cases that pass successfully. Third, previous researchers focus only on introducing new algorithms to reduce the resource costs of finding the culprit change in a batch. They did not perform an indepth investigation into the impact on the feedback time of batching.

In this work, we study the impact of parallel testing and batch testing on one of Ericsson's products. We have two input parameters: the degree of parallelization (*i.e.* the number of machines) and the batching techniques: *ConstantBatching*, *TestDynamicBatching*, and *TestCaseBatching*. Through simulation we evaluate the change in Feedback time, *i.e.* wall time, and Resource usage, *i.e.* CPU time. We present a brief overview of the batching algorithms and our results.

TestAll baseline: The most common testing approach is to test each change individually, *i.e.* TestAll. In this work, we simulate TestAll by varying the number of parallel machines. We find that by increasing the number of machines, the feedback time will decrease. Interestingly the relationship between machines and feedback time is nonlinear because the delays are compounded and are propagated to all changes waiting in the queue.

ConstantBatching: TestAll is expensive and often infeasible for large companies such as Ericsson [70] or Google [104]. To solve this problem, *ConstantBatching*, *e.g.,* Batch4 [13] groups 4 changes in a batch and tests the union of their required test cases. The common test cases requested by each change are run only once saving both CPU time and decreasing feedback time. If there is a failed test case, we need to identify the change that caused this failure using a culprit finding technique, *e.g.,* BatchStop4 [13]. Since only the failed test case needs to be re-run, there is often still savings.

We vary the batch size from 2 to 32 and the number of machines. We plot the tradeoff between machines and batch size, but note that large batch sizes are ineffective because machines can sit idle while waiting for changes and there is a greater likelihood of a failing batch that requires bisection.

However, in a resource constrained environment with few machines *ConstantBatching* effectively decreases feedback time.

TestDynamicBatching: Prior work on batching uses a constant batch size for the entire lifetime of a project [70, 13]. We suggest *TestDynamicBatching* which immediately runs a batch with all the changes in the queue, *i.e.* batch size = queue size. This approach adapts to the environment. With limited available resources, we can process changes sooner by creating larger batch sizes. In contrast, if the change queue is empty, any new change will be processed immediately in a "batch" of size 1. By adapting to the change queue, *TestDynamicBatching* reduces the average feedback time and resource consumption.

We vary the number of machines available to *TestDynamicBatching* in simulations. We plot the savings and discuss why the savings are nonlinear. We find that with only 4 machines we are able to decrease the feedback time by 80.38% and reduce the CPU usage by 32.29%.

TestCaseBatching: All prior works batch changes, which can increase the test scope as each change may request different tests. In contrast, we suggest batching at the test case level. We create a test queue that contains all the requested tests from the change queue. We then batch all the changes and run the first test in the test queue. After the test completes, we add any new changes that have arrived in the change queue. If the new change requests tests not already in the test queue, these are added to the test queue. Once a change has had all its tests run, the verdict is reported for that change. In contrast to change batching that must run all the tests for the changes before making a new batch, *TestCaseBatching* ensures that the change queue is always emptied every time an individual test completes. We find that **TestCaseBatching** provides feedback an average of 19.47% and 84.20% faster than *TestDynamicBatching* and *TestAll* and reduces the CPU resources by 2.82% and 34.19% with four machines, respectively. TestCaseBatching is highly effective at Ericsson.

This article is structured as follows. In Section 4.2, we provide definitions, background, and the advantages and disadvantages of each batching algorithm. In Section 4.3, we describe our data, simulation methodology, and evaluation metrics. In Section 4.4, we describe the results from the simulation for each batching algorithm with a variable number of machines. In Section 4.5, we discuss threats to validity. In Section 4.6, we position our work in the literature. In Section 4.7, we conclude the paper.

4.2 Background and Definitions

This paper evaluates the change in Feedback time, *i.e.* how long it takes for a final test verdict, and the CPU time, *i.e.* how many machine cycles are necessary to test the change. We vary the number of machines and the batching algorithm. We describe our metrics and algorithms below.

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

$$\text{FeedbackTime} = \text{Time}_{\text{TestVerdicts}} - \text{Time}_{\text{Commit}} \tag{18}$$

For example, if a developer commits a change at 9 am and receives the feedback that the tests passed successfully on that change at 10 am, the feedback time will be 1 hour for that change. In contrast, if the change was queued for 1 hour, then the feedback time would be 2 hours, a doubling in feedback time.

CPU time. The CPU time is the number of CPUs that are in use to test a change. For example, if we are running 3 batches then we will be using 3 machines. If each is in use for 2 hours then the CPU time is $2 * 3 = 6$ hours. CPU time is formalized below:

$$\text{CPU Time} = \sum_{m=1}^{M} \text{CPU}_m \tag{19}$$

Number of machines. Testing large software systems is time-consuming. To solve this problem, companies such as Google [37] parallelize their test suites. The test parallelization process distributes the tests across multiple machines to reduce the feedback time [63]. Suppose a developer commits a change. To verify and merge the commit to the main branch, it needs to pass tests A, B, and C. Each test takes 1 hour to run. If we use a single machine, testing the change will take 3 hours. If the testing starts immediately after the committing, the feedback time will be 3 hours. However, by using 3 machines, we can execute each test on a different machine in parallel. As a result, the testing process will take only 1 hour, and the feedback time will decrease by 2 hours. However, in both cases, the CPU time will still be 3 hours ($1 * 3 = 3 * 1 = 3$ hours). In this work,

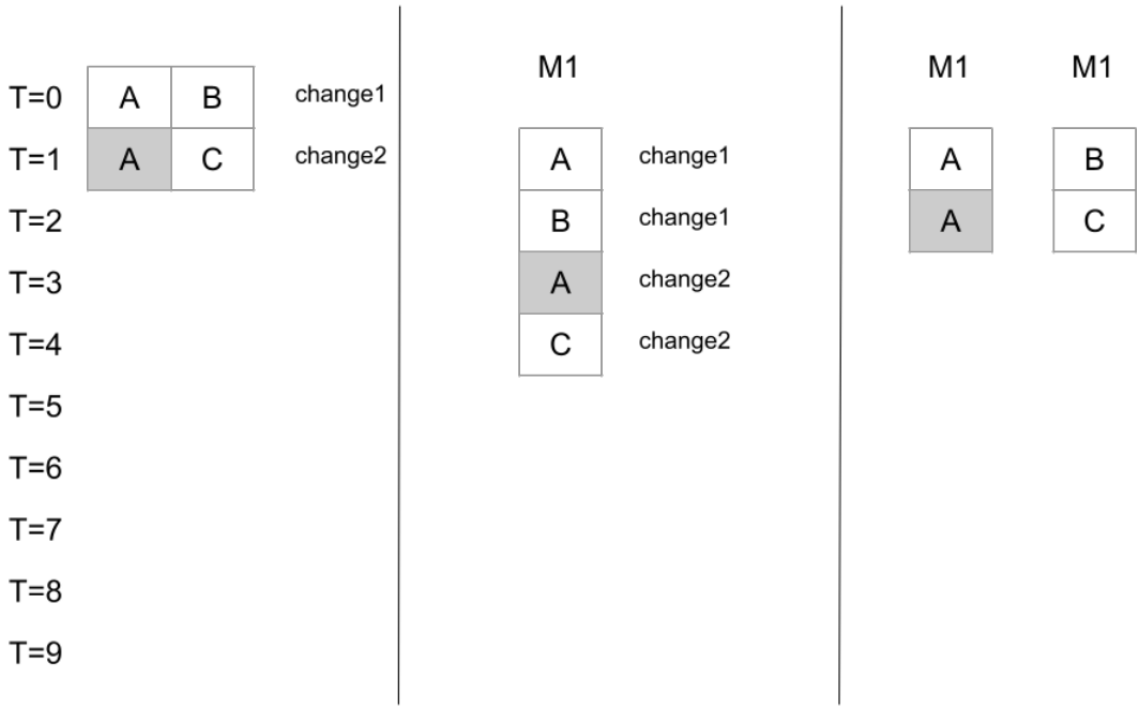

Figure 4.1: TestAll: In this example, the total number of changes that are tested is 2. The feedback time for change1 and change2 by assuming that there is only one machine will be 2 and 3 respectively. By assuming two machines for parallel testing, the feedback time for both of the changes would be 1. In both scenarios, the total CPU time would be 4

we vary the number of machines and simulate the impact on Feedback time and CPU time.

4.2.1 Background on batching

Although parallel testing can help to reduce the feedback time, even at large companies using farms of servers to run tests in parallel, they still need to batch changes to further reduce CPU time [104]. We describe the batching algorithms that are used in practice and introduce novel ones.

4.2.2 TestAll

Ideally, each change would be tested immediately and in isolation. This approach works well on small projects that are not resource constrained. The total CPU time for this technique is equal to the total execution time of the tests that have to be run for each change. However, the feedback time for each change varies and depends on the time that a change waits in the queue.

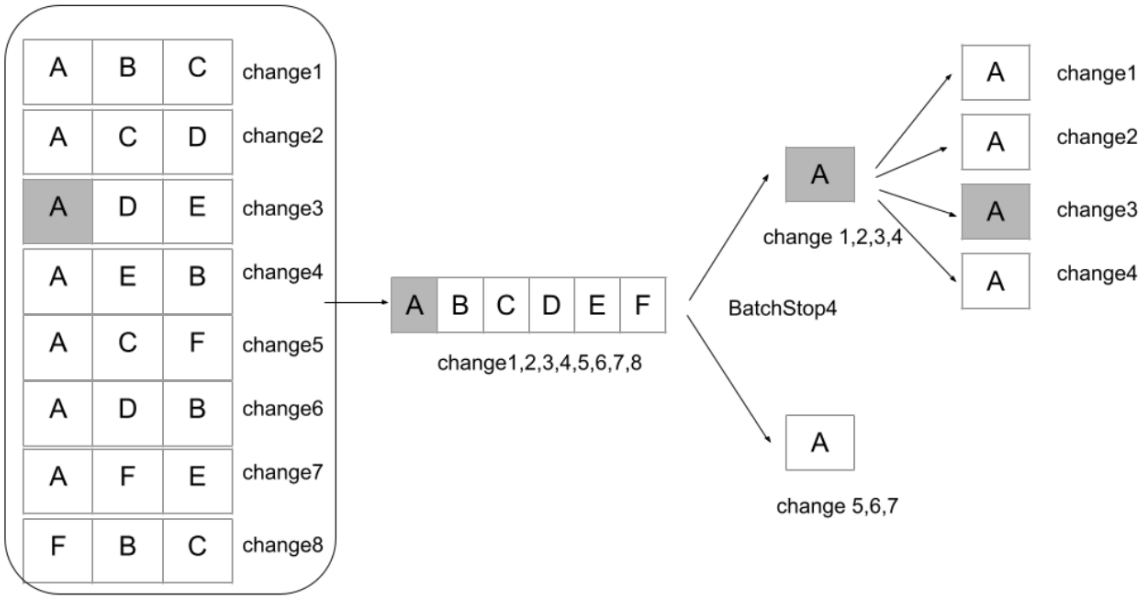

Figure 4.2: Batching and BatchStop4: In this example, there are 8 changes needing to be tested. To test them all using batching, we combine them and execute the union of their required test cases on a batch. We see five tests will pass and test A fails. To find the culprit changes we perform a BatchStop4 on the changes and test A. The total number of executions for testing using batching would be 12 compared to TestAll which needs 24 executions, we can reduce the 50% of the number of executions

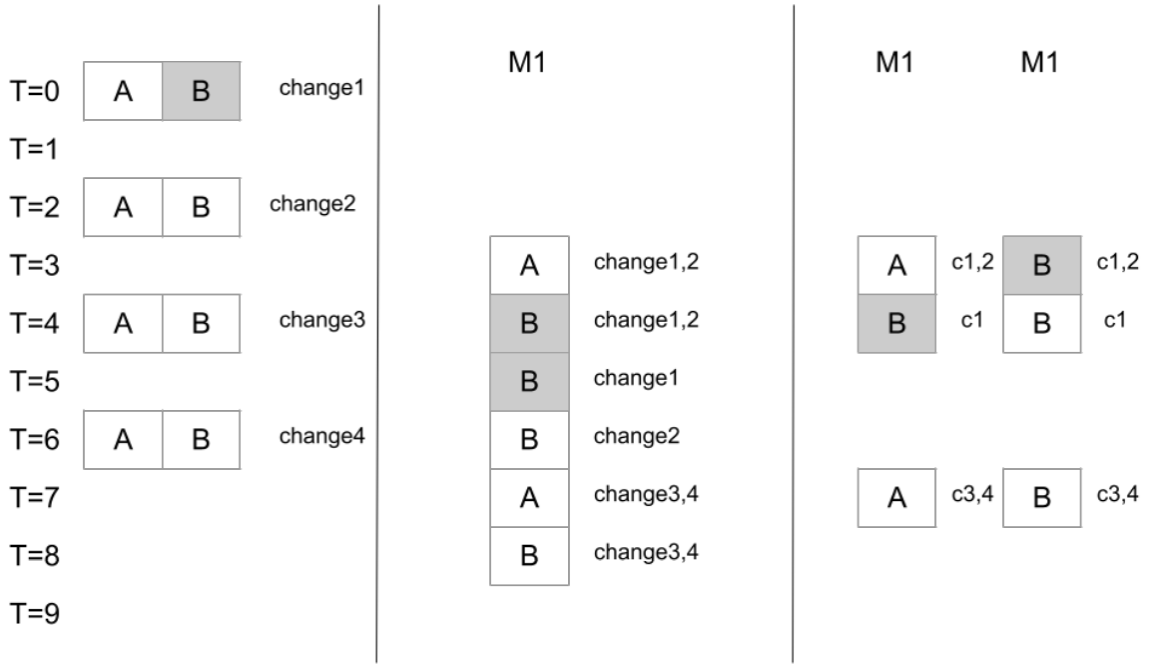

Figure 4.3: *ConstantBatching*: In this example, we use batches of size 2 *i.e.* batch2. Using a single machine, the feedback time would be 5, 4, 4, and 2 respectively. Using two machines for parallel testing, the feedback time would be 4, 2, 3, and 1 respectively. In both situations, the CPU time is 6. If there are fewer changes than the batch size, then there will be unused resources.

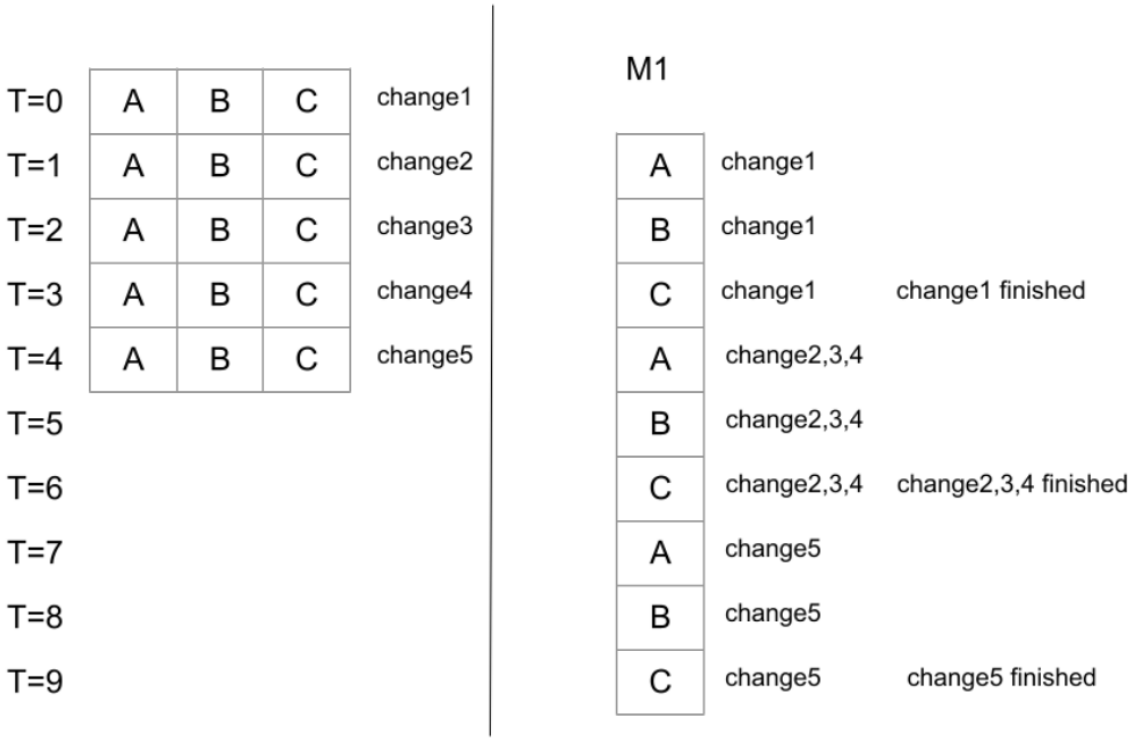

Figure 4.4: *TestDynamicBatching*: After committing the change1, we immediately allocate resources and start testing it. As a result, the size of the first batch would be 1. After finishing testing the first batch, there are 3 changes waiting for the testing process, i.e., change 2, 3 and 4, and we batch all three. Change4 is waiting when the batch finishes so we would start the third batch of size 1. The total resource usage would be 9. The feedback time for each change would be 3, 5, 4, 3, and 5 respectively.

