8-2017

Efficient And Flexible Continuous Integration Infrastructure to Improve Software Development Process

Daehyeok Mun
Utah State University

EFFICIENT AND FLEXIBLE CONTINUOUS INTEGRATION INFRASTRUCTURE TO IMPROVE SOFTWARE DEVELOPMENT PROCESS

by

Daehyeok Mun

A report submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Science

Approved:

Young-Woo Kwon, Ph.D. Major Professor	Stephen Clyde, Ph.D. Committee Member

Tung Nguyen, Ph.D.
Committee Member

UTAH STATE UNIVERSITY
Logan, Utah

2017

ABSTRACT

Efficient And Flexible Continuous Integration Infrastructure to Improve Software Development Process

by

Daehyeok Mun, Master of Science
Utah State University, 2017

Major Professor: Young-Woo Kwon, Ph.D.
Department: Computer Science

Continuous Integration (CI) is a proven software-engineering practice that can solve several issues occurring when a programmer integrates code changes into a shared source repository. Although extensive academic research efforts and implementations have been made in the industry or academia, due to the complexity of software execution environments and limited resources of CI infrastructures, CI services need further improvements. The aim of this thesis is to increase the utility and efficiency of a CI framework by adding new features as well as enhancing existing CI frameworks. First, we introduce a configuration method that can dynamically compose software execution conditions to reflect real software execution environments. Then, we present resource-usage profiling techniques that can provide statistical data collected during testing, so that a programmer can easily understand how his or her code changes impacted on the the current version of a program. Finally, we present a scheduling algorithm that can efficiently utilize limited resources of a CI infrastructure. Our experiments indicate that the proposed approaches can help programmer discover potential problems of a program and understand the impact of code changes as well as increase the efficiency of the overall CI service.

(55 pages)

PUBLIC ABSTRACT

Efficient And Flexible Continuous Integration Infrastructure to Improve Software Development Process

Daehyeok Mun

Continuous Integration (CI) is a popular software-engineering methodology for co-working between programmers. The key function of CI is to run, build and test tasks automatically when a programmer wants to share his or her code or implement a feature. The primary objectives of CI are to prevent growing integration problems, and to provide feedback with useful information for resolving these issues easily and quickly. Despite extensive academic research and popular services in the industry, such as TravisCI, CircleCI or Jenkins, there is practically have limitations, which result from limited available resources, including budget and low computing power. Moreover, the diversity of modern computer environments, such as different operating systems, libraries or disk sizes, memory, and network speeds, increase both the costs for CI and difficulties in finding bugs automatically in every cases.

This study aims to propose supplemental external and internal methods to solve the above obstacles. First, our approach enables programmers to configure different execution environments such as memory and network bandwidth during CI services. Then, we introduce an enhanced CI infrastructure that can efficiently schedule CI services based on resource-profiling techniques and our time-based scheduling algorithm, thereby reducing the overall CI time. Our case studies show that the proposed approaches can report the resource usage information after completing a CI service as well as improve the performance of an existing CI infrastructure.

CONTENTS

Page

ABSTRACT iii

PUBLIC ABSTRACT iv

LIST OF TABLES vii

LIST OF FIGURES viii

1 INTRODUCTION 1

2 BACKGROUND 3
- 2.1 Continuous Integration and workflow development 3
- 2.2 Approach overview 4

3 DYNAMICALLY GENERATED EXECUTION ENVIRONMENTS 6
- 3.1 Introduction 6
- 3.2 Background and motivation 7
  - 3.2.1 Diversity on software execution environments 7
  - 3.2.2 Existing problem and gaps in CI 7
- 3.3 Approach 8
  - 3.3.1 Proposed design architecture overview 9
  - 3.3.2 Environments composer 10
  - 3.3.3 Environment manager 13
  - 3.3.4 Resource-usage profiler 13
- 3.4 Use case studies 14
  - 3.4.1 Android application 14
  - 3.4.2 Python script for reading file contents 16
- 3.5 Summary 19

4 DYNAMIC CI SCHEDULING 21
- 4.1 Introduction 21
- 4.2 Background and motivation 22
- 4.3 Approach 23
  - 4.3.1 Problematic situation 23
  - 4.3.2 Small commit 24
  - 4.3.3 Time-based dynamic scheduling overview 25
- 4.4 Continuous Integration simulation 26
  - 4.4.1 Comparison algorithms 26
  - 4.4.2 Experiment-scenario generator 27
- 4.5 Evaluation 30
  - 4.5.1 Experimental environment 30
  - 4.5.2 Baseline benchmarks 30

4.5.3 Experimental result 31
4.5.4 Summary 38
4.5.5 Threats to validity 38
4.6 Summary 39

5 RELATED WORK 40

6 CONCLUSION AND DISCUSSION 43

REFERENCES 45

LIST OF TABLES

Table Page

3.1 Example of buggy python code and desirable testing environment components 16

3.2 Number of cases with example 17

3.3 Number of cases for environment composing 18

3.4 Number of cases for Python 3.6 is combined with Windows (total 84) 19

4.1 CI task time for Git 23

4.2 Baseline Times (seconds) 30

4.3 Experiment results when CI frameworks are used by one project 31

4.4 CI task triggered time for first experiment 32

4.5 Experiment result when CI frameworks are shared by two different projects . 35

4.6 CI task triggered time for second experiment 36

LIST OF FIGURES

Figure Page

2.1 Workflow development with a CI tool 3

2.2 The overview of proposed CI framework 4

3.1 The approach overview 9

3.2 Architecture of CI framework 9

3.3 Build and test processes in container overview 14

3.4 The CI test with two different network conditions 15

3.5 Example of feedback with resource usage 16

4.1 The comparison of Hypervisor and Container 22

4.2 The CI task scheduling problem 24

4.3 CI task time for 30 recent PHP pull-request 24

4.4 Time range and mean value for the first experiment case 32

4.5 The CI experiment time line with PHP 33

4.6 The CI experiment time line with GIT 34

4.7 Time range and mean value for second experiment case 35

4.8 The CI experiment time line with PHP and GIT 37

CHAPTER 1
INTRODUCTION

In software engineering, CI has been used to refer to practices or tools that help programmers merge working copies into a shared mainline repository. One benefit of CI is integrating automatic build and test processes into a version-control system. When a programmer edits code, his or her changes are integrated a main branch or master after automatically building and testing. As a result, such a automated process enables programmers to find code conflicts or bugs at the early stage of software development, thereby improving their productivity and the quality of software.

A recent survey conducted by Hilton [1, 2] has demonstrated benefits of CI services in open-source and private projects. According to his research, open-source projects that use CI show the following three tendencies: 1) accepting pull-requests early, 2) releasing code twice as often, and 3) less concerning about breaking a build process compared to projects that do not utilize CI. The statistical results indicate that 70% of popular projects hosted at GitHub more extensively use CI services than other projects.

Even though the usefulness of CI has led to the growth of frameworks for in-house implementations and Software As A Service (SAAS) in the past few years, the demanding nature of CI tasks still remains a concern due to the limited capability of available resources and the diversity of software environments. The dependence of software and execution environments such as network bandwidths, memory size might affects to software's behaviors. Consequently, CI tasks should be progressed multiple times with various environments, since some bugs do not arise every time. However, testing with all possible cases for each commits is almost impossible because it might require high costs.

For these reasons, we address the following challenges faced when employing a CI service:

- Selecting and configuring the software-execution environment to run multiple CIs with different combinations in the limited resource pool.
- Improving the CI framework's performance, which is implemented with multiple servers to provide timely feedback to the developers, and increase the infrastructure's capability.

In this report, we introduce supplemental services and algorithms in terms of usability and efficiency. First, we present a configuration method to help programmers test their code changes under different execution environments. Then, to enhance the efficiency of a CI infrastructure, we design an algorithm that can fairly schedule Ci services by estimating the resource usage and the completion time of a CI service for the given CI service. The primary contributions of our work are as follows:

- We purpose configuration methods to increase the possibility of discovering potential problems while maintaining the infrastructure with limited resources. We also propose feedback with supplemental stats, which automatically collect resource usage during the test. This data will be helpful in revealing changes that affect the software's quality; however, this is difficult to detect without the developer's benchmark program.
- We present use cases which may discovering potential problems with our approaches but difficult with existing CI frameworks.
- Finally, we develop scheduling algorithms with execution-time prediction for CI infrastructures in the cloud system.

CHAPTER 2
BACKGROUND

2.1 Continuous Integration and workflow development

The term ‘CI’ was introduced by Grady [3], and has been used to refer to the software-engineering practice of merging all local change sets to a shared code repository. In terms of workflow practices, the CI task is started by multiple triggers. First, all submitted commits are built, and a test script is executed to verify the minimal quality of code; this test script provides useful information to a user prior to merging the code changes. Figure1 illustrates the workflow of this type with a version-control system and a CI tool. Then, the CI tool either periodically checks the last available commit or manually checks the last fixed commit before releasing a new version of the product. In this case, tasks should be triggered less frequently and more test units should be checked, which require more resources than tasks for each commits.

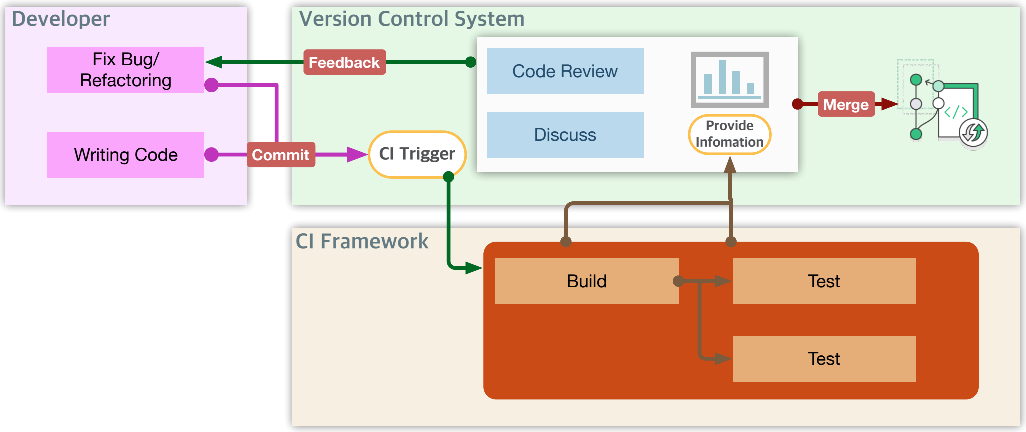


Fig. 2.1: Workflow development with a CI tool

Throughout this thesis, the term ‘CI’ will be used in its broadest sense to refer to a framework or infrastructure that automatically run build and test tasks for every commits.

2.2 Approach overview

To enhance the utility of CI, we suggest the following requirements based on previously mentioned assumptions:

- The CI framework listens to events from the version-control system, and automatically triggers tasks.
- The CI framework provides methods to flexibly configure build environments, including software dependencies and physical environments.
- The CI framework dynamically composes environment components to use limited resources efficiently.
- The CI infrastructure should allocate tasks efficiently into server clusters, which are constructed with multiple worker nodes.
- When the build or test run is completed, CI feeds results and other information back to developers.

The remaining part of this thesis describes several components in the CI framework to address the above requirements. Figure 2.2 provides a brief overview of our proposed CI framework and overall workflow.

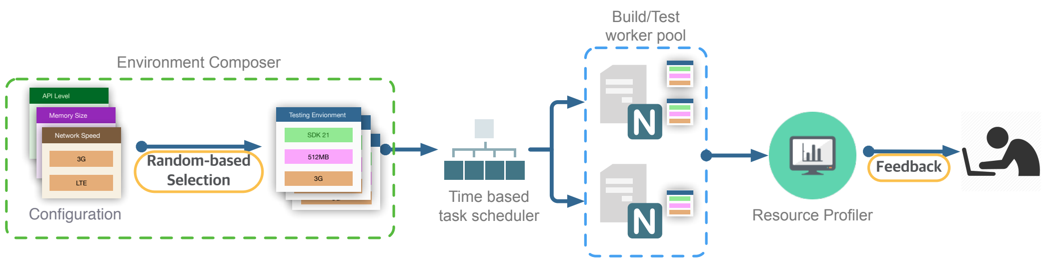


Fig. 2.2: The overview of proposed CI framework

For handling triggered tasks, CI frameworks compose build environments from a user's configuration file. Each task is deployed to one of the worker servers, and is then run with the composed environments combination. When the task is completed, CI provides task results and resource usages, which are collected by a profiler. in addition to conventional methods, to dynamically compose environment combinations, we present a random-based selection, which is described in chapter 3 along with resource profilers. For increasing internal efficiency, we argue the necessity of a scheduler, which specializes in CI tasks, and present time-based task schedulers in chapter 4.

CHAPTER 3
DYNAMICALLY GENERATED EXECUTION ENVIRONMENTS

3.1 Introduction

Automatic testing by CI services is used for improving the software-development process. Previous studies suggest that the use of CI often results in higher productivity, with a shorter release cycle and time to merge code changes [2, 4]. It has also been found that most of the costs of managing CI are spent configuring build matrixes (environments).

While the response regarding improved productivity with CI is positive, most users believe that CI does not help find bugs [2, 5]. One reason for this might be the diversity of execution environments. Various components of environments can lead to abnormal testing results. A shared object is a useful illustration of this problem. Software written with deprecated API will not work on other machines that use the newest version of the shared library. Moreover physical environments, such as hardware specifications and network conditions affect software behavior, for example, network-connection failures or timeout, and I/O errors. Therefore, CI services should provide features for running test applications in varied environments. However, the practical number of testing environments is limited by usable resources and time.

To address the above issues, this chapter describes new features that enable testing on the dynamically generated run-time environments and resource-usage profiling without a user's extra effort; we specifically argue that CI should also control physical conditions. Then we present a solution based on random selections for testing environment generation. To cover all of the components of environments with minimizing or reduced entries, CI provides different sets for each triggered job. In addition, CI yields the resource-usage data collected during the test process without other extra profiling tools.

We also evaluate the usefulness of the approach by applying it to the use cases, which indicate that the presented method provides more chances of finding bugs.

3.2 Background and motivation

3.2.1 Diversity on software execution environments

We divide the diversity of the software-execution environment into two groups,namely software diversity and physical environment diversity, and then briefly explain potential problems.

Software diversity Many applications rely on existing components in both the compilation (development) and execution phases. For example, a web application written by Python might use Flask[1] as a web framework and SQLAlchemy[2] for abstracting SQL code. Then, it might run with MySQL[3] or PostgreSQL[4] as a data store, and Apache[5] or NginX[6] as a web server, depending on the user's choice. A well-known problem called 'dependency hell' arises when different versions of the same package or library are required, but cannot work at the same time.

Physical environment diversity Modern computing devices have a wide range of computation and memory capabilities, disk sizes, and network bandwidths. This diversity may cause errors or abnormal behavior. For instance, an application that consumes too much disk space could experience I/O errors on limited disk devices. Even if the application does not throw an error directly, it could still have a potential problem, such as network-connection failure or testing timeout, depending on the physical conditions

3.2.2 Existing problem and gaps in CI

[1]`https://flask.pocoo.org`
[2]`https://www.sqlalchemy.org`
[3]`https://www.mysql.com`
[4]`https://www.postgresql.org`
[5]`https://httpd.apache.org`
[6]`https://www.nginx.com`

In this subsection, we argue that existing CI services cannot sufficiently cover the following problems or gaps:

Fixed physical environment While existing CIs already support software-execution environments to select or compose multiple options for build and test tasks, features for verifying software with the different physical environments is relatively ignored. Although some components, such as network bandwidth and maximum memory size, could be controlled dynamically by virtualization of a platform, most of the physical conditions are fixed to the service provider's setup or predefined framework's configuration.

A ruinous number of tasks Multiple tasks can be included in a single CI to test in various environments, and multiple projects or a development team can share the CI infrastructure. Thus, the workload for CI could be increased dynamically, for example, according to Google's presentation [6], 150 million tests are executed per day on their internal infrastructure, which might require unacceptable maintenance cost. Therefore, users should select envrionments that work on CI. However, most existing CIs do not support methods for dynamically composing environment components; instead, they perform hand-picked combination or generate all possible combinations.

Insufficient information on feedback Most current CI implementations focus on reducing user effort for constructing CI infrastructure, and providing configuration methods to automatically run the user's build or test script. Except for the success or failure of the script, the feedback content from CI is totally dependent on the output of the user's test application.

3.3 Approach

The presented approach aims to fill the gap described in the previous section, and to provide complementary features for utilizing CI with limited resources.

The following three features play key roles in our proposed CI framework design:

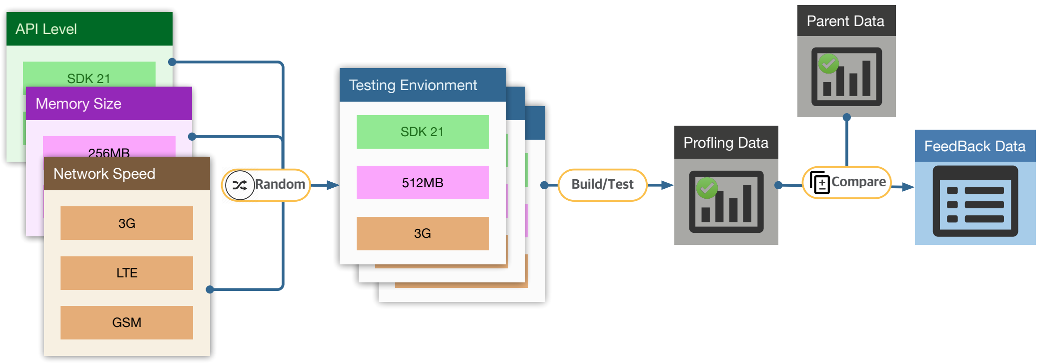


Fig. 3.1: The approach overview

1. Present CI feature to configuration physical environment like software component
2. Compose environment component based on random selection
3. Provide resource-usage data

3.3.1 Proposed design architecture overview

As with other CI frameworks, our design is constructed based on the master/slave model: the controller server manages the receiving and sending of data from the source repository, and the generating and running of a task for the worker; the worker's environment is generated by a virtualization platform (for example, Hypervisor, Docker, or an Android device emulator). Figure 3.2 provides an overview of the architecture for the proposed CI framework.

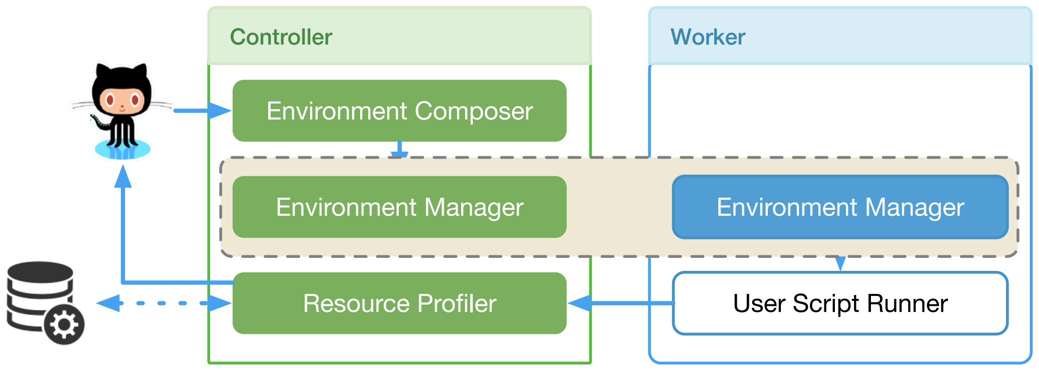


Fig. 3.2: Architecture of CI framework

The proposed framework has three key components: the environment composer, the environment manager, and the resource profiler. The environment composer selects the environment list for build and test tasks based on the user's configuration. In addition to fixed component composition, our composer also uses alternative methods to increase coverage in the limited resource pool. The environment manager initializes each component, including the configurable physical or software environment that the composer generates. The resource profiler profiles resources used for the test, and then provides that information to the user. In the following pages, we will describe how each component works.

3.3.2 Environments composer

Configuration file

Yet Another Markup Language (YAML) is a simple human-readable, data-serialization language that is widely used to define configurations for task including commands, virtual machine images, and environment variables in decent CI, such as TravisCI and DroneCI.

We use YAML file which contained in the code repository with other source files instead of other configuration methods such as external web UI or API due to following benefits:

- The build configuration can be shared with the source code, which helps developers to build and test more easily in local environments.
- Different configurations could be used depending on the branches or versions.
- As with other files, changes can be reviewed before they are merged to the shared code base, and can be reverted to undo negative change.

In this thesis, we modified DroneCI to experimental implementation, and added a 'shuffle' definition to the configuration syntax to support our random-based environment selection.

Listing 1 presents an example of CI configuration with YAML in which each CI task is composed of two parts: first, the worker is lunched with golang virtual machine image to build the process, then, the built binary is tested on the Ubuntu environment.

Three different GO_VERSIONs and two different RELEASE_TYPEs could be combined for each task. DroneCI already supports the ‘matrix’ syntax to automatically run with all possible combinations. In this case, CI tasks expand to six individual (3×2) jobs.

Listing 1 Example of a YAML configuration file

```
pipeline:
  build:
    image: golang:${GO_VERSION}
    commands:
      - go build

  test:
    image: ubuntu:16.04
    commands:
      - ./test.sh
    environment:
      - RELEASE_TYPE=${RELEASE_TYPE}

#Include all possible combinations.
matrix:
  go: [1.6, 1.7. 1.8]
  RELEASE_TYPE: [release, debug]

#Include combinations which generated by our approach.
shuffle:
  go: [1.6, 1.7. 1.8]
  RELEASE_TYPE: [release, debug]
```

In the following section, we describe how to compose the environments with the ‘shuffle’ syntax, and our approach.

Random-based environment selection

The presented algorithm 1 demonstrates our approach as a pseudo code. The role of this procedure is to return randomly composed environment sets, and the results are produced by the following operations:

1. Let component_dict be the associative array that has an environment’s name as a key, and list of options in that category as a value. Iterating all of the categories in

given component_dict (line 8).

2. If previously un-selected options still exist in the category, randomly pick from the un-selected options in that category, then move that option to selected_group (line 9-11).

3. If all of the options in the category are selected at least once, then randomly select from the selected group (line 13).

4. Append composed matrix to env_list (line 17), then repeat the above process until component_dict is empty

5. Return env_list, which contains all of the composed environment combinations

Algorithm 1 Random Testing Environment Selecting

```
Require: component_dict                                        ▷ associative array
 1: procedure randomSelector(component_dic)
 2:     env_list ← empty
 3:     keys ← {key values in component_dict}
 4:     selected_group ← empty
 5:     repeat
 6:         env ← Empty                                ▷ new environments composition
 7:         for all category ∈ keys do
 8:             if component_dict is not empty then
 9:                 $E_c$ ← Random(component_dict[category])
10:                 Delete(component_dict[category], $E_c$)
11:                 Append(selected_group[category], $E_c$)
12:             else
13:                 $E_c$ ← Random(selected_group[category])
14:             end if
15:             env[category] ← $E_c$
16:         end for
17:         Append(env_list, env)
18:     until component_dict is not empty
19:     return env_list
20: end procedure
```

Result sets produced by our approach have the following two characteristics: each component option is guaranteed to be included at least once, and the size of the result set is equal to the length of the longest list in the given component array.

3.3.3 Environment manager

Prior to running a build or testing process, CI should initialize the selected environment. To dynamically configure both the physical and software environments, the environment manager should work on both the controller and the inside of the virtualized working environment.

Environment manager in controller Manager in controller initializing physical environment and create virtual environment, Configurable options are might be various depend on used virtualization platform. Most of virtual machine's option's change is impossible, after boot up machine, This process should be processed by the controller. For example, if CI is working with Hypervisor, then CI cloud flexible limit or provide memory, number of core, disk size. On the other hand, CI run for mobile device with emulator, disk size also cloud be selectable.

Environment manager in worker This component have a charge to initializing software environments, including installing third-party software and setting up shell variables. Most software could be installed through a well-maintained package manager (for example, Advanced Package Tool[7], Chocolatey[8], Nuget[9]). Combination of OS and program also dynamically by our composing approaches. Using package manager, Eliminating running user-script for initalize envrionments is necessary.

3.3.4 Resource-usage profiler

This component aims to collect resource-usage data during testing, either without or with minimum supplemental configuration effort from a user. Most of the profiling tools for sampling statistics require at least two inputs indicated at the start and stop positions. As illustrated in Figure 3.3, when CI works with a virtualization platