8-2016

Improving the Efficiency of CI with Uber-commits

Matias Waterloo
University of Nebraska - Lincoln, waterloo.matias@gmail.com

IMPROVING THE EFFICIENCY OF CI WITH UBER-COMMITS

by

Matias Waterloo

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfillment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Sebastian Elbaum and Suzette Person

Lincoln, Nebraska

August, 2016

IMPROVING THE EFFICIENCY OF CI WITH UBER-COMMITS

Matias Waterloo, M.S.

University of Nebraska, 2016

Advisor: Sebastian Elbaum and Suzette Person

Continuous Integration (CI) is a software engineering practice where developers break their coding tasks into small changes that can be integrated with the shared code repository on a frequent basis. The primary objectives of CI are to avoid integration problems caused by large change sets and to provide prompt developer feedback so that if a problem is detected, it can be easily and quickly resolved. In this thesis, we argue that while keeping changes small and integrating often is a wise approach for developers, the CI server may be more efficient operating on a different scale. In our approach, the CI server monitors the queue of commits to be integrated and merges multiple commits into a single *Uber* commit, thus avoiding the redundant operations, e.g., testing, associated with integrating each commit individually. If an Uber commit fails during the merge, build or test process, our approach uses a culprit analysis to find the commit(s) causing the failure. An analysis of our approach on an open source project shows that Uber commits can improve both CI server efficiency by 7% to 11%, and reduce developer feedback time by 7% to 30%.

ACKNOWLEDGMENTS

I would like to thank Dr. Sebastian Elbaum and Dr. Suzette Person for their support, encouragement and guidance. I will always be grateful for what I have learned working with you in the last two years. I also thank Dr. Myra Cohen for agreeing to serve on my committee. Special thanks to my family for their unconditional love and support and a thanks to all my friends in Lincoln for making this experience much more fun.

This work was partially supported by an Argentine Presidential Fellowship Award and the National Science Foundation under awards #1526652 and #1218265.

Contents

List of Figures

Chapter 1

Introduction

Integration testing exposes defects in the interactions between software components. Prior to the practice of Continuous Integration (CI) [1, 2, 3], integration testing was performed at the end of a development cycle, which could last anywhere from a day to a period of weeks or months. Even when developers faithfully perform unit testing on their code changes prior to integration, long periods of time between integration testing have the potential to increase the risk of code change conflicts. Moreover, multiple changes from multiple developers increase the difficulty of locating and resolving any problems that are detected during integration testing.

Continuous Integration aims to reduce integration problems and to provide prompt developer feedback so that if a problem is detected, it can be easily and quickly resolved. By leveraging build servers and powerful testing frameworks, CI environments enable developers to break development tasks into small code changes that can be integrated and automatically tested with the mainline codebase at frequent time intervals. This approach to continuous quality control can reduce the amount of rework and improve developer productivity, ultimately enabling organizations to deliver software more rapidly. CI servers have also provided tremendous value to open-source projects by enabling core team members to vet changes from large numbers of contributors through pull requests that are automatically built and tested before they are reviewed by a core developer.

Despite sophisticated tool support and the pervasive adoption of CI by a wide range of organizations including Google, Facebook, Netflix, Amazon, Twitter and tens of thousands of open source projects, CI environments continue to face challenges. One significant challenge is the staggering pace at which code changes occur, especially when developers are encouraged to commit their changes early and often. For example, a few years ago, Amazon reported one code commit every 11 seconds [4], Google reported an average of 20 code commits per minute [5], Facebook reported approximately 500 commits per day affecting thousands of files [6], and LibreOffice has had peak days with 12 commits per minute [7].

For each commit, the CI server builds the code and then runs potentially huge numbers of test cases. Some of the tests analyze the changed code, while other tests analyze the interactions of changed code with other affected code. At Google, this amounts to over 100 million individual test case executions per day [8]. Even when organizations utilize huge server farms to run tests in parallel, or execute tests in the "cloud," projects have a tendency to expand the testing phase to utilize all of the available resources. And then they continue to expand beyond that [8], ultimately causing the CI server to become a bottleneck when large numbers of changes are committed in a short period of time.

Various techniques have been developed to improve feedback time in CI environments, however, existing techniques have primarily focused on optimizing builds at the individual level by improving their activities [25] or eliminating unnecessary work (i.e., test activities [9,10], compilation [11,12,13]) or by applying parallelization when optimizing at the global level [14, 15, 16], which is scalable as long as you have the necessary machinery.

In this work, we introduce a technique that aims to improve the efficiency of CI servers to reduce the bottleneck of building and testing changes, while maintaining timely feedback to the developers. The key insight of our work is that while small changes and frequent integration is a wise approach for developers, CI servers need not follow the same workflow as the developers. Instead, when the CI server queue contains multiple commits for integration, the CI server workflow can be modified in a way that improves developer feedback, but yet is transparent to the developers. In the modified CI server workflow, commits in the queue are first merged into an *Uber* commit that is built and tested as a single unit of work, eliminating the redundancies associated with building and testing each commit individually. If an Uber commit fails during the merging process, each commit is processed individually with only the overhead of the attempted code merge. If the Uber commit fails during the build or test phase, our approach applies a culprit analysis to find the commit(s) causing the failure.

The primary contributions of our work are:

- A novel approach to improve the efficiency of CI servers that is transparent to developers and leverages the redundancies in building and testing multiple commits while maintaining timely feedback to developers. Our approach is also complementary to other techniques focused on increasing the resources available to the CI server.

- A novel approach, based on culprit analysis, to process the case where an Uber commit fails during the build or test process. It provides results with adjustable precision.

- A simulation of our approach on the version history of three open source projects that demonstrates that Uber commits can improve CI server efficiency and developer feedback time. Our experiments show improvements in both, CI server efficiency (7% to 11%) and developer feedback time (7% to 30%).

Chapter 2

Background & Motivation

2.1 Continuous Integration

Fowler [2] defines CI as:

> Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible.

Although the practice of CI does not require any special tooling to deploy, CI services have facilitated its wide adoption. Servers like Travis-CI [17], Jenkins [18] and CircleCI [19] can be used with popular source code hosting services such as Github [20]. Staahl and Bosch provide a systematic overview of continuous integration practices and how they vary [21]. In this section, we briefly describe the common elements found in CI servers that are most relevant to our approach.

In general, a CI environment is composed of one or more servers waiting for event notifications to trigger builds. CI servers are generally configured for each project individually, however, the most common configuration is for the server to trigger a build when a new commit has been submitted to the project's code repository. Nonetheless, builds could be also triggered manually by a developer, by a *cron* process or by another build. The CI server then creates a temporary copy of the updated code from the source code repository, starts a build of the code and monitors the state of the build. The build script can include commands to test the build, deploy the code, build documentation, and perform clean-up, all depending on the status of the build.

Most CI servers provide multiple build configuration options to provide flexibility in how the project is built. For example, it is possible to trigger multiple builds from a single commit to test the resulting product on multiple machines or operating system configuration, e.g., one build may be executed on a Microsoft Windows machine and another on a Unix machine. In addition, most CI servers make use of parallelization to reduce the amount of time a developer waits for the results of the build and test phases.

In terms of workflow practices, Fowler describes two types of builds that should be common in CI environments [2]. First, there should be builds that are triggered on every submitted commit and execute a series of fast tests to validate no obvious bugs are being introduced in the software. Then, there should be others that are triggered less frequently, which are built on the last available commit, and execute more expensive tests to have a complete validation of the system. Nightly builds are a classic example of this type. These types illustrate the need to differentiate builds and adjust the workflow in order to accelerate developer's feedback. Builds of the first type are run on each commit because it is assumed that including two or more into a single build leads to a loss in precision on finding changes that introduce faults [22, 23].

Builds of the second type, however, include all commits since the previous execution into a single build. In this case the loss in precision is accepted because this practice reduces the resource utilization for these builds allowing more builds of the first type to get executed and because the risk on failure is lower for this type of builds, commits have already passed the first series of tests.

In this thesis, we argue the CI server could also benefit of reduction of resource utilization when working with the first type of builds by including two or more commits pending on the queue into single builds. We show the loss in precision can be mitigated with small extensions to the CI process. In the next section, we provide an example of how the processing of individual commits can impact the time it takes for the server to provide feedback to developers and how the impact can be reduced by merging commits into an *Uber* commit that is built and tested as a single unit of work.

2.2 Motivating Example

In a traditional CI development environment, developers typically build and test the changes locally prior to integration with the mainline codebase. Once the changes pass local tests, the developer then updates her working copy of the project with any changes that were made to the mainline code while she was working on her changes. This step ensures the compatibility of her changes with changes committed by other developers. Once her working copy is synchronized with the mainline codebase, she then commits her changes or creates a pull request, which triggers a build job on the CI server. Depending on the CI server load, her job may be processed immediately and the *feedback time* of her changes, which is the time she will have to wait to have results, will include only the amount of time necessary to process her job. When the CI server is busy, i.e., multiple commits or pull requests are waiting to be processed, the feedback time will include not only the time to process her job, but also the time to process all of the jobs waiting in the queue ahead of her job. Depending on factors such as the culture of the project, the phase in the development cycle, and the size of the project, multiple jobs may be queued at any given time. In a sample of 100 projects from Travis, and 100 of their builds[1], we found that 30% of the projects reported having more than one commit in the queue for at least 20% of the time the server was about to pick a new commit to be processed.

Various techniques have been proposed to deal with a large number of commits such as parallelizing the execution of independent builds [3, 14, 15, 24], minimizing and parallelizing activities inside the build [9,25], scoping the compilation and test execution to the parts of the system that were affected by changes [10, 11, 12, 26] and efficient handling of the propagation effect of new builds over dependent projects [13, 16].

In spite of these efforts, large companies are reporting problems to trigger a build on each change because their code change frequency is too high for this approach to be feasible [8, 24]. We believe our approach can provide benefits to these companies because it aims to eliminate redundancies between builds already in the queue and reduce the number of builds the server needs to process. These redundancies especially occur when all builds are successful since the same results can be obtained by executing a single build containing all changes. Such redundancies are common, we found in the sample of 100 projects from Travis that 32% of the projects report a low failure rate (20% or less) for commits processed by the CI server.

In this thesis, we propose to modify the way commits and pull requests are built and tested by adding an option to the CI server that would enable it to merge the next

[1]100 projects were randomly selected from all available projects on Travis-CI on May 17, 2016 that reported more than 1000 builds, 10 or more Github contributors and a Test Suite. Either the last 10000 builds or builds from the last 6 months were selected for each project. Builds were collected on July 13, 2016.

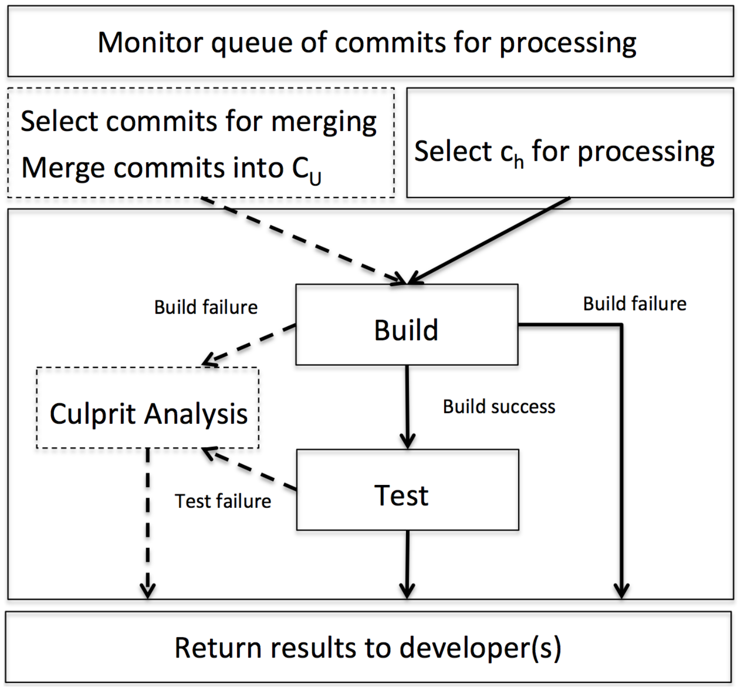
Monitor queue of commits for processing
Select commits for merging
Merge commits into C_U
Select c_h for processing
Build failure
Build
Build failure
Culprit Analysis
Build success
Test failure
Test
Return results to developer(s)

Figure 2.1: CI Server workflow for traditional (solid line) and Uber approach (dashed line).

commit to be processed, i.e., the head of the queue, with other compatible commits in the queue to form an Uber commit. Fig. 2.1 shows the workflow of a CI server applying this approach and compares it with the traditional. In the later, shown in solid lines, the server monitors the queue and always picks the commit in the head to be processed individually. The selected commit is then built and tested and the results of these operations are finally reported to the developers. Under the Uber approach, the server workflow, shown in dashed lines, is extended with a selection and culprit analysis processes. The first one is applied to select candidates commits that will take part of an Uber commit and the second to identify the commits that make the build fail.

Applying the new workflow would be transparent to the developers and can be employed as needed, based on the server load. Uber commits do not affect the developer workflow, but have the potential to improve developer feedback time and reduce the

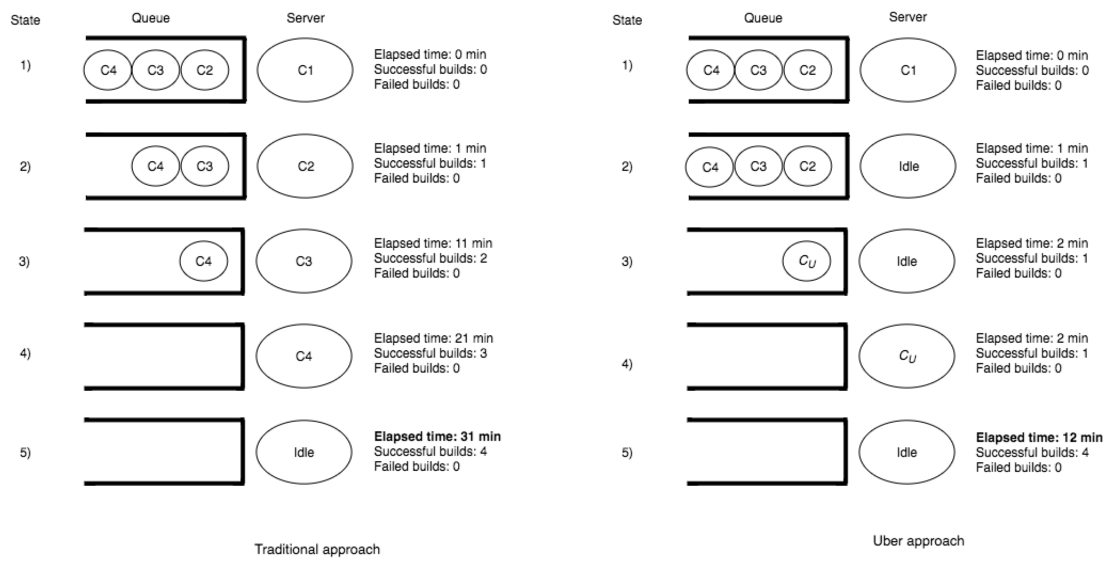

Figure 2.2: **(a)**: Traditional CI server approach for processing code change. All submissions are processed in 31 minutes. **(b)**: CI server processing code change as an *Uber* commit. All submissions are processed in 12 minutes.

number of CI server cycles that would have been spent building and testing each commit individually. This additional processing first requires that candidate commits are checked for compatibility. When commits are determined to be compatible, the source code is then merged, and the processing of the Uber commit proceeds in the same manner as processing as traditional CI server processing.

Figure 2.2 illustrates both the traditional approach (a), and our proposed approach (b), when multiple jobs are waiting in the queue for processing. In the figure, each state represents the state of the environment at a particular point of time and all together show the evolution of the environment over the progression of time. Queue and server represent the commits available on the queue or being processed by the server at a given state. In both examples, three commits (jobs) are waiting in the queue for processing. The traditional CI server will process each commit individually, consuming 31 minutes. By creating an Uber commit, i.e. merging the commits, the proposed approach can process all three commits in 12 minutes. The merging process

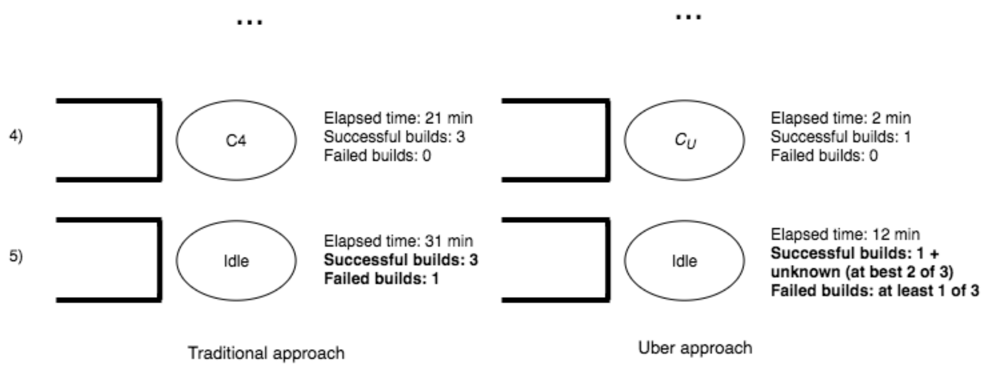

Figure 2.3: Example of loss in precision on culprit detection by changing the integration scale. Same example used in Fig. 2.2 but C4 fails in this case.

adds a small amount of overhead (1 minute) but there is a saving in the cost of building and testing each commit individually. All commits are assumed to take 10 minutes to be run and introduced changes are assumed to be small for C_U to also take 10 minutes of execution time.

It is possible that not all Uber commits will successfully build and pass the testing phase. Figure 2.3 shows an example of this situation. The traditional approach is able to identify the change which caused the failure, i.e., the culprit. In order to match such capability, the Uber approach also needs a culprit analysis to identify what change(s) introduce the fault. In Section 3.3, we propose an analysis to handle this situation, however, identifying compatible commits before the build and testing phases has lower overhead, and can be done using information about the project and changes to reduce the risk of failure. For example, by choosing commits that, based on project history, are not likely to fail. As seen before, 32% of the sample projects report a low failure rate (20% or less) for commits processed by the CI server.

In spite of the challenges introduced by Uber commits, the potential savings in developer feedback time and the transparency of processing Uber commits to the developers, suggests that our approach to continue integration has the potential to improve developer feedback time and improve CI server efficiency. To the best of our knowledge, no previous work has been done to modify the CI server workflow to remove its dependency on the developer workflow in order to reduce the bottleneck created by the server when multiple jobs are waiting for processing without losing the precision of results.

Chapter 3

Our Approach

The primary objective of our approach is reducing the amount of time developers wait for feedback from the CI server when the queue contains multiple jobs to be processed. In this chapter, we first define the underlying problem and then describe our insights that led to using Uber commits to improve developer feedback time.

3.1 Problem Definition

Let S represent a CI server configured to build and test code; and, $C_S = c_h, ..., c_t$, be the queue of commits waiting to be built and tested on S, where c_h is the next commit to be processed, i.e., the head of the queue, and c_t is the last commit submitted for processing by S. In a traditional CI server environment, where each commit in the queue is processed individually, the total time necessary for S to build, test and return the results, $r_{c_h}, ..., r_{c_t}$, for all of the commits in C_S is $t_{C_S} = t_{c_h} + ... + t_{c_t}$. We refer to each t_{c_i} as the *developer feedback time* for c_i.

Key insight: for CI projects, each developer feedback time, t_c, includes the time a commit spends waiting in the queue, plus the time necessary to process the commit. For CI projects where most commits can be successfully built and tested on the first attempt, and where the CI server is unable to process jobs as soon as they arrive in the queue ($|C_S| > 1$), we assert that it is possible to omit redundant processing steps resulting from processing individual commits which can lead to faster feedback to developers and improve the efficiency of the CI server. Our approach avoids these redundancies by merging the first commit to be processed, i.e., c_h, with other compatible commits waiting in the queue, to create an Uber commit that is built, tested, and integrated as a whole, thus avoiding the redundancies associated with building and testing each commit individually.

The underlying challenge of our approach is to find a transformation function, $f_c(C_S) \rightarrow C_U$, that identifies and merges compatible commits waiting to be processed, $c_h, ..., c_t$, [1] such that $t_{C_U} < t_{C_S}$, and $r_{C_U} = pass$, and $\forall r_{c_*} = pass$. In other words, f_c must be able to successfully merge commits in the queue into an *Uber* commit, C_U, to reduce the overall building, testing, and integration time while retaining the same results as if the commits were integrated and tested individually.

For our process to build and test an Uber commit, three conditions must hold:

Condition 1: The queue of commits waiting to be processed must be greater than one, i.e., $|C_S| > 1$. As the number of commits waiting in the queue increases, the potential to incorporate more commits into an Uber commit improves, thereby improving the chance to reduce the number of redundant steps, and ultimately reduce the developer feedback time. As we saw earlier, for a sample of projects, almost 30% report queued commits for 20% of the time or more a server picks a new commit to be processed, which supports the presence of this condition in current CI environments. When $|C_S| \leq 1$, applying f_c is equivalent to operating according to the current CI

[1]Without loss of generality, C_U can include just a subset of the commits. We include all commits to simplify the presentation.

practices, where every commit is built, tested, and integrated individually. The best opportunity for improving response time occurs when $|C_S|$ is large and all commits in the queue can be successfully merged into C_U. However, the cost of the culprit analysis increases as the number of merged commits in an Uber commit increases. Similarly, the probability of a successful merging decreases as $|C_S|$ increases.

Condition 2: Candidate commits must target the same development branch and be executed under the same configuration. The branch condition can be verified by the CI server with an inexpensive comparison of the branch or tag identifier contained in the event notification data for each commit. An alternative approach would be to establish separate queues on the CI server for each branch. The test for the configuration condition is also an inexpensive comparison. In this case, comparing a hash of the build script for each commit.

Condition 3: Candidate commits must merge without source code conflicts. Conceptually, we perform the merge of candidate commits based on the arrival order in the queue, beginning with the commit at the head of the queue, c_h, stopping when a conflict is detected or the end of the queue is reached. This approach enables us to rollback to the previous version with low overhead when a conflict is detected. Keeping order also increases the probability of successful merge and maintains fairness in the queue, i.e. commits tend to be processed in the order they arrived to the queue.

For our Uber approach to provide gains, candidate commits should share some portion of the build and test processes as the approach is based on reducing the redundancies in them. If there is no overlap between the build and test processes among commits then the C_U best possible performance can only be C_S. In practice, we see that many commits tend to trigger similar dependencies, leading to overlap in the builds, and also execute similar batches of tests, rendering support for this condition.

Additionally, Uber commits that fail to build or tests will need to be analyzed and partitioned such that each partition is treated independently, a process known as *culprit analysis* which we describe in Section 3.3. This analysis introduces extra overhead that is often small but in some cases could erase potential gains. Therefore, low failure occurrence is expected for this approach to improve the CI process. As we saw earlier for a sample of projects, almost 32% have failure rates under 20%, which supports the presence of this condition in current CI environments.

In the next section, we decribe the components of developer feedback time and how modifying the CI server to process Uber commits can reduce this time.

3.2 Reducing Developer Feedback Time

The are four main elements contributing to the developer feedback time for each commit, t_c, processed by the CI server:

- *t_wait*: the amount of time a commit spends *waiting* in the queue until S is available
- *t_fetch*: the amount of time for S to *fetch* the code
- *t_build*: the amount of time to *build* the code
- *t_test*: the amount of time to *test* the build, (this time may also include additional time for other analyses, e.g., static analysis).

The time for the CI server to process an Uber commit, t_{C_S} includes time for fetching, building and testing the Uber commit. It also include the overhead of performing the transformation function, tf_c, which is composed of:

- *t_ident*: the time spent *identifying* the commits to be merged

- *t_merge*: the time spent *merging* the commits into C_U.

In the remainder of our discussion, we omit *t_wait* and *t_fetch* from further consideration. The amount of time a commit spends waiting in the queue for processing depends on the rate of incoming commits and the CI server's available resources. Since we cannot control the rate of incoming commits, we focus on reducing the amount of time spent building and testing commits, and thus omit *t_wait*. *t_fetch* can also be omitted from further consideration as its value will be the same regardless of the approach; CI processing of traditional commits and CI processing of Uber commits both require each commit to be fetched once.

Then, in order for Uber commits to reduce developer feedback time, it must be the case that:

$$tf_c + t\_build(C_U) + t\_test(C_U) < \sum_{c \in C_S} (t\_build(c) + t\_test(c)) \tag{3.1}$$

To better understand why and when Uber commits can improve over traditional CI server processing, we analyze each of the remaining times individually.

The *build* activity can be viewed as three main steps: 1) obtain the dependencies, 2) compile the source code, and 3) generate packages to be distributed (with some variations, depending on the project). We can assume the execution time to get dependencies is constant across commits. The time required by compilation and distribution, however, depends primarily on the amount of code committed and its dependencies. When commits share code, merging them into a C_U can avoid recompiling the shared code (and their dependencies). Even when code is not shared across commits, they may share dependencies that are recompiled (although this can be mitigated with sophisticated catching mechanisms that avoid unnecessary recompilations).

Similarly, the *test* activity can benefit from processing Uber commits when there is overlap between the tests triggered by different commits. The counter argument also holds in this case: our approach cannot improve on commits that trigger distinct tests. However, it is common for projects to include test suites that are used to exercise every commit. More concretely, it is almost standard to have a set of smoke-like tests used on every pull-request before it is considered for integration into the main branch. In such cases, the proposed approach would save time by executing those tests only once for C_U.

Processing Uber commits has the additional overhead of applying the transformation function, f_c. We note that the commit queue in a CI server often contains commits that belong to different development branches which means that they are not amenable to be merged as an Uber commit as they operate on different code bases. Similarly, some commits are submitted with different target testing configurations. In the former case, the function f_c then needs to differentiate among branches either by maintaining independent queues or by performing a selection before merging. In the later case, when the differences are just the configurations, f_c can merge the commits, but it must exercise the C_U with the conjunction of the test configurations.

The cost of this function has two components, 1) the time spent identifying commits for merging into C_U, and 2) the time spent performing the merge. In the worst case, commit identification should involve a linear scan of commits in the queue and performing an evaluation on each one checking data already in memory. The merge process is delegated to the VSC working with the server because modern systems already implement a merge operation. Commits are merged incrementally from the earliest, C_h, to the newest until all commits are merged or a conflict if found. In the last case, a rollback operation is performed to remove conflicted changes from the Uber commit. If the first commit is part of a branch intended to be merged with another, i.e. the target branch, it is merged with the last available commit from the this branch at the time of its submission. In a MacBook Pro with 2.6 GHz Intel Core i5 and 8 GB of Memory, merging 2, 10 and 100 commits from a local Git repository for project Joomla took 1.98 sec, 3.21 sec and 31.07 sec respectively.

All these f_c activities are relatively light weight, and although they consume time, it is insignificant compared with the potential savings of avoiding building and testing redundancies.

3.3 When C_U Fails: Culprit Analysis

Despite a successful merge, a C_U may fail at either the build or test stage requiring an additional step, i.e., a culprit analysis, to determine which commit or commits caused the problem. Ideally, the analysis must be capable of detecting culprit commits with the same accuracy of processing individual commits, and it must be able to detect faults that are introduced by individual commits as well those caused by integration issues. Algorithm 1 provides a high-level description of a culprit analysis. At its core, the analysis partitions the C_U recursively, searching for smaller C_Us that can be built and tested successfully. If a search *bound* is reached, the approach resorts to processing the commits individually through $standardCI()$ (line 11). The analysis calls the *partition* function (line 3) on a failed C_U to divide it into parts. Each part can contain multiple commits, which are then merged into a new C_U (line 5) that is built and tested (line 6). If the new C_U fails, then it is subjected to further analysis by calling *ca* again (line 7). The effect of the culprit analysis to localize the source of the problem(s) by partitioning C_U into multiple groups for processing by the CI server. Each partition can vary in size, depending on the number of failures in C_U and their locations.

Algorithm 1 Culprit Analysis

```
1:  function $ca(failedC_U, bound)$
2:      if $bound > 0$ then
3:          $C_U parts[] \leftarrow partition(failedC_U)$
4:          for each $part$ in $C_U parts[]$ do
5:              $C_U \leftarrow mergeElementsIn(part)$
6:              if $\neg success(build(C_U), test(C_U))$ then
7:                  $ca(C_U, bound - 1)$
8:              end if
9:          end for
10:     else
11:         $standardCI(failedC_U)$                    ▷ Reached $bound$
12:     end if
13: end function
```

We note that the *partition* function is configurable and can implement various strategies in terms of the number of partitions and how they are selected. Our implementation of Algorithm 1 follows a binary search process, similar to that of finding the input inducing failure [27], with the addition of a temporal element as the commits submission sequence is known. The cost of this algorithm will depend on the number of commits included in the Uber commit and the failure rate of these commits.

The *bound* argument in Algorithm 1 is used to control the cost of the culprit analysis by limiting the depth of the partitioning process. When C_U fails, assuming a binary search implementation, the culprit analysis partitions $failedC_U$ into two partitions, i.e., two Uber commits, introducing $t\_build(C_{Ui}) + t\_test(C_{Ui}) + t\_build(C_{Ui+1}) + t\_test(C_{Ui+1})$. Since we do not know in advance when those costs will accumulate to surpass t_{C_S}, *bound* is set empirically to control the cost of the culprit analysis. If *bound* is set to 0, then the proposed approach performs a single attempt at forming a C_U, and if it fails it resorts to traditional CI. When *bound* is set to $log_2 \quad n$, where n

is the maximum number of commits in a queue, then ca will partition C_U all the way down to individual commits, if necessary.

Chapter 4

Simulation Framework

To analyze the effects of Uber commits on developer feedback time and study various instantiations of the proposed approach, we developed a simulation framework that mimics a continuous integration environment. Designing and implementing our own simulation framework was necessary to enable us to explore drastic changes that could have, and often did have, negative effects on the integration and testing outcomes. It also allowed us to repeat our simulations in order to develop a better understanding of how the proposed approach affects CI server processing.

4.1 Architecture

Fig. 4.1 shows a diagram of the architecture of the framework. It has four key components: the Data collector, the Data repository, the Model generator and the Simulator. The first component enables the consumption of data from existing CI servers. The current implementation can retrieve data from Travis-CI [17], Github [20] and CircleCI [19], utilizing a combination of web service (WS) or the API as needed. For example, the process of data collection starts by first fetching *Request* records for

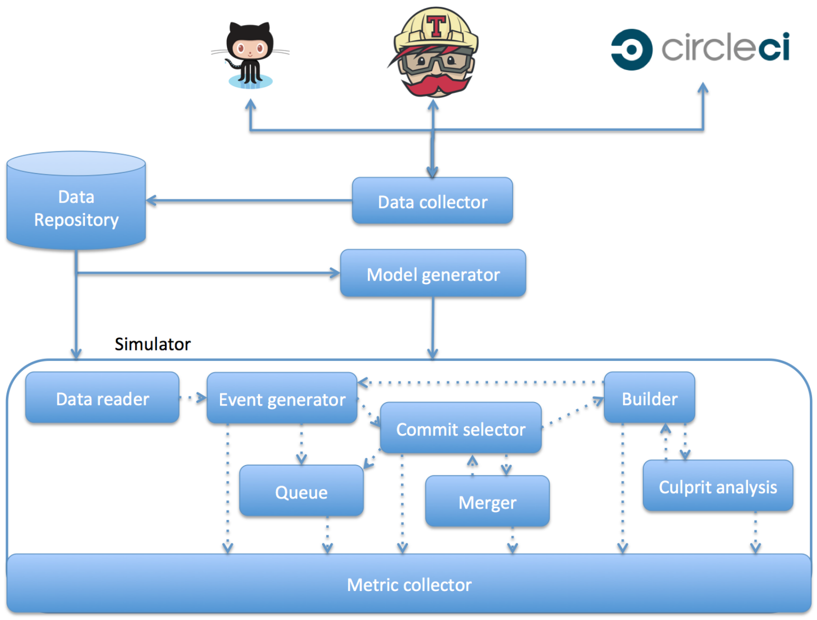

Figure 4.1: Diagram of the architecture of the simulator framework.

a selected repository through the WS. The component obtains these records from the url `http://api.travis-ci.org/requests` using an HTTP client. They are retrieved sequentially from the newest to the oldest, and records not associated with a build, i.e., the field *build_id* is empty, are discarded. With this id, the component uses the Ruby client library to obtain the *Build* record and concatenate it with the request. Records from builds that are currently running, i.e., *state* equal to *created* or *started*, are also discarded. It also concatenates the *Job* records associated this build together with the execution logs attached to each job record. Working with CircleCI is simpler because the Web Service provides a single entity, the *Build*, with all of the data needed to generate the input for the simulator. In the end, the data-collector generates a csv-file, which is saved in the Data repository, containing raw data such as the build-id, commit sha, branch, configuration, arrival time, and whether it failed while at building or testing time, as well as data that requires some basic computation such as the time spent on building and testing activities, which are extracted from the execution logs.

The data stored in the Data repository can be consumed by either the Simulator or the Model generator. The latter will consume this data to generate statistical models that mimic real CI environments but can be tuned to modify their behavior. The component provides tools to help you choose which model fits better the input data. For example, we explored the CI environment for Rails using a model consisting of a binomial distribution for modeling the failure outcome of the commit, two different gamma distributions for modeling the build and test duration, a weibull distribution for modeling the arrival-time between commits and a multinomial distribution for modeling the branch of the commit.

The simulator component performs a walk through the data stored in the Data repository or provided by the Model generator according to an integration strategy, while collecting the assessment metrics. Conceptually, it supports two basic policies described in the previous section *traditional* and *Uber*, applying the appropriate workflow as shown in Fig. 2.1.

This component has 8 key subcomponents: the Data reader that is responsible for reading the input data from the Data repository or Model generator and generating the proper data structures in memory. The Queue that simulates a queue in the CI environment. The Event generator that is responsible for simulating the arrival of new commits to the queue and notify the Commit selector that new commits are pending in the queue and the Builder is idle. The Selector inspects the queue and selects candidate commits to be part of an Uber commit. The Merger takes these commits and performs the actual merge on a git repository. Commits that are merged successfully are given to the Builder subcomponent and those that are remaining after a conflict was found are returned to the queue. The Builder takes commits or Uber commits and simulates their build executions using the execution times and build statuses provided in the input dataset, no build or test is actually run. In the case of a regular commits, the value of these fields are directly used for calculating the results of the simulation. In the case of an Uber commit, the values for these fields are calculated based on the largest values among the candidates before calculating the results of the simulation. Builds that are failing after the Builder has completed their simulation are taken by the Culprit analysis subcomponent to identify individual commits causing the failure status on the Uber commit. This subcomponent implements Alg. 1 and interacts with the Builder to simulate the build execution for partitions generated by the algorithm. Finally, the last subcomponent is the Metric collector that is responsible for collecting the metrics from the different subcomponents and writing them down in a file.

The Simulator component can work with different implementations of these subcomponents. In practice, the two basic policies are implemented as different implementations of the Commit selector. We have also experimented with a few Culprit analysis and Merger implementations.

4.2 Design Decisions

The simulator follows the approach as described in the previous section, but there are a few design decisions we made that require further explanation as they affect the fidelity of the simulation.

First, the simulator linearizes the execution of builds, serializing all parallel execution of jobs or threads. This choice was mainly made because we wanted to study this approach in a simple set up first, before moving to more complex scenarios. We can understand better the effect of this approach without parallelization. However, we need not consider the approaches to be exclusive and we will explore their combination in future work. Additionally, Travis-CI overwrites part of its history when builds are re-executed by a manual trigger, limiting the possibility of recreating the actual execution history.

Second, our implementation for f_c collects from the queue commits with the same branch value and same set of configuration ids (there is an id for each job). Then it relies on the *git merge* command to create the Uber commit and detect conflicts. Merges are performed one commit at a time in queue-arrival order and stop as soon as a conflict is found, which is reverted with the command *git reset –merge*. The checkout time for an Uber commit is calculated as the sum of the checkout out time of all jobs. The checkout time of an Uber commit job, which runs on configuration c, is calculated as the maximum *repo_ checkout_ duration* field value among all jobs from the individual commits that run on c plus the sum of *pr_fetch_ duration* field value from the same jobs. However, when two or more commits come from the same pull request, the *pr_fetch_ duration* value is included just once by selecting the maximum value. Note that we are considering the time of the merge operation to be zero. This decision was made because we cannot obtain this value from Travis-CI and we have already seen in Section 3.2 that this value is very low. If a commit cannot be found on the git repository, it will be considered as not mergable and will be marked to be run individually.

Third, this framework does not build the Uber commit nor execute its related tests. It also assumes there will not be an integration issue if git does not detect an integration conflict. In section 6.3 we present an experiment to validate how realistic our results are. The server time for an Uber commit is approximated using the values of the individual commits. The building time for an Uber commit is calculated as the sum of its jobs' building time and the value for each job is calculated as the maximum *building_ duration* field value among jobs from individual commits that run on the same configuration. Similarly, for testing results we utilize the results of individual commits. The testing time for an Uber commit is calculated similarly. However, when an Uber commit contains individual commits whose builds were canceled, the building (and similarly the testing time) is calculated as the sum of building time of canceled commits plus the building time of the Uber commit as if canceled builds were removed from it.

Finally, our implementation for the culprit analysis follows a binary search strategy, every partition generates two parts, and relies on the *build status* field of the individual builds to decide if the Uber commit has issues or not. A new partition is not executed immediately, it is returned to the queue in the position of the commit that is first in the queue and is also part of this partition.

Chapter 5

Study Design

5.1 Research questions

We applied our simulation framework to a set of projects working with real CI environments. We started collecting data from projects working with Travis-CI or CircleCI that presented characteristics that should support the Uber commit generation, i.e. low failure rate, commits being queued up and overlap of source and test code, and we performed simulations of the operations of CI server applying the traditional and our proposed approach. Our first goal of the study was to verify if Uber commits could be constructed as frequent as we expected them to be. Then, we verified whether the application of Uber commits could improve the efficiency of the CI environments. We divided the study into the following research questions:

RQ1: How often could Uber commits be generated? This question was aimed to understand the frequency in which Uber commits could be generated by the simulation using data from real CI environments and their characteristics.

RQ2: What are the benefits of Uber commits over individual commits? This question was aimed to understand what benefits, if any, can provide the application of Uber commits. We had in mind two aspects: the developer feedback time and the resource utilization of the server.

RQ3: How realistic are the Uber commits presented in the simulations? This question was aimed to verify if simulation results could be translated to real CI environments since simulations could not reproduce completely these environments.

5.2 Design

We designed two experiments, *Traditional CI vs Bounded Uber CI* and *Traditional CI vs Unbounded Uber CI*, involving the simulations of a CI server operating over a series of commits collected from a real project. In both experiments two simulations are performed for the same input: one of the server applying the traditional approach to CI and another applying our approach. The main difference between the two experiments is the configuration for the culprit analysis, in particular, the parameter *bound* of Alg. 3.3. In one case this analysis is bounded, $bound = 0$, to stop after the first the execution of an Uber commit fails and to execute its commits individually. In the other case, the analysis is unbounded, $bound = log_2 \quad |C_U|$, making the server to create partitions (smaller Uber commits) as along as it is possible in order to find the culprit commit(s).

It is important to highlight the simulations, as explained in Chapter 4, do not compile code or execute tests. We made this decision to avoid the complexity of recreating the environments required for projects to work. This allowed us to potentially perform simulations on several projects and a great volume of commits, especially old ones. However, this imposes a limitation on the experiments, Uber commits can only be evaluated with respect to the merge success at the text-merge level, command *git merge*, but not at the build or test level. RQ3 was proposed to address this issue and we designed a new experiment to mitigate this limitation. We can sample Uber commits resulting from the simulation of a project for which we can reproduce its environment on Travis-CI, submit these commits to the server and then compare the results to verify whether Travis-CI would behave as the simulation or not.

5.3 Metrics

The simulation framework generates a set of metrics in order to understand the behavior of the server. The relevant metrics for this study are the following:

- Uber commit individual commits: the commits taking part of each Uber commit.
- Uber commit creation time: The time the server started the selection process that ended up in a new Uber commit.
- Uber commits reporting failures: calculated as Uber commits that triggered the culprit analysis process.
- Queue size: the number of builds in the queue before picking a new commit or commits in the case of an Uber commit.
- Feedback time of a commit: calculated as the time the server reports the results for this commit minus the time it was added to the server queue.
- Server execution time for commits or Uber commits: calculated as the time the server reports the results for this commit minus the time it was selected by the server.

The first four metrics are used for measuring frequency of occurrence of Uber commits and their characteristics and the others are use for measuring their performance either from developer or resource utilization perspective.

Note, the results of a successful Uber commit will be used as the results for its individual commits, while the culprit analysis will provide individual results for commits after the execution of the Uber commit.

5.4 Artifacts

We started exploring the ideas of this thesis with Ruby on Rails, the Web application framework. We inherited it from a previous project and we considered it was good candidate to work with because it is a well-know project, it is a mature project with an active development on Github, it contains several satellite projects around it and it works with Travis-CI. At the time of this writing, it reports 27 branches, 58951 commits, 3112 contributors and 16948 Pull-Requests (631 are still open) on Github. This repository has been active for 4209 days (11 years approximately) and it reports 36512 builds executed on Travis-CI. We soon realized the Uber approach would not work all the cases and decided to pick two more project. At that time, the Uber approach was presenting positive results on Rails. We wanted to pick one candidate for which the Uber approach could not work and other for which it could excel. We obtained a list of all projects available on Travis-CI with the number of executed builds between May 13, 2016 and May 17, 2016. We inspected the top 25 projects and selected Joomla and Code.org. These two projects were met the criteria of reporting low failure rate, having more activity than Rails, being well-know projects and reporting good results with the Uber approach. Since simulations for Rails started to report bad performance of the Uber approach when we removed a few assumptions from the simulator, we decided to select Rails as the bad candidate.

Joomla is a Content Management System. We selected it because it is well-known project in the Web community, has high activity in Travis-CI and Github, is a mature project, reports code coverage and we can re-execute its builds on Travis-CI by forking the project. At the time of this writing, it reports three branches, 26767 commits, 486 contributors and 8247 Pull-Requests (303 are still open) on Github. This repository has been active for 3950 days (11 years approximately). 25880 builds have been executed on Travis-CI and 4424 builds on their Jenkins server. New commits or PR are submitted to the *staging* branch, which is tested on Travis-CI, and if they are successful, they are merged to the branch *master*, which is tested on Jenkins. Its latest reports show test coverage of 46.9%.

Code.org project contains the source code for the website and the Code Studio Platform that is used for teaching Computer Science. We selected it because Uber approach reports good results for it, it was one of the most active project on Github and Travis-CI, it was also well-known project in the community and it presented reports of code coverage. At the time of this writing, it reports 261 branches, 36298 commits, 46 contributors and 9733 Pull-Requests (46 are still open) on Github. This repository has been active for 700 days (2 years approximately). 45340 builds have been executed on Travis-CI. However, CircleCI is the main environment where most of the tests gets executed. Its latest reports show test coverage of 88%.

In the case of Rails, we inherited a dataset with 4006 builds extracted from Travis-CI. These are builds between ids #53890000 and #84190000. We calculated form this sample that builds contain 28 job on average, commits or Pull-Requests are submitted every 1.26 hours on average, build last 3 hours on average after linearizing the execution and failure rate is 28.45%.

For Joomla, we started collecting build data from Travis-CI from the last available build until we had 10000. At the end we collected 10042 builds with ids between #54100398 and #130600329. We decided to gather this amount of builds in order to capture different periods of the development process. We obtained from this sample that a commit or Pull-Request is submitted every 11 minutes and builds last 22 minutes on average. The failure rate for these builds is 16% and each build contains 5.2 jobs on average.

Finally, we extended our simulation framework to work with CircleCI and gathered data from Code.org. We collected the last 10000 executed builds, which were those between ids #2902 and #13750. This dataset reported builds with single jobs, arrivals of new commits or Pull-Requests every 25.6 minutes, build execution times of 21 minutes on average and failure rate of 13.8%.

5.5 Data Sanitization process

While most records for the input dataset of our framework were obtained without issues, some inconsistencies were found and we had to apply a sanitization process.

Twenty-three and 73 execution logs reported parsing errors on Joomla and Rails respectively. These issues may be the result of infrastructure failures where the job was halted before logging the required data. We also found cases where Travis-CI logs are erroneous even for passing builds. For instance, the logs present an invalid format or their build duration and finish time are incomplete even though they have finished successfully. Nineteen and 49 builds were affected by these issues on each of the projects. We decided to deleted these job records as we could not obtain the real value. Note, this decision is biased against our approach because it reduces the execution time for these builds as well as the possibility of commits to queue up. Besides, we made sure to keep the failure status on builds that any of their jobs was reporting failures and was selected to be deleted.

A second kind of issue involved jobs reporting a negative duration. This is common on canceled jobs that could not get to execute but there are also records where

Travis-CI failed to save the real value. We converted the value of these records to zero since this is the intended value for jobs that do not get to execute and then we proceed to delete all job record whose testing duration lasted longer than its own execution. This issue affected 190 and 138 jobs (40 and 18 build) from Joomla and Rails respectively.

Code.org did not present any issue on the data collection stage but sanitization was required when mapping the CircleCI states for build outcome to the framework states. In particular, there is one state called *no_ test* that is related to builds that are intentionally triggered to build commits without test execution. This state is reported when these operations complete successfully. It was not clear if the purpose of these builds was to perform a light test, to check if the code compiles, or to generate builds for other purpose than testing. In the last case, providing an Uber commit instead of the specified build may not adequate. Therefore, we opted to be conservative and mark them to be executed individually on our simulations.

5.6 Threats to validity

Our results were obtained through a simulation process and, although our framework tries to mimic real CI environments, simplification were made to make it practical to obtain results. For instance, we are not building or testing the code. Therefore, results may differ in practice. This also implies that the approach may have to be revised when attempting to implement it on real environments since there are aspects that cannot be captured by the simulator, like integration issues.

We are excluding from this study other techniques to improve the CI efficiency, not even standard practices such as parallelization and test selection and prioritization. These other techniques may reduce the size of the queue considerable to make our approach inapplicable. Moreover, the applicability and adoption of this approach on real CI environments will be limited if it is not extended to work with other techniques.

We are assuming a 100% of redundancy of source and test code between commits that take part in the same Uber commit.

We explored this idea with three projects that were selected purposefully. Although they differ to each other and provide insights of how the Uber approach would perform under different contexts, findings may not generalize to other projects working on CI environments because they may not share the same characteristics, which are shaped by factors such as the development process, the number of developers and contributors, popularity of the project, type of software, if the project is open source or private, understanding of CI practices and the reported failure rates and rate of change. Additionally, we tested out approach on the ecosystem of Travis-CI and Github but there are many CI technologies that can be integrated with several Version Control, Building and Testing Systems. Some CI servers provide interfaces to extend the functionality through plugins. We cannot guarantee our approach will work on all possible CI environments.

Finally, the examination of Uber commits on Travis-CI was based on build results and the code was not inspected. There is a risk that integration issues went undetected making the Uber commits generated by the simulation less realistic than they appear to be. Moreover, Joomla Test suites report a Code coverage of 46.9% that seems to be reasonable to developers but it is not high enough to mitigate concerns that tests may not be good enough to detect integration issues that could result of generating Uber commits.

Chapter 6

Study Results

We focus on the results for the research questions in this section and we discuss their implication in the next section.

6.1 How often could Uber commits be generated

Simulations with Joomla reported 1251 and 1213 Uber commits could be generated on the experiments with bounded and unbounded culprit analysis respectively involving 4898 and 5076 builds on each experiment. These commits account for the 48.77% and 50.54% of the builds under study in each case. In the case of Rails, 413 Uber commits could be created in simulations applying both the bounded and unbounded culprit analysis. While this number may appear small, these Uber commits were generated from 3745 of the 4006 the commits used in the simulations. Finally, 1167 Uber commits were reported for Code.org simulation with the bounded culprit analysis[1] involving 9084 (83%) of the commits processed in the simulation.

[1]Results for the Uber approach with unbounded culprit analysis for Code.org could not be provided in this work because the simulation was still running after 200 hours of execution and only 3740 commits were processed.

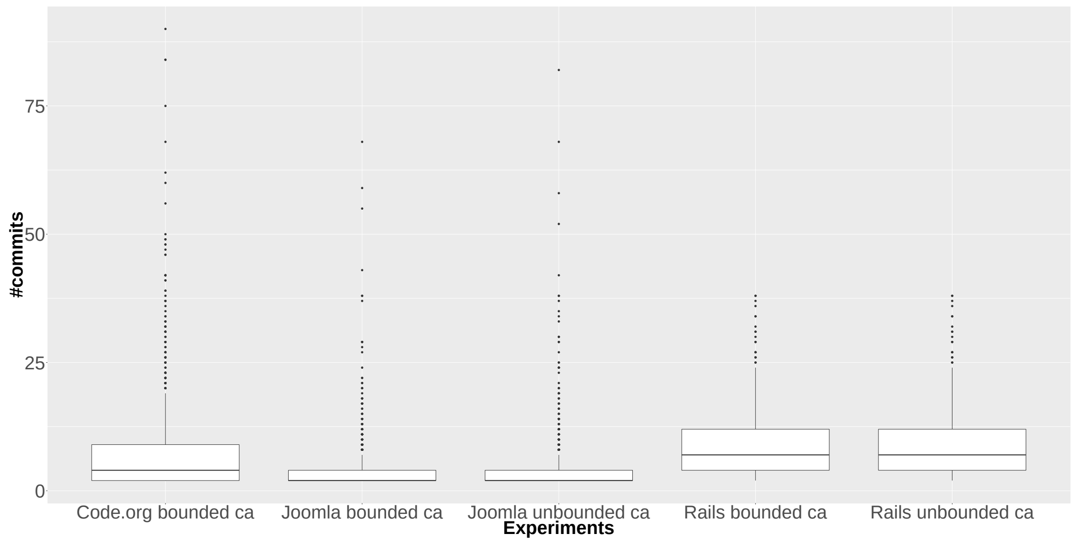

Figure 6.1: Uber commit size for the different experiments.

These results show commits queue often and they can be transformed in Uber commits but note some of the commits taking part of these Uber commits ended up running individually due to failures reported by the latter. This is the particular case for Rails in which 76.5% of Uber commits reported failures. We will discuss below that this phenomenon is significant because it affects dramatically the performance of the Uber approach. In the case of Joomla, 31.65% and 32.23% of the Uber commits presented failures for the simulations with bounded and unbounded culprit analysis respectively and 42.3% for Code.org with bounded culprit analysis.

Fig 6.1 shows the number of commits involved on each Uber commit on each of the simulations. In most of the cases, this number is small, the average number on each case range between 3.91 and 9 commits, but values above 40 are reported. It can be observed that Joomla presents smaller Uber commits than the other projects and Rails presents the largest ones. Nonetheless, Fig 6.2 shows the percentage of commits from the queue that ended up taking part of each Uber commit. It can be seen that most of the time the Uber commits from Joomla are formed with 83% of commits in

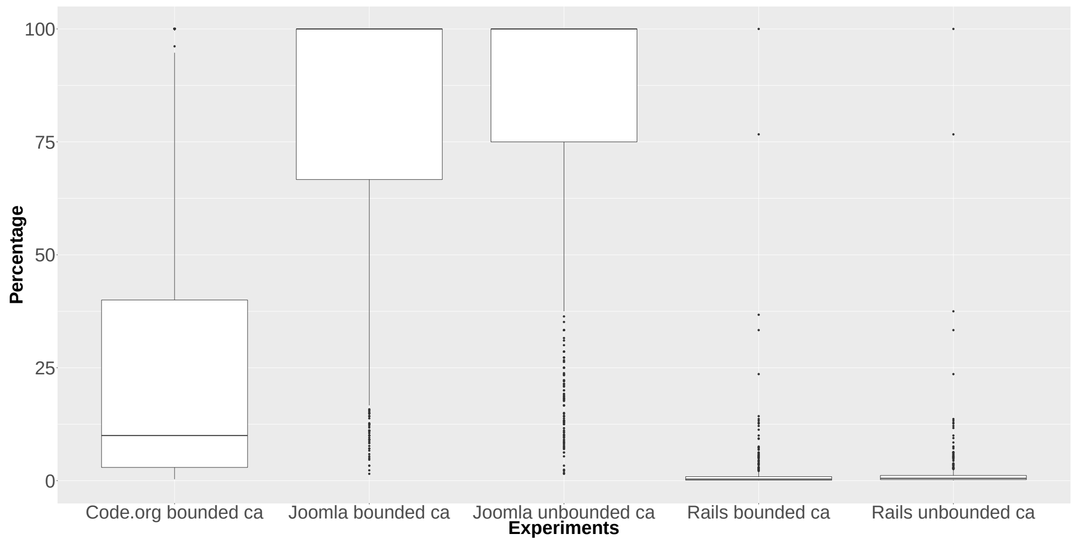

Figure 6.2: Percentage per Uber commit of the numbers of commits in the queue taking part in this Uber commit.

the queue, 25.83% of commits in the case of Code.org but only 1.5% of commits in the case of Rails.

6.2 What are the benefits of Uber commits over individual commits?

In the case of Joomla, the simulations reported 3870.66 hours of execution for the server, i.e. the amount of time the server was processing commits, applying the traditional approach, 3423.15 hours for the Uber approach with bounded culprit analysis and 3577.87 hours for the Uber approach with unbounded analysis. The difference of the traditional with each of the other simulation results are 447.5 hours and 292.79 hours respectively, representing the 11.56% and 7.56% of the server execution time for the traditional approach. Fig. 6.3 shows the difference between the cumulative server execution time for the traditional approach and those obtained for the Uber approach with bounded (solid line) and unbounded (dashed line) culprit analysis on each case.

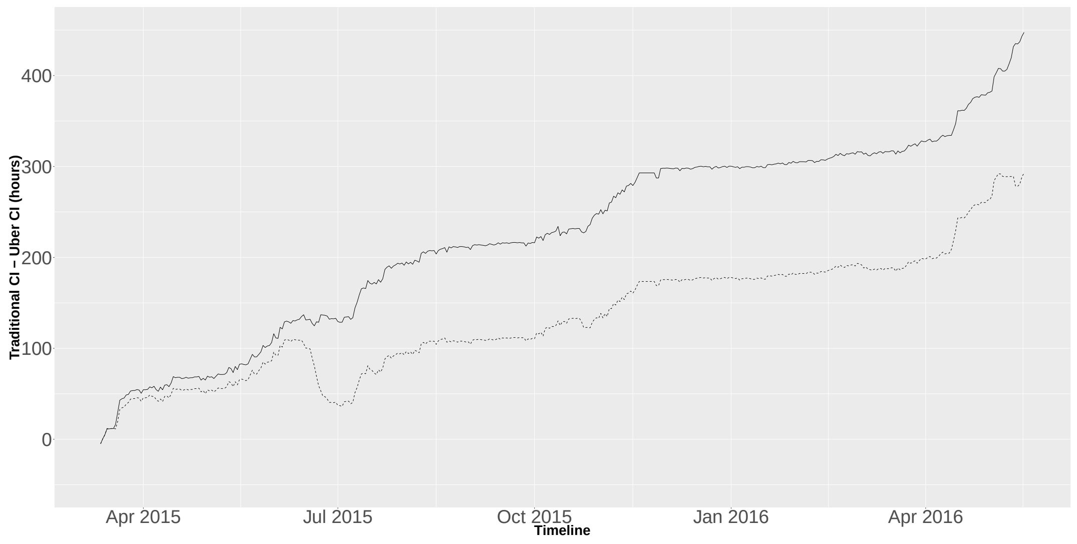

Figure 6.3: Server execution cumulative time difference between traditional and Uber approaches for Joomla project. The solid and dashed lines show the values for the Uber approach with bounded and unbounded culprit analysis.

Both curves present non-negative values with increasing tendency that show the Uber approach reduces the resource utilization. The solid curve presents no significant decreasing periods, in which the traditional approach utilized less resources than the Uber approach, but there are periods where the difference remains approximately constant, which means the same amount of resources were utilized for both approaches. The dashed curve shows a slower growth, periods of approximately constant value last longer. It also presents one decreasing period of significant value around July 2015.

Code.org reported similar results, the traditional approach required 3776.419 server execution hours to process all the commits while the Uber approach required only 3380.207 hours. The saving achieved by applying the Uber approach with bounded culprit analysis is 396.21 hours, which represents the 10.5% of the server time required by the traditional approach. Fig. 6.4 is similar to Fig. 6.3 from Joomla, the cumulative difference presents no negative value and an increasing tendency showing the Uber approach in general consumes less resources than the traditional. However, this figure

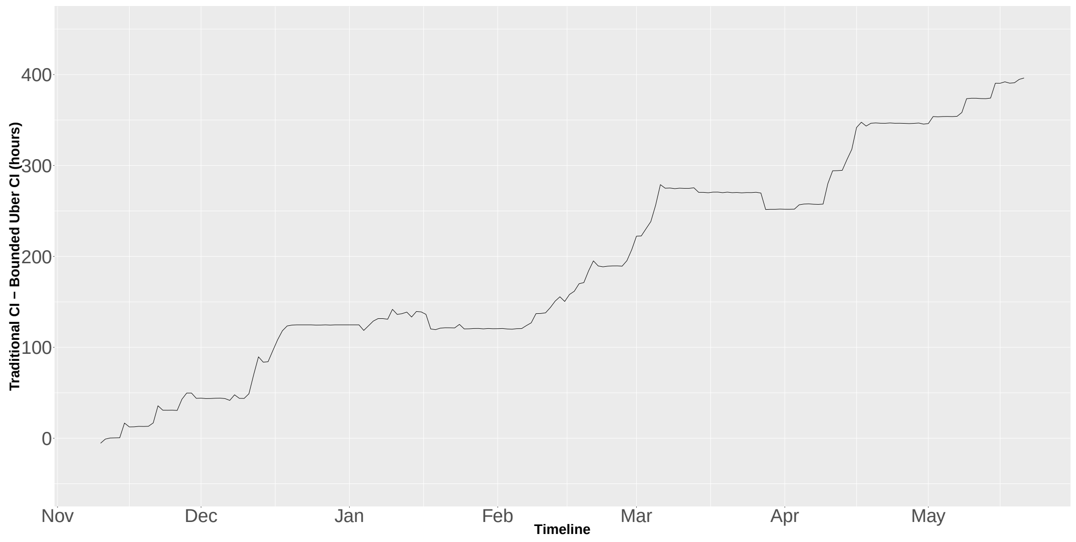

Figure 6.4: Server execution cumulative time difference between traditional and Uber approaches with the bounded culprit analysis for Code.org project.

presents longer periods where the difference is constant, in which both approaches consumes the same resources, and two periods were the difference was reduced on middle January and March.

Rails presented a different scenario, the Uber approach was not able to produce saving. The traditional approach required 12324.66 hours of server execution while the Uber approach required 12891.75 hours and 16480.55 hours with the bounded and unbound culprit analysis. The overhead introduced by the approach applying the bounded analysis was small, 567.09 hours (4.6%) but the one introduced by the Uber approach with the unbounded analysis is significant, 4155.89 extra hours of server execution (33.72%) were required to process all commits. Fig. 6.5 shows the server cumulative time difference between the traditional approach and the Uber approach was negative for most parts of the simulations and was oscillating around value -5 hours until July 2016. At that point, the difference abruptly dropped until getting to the final values, -567.09 hours and -4155.89 hours on each of the cases. In order to understand this behavior, we need to understand how the linearization of build execution affected Rails. The dataset used as input for the simulations reports that on average 19 commit are submitted per day. Travis-CI can process this amount in a single day because builds are completed on 40 minutes on average. Note this time is achieved by applying techniques, like paralllelization, to reduce the execution time of builds. The simulator does not apply these techniques and builds require on average 3 hours to be processed. The simulator is not able to process all the commits that arrive on a single day and they start queuing. As a result, the server is constantly working since the queue is constantly growing. The oscillation reported on Fig. 6.5 is caused by this behavior. The cumulative difference does not grow because none of the server is idle during this period. The main difference between these servers is that the one working with the traditional approach processes individual commits while the other also processes Uber commits. However, most of the time the server spends on Uber commits is wasted because of the failures and the overhead introduced by the culprit analysis. In general the culprit analysis introduces a small overhead that is usually insignificant but it starts accumulating if a series of Uber commits are executed one after the other and most of them incur into the culprit analysis. We have seen this is the context of Rails since 93.49% of the commits were part of an Uber commit and 76.5% of the Uber commits reported failures. Consequently, on July 2016, the server under the traditional approach is done while the other is still working due to the delay introduced by the culprit analysis.

The overhead on server execution time could be tolerated if there are significant gains on feedback time of processed commits. Fig. 6.6 presents the developer feedback time difference between the traditional approach and the Uber approach for Joomla applying bounded (top graphic) and unbound (bottom graphic) culprit analysis. Note this is a similar metric to the server execution time but it also accounts for the period commits were waiting in the queue. This metric is used for measuring how much time

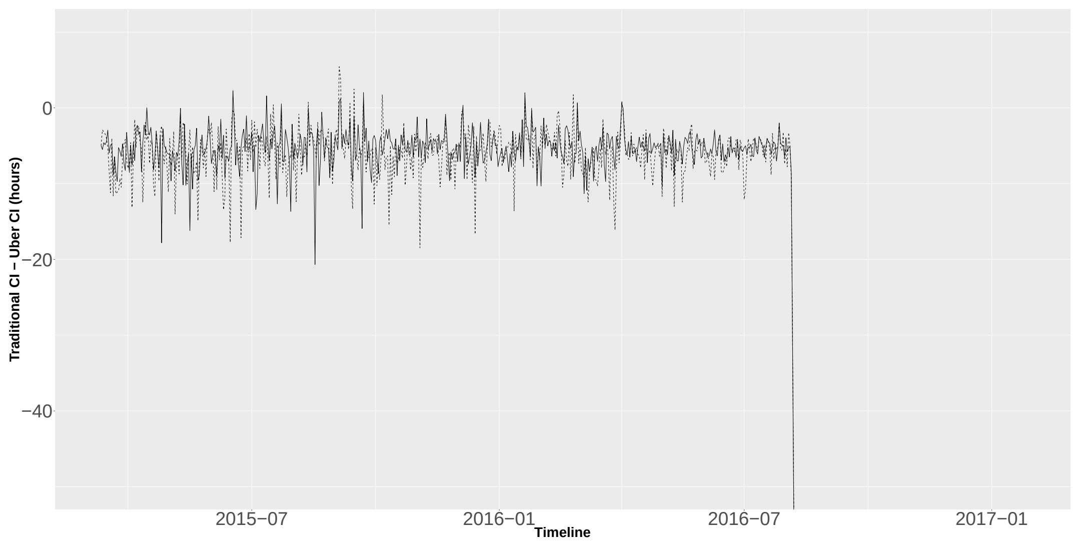

Figure 6.5: Server execution cumulative time difference between traditional and Uber approaches for Rails project. The solid and dashed lines show the values for the Uber approach with bounded and unbounded culprit analysis. The final values are not shown. The difference for the Uber approach with the bounded analysis keeps dropping until reaching the value of -567.09 hours on September 2016 and the difference for the Uber approach with unbounded analysis drops until reaching -4155.89 hours on January 2017.

developers need to wait to receive results from the server. The x-axis can be thought as a timeline since every point is a build and they are ordered by their queue-arrival time. Fig. 6.7 shows the cumulative feedback time difference, which is the same data with a different perspective. The experiment with bounded culprit analysis reports savings of 19034.85 hours (30.92%) while the other reports savings of 4307.09 hours (7%). These two figures also show that Uber commits occurrences are distributed along the timeline and are not focused on a single point of time. Nonetheless, most significant savings were achieved at the beginning and at the end of the simulations.

A build reports positive value when its wait time is reduced, which can be caused by two effects, either the server reports a shorter execution time by applying the Uber approach for any of the commits in the queue that are in front of the commit of this build or its commit is promoted to be executed as an Uber commit with the

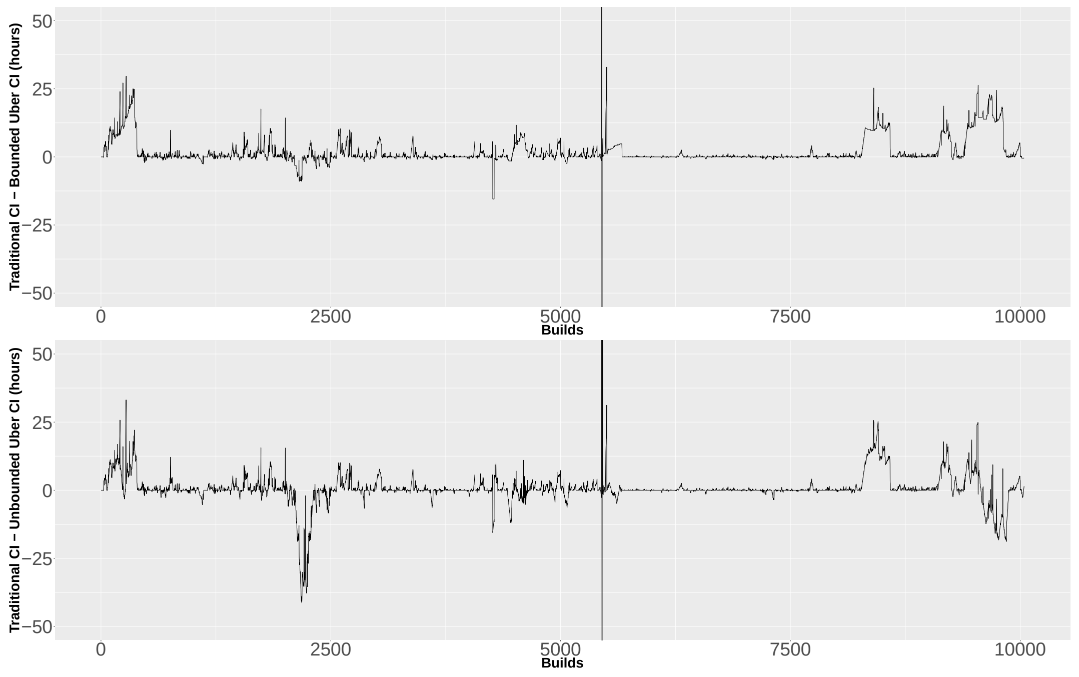

Figure 6.6: Feedback time difference between traditional and Uber approaches applying the bounded (top) and unbounded (bottom) culprit analysis for Joomla project. The highest peak and lowest valley in the middle of both graphics are partially shown. They real values for the top one are 191.09 hours and -192.037 hours respectively and 192.09 hours and -193.09 hours for the other.

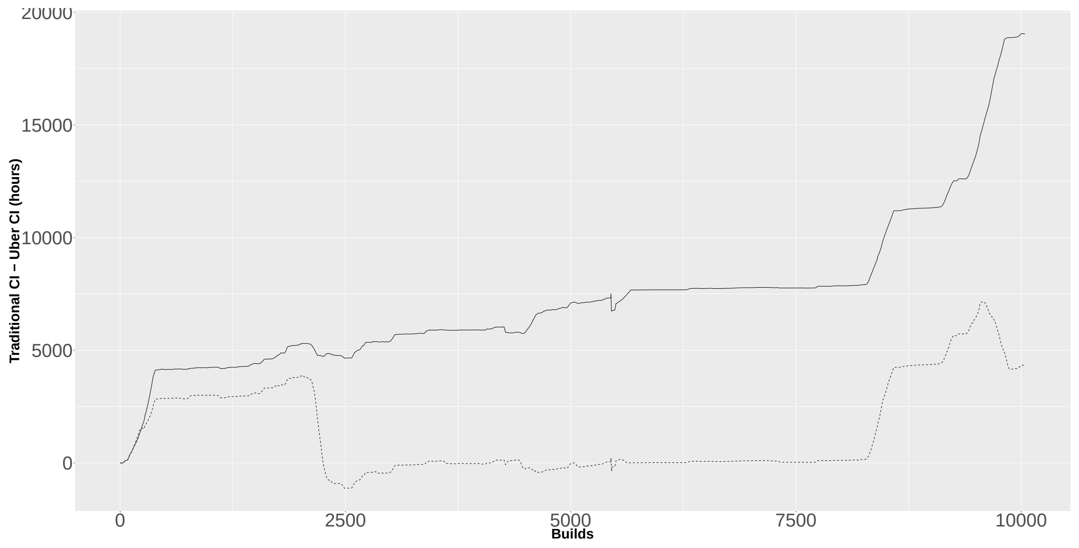

Figure 6.7: Cumulative Feedback time difference between traditional and Uber approaches applying the bounded (solid line) and unbounded (dashed line) culprit analysis for Joomla project.

first commit in the queue. The positive peak of 191 hours shown in Fig. 6.6(top) is caused by the second effect. In the traditional approach this commit is executed after a commit taking 190 hours but, in the Uber approach, it is able to avoid waiting for this commit to be over by taking part of an Uber commit that is executed first.

The negative results in Fig. 6.6(bottom) are also caused by a high failure rate and the overhead introduced by the culprit analysis. *ca* performs the analysis until all commits are executed individually. As a result, the execution of partitions provides anything but extra overhead to the process. This overhead will cause more commits to queue up and the cycle will repeat until the failure rate decreases. Negative values in Fig. 6.6(top) are also caused by the overhead introduced by the culprit analysis but in this case the depth of the analysis is bounded. Additionally, commits could suffer a penalization if they are the first in the queue and merged with another commit that requires much more server time and all of them are selected to be part of an Uber commit. This occurs because the simulator calculates the cost of the Uber commit based on the highest values reported by the candidates. This penalization is usually small in comparison to the overhead introduced by the culprit analysis. The large negative valley of value -192 hours in Fig. 6.6(top) is an example of a significant penalization caused by this phenomenon. In this case, a few commits in front of the commit requiring 190 hours of server time are selected to be part of an Uber commit with this one suffering an unnecessary delay.

Figures 6.8 and 6.9 show the cumulative and non-cumulative developer feedback time difference between the traditional approach and the Uber approach for Rails project. They are a clear example that the Uber approach cannot be applied blindly since they show how commits are being delayed as Uber commits trigger the culprit analysis. At the end of the simulations, the Uber approach introduced delays of 1533033 hours (8.66%) and 9354617 hours (52.87%) under the bounded and unbounded culprit

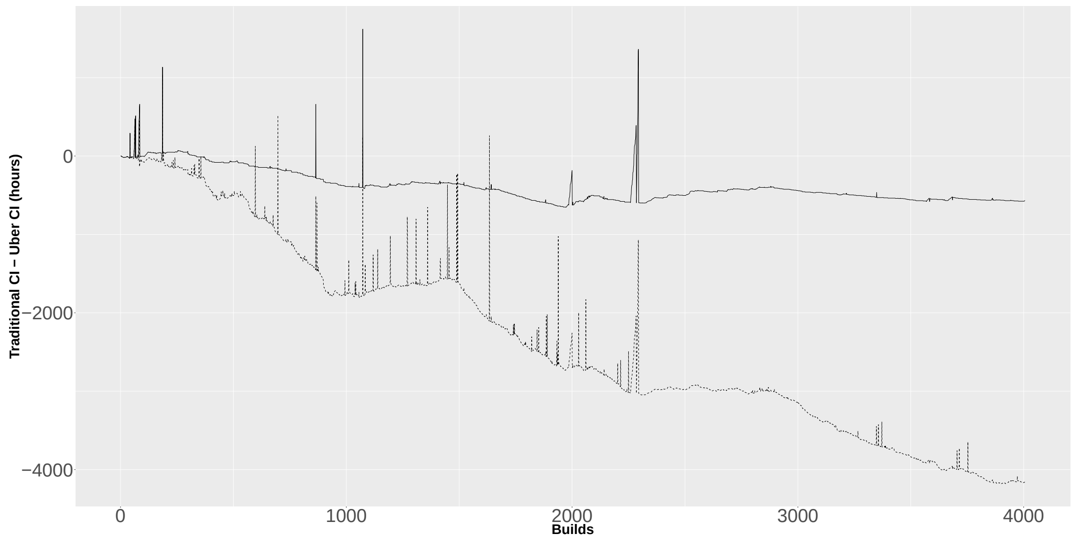

Figure 6.8: Feedback time difference between traditional and Uber approaches applying the bounded (solid line) and unbounded (dashed line) culprit analysis for Rails project.

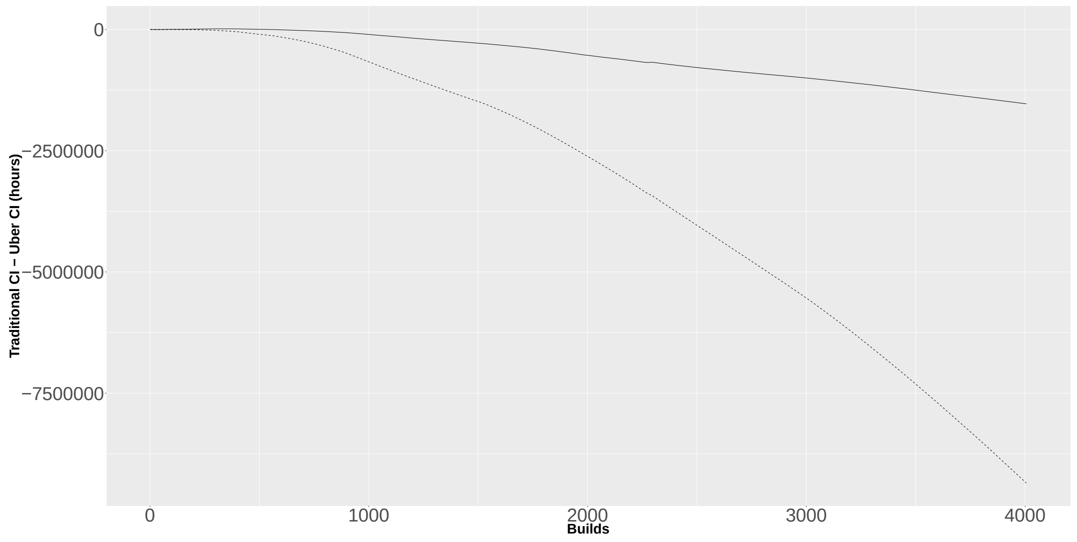

Figure 6.9: Cumulative Feedback time difference between traditional and Uber approaches applying the bounded (solid line) and unbounded (dashed line) culprit analysis for Rails project.

analyses respectively. On the other hand, Figures 6.10 and 6.11 show a total saving of 63047.65 hours (21%) for project Code.org. Although this saving is smaller that the one achieved for Joomla, the figures show significant gains are distributed through all the simulation period.

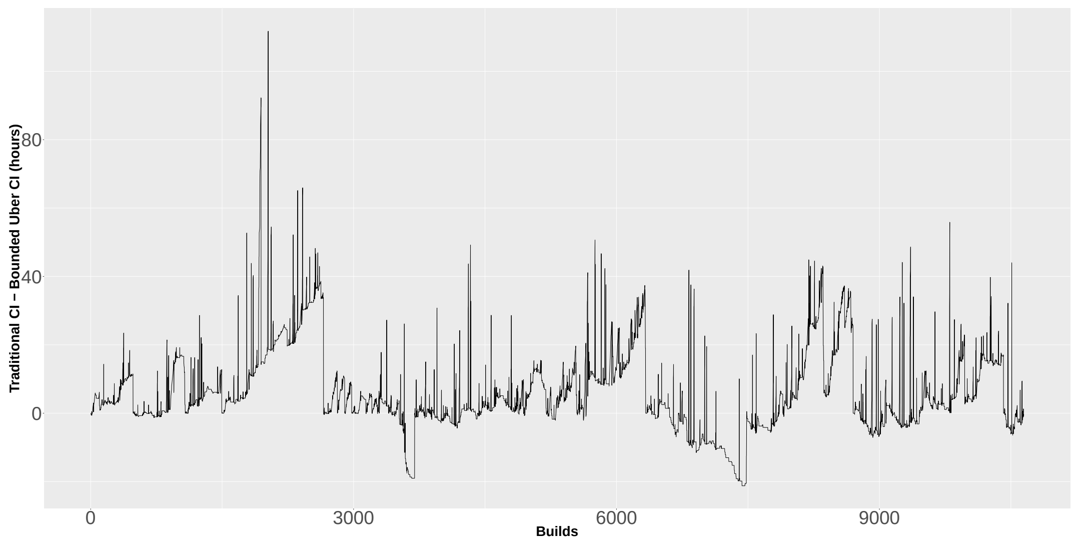

Figure 6.10: Feedback time difference between traditional and Uber approaches applying the bounded culprit analysis for Code.org project.

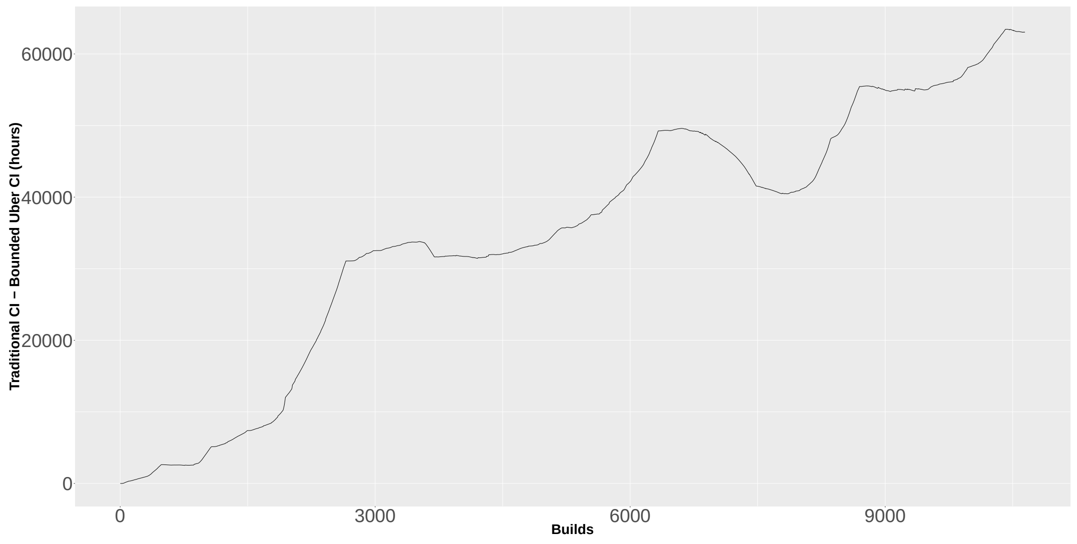

Figure 6.11: Cumulative Feedback time difference between traditional and Uber approaches applying the bounded culprit analysis for Code.org project.

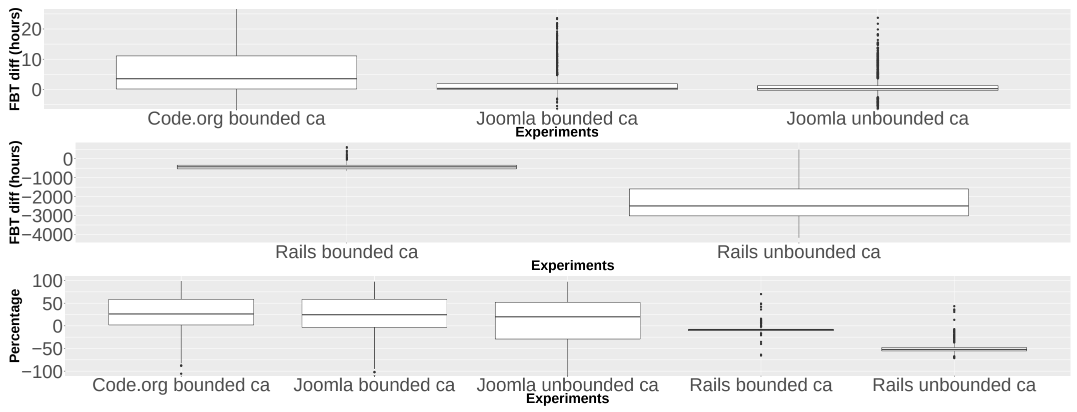

Figure 6.12: Average feedback time savings achieved per commit by being part of an Uber commit. Top and middle plots show the savings on hours while the last one shows the percentage.

Previous figures showed the global effect of applying the Uber approach. We can understand better the local effect by looking at Uber commits individually. Fig. 6.12 presents the savings on developer feedback time achieved per commit by taking part of an Uber commit. The top and middle box-plots show the savings (negative values are overhead) achieved per commit in concrete hours and the last box-plot show the percentage that these savings represent. On average Joomla reports per commit savings of 13.73% and 1.92 hours for the Uber approach with bounded culprit analysis and -23% and 43.2 minutes for with the unbounded approach. Rails reports -7.96% and -372.1 hours for the first case and -48.33% and -2243 hours for the second. Finally, Code.org reports 26.95% and 6.56 hours for the Uber approach with the bounded analysis. It can be observed in the figure that savings achieved for Rails are outliers and that the Uber approach provides similar gains on percentage to Joomla and Code.org. In concrete hours, larger gains are provided for Code.org.

We can also inspect the server execution time for Uber commits and their individual commit execution to understand the local effect of the former on resource utilization. The Uber approach applying the bounded and unbounded culprit analysis presents average savings per commit of 26.08% and 22.31% respectively on Joomla, -6.13% and -30.72% on Rails and 21.06% on Code.org. In concrete hours, the savings are 21 minutes and 14.4 minutes on Joomla, -1.37 hours and -10.06 hours on Rails and 20.4 minutes on Code.org. Fig. 6.13 shows the distribution of the savings reported by Uber commits. 857 (68.5%) and 861 (70.98%) Uber commits reported savings for each type of culprit analysis on simulations for Joomla, 98 (23.7%) and 129 (31.23%) for Rails and 674 (57.75%) for Code.org. Observe, Uber commits presenting savings with the bounded analysis are those that do not report failure and do not report overhead for the merging process while commits reporting savings with the unbounded analysis be also include failing Uber commits that the culprit analysis could identify without incurring into overhead. This is group is small, only 9.9% of failing Uber commits reported for Joomla and 10.12% for Rails. On the other hand 394 (31.49%) and 352 (29.01%) Uber commits introduced overhead to the server when the bounded and unbounded culprit analysis were applied respectively on Joomla simulations, 315 (76.27%) and 284 (68.76%) on Rails and 493 (42.24%) on Code.org. Fig 6.14 shows the percentages of the overhead introduced by these Uber commits. It can be observed the unbounded culprit analysis introduces more overhead than the bounded one, reporting values even higher than 100%.

6.3 How realistic are the Uber commits presented in the simulations?

We sampled 100 Uber commits from the 1251 reported by Joomla with the bounded analysis and reproduced them manually on Travis-CI through a Github fork. All the commits involved in an Uber commit were merged, following the proper order, into a

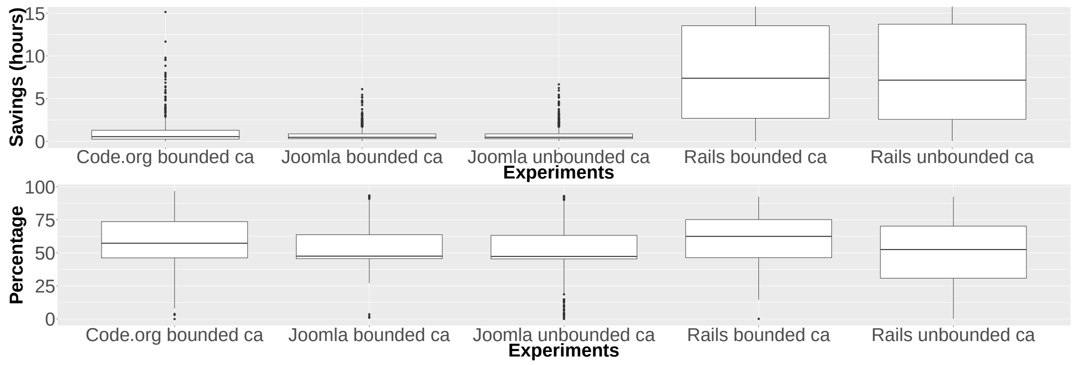

Figure 6.13: Savings achieved by Uber commits with respect the individual run of their composing commits. Joomla experiments present 857 (bounded ca) and 861 (unbounded ca) Uber commits. Rails experiments present 98 (bounded ca) and 129 (unbounded ca) Uber commits. Code.org experiment presents 674 (bounded ca) Uber commits. UC introducing overhead are excluded.

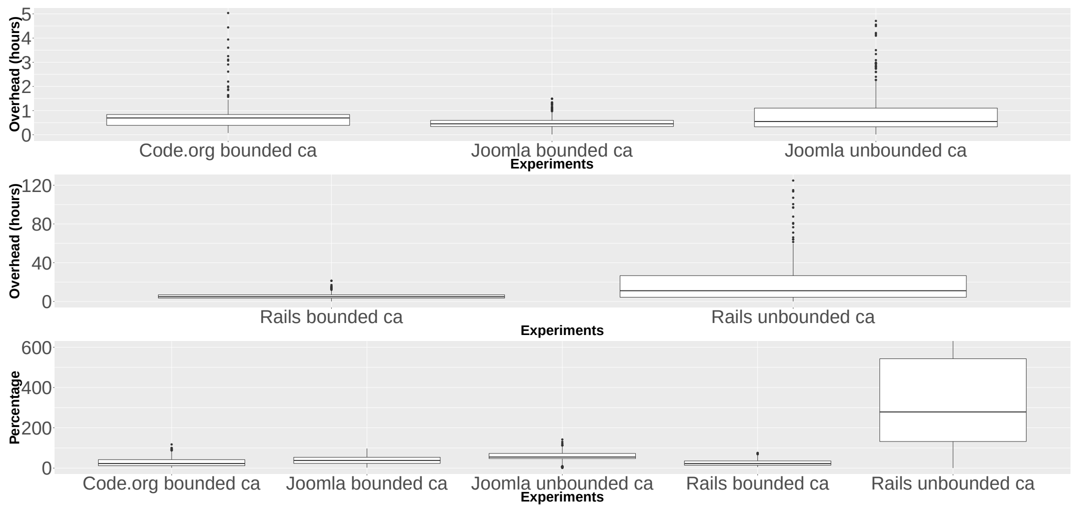

Figure 6.14: Overhead introduced by Uber commits with respect the individual run of their composing commits. Joomla experiments present 394 (bounded ca) and 352 (unbounded ca) Uber commits. Rails experiments present 315 (bounded ca) and 284 (unbounded ca) Uber commits. Code.org experiment presents 493 (bounded ca) UC introducing savings are excluded.

new branch of this fork and a build was triggered on Travis-CI for this branch when the merge process was completed. Note the simulator generates the build status for Uber commits based on the build status of their individual commits. Therefore, we also needed to execute these commits individually on Travis-CI to check if they still reported the same build status. An Uber commit could only be reproduced successfully if all the individual commits reported the same build status on the simulation and our Travis-CI environment. 27 of the Uber commits could not be reproduced because either the original builds reported temporal infrastructure errors or the new commits could not find all the necessary dependencies to work. For those Uber commits that could be reproduced successfully, 89% of the Uber commits reported the same results as the simulation, 1.36% (1 Uber commit) reported integration issues, two commits that passed when run individually report failure as part of an Uber commit. Finally, 9.58% reported fixes, i.e. when two commits are part of an Uber commit and one introduces a fault but the other provides the fix, that the simulation is not able to detect.

Once we completed this experiment, we wanted to know if the Uber approach would still provide significant gains if only the 89% of the attempts to create an Uber commit were successful. We adapted the simulator to make the Uber commit creation process to fail following a probability of failure provided as parameter. We run a new and more realistic experiment comparing the traditional approach with the Uber approach with bounded culprit analysis. In this case only the 89% of the attempts to create an Uber commit could succeed. Figures 6.15 and 6.16 show the Uber approach performing better than the traditional approach but, as expected, gains are lower than the one obtained the Uber approach with 100% success probability.

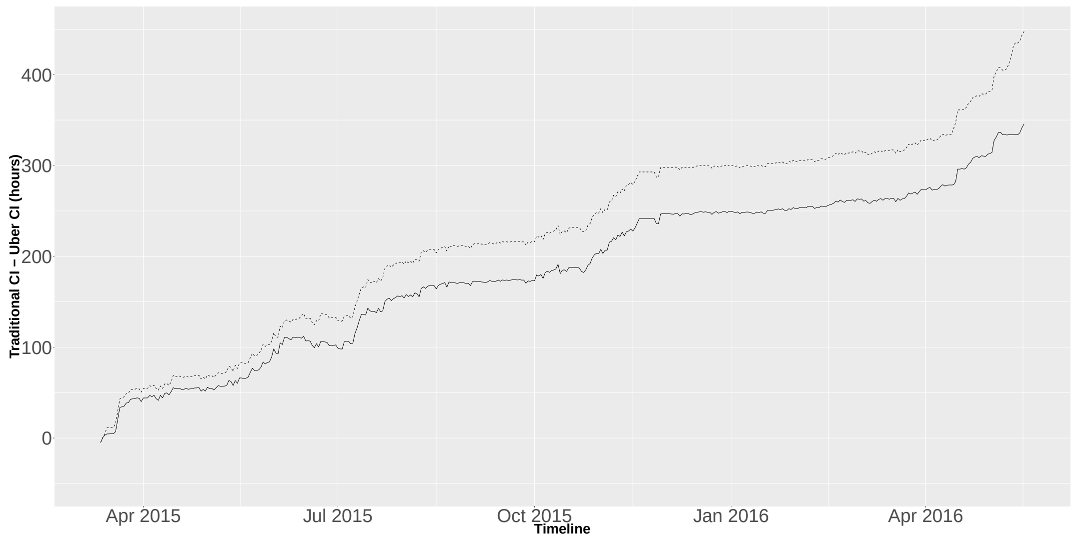

Figure 6.15: Server execution cumulative time difference between traditional and Uber approaches for Joomla project. The solid and dashed lines show the values for the Uber approach with bounded culprit analysis with 89% and 100% of Uber commit success probability.

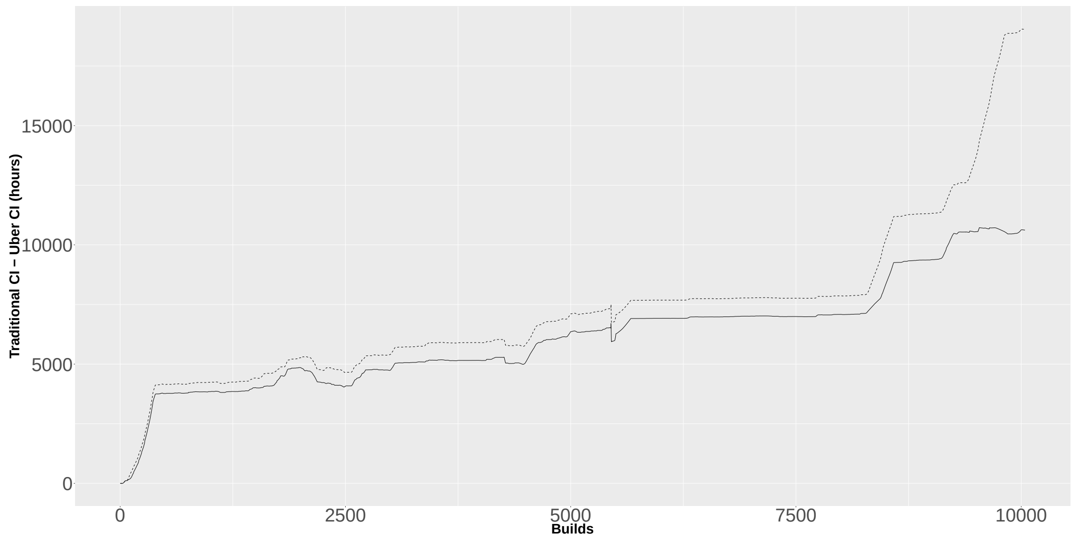

Figure 6.16: Cumulative Feedback time difference between traditional and Uber approaches applying the bounded culprit analysis for Joomla project with 89% (solid line) and 100% (dashed line) of Uber commit success probability.

Chapter 7

Applicability of the approach

7.1 Study Implications

Uber commits were present on all the simulations involving between 50% to 90% of the commits under study. This suggests we can expect to generate Uber commits on other projects with similar characteristics (high commit activity, low failure rate). RQ2 showed that when the conditions are given it can produce significant improvements (reductions of 30% on feedback time and 11% on resource utilization). However, this approach cannot be applied blindly because doing it under the wrong conditions will result on overhead. This behavior was seen on every simulation, not only Rails simulations. More work is required on the culprit analysis in order to break the high failure rate barrier. This was the goal of the unbounded analysis version but only worked 10% of the time. Note localized failure rate is the real problem. 42.3% of the Uber commits failed on the simulation for Code.org and a reduction of 20% on feedback time was still obtained. The Uber approach was only negatively affected on the periods where faults were concentrated.

This highlights another direction further work should focus, the selection process.

The results show the Uber approach with bounded culprit analysis can be effective as it is but it could be even more if the selection process were able to avoid candidates that are likely to introduce overhead. This could be achieved by collecting historical data but also exploring other simple heuristics for this process. For example, one idea inspired by Joomla results that reports small Uber commits but achieved the best gains is limiting the size of Uber commits. It is clear that more redundancies are likely to be reduced when more candidates are considered to be part of an Uber commit, assuming they are compatible, however, there are also more risks to find integration conflicts or failures as well.

Finally, the experiment to reproduce Uber commits on Travis-CI reported that Uber commits are slightly less frequent on this environment, only 89% of the Uber commits reported the same results as in the simulation. However, a new simulation was carried out on Section 6.3 to simulate an Uber approach on which only the 89% of the time Uber commits are created successfully to measure if gains could still be achieved with less Uber commits. This simulation shows a good performance of the Uber approach as well.

7.2 Challenges

In addition to the issues highlighted on the previous section, there will be challenges when trying to implement this approach on a real CI environment. They can be divided into two categories, technical and social. The first category includes the challenges related to combining this approach with other techniques set in place to improve the environment and the stack of technologies that compose it. The first challenge will be adapting the culprit analysis to work with integration issues since the implementation used on the simulations is not aware of them. Similarly, commits on the environment may not be as homogeneous as assumed on the simulations, i.e. the redundancies between them may be small or their processing time may vary significantly. One possible approach to deal with these issues is to apply a cheap analysis to detect commits that are likely to be problematic and exclude them as candidates. Other approach may allow the user to interact with Uber commits or the candidates. For instance, candidates can be reviewed by developers before executing the Uber commit and this one can also be reviewed if it fails before applying the culprit analysis. These approaches may not be feasible on large environments since they may add more overhead to the process but may be useful if they could be performed faster and with better accuracy than the our selection and culprit analysis implementations. The other technical challenge is how to combine the Uber approach with other techniques such as parallelization. In this case the server needs to decide how to make good use of all the techniques and resources. However, making this decision should not introduce more overhead than the savings obtained.

Among the social challenges, we tried to make the approach as transparent as possible to the developers but aspects of the environment will change and developers may react to them in different ways. First, if feedback time and resource utilization are reduced as result of applying Uber commits, developers may change their workflow since the may have room for extra work. Second, the developers will have to accept that results for some commits could get delayed with the Uber approach in order to improve the overall system performance. For example, the commit at the head of the queue does not benefit from being part of an Uber commit but doing it so can reduce the server utilization and feedback time of other commits in the queue. In fact, it is likely this commit have been benefited from previous Uber commits. Finally, users may not want this process to be transparent because they may find other useful applications for Uber commits that were not considered in this work. For instance, they may believe the best approach to work with Uber commits is interacting with them manually. In other words, they could control when to trigger them and which commit candidates would be taking part of them. They could also inspect failing ones before applying the culprit analysis as mentioned before.

Chapter 8

Related work

To the best of our knowledge, only two other works describe a CI approach involving the merging of queued commits to improve the efficiency of the server. Lacoste presents the success story of implementing a CI environment [28]. A few ideas are described in the paper. One of them, similar to our approach, was to include queued commits on a single build before being built and tested. However, it is briefly explained in the paper and, even though the author presented a quantitative evidence of the improvements of achieved with the application of a CI environment, there is no individual analysis focusing on the applicability or savings achieved by this idea in particular. He presents a figure showing the number of integrations performed per day together with the number of errors and tests failures found on each day before and after implementing the CI environment. The figure shows a significant improvement but there is no analysis of the impact of each applied idea to the final results. The second work is Openstack's Zuul project [24], the gating system developed and used by the organization in its CI environments. This approach improves the efficiency of the server by parallelizing the execution of builds in the queue. Uber-commit-like elements are used for keeping the dependencies of commits with those in front of them in the queue. For instance, if three commits are pending on the queue, the server will executed them on three different job at the same time. The first job will exercise a copy of the master branch integrated with the first commit, the second job will work on a copy of the master branch integrated with the first two commits and the third job will exercise the copy with three commits. The three commits are merged if the three jobs are successful. If the first job fails, the other two are re-executed without the changes introduced by this commit. The Uber approach, in comparison, would execute a single job with the three commits together and the other two would be executed only if the culprit analysis requires them. Similar to [28], the approach is not explained formally and no quantitative analysis is provided to understand the benefits over the traditional approach.

Other works have proposed improvements to CI environments but are not based on changing the server integration workflow. Many authors [3, 29, 30, 31, 32, 33] propose applying different types of builds with different frequency and target. With these approaches only the fastest tests are executed on every commit and the rest of the tests will be executed later with less frequency keeping a balance between executing expensive tests and providing fast feedback. Frequently these ideas come from industry consultants and do not provide a quantitative analysis of their benefit. Kawalerowicz [30] proposed four types of builds, one that is triggered on each commit and executes unit tests, a nightly build that executes unit tests and possibly some acceptance tests, a weekly build that executes unit tests and acceptance tests and a release build that performs a complete testing of the system. Modesto [32] takes ideas from Larman et al. [33] and proposes a software development process around CI. In this development process CI is applied at component, subsystem and product levels. Each successful build on one level triggers builds at the level on top of it. Tests at the lower level are fast but exercise a small part of the system and tests at the upper level are slow and exercise the whole system.

A few authors propose implementing local CI environments for particular teams or components [3, 13, 29, 30]. Related to this idea, Van der Storm [11] proposed triggering builds at the component level and executing only those that have been affected by the last change. In addition, a backtracking approach is applied to always provide working version of the system. This approach was evaluated in a period of 32 weeks and reported that the development speed decreased, the number of commit decreased one third, but the number of failing builds also decreased 43%. Dosinger et al. [16] propose a communication system between servers in order to improve the effectiveness of testing activities. Every time a build of a project is completed successfully, the CI server sends a notification to CI servers of projects depending on this one to validate if the introduced changes break the builds of these projects. The goal is to detect issues before a new version of the product is released.

Eyl et al. [12] propose an approach to reduce feedback time by saving binaries on the code repository so it can be used as a cache and triggering only the tests that have been affected by a change. Roberts also proposed, in [13], to use binary dependencies together with modularization to facilitate the development of related software components with independent life-cycles.

Popular CI tools [18, 34, 35] provide mechanism to parallelize build executions. Beaumont et al. [15], propose an approach to improve the scheduling of parallel builds to provide faster feedback. Gambi et al. [14] present an approach with a similar reasoning to ours, they propose to eliminate infrastructure redundancies in cloud-base CI.

Finally, the research community has presented several works focused on the individual tasks composing CI like Testing activities [9, 10] and Building Systems [26].

Improvements on individual areas are likely to improve the overall performance of the CI environment.

Chapter 9

Conclusion

We presented a novel approach to improve the efficiency of CI server based on changing the integration scale of the server that is transparent to the developers and can provide gains in the resource utilization and feedback time for developers. We carried out a study to assess this approach by simulating a real CI environment. We obtained promising results that suggest this approach could benefit real projects as it is but further studies are required to improve its applicability.

In terms of future work, this work can be extended in the following directions:

- Similar experiments should be carried out on new projects to analyze the performance of this approach under different toolsets, development processes and types of software.

- New experiments should consider the combination of this approach with standard optimization approaches, such as parallelization and test selection. We believe further improvements could be achieved.

- Improvements to f_c and ca should be studied to reduce the overhead introduced on the worst case scenarios. For instance, an f_c implementation could learn from failure history to determine if an Uber commit would incur into an expensive culprit analysis overhead and avoid the merge of commits.

- The simulator could be improved to make it more realistic in order to obtain more accurate results. In particular, it should be extended to consider different degrees of overlap of source and test code among commits.
- New experiments should explore the balance between size of Uber commits and the risk of incurring into integration issues.
- The long term goal is to have at least one implementation working on one or more real CI environments to study the actual benefits of the approach, detect the main challenges to bring this approach into practice and collect new ideas of how to extend the it.

Bibliography

[1] K. Beck, *Extreme programming explained: embrace change*. addison-wesley professional, 2000.

[2] M. Fowler and M. Foemmel, "Continuous integration," *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf*, p. 122, 2006.

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

[4] https://www.youtube.com/watch?v=dxk8b9rSKOo.

[5] P. Gupta, M. Ivey, and P. J., "Testing at the speed and scale of google," http://googletesting.blogspot.com/2011/06/testing-at-speed-and-scale-of-google.html.

[6] D. Feitelson, E. Frachtenberg, and K. Beck, "Development and deployment at facebook," *IEEE Internet Computing*, vol. 17(4), pp. 8–17, 2013.

[7] http://www.libreoffice.org.

[8] J. Micco, "Continuous integration at google scale," http://eclipsecon.org/2013/sites/eclipsecon.org.2013/files/2013-03-24%20Continuous%20Integration%20at%20Google%20Scale.pdf, 2013.

[9] 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.* ACM, 2014, pp. 235–245.

[10] E. Knauss, M. Staron, W. Meding, O. Söder, A. Nilsson, and M. Castell, "Supporting continuous integration by code-churn based test selection," in *Proceedings of the Second International Workshop on Rapid Continuous Software Engineering.* IEEE Press, 2015, pp. 19–25.

[11] T. Van Der Storm, "Backtracking incremental continuous integration," in *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on.* IEEE, 2008, pp. 233–242.

[12] M. Eyl, C. Reichmann, and K. Müller-Glaser, "Fast feedback from automated tests executed with the product build," in *Software Quality. The Future of Systems-and Software Development.* Springer, 2016, pp. 199–210.

[13] M. Roberts, "Enterprise continuous integration using binary dependencies," in *International Conference on Extreme Programming and Agile Processes in Software Engineering.* Springer, 2004, pp. 194–201.

[14] A. Gambi, Z. Rostyslav, and S. Dustdar, "Poster: Improving cloud-based continuous integration environments," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2. IEEE, 2015, pp. 797–798.

[15] O. Beaumont, N. Bonichon, L. Courtès, E. Dolstra, and X. Hanin, "Mixed data-parallel scheduling for distributed continuous integration," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International.* IEEE, 2012, pp. 91–98.

[16] S. Dösinger, R. Mordinyi, and S. Biffl, "Communicating continuous integration servers for increasing effectiveness of automated testing," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2012, pp. 374–377.

[17] "Travis-ci," https://travis-ci.org.

[18] "Jenkins," https://jenkins.io/.

[19] "Circleci," https://circleci.com/.

[20] https://github.com/.

[21] D. Ståhl and J. Bosch, "Modeling continuous integration practice differences in industry software development," *Journal of Systems and Software*, vol. 87, pp. 48–59, 2014.

[22] A. Miller, "A hundred days of continuous integration," in *Agile, 2008. AGILE'08. Conference.* IEEE, 2008, pp. 289–293.

[23] B. Adams and S. McIntosh, "Modern release engineering in a nutshell–why researchers should care," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 5. IEEE, 2016, pp. 78–90.

[24] "Openstack zuul documentation," http://docs.openstack.org/infra/zuul/gating.html.

[25] J. Rasmusson, "Long build trouble shooting guide," in *Conference on Extreme Programming and Agile Methods.* Springer, 2004, pp. 13–21.

[26] M. Vakilian, R. Sauciuc, J. D. Morgenthaler, and V. Mirrokni, "Automated decomposition of build targets," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 123–133.

[27] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[28] F. J. Lacoste, "Killing the gatekeeper: Introducing a continuous integration system," in *Agile Conference, 2009. AGILE'09.* IEEE, 2009, pp. 387–392.

[29] R. O. Rogers, "Scaling continuous integration," in *International Conference on Extreme Programming and Agile Processes in Software Engineering.* Springer, 2004, pp. 68–76.

[30] M. Kawalerowicz and C. Berntson, *Continuous integration in. NET.* Manning, 2011.

[31] J. Humble and D. Farley, *Continuous delivery: reliable software releases through build, test, and deployment automation.* Pearson Education, 2010.

[32] R. Modesto de Abreu, "Multi-stage continuous integration: Leveraging scalability on agile software development," 2013.

[33] C. Larman and B. Vodde, *Practices for scaling lean & Agile development: large, multisite, and offshore product development with large-scale scrum.* Pearson Education, 2010.

[34] "Teamcity," https://www.jetbrains.com/teamcity/.

[35] "Atlassian bamboo," https://www.atlassian.com/software/bamboo.