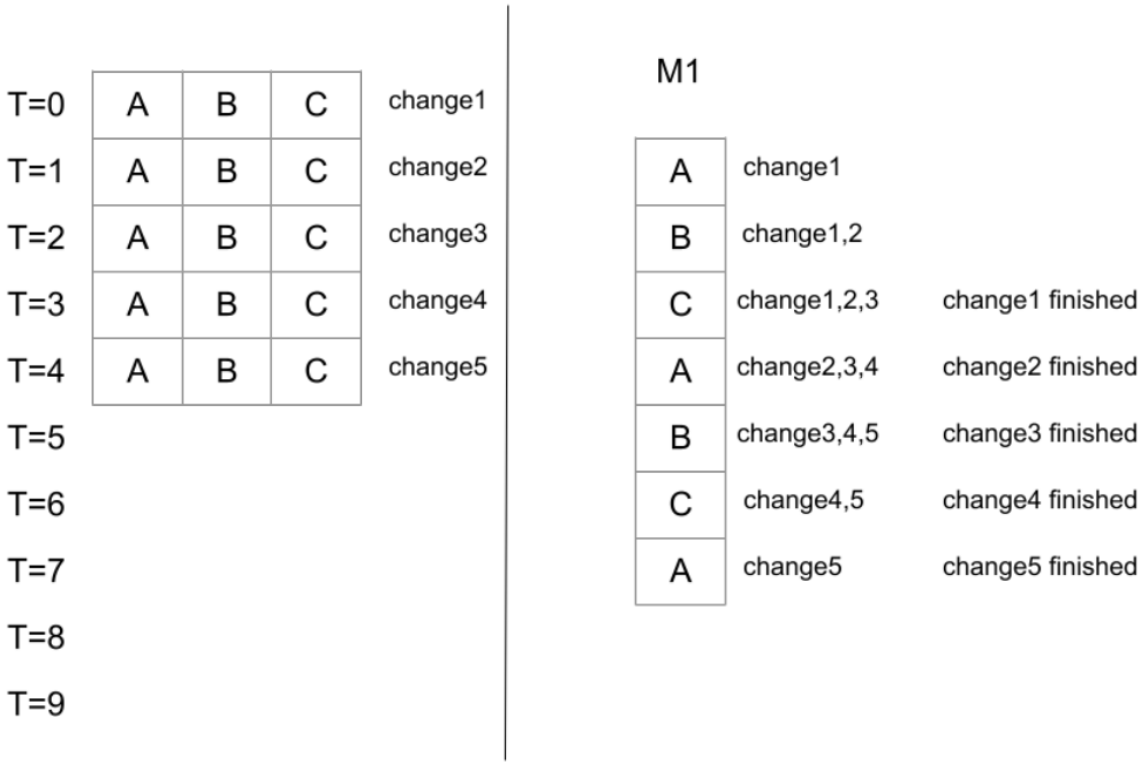

Figure 4.5: *TestCaseBatching*: After each test finishes, we include any waiting builds and run the next requested test in the test queue. We have to run Test A three times, because it has finished for Change1 before Changes 2, 3, and 4 and we have to run it independently for Change 5. In contrast, we must only run B and C twice as they overlap when more changes are available. The feedback time for each change is 3. The total CPU time is 7.

Figure 4.1 shows an example of TestAll. In this example, there are two changes that are committed for testing. The time of committing each change is T=0 and T=1 for Change 1 and Change 2 respectively. Change 1 requests tests A and B. Change 2 requests tests A and C. Executing each of the tests needs exactly 1 unit of CPU time. We run each of the tests for each change individually. Each change has exactly two tests. As a result, the total CPU time for testing this example regardless of the tests' outcome is

$$\text{TCT}_{\text{TestAll}} = CT_A + CT_B + CT_A + CT_C = 4 \tag{20}$$

In which CT means CPU Time and TCT means the Total CPU Time for testing these two changes.

In contrast, the feedback time will vary depending on the number of available machines. In Figure 4.1, we show the result of testing Change 1 and 2 on a single machine as well as having two machines for parallelization. In a single machine environment, after committing Change 1, the tests run. At time $T = 2$, the execution of both the tests A and B finish. As a result, the feedback time for Change 1 is equal to 2 units of time. After finishing the testing process of Change 1, we start running tests A and C for Change 2. The results of the test executions will be available at T=4. As the time of committing Change 2 is $T = 1$, the feedback time for this change will be equal to 3 units of time.

Using two machines, M1 and M2 will change the feedback time. We show an example of this situation in the Figure 4.1 for TestAll. For testing Change 1, test A is run on M1, and test B at the same time is run on M2. Due to test parallelization, the result of the tests will be ready at T=1. The feedback time for Change 1 is equal to 1 unit of time. After finishing the testing process of Change 1, the tests A and C of Change 2 are executed on machines M1 and M2 in parallel, respectively. The result will be available at T=2. As a result, the feedback time for Change 2 is equal to 1 unit of time.

We can see that the average feedback time by having a single machine is equal to 2.5 units of time. However, using two machines and parallel testing, the average feedback time will decrease to 1. In both situations, the CPU time remains constant at 4.

4.2.3 Batch Testing and BatchStop4

TestAll is expensive and sometimes impossible in large companies, *e.g.,* Ericsson [70] or Google [104]. Instead of testing every change individually, we can combine multiple changes and run the union of their requested tests in a batch. If a batch passes, we save resources and provide feedback more quickly. However, if a test fails, we need to find the culprit change(s) that are responsible for the failure. If the intersection of the requested tests for the batched changes is large, and most tests pass, the saving could be substantial. In an extreme example, if we batch 50 changes, and each change requests the same tests, when the batch passes we save 49 build test executions. This results in a reduction in CPU time of 98%.

If the batch fails, we need to bisect and rerun tests to find the culprit change(s). There are several approaches for culprit finding in a failed batch including Dorfman [34] for medical pool testing, bisection [70], and the current state-of-the-art BatchStop4. BatchStop4 [13] has been shown both mathematically and empirically to be the top performing culprit finding algorithm when using constant batch sizes. Figure 4.2 shows an example of batch testing and BatchStop4 for culprit finding. To simplify the example, we focus the example only on the process of BatchStop4 for culprit finding and do not show the feedback time or the number of machines. For the example, there are a total 8 changes that need to be tested. For each change, we need to execute 3 tests among the set of tests {A, B, C, D, E, F}. The execution time for each test is 1 unit of time. The requested tests for each change vary. Using TestAll, we need to run every test for every change, which requires a total of 24 test executions. However, by combining all the changes, we run the union of their test cases which means running 6 test cases. If the batch passes for all tests, we would save in total 18 executions. This means saving 75% in terms of CPU time. However, with a failure of a test, *e.g.,* test A which needs to pass in 7 different builds, we need to localize the faulty builds. We apply BatchStop4 to test A and find the culprit changes that are responsible for the failure of the batch. After finishing the process, the total number of test executions for testing all the 8 builds would be 12, which leads to a saving of 50% of the resources compared to TestAll.

4.2.4 ConstantBatching

Prior works have selected a constant batch size for testing [70, 13]. In the *ConstantBatching* technique, we group n changes together and test them in a batch. For example, with $n = 8$, we batch every 8 changes together for testing. Figure 4.3 shows an example of Batch2 using a single machine as well as having two parallel machines. There are in total 4 changes. The time of committing the changes is equal to T=0, T=2, T=4, and T=6 respectively. Similar to the previous examples, for simplicity we assume each test execution takes 1 unit of time. For both the single machine and two machines environments, the CPU time is equal to 6 hours. In a single machine configuration, the feedback time for each build would be 5, 4, 4, and 2 units of time respectively. By using two machines, the feedback time would be 4, 2, 3, and 1 unit of time respectively.

We can see using two parallel machines, there is a time that machines are free and no test has been assigned to them for execution. Although it does not have any effect on the CPU time, it affects the feedback time for Change 3. As the approach is Batch 2, even though there are resources available, we have to wait until there are two changes available for creating a batch.

4.2.5 TestDynamicBatching

The assumption of a constant batch size introduces problems. First, the rate of committed changes varies over time. For example, during the peak of the workday, there may be 1000's more commits than at night. We need to vary the batch size based on the change queue. In *TestDynamicBatching*, when there are resources available, all the waiting changes, are grouped and the union of required tests are run for the batch. When the testing process of the batch finishes and the corresponding resources are free, another batch is created using all the current waiting changes and the resources are allocated to the new batch. The Algorithm 2 shows the high-level procedure for *TestDynamicBatching*.

Figure 4.4 shows an example of *TestDynamicBatching* with a single machine. We assume that there is no failure and all the changes pass the tests. Using *TestDynamicBatching*, after committing the first change, the resources are immediately allocated to it for testing. Change 1 arrives first, and only after we finish testing Change 1, can we batch all the changes that are now waiting, *i.e.* changes

Algorithm 2: TestDynamicBatching procedure for batching and resource allocation

```
Whenever there are resources available for testing;
Pull all the waiting changes from change queue;
compute the union of the test cases that need to be run on the changes;
Run all the tests on the batch of the changes;
if all the tests pass then
    consider the batch as a passing batch ;
else
    for each test that fails do
        Perform BatchStop4 on that test to find the culprit changes
    end
end
Release all the busy resources;
Repeat the process for other changes in the queue;
```

2, 3, and 4. After testing the second batch of size 3, we test Change 5 in a batch of size 1 because no other changes are waiting for testing. The total CPU time will be 9 units of time. However, the feedback time for each change will be 3, 5, 4, 3, and 5 respectively.

4.2.6 TestCaseBatching

TestDynamicBatching can decrease the feedback time by reducing the idle time of resources. However, when all resources are utilized for testing, new changes must be queued until *all the tests* for the current batch complete. With *TestCaseBatching* new changes are added to the batch when any test finishes rather than having to wait for all the tests to finish. This approach requires the requested tested to be queued. To manage the test queue, the requested test cases for each change are added to the queue, *i.e.* the ChangeID, TestID. When a test finishes, any new changes are added to the batch and the next test in the queue is run. Once a change has had all its tests run, the results are reported (see Algorithm 3).

In Figure 4.5 we provide an example of *TestCaseBatching*, and see that we need only 7 CPU time units compared to the 9 need for *TestDynamicBatching* a reduction of 22.23%. The average feedback time is reduced to to 3 compared to the 4 needed for *TestDynamicBatching*, meaning that we get feedback to developers 25% sooner.

Algorithm 3: TestCaseBatching procedure for batching and resource allocation

test queue management;

Whenever a change is committed;
compute the required test cases for that change;
add the required test cases to the test queue with the format (testID, changeID);
build a version of software that includes the current change;

---

resource management;

---

whenever the resources are available;
pull a test from the queue;
pull all the tests with the same testID from the queue;
if *all the tests pass* **then**
  consider the test as a passing test ;
else
  Perform BatchStop4 on that test to find the culprit changes;
end
Release all the busy resources;
Repeat the process for other builds in the queue;

4.3 Project, Data, and Simulation Methodology

In the previous sections, we describe the theory behind batching and introduced our metrics. These approaches can be applied to any project that uses continuous integration of changes and has resource constraints. In this section, we provide the background on the Ericsson project used in this study.

Testing at Ericsson is an expensive multistage process. In order for a chance to be integrated into the released product, it has to pass multiple testing levels. In this paper, we focus on Confidence level 2 and Confidence level 3, which are the second and third phase of integration testing.

Ericsson tests the software that runs on cellular base stations. In this context, the machines used for testing are extremely expensive and limited in number. The unit of integration testing is called an Upgrade Packages (UP) and is the unit for our study. Each upgrade package consists of smaller change units that are grouped into Load Module Containers (LMC). Every day there are thousands of changes grouped into LMCs and into UPs for integration testing. We evaluate the integration testing phase at the UP level for a project at Ericsson for the period of six weeks from January

to February 2021. This time period includes over 11,000 changes. For each change, we have the test cases that are run, the outcome of each test case, the execution time for each test case, and the time of creating each change. Over this time period the average feedback time was 8.33 hours. The number of available machines is difficult to determine exactly because the test machines are also used by other projects, however, our simulations results provide approximations of available machines based on the processing time.

The data necessary to conduct the simulation is not company specific and is simply the time a change, *e.g.,* UP, is ready for test, the requested tests, and the running time for each test. We use the common incremental simulation methodology that is used in software engineering literature [70, 19, 96, 49, 46]. For *ConstantBatching*, we simulate the batches of sizes from 2 to 32. We note that above batch size 8 there is little to no improvement. We varied the number of machines from 1 to 16. Although 16 seems like a small number of machines, the specialized hardware makes it impossible to increase resources beyond this number. Furthermore, we will show that after 9 machines there is little to no decrease in feedback or CPU time, and additional resources would be underutilized.

4.3.1 Outcome measures

Our goal is to reduce both feedback time and resource consumption. We define the *AvgFeedback* and the *AvgCPU*. The *AvgFeedback* is the sum of the feedback times for each change in the project divided by the total number of changes for the project. The equation below shows the *AvgFeedback* for batching algorithm A across C changes with m machines.

$$\text{AvgFeedback}_m(A) = \frac{\sum_{c=1}^{C} \text{Feedbacktime}_c}{c} \tag{21}$$

In the example in Figure 4.1, the *AvgFeedback* with a single machine is 2.5 units of time.

The average CPU time is the sum of the CPU time for each change in the project divided by the total number of changes, C. For batching algorithm A with m machines the *AvgCPU* is

$$\text{AvgCPU}_m(A) = \frac{\sum_{c=1}^{C} \text{CPUTime}_i}{c} \tag{22}$$

Table 4.1: The *FeedbackReduction* for each algorithm relative to TestAll with a m machines. With the exception of 1 or 2 machines, *TestCaseBatching* has the greatest *FeedbackReduction*. *ConstantBatching* can reduce the Feedback time in an extreme resource constraint environment, but ineffectively utilizes additional machines.

Algorithm	m=1	m=2	m=3	m=4	m=5	m=6	m=7	m=8	m=9
Batch2	27.25%	33.45%	42.39%	55.90%	67.50%	49.64%	-3.33%	-31.09%	-51.48%
Batch4	44.08%	52.80%	65.49%	74.64%	66.06%	28.75%	-65.99%	-127.69%	-174.79%
Batch8	49.02%	56.73%	65.38%	66.96%	43.95%	-28.52%	-215.75%	-345.87%	-450.70%
TestDynamicBatching	40.15%	56.57%	72.27%	80.38%	78.46%	64.43%	31.56%	20.25%	15.78%
TestCaseBatching	32.85%	55.56%	73.67%	84.20%	82.13%	69.37%	38.69%	26.73%	19.84%
Minimum	Batch2	Batch2	Batch2	Batch2	Batch8	Batch8	Batch8	Batch8	Batch8
Maximum	Batch8	Batch8	TestCase	TestCase	TestCase	TestCase	TestCase	TestCase	TestCase

In the example in Figure 4.1, the *AvgCPU* with two machines is 2 units of CPU time.

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

$$\text{FeedbackReduction}_m(A1, A2) = (1 - \frac{\text{AvgFeedback}_{A1}}{\text{AvgFeedback}_{A2}}) * 100 \tag{23}$$

The *CPUReduction* for algorithm $A1$, is relative to percentage decrease in *AvgCPU* compared to to algorithm $A2$ with m machines.

$$\text{CPUReduction}_m(A1, A2) = (1 - \frac{\text{AvgCPU}_{A1}}{\text{AvgCPU}_{A2}}) * 100 \tag{24}$$

We use the historical test outcomes of 11k changes and to understand how each batching algorithm would have changed the *AvgCPU* and *AvgFeedback*.

4.4 Results

4.4.1 Result: TestAll

As a baseline we simulate testing each change individually, we run the historical simulation and vary the number of machines and plot the *AvgCPU* as well as *AvgFeedback*. In Figure 4.6 the *AvgFeedback* for TestAll is 721.60, 290.15, 146.33, 75.78, 35.30, 13.76, 5.18, 3.45, 2.66 hours for 1 to 9 machines, respectively. As we can see, the *AvgFeedback* is reduced dramatically with the

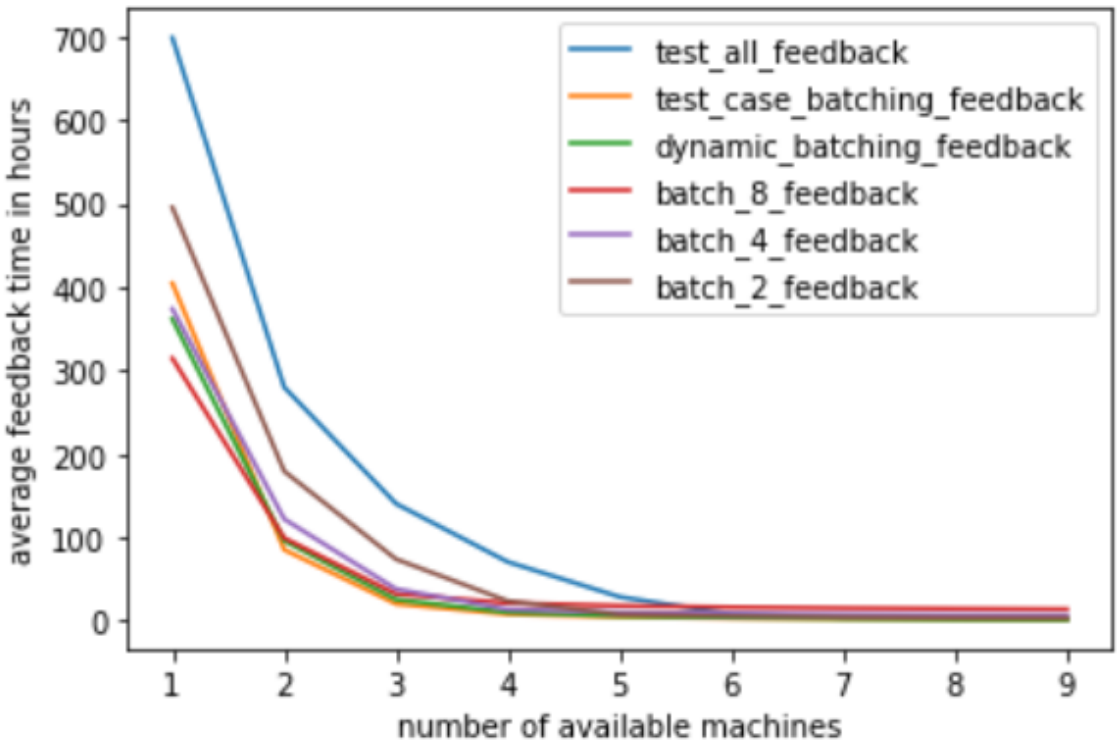

Figure 4.6: The average feedback time(*AvgFeedback*) for each algorithm with a variable number of machines. We see by increasing the number of parallel machines, the feedback time will decrease. However, because of the delay consolidation in future changes, the relation is nonlinear.

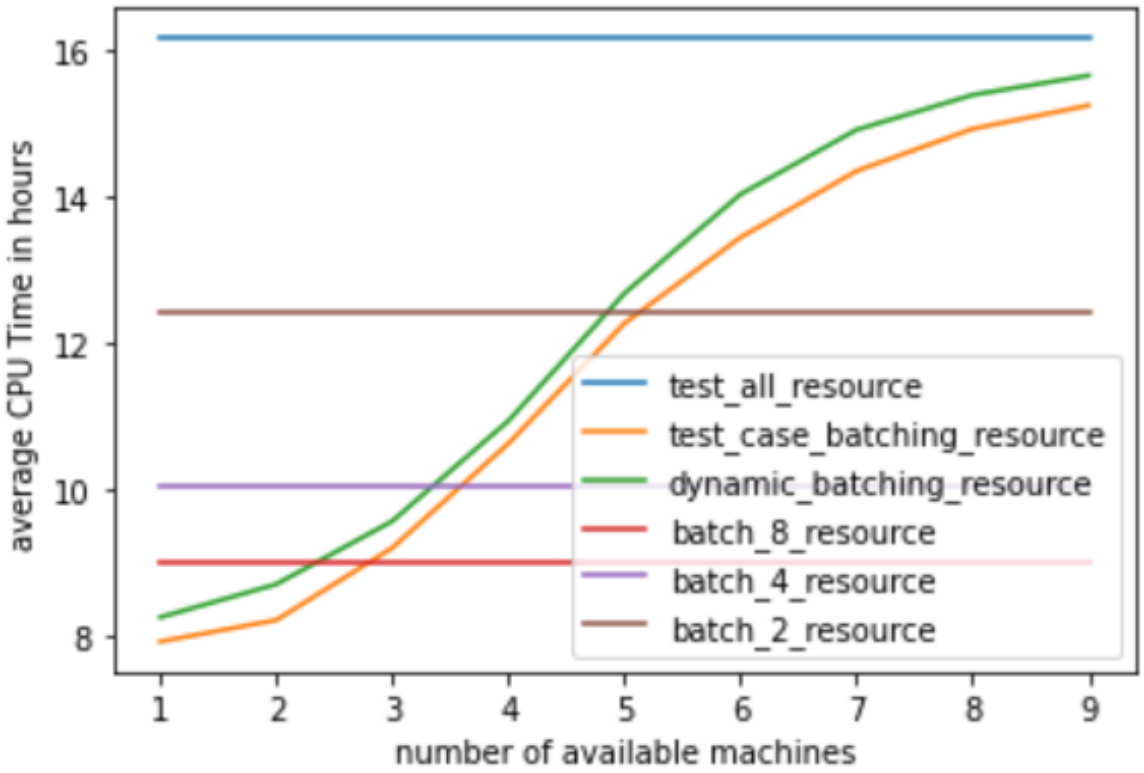

Figure 4.7: The average CPU Time(*AvgCPU*) of each algorithm varying the number of machines. We see by larger batch sizes save more resources. Since *TestDynamicBatching* and *TestCaseBatching* attempt to maximize the utilization of each machine, their *AvgCPU* will increases with more machines.

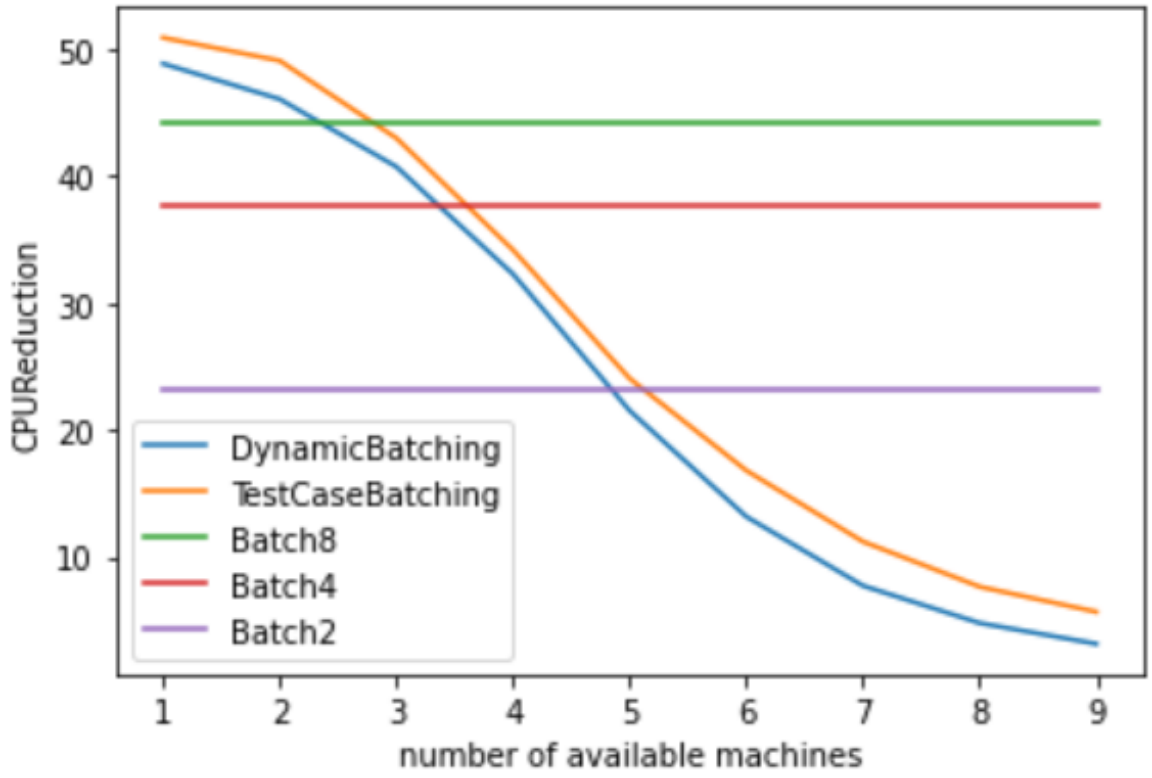

Figure 4.8: The *CPUReduction* of each algorithm relative to TestAll. We see the *CPUReduction* for Batch2, Batch4, and Batch8 is 25%, 40%, and 50% respectively. However, for *TestDynamicBatching* and *TestCaseBatching*, it is not constant and decreases as we increase the number of machines.

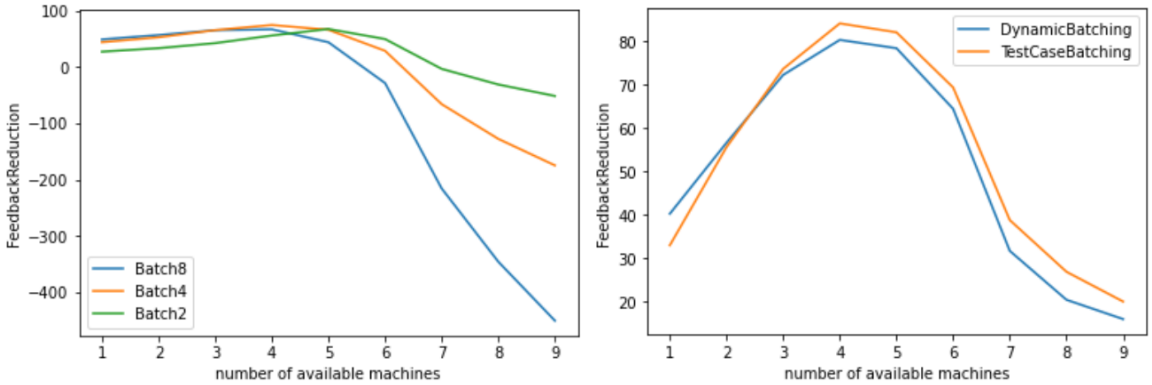

Figure 4.9: The *FeedbackReduction* each algorithm relative to TestAll. With *TestDynamicBatching* and *TestCaseBatching*, we always decrease the feedback time compared to TestAll. We note that *ConstantBatching* is highly ineffective adapting to an increase in resources.

number of parallel machines. For example, by increasing the number of machines from 4 to 5, the *AvgFeedback* decreases from 75.78 hours to 35.30 hours. The relationship between machines and *AvgFeedback* is clearly non-linear, with an increase in machines from 4 to 5, *i.e.* 25%, we see a reduction in *AvgFeedback*) time of 53%. The improvement is nonlinear because without sufficient resources the delay is compounded to all subsequent changes leading to large delays. An illustrative example of the compounded delay can be found in background Section 4.2.1.

In contrast, for TestAll, the resource consumption is constant because regardless of whether the test runs are parallelized we still need to run each request test on each individual change. For TestAll, the *AvgCPU* is 16.15 hours.

> With TestAll, any delay will be compounded in delays to all subsequent changes in the queue. By increasing the number of parallel machines for testing, the feedback time will decrease in a nonlinear manner, *e.g.,* from 721.60 hours with one machine to 5.18 hours with 7 machines, a decrease of 99.3%. The CPU time is constant with TestAll because commits are tested independently with an *AvgCPU* time of 16.15 hours.

4.4.2 Result: ConstantBatching

Batching changes reduces the number of test executions when the changes request the same tests. Prior works [70, 13] used a constant batch size, and we replicate the state-of-the-art approach on a project at Ericsson. We varied the batch size from 2 to 32, but as we can see in Figure 4.8, the savings plateau, so we report results for batch sizes of 2, 4, and 8, *i.e.* Batch2, Batch4, and Batch8 in the body of the paper.

Figure 4.8 shows the *CPUReduction* for these batching technique relative to TestAll. The CPUReduction for Batch2, Batch4, and Batch8 is 23%, 37%, and 44%, respectively. The reduction increases with batch size because the majority of the tests pass. As noted by prior work [13], the reduction is limited by the number of failing tests because a failure requires additional executions to find the culprit changes.

Prior works have focused on resource savings and largely ignored or simplified feedback time [70,

13]. If we strictly follow a constant batch size, then we see that commits can wait for extended periods of time. Figure 4.9 plots the result of *FeedbackReduction* compared to TestAll. When resources are constrained and the queue is full, in this case up to 5 machines, Batch2, Batch4, and Batch8 all reduce the average feedback time. However, as the machines increase up to 9 the constant batch size increases the feedback time by -51.48%, -174.79%, and -450.70% respectively. In practice, with constant batching, developers set a maximum waiting time and then process all available commits regardless of batch size.

> A constant batch size of 2, 4, and 8 reduce the number of executions by 23%, 37%, and 44% relative to TestAll. Feedback time can increase or be reduced depending on the number of available machines for parallelization. Large batch sizes lead to feedback delays.

4.4.3 Result: TestDynamicBatching

ConstantBatching can save resources and can decrease the feedback time in an extreme resource constraint environment where there are many commits in the queue. However, the queue size varies over time, with peak changes happening during working hours. To better utilize the available resources, we suggest *TestDynamicBatching* which batches all available changes in the queue. Section 4.2.1 provides the background.

Varying the number of machines from 1 to 9, we see a *FeedbackReduction* of 40.15%, 56.57%, 72.27%, 80.38%, 78.46%, 64.43%, 31.56%, 20.25%, and 15.78% relative to TestAll. Figure 4.9 and Table 4.1 compare the effectiveness against the other approaches. The maximum reduction in feedback time is achieved with four machines.

In terms of *CPUReduction*, varying the machines from 1 to 9 results in a respective reduction of in 48.87%, 46.08%, 40.75%, 32.29%, 21.52%, 13.16%, 7.71%, 4.76%, 3.13% compared to TestAll. Figure 4.8 compares against all other techniques. We can see that constant batching can outperform dynamic batching in terms of CPU reduction, however, this comes at a cost in feedback time.

We see the *FeedbackReduction* for *TestDynamicBatching* is substantial and the largest saving relative to TestAll is 80.38% decrease in feedback time. To attain this improvement we need four machines, which results in a 32.29% reduction in CPU time.

4.4.4 Result: TestCaseBatching

TestDynamicBatching process all available changes. However, any change that arrives when no machines are available has to wait until all the tests for a batch have completed. In background Section 4.2.1, we introduced *TestCaseBatching* that queues the request tests across all changes and includes any new change after each test completes (rather than waiting for all tests to complete).

We vary the number of machines from 1 to 9, and see a respective *FeedbackReduction* of 32.85%, 55.56%, 73.67%, 84.20%, 82.13%, 69.37%, 38.69%, 26.73%, and 19.84% relative to TestAll. Figure 4.9 and Table 4.1 compare the effectiveness against the other approaches. Relative to *TestDynamicBatching* the corresponding change is -12.18, -2.33, 5.04%, 19.47%, 17.08, 13.9%, 10.42%, 8.13%, and 4.83%, respectively. We see that *TestCaseBatching* is more effective than dynamic batching with 3 or more machines and can be substantially more effective with 7 or 8 machines.

For *CPUReduction* the corresponding reduction is 50.92%, 49.11%, 43.01%, 34.19%, 24.05%, 16.80%, 11.19%, 7.63%, and 5.65% relative to TestAll. Relative to *TestDynamicBatching*, it can reduce CPU resources by 4.01%, 5.62%, 3.81%, 2.82%, 3.23%, 4.21%, 3.78%, 3.01%, and 2.61%. In Figure 4.8 we that *TestCaseBatching* is always more effective.

TestCaseBatching has the largest savings in both feedback time and CPU time of any technique. With four machines, compared to TestAll it reduces feedback time by 84.20% and CPU time by 34.19%. Relative to *TestDynamicBatching* it reduces feedback time by 19.47% and CPU time by 2.82%.

4.5 Threats to validity

We selected a large project with over 11k changes at Ericsson. While our definitions and algorithms are not tied to the project, the approach that works the best may change with different project environments. For example, on a project that tests in the cloud, additional machines may be very cheap. However, we have shown that additional machines do not always improve the results, and as Microsoft noted, cloud test machines are not free and can add up quickly [45].

The historical simulation simplified parts of the Ericsson's testing processes. Developers can stop the testing of a build or manually batch select changes for testing. Since we cannot model these manual interventions, we excluded them from our simulation.

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

We have two thresholds in our work, the batch size is varied from 2 to 32 and the number of machines is varied from 1 to 16. For the studied project, our results show that these ranges are appropriate, with the outcome measures either reaching a plateau or showing a downward trend. Our discussion of compounded delays and utilized resources also demonstrate appropriate threshold ranges.

4.6 Related Work

4.6.1 Test parallelization

Test parallelization distributes testing across multiple machines to reduce feedback time. Previous works widely studied the impact of test parallelization on software testing and introduced algorithms to run tests in parallel [51, 69, 10, 64]. For example, Arabnejad *et al.* [6] investigated using GPUs for running tests in parallel. The most popular algorithms for parallelizing tests are scheduling tests across the machines based on their IDs and their historical execution time [2].

Candido *et al.* [21] investigated the impact of test parallelization on open source projects. By analyzing more than 450 Java projects, they found that less than 20% of major projects use test parallelization due to concerns with concurrency issues. They provide recommendations to practitioners to facilitate their parallel testing, such as refactoring tests for load balancing and grouping tests based on their dependencies and running tests with dependencies on the same machine. Bell *et al.* [16] studied the impact of test dependency on test parallelization. They introduced the *ElectricTest* approach to detect dependencies before scheduling them across the machines. Ding *et al.* [32] proposed a software behavior oriented test parallelization to reduce conflicting behaviors.

At the Ericsson, we run tests that have already been parallelized. We note that none of the prior works studied the effect of test parallelization in the context of batch testing.

4.6.2 Batch Testing and CI/CD

Continuous integration and delivery is an essential aspect of modern software development [50]. Using a CI infrastructure, developers can automatically build and test their changes to make sure it does not break software functionality [78, 47, 88, 65].

In a CI pipeline, we need to test each commit before merging it to the main branch. In large software systems, testing every change individually is impossible [45]. Even Google with huge server farms is not able to test every single commit independently [37]. There are multiple solutions to solve this problem including test selection [81, 40, 23], test minimization [67, 91, 24], test prioritization [9, 102, 84], and batch testing [70, 13]. The first three techniques have been widely studied and various articles about them are available in the literature. However, there are a few papers that studied batch testing.

Batch testing is used to decrease the CPU time and the feedback time in resource constraint environments [28, 22]. Instead of testing every single change in isolation, we batch them and run them all at once. If the batch fails, we have to find the culprit change(s) responsible for the failure. GitBisection [1] is the most well-known culprit finding technique. It performs a binary search on the changes to find the culprit. GitBisection can find the culprit with $log(n)$ executions. However, in the condition that there is more than one culprit, GitBisection only finds the first one. To solve this problem, Najafi *et al.* [70] introduced a bisection that use a divide-and-conquer algorithm to

find all the culprits. It divides the failing batch into two subbatches and tests both of them to find if they include a culprit change. As it performs a complete search, the total number of executions is between $2log(n)$ and $2n + 1$ when all the changes in the batch are culprits. Beheshtian *et al.* [13] showed that for batches with a size of 4 or fewer, bisection only increases the number of executions. They propose BatchStop4 which tests every change individually in the batches with size 4 or fewer. They mathematically show that using BatchStop4 always outperforms batch bisection.

These previous works studied batch testing at the change level and with a single machine. They did not investigate the impact of parallel testing. In this work, we study the combination of batching and parallel testing for the first time and show the impact of batching by varying the number of parallel machines.

4.7 Contributions and Concluding Remarks

In this paper, we make the following contributions.

(1) We introduce metrics to gauge the change in feedback time and CPU resource usage for batch testing.

(2) We introduce two novel batching strategies. *TestDynamicBatching* builds batches out of all the waiting changes and runs culprit finding on failure, allowing better resource utilization. *TestCaseBatching* batches all changes that request a particular test, and adds new changes immediately after each test case finishes.

We report the following findings from our historical simulations at Ericsson varying the number of machines available for each batching algorithm.

(1) **TestAll.** If changes are tested individually, any delay will be compounded in delays to all subsequent changes in the queue. By increasing the number of parallel machines for testing, the feedback time will decrease in a nonlinear matter, *e.g.,* from 721.60 hours with one machine to 5.18 hours with 7 machines, a decrease of 99.3%. The CPU time is constant with TestAll because commits are tested independently with an *AvgCPU* time of 16.15 hours.

(2) **ConstantBatching.** A constant batch size of 2, 4, and 8 reduce the number of executions by 23%, 37%, and 44% relative to TestAll. Feedback time will increase as changes wait for the batch size before being run.

(3) **TestDynamicBatching.** The feedback reduction for *TestDynamicBatching* is substantial and the saving relative to TestAll is 80.38% decrease in feedback time. To attain this improvement we need only four machines, which results in a 32.29% reduction in CPU time.

(4) **TestCaseBatching** has the largest savings in both feedback time and CPU time of any technique. With four machines, compared to TestAll it reduces feedback time by 84.20% and CPU time by 34.19%. Relative to *TestDynamicBatching* it reduces feedback time by 19.47% and CPU time by 2.82%.

We show that *TestCaseBatching* rather than batching at the change level can be highly effective at Ericsson. We hope to see other reports on its usage.

Chapter 5

Conclusion

Testing every change is expensive and sometimes impossible for large companies *e.g.,* Ericsson[70] and Google[37]. In this work, we have done an extensive study on the impact of batching and parallel testing in continuous integration environments. We extended the work that have been done by Beheshtian *et al.* [13] and study the impact of batching in more diverse open source projects with a wider range of failure rates. We have seen that batching is still effective by having a failure rate of up to 50%. We then introduced new batching techniques based on a dynamic batch size for projects under study. In the *DynamicBatching* technique, instead of assuming a constant batch size for all the batches over time, we calculate the batch size based on the recent failure rate of the project. Our simulations show that *DynamicBatching* decreases the number of executions by 47.85% over TestAll with an improvement over the state-of-the-art Batch4 of 5.17 percentage points.

We describe a theoretical limit for the savings that can be achieved in batch testing. We show that using *DynamicBatching*, we achieve an across project average of 58.91% of the theoretical limit. Although batching is highly effective, there is still substantial room for improving batching relative to the theoretical batch savings limit.

All the previous approaches including *DynamicBatching* perform batching in the build level of projects. To find a culprit build in a failed batch, they rerun all the test cases to find which change is responsible for the failure of the batch. However, in large industrial projects, there are three additional assumptions which need to be considered. First, the test cases need to be run could vary among different builds. Second, we have access to test outcomes of each build in the test case, level.

This means in the case of failure of a batch, we can just rerun the failed test cases to save resources and time. In addition, in Chapter 2 and 3, we only considered testing builds in a single machine. But in industrial projects, we have access to multiple machines to parallelize test runs. We also have to consider the feedback time of our batching approaches because it is a critical metric for CI.

Using the above assumptions, we extend our study to a real world industrial project at Ericsson to study the impact of batching and parallel testing at the test case level. We introduce metrics to measure the change in feedback time and CPU resource usage for batch testing. We introduce two novel batching strategies. *TestDynamicBatching* builds batches out of all the waiting commits and runs culprit finding on failure. *TestCaseBatching* groups all changes that request a particular test. We find that the feedback reduction as well as the CPU time reduction in both the proposed approaches is substantial. We find among all the approaches *TestCaseBatching* has the largest savings in both feedback time and CPU time of any technique. With four machines, compared to TestAll it reduces feedback time by 84.20% and CPU time by 34.19%. Relative to *TestDynamicBatching*, it reduces feedback time by 19.47% and CPU time by 2.82%. We conclude that *TestCaseBatching* rather than batching at the change level can be highly effective at Ericsson. We hope to see other reports on its usage.

Bibliography

[1] git-bisect manual page, 2015.

[2] Run tests in parallel using the visual studio test task. 2019.

[3] Travis CI Queue Dashboard. https://www.traviscistatus.com/#week, 2020.

[4] B. A. Alexeevich and D. M. Borisovich. Test bundling and batching optimizations, May 16 2019. Patent App. 16/206,311.

[5] J. Anderson, S. Salem, and H. Do. Improving the effectiveness of test suite through mining historical data. MSR 2014, page 142–151, New York, NY, USA, 2014. Association for Computing Machinery.

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

[7] D. Aragón-Caqueo, J. Fernández-Salinas, and D. Laroze. Optimization of group size in pool testing strategy for sars-cov-2: A simple mathematical model. *Journal of Medical Virology*, 2020.

[8] L. Aversano, L. Cerulo, and C. Del Grosso. Learning from bug-introducing changes to prevent fault prone code. In *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*, pages 19–26, 2007.

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

[10] T. Bagies. Parallelizing unit test execution on gpu. 2020.

[11] A. H. Bavand and P. C. Rigby. Replication package. https://github.com/CESEL/DynamicBatchingICSME, 2021.

[12] M. J. Beheshtian. Software batch testing to reduce build test executions, 2020.

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

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

[15] M. J. Beheshtian and P. C. Rigby. BatchBuilder GitHub App. https://github.com/apps/batchbuilder, 2020.

[16] J. Bell, G. Kaiser, E. Melski, and M. Dattatreya. Efficient dependency detection for safe java test acceleration. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 770–781, New York, NY, USA, 2015. Association for Computing Machinery.

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

[18] M. Beller, G. Gousios, and A. Zaidman. 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)*, pages 447–450. IEEE, 2017.

[19] P. Bhattacharya and I. Neamtiu. Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10, 2010.

[20] A. Z. Broder and R. Kumar. A note on double pooling tests. *arXiv preprint arXiv:2004.01684*, 2020.

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

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

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

[24] T. Y. Chen and M. F. Lau. Dividing strategies for the optimization of a test suite. *Information Processing Letters*, 60(3):135 – 141, 1996.

[25] X. Chen, Y. Shen, Z. Cui, and X. Ju. Applying feature selection to software defect prediction using multi-objective optimization. In *2017 IEEE 41st annual computer software and applications conference (COMPSAC)*, volume 2, pages 54–59. IEEE, 2017.

[26] Y. Chen, R. L. Probert, and D. P. Sims. Specification-based regression test selection with risk analysis. In *Proceedings of the 2002 conference of the Centre for Advanced Studies on Collaborative research*, page 1. IBM Press, 2002.

[27] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

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

[29] L. Crispin and J. Gregory. *Agile testing: A practical guide for testers and agile teams*. Pearson Education, 2009.

[30] R. Cruz. Ricardo cruz.

[31] D. A. Da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan. A framework for evaluating the results of the szz approach for identifying bug-introducing changes. *IEEE Transactions on Software Engineering*, 43(7):641–657, 2016.

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

[33] R. Dorfman. The detection of defective members of large populations. *The Annals of Mathematical Statistics*, 14(4):436–440, 1943.

[34] R. Dorfman. The detection of defective members of large populations. *Ann. Math. Statist.*, 14(4):436–440, 12 1943.

[35] T. Durieux, R. Abreu, M. Monperrus, T. F. Bissyandé, and L. Cruz. An analysis of 35+ million jobs of travis ci. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 291–295. IEEE, 2019.

[36] P. M. Duvall, S. Matyas, and A. Glover. *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

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

[38] S. Elbaum, G. Rothermel, and J. Penix. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 235–245, New York, NY, USA, 2014. ACM.

[39] D. Engler and M. Musuvathi. Static analysis versus software model checking for bug finding. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 191–210. Springer, 2004.

[40] E. Engström, P. Runeson, and M. Skoglund. A systematic review on regression test selection techniques. *Information and Software Technology*, 52(1):14–30, 2010.

[41] Y. Gajpal, S. Appadoo, V. Shi, and Y. Liao. Optimal multi-stage group partition for efficient coronavirus screening. *Available at SSRN 3591961*, 2020.

[42] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304, 2011.

[43] D. Hao, L. Zhang, L. Zhang, G. Rothermel, and H. Mei. A unified test case prioritization approach. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(2):1–31, 2014.

[44] M. Harman and P. O'Hearn. From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 1–23. IEEE, 2018.

[45] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy. The art of testing less without sacrificing quality. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, page 483–493. IEEE Press, 2015.

[46] K. Herzig and N. Nagappan. Empirically detecting false test alarms using association rules. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 39–48, 2015.

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

[48] D. Jeffrey and N. Gupta. Test case prioritization using relevant slices. In *Proceedings of the 30th Annual International Computer Software and Applications Conference - Volume 01*, COMPSAC '06, pages 411–420, Washington, DC, USA, 2006. IEEE Computer Society.

[49] H. Jiang, X. Li, Z. Yang, and J. Xuan. What causes my test alarm? automatic cause analysis for test alarms in system and integration testing. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, page 712–723. IEEE Press, 2017.

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

[51] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, page 467–477, New York, NY, USA, 2002. Association for Computing Machinery.

[52] L. Jonsson, M. Borg, D. Broman, K. Sandahl, S. Eldh, and P. Runeson. Automated bug assignment: Ensemble-based machine learning in large scale industrial contexts. *Empirical Software Engineering*, 21(4):1533–1578, 2016.

[53] R. Just, G. M. Kapfhammer, and F. Schweiggert. Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*, pages 11–20. IEEE, 2012.

[54] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, 2012.

[55] A. Kaur and S. Goyal. A genetic algorithm for regression test case prioritization using code coverage. *International journal on computer science and engineering*, 3(5):1839–1847, 2011.

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

[57] S. Kim, E. J. Whitehead, and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, 2008.

[58] S. Kim, T. Zimmermann, K. Pan, E. James Jr, et al. Automatic identification of bug-introducing changes. In *21st IEEE/ACM international conference on automated software engineering (ASE'06)*, pages 81–90. IEEE, 2006.

[59] P. Konsaard and L. Ramingwong. Total coverage based regression test case prioritization using genetic algorithm. In *2015 12th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)*, pages 1–6. IEEE, 2015.

[60] M. Kumar, A. Sharma, and R. Kumar. An empirical evaluation of a three-tier conduit framework for multifaceted test case classification and selection using fuzzy-ant colony optimisation approach. *Software: Practice and Experience*, 45(7):949–971, 2015.

[61] M. Laali, H. Liu, M. Hamilton, M. Spichkova, and H. W. Schmidt. Test case prioritization using online fault detection information. In *Ada-Europe International Conference on Reliable Software Technologies*, pages 78–93. Springer, 2016.

[62] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie. idflakies: A framework for detecting and partially classifying flaky tests. In *2019 12th ieee conference on software testing, validation and verification (icst)*, pages 312–322. IEEE, 2019.

[63] W. Lam, A. Shi, R. Oei, S. Zhang, M. D. Ernst, and T. Xie. Dependent-test-aware regression testing techniques. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, page 298–311, New York, NY, USA, 2020. Association for Computing Machinery.

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

[65] M. Leppänen, S. Mäkinen, M. Pagels, V.-P. Eloranta, J. Itkonen, M. V. Mäntylä, and T. Männistö. The highways and country roads to continuous deployment. *Ieee software*, 32(2):64–72, 2015.

[66] J. Li. Successfully merging the work of 1000+ developers [at shopify]. https://engineering.shopify.com/blogs/engineering/successfully-merging-work-1000-developers, 2019.

[67] M. Marré and A. Bertolino. Using spanning sets for coverage testing. *IEEE Trans. Softw. Eng.*, 29(11):974–984, Nov. 2003.

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

[69] S. Misailovic, A. Milicevic, N. Petrovic, S. Khurshid, and D. Marinov. Parallel test generation and execution with korat. ESEC-FSE '07, page 135–144, New York, NY, USA, 2007. Association for Computing Machinery.

[70] A. Najafi, P. C. Rigby, and W. Shang. Bisecting commits and modeling commit risk during testing. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 279–289, 2019.

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

[72] J. Nam, W. Fu, S. Kim, T. Menzies, and L. Tan. Heterogeneous defect prediction. *IEEE Transactions on Software Engineering*, 44(9):874–896, 2017.

[73] C. Nguyen, P. Tonella, T. Vos, N. Condori, B. Mendelson, D. Citron, and O. Shehory. Test prioritization based on change sensitivity: an industrial case study, 2014.

[74] W. Niu, X. Zhang, X. Du, L. Zhao, R. Cao, and M. Guizani. A deep learning based static taint analysis approach for iot software vulnerability location. *Measurement*, 152:107139, 2020.

[75] T. B. Noor and H. Hemmati. Studying test case failure prediction for test case prioritization. In *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 2–11, 2017.

[76] S. K. Pandey, R. B. Mishra, and A. K. Tripathi. Bpdet: An effective software bug prediction model using deep representation and ensemble learning techniques. *Expert Systems with Applications*, 144:113085, 2020.

[77] H. Peijie, J. Tebbs, C. Bilder, and C. Mcmahan. Hierarchical group testing for multiple infections. *Biometrics*, 73, 09 2016.

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

[79] X. Qu, M. B. Cohen, and G. Rothermel. Configuration-aware regression testing: an empirical study of sampling and prioritization. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 75–86, 2008.

[80] D. Radjenović, M. Heričko, R. Torkar, and A. Živkovič. Software fault prediction metrics: A systematic literature review. *Information and software technology*, 55(8):1397–1418, 2013.

[81] G. Rothermel and M. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, 1996.

[82] G. Rothermel and M. J. Harrold. A framework for evaluating regression test selection techniques. In *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, pages 201–210, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[83] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(2):173–210, 1997.

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

[85] A. Sarkar, P. C. Rigby, and B. Bartalos. Improving bug triaging with high confidence predictions at ericsson. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 81–91, 2019.

[86] M. Shepperd, D. Bowes, and T. Hall. Researcher bias: The use of machine learning in software defect prediction. *IEEE Transactions on Software Engineering*, 40(6):603–616, 2014.

[87] M. Sherriff, M. Lake, and L. Williams. Prioritization of regression tests using singular value decomposition with empirical change records. In *Proceedings of the The 18th IEEE International Symposium on Software Reliability*, ISSRE '07, pages 81–90, Washington, DC, USA, 2007. IEEE Computer Society.

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

[89] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige. Reinforcement learning for automatic test case prioritization and selection in continuous integration. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, page 12–22, New York, NY, USA, 2017. Association for Computing Machinery.

[90] A.-B. Taha, S. Thebaut, and S.-S. Liu. An approach to software fault localization and revalidation based on incremental data flow analysis. In *Computer Software and Applications Conference, 1989. COMPSAC 89., Proceedings of the 13th Annual International*, pages 527–534, Sep 1989.

[91] S. Tallam and N. Gupta. A concept analysis inspired greedy algorithm for test suite minimization. *ACM SIGSOFT Software Engineering Notes*, 31(1):35–42, 2005.

[92] A. Viehweger, F. Kühnl, C. Brandt, and B. König. Increased pcr screening capacity using a multi-replicate pooling scheme. *medRxiv*, 2020.

[93] S. Wang, J. Nam, and L. Tan. Qtep: quality-aware test case prioritization. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 523–534, 2017.

[94] X. Wu, V. Kumar, J. R. Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, S. Y. Philip, et al. Top 10 algorithms in data mining. *Knowledge and information systems*, 14(1):1–37, 2008.

[95] S.-Q. Xi, Y. Yao, X.-S. Xiao, F. Xu, and J. Lv. Bug triaging based on tossing sequence modeling. *Journal of Computer Science and Technology*, 34(5):942–956, 2019.

[96] J. Xuan, H. Jiang, Z. Ren, and W. Zou. Developer prioritization in bug repositories. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 25–35, 2012.

[97] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun. Deep learning for just-in-time defect prediction. In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 17–26. IEEE, 2015.

[98] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

[99] L. Zhang. Hybrid regression test selection. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 199–209, 2018.

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

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

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

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

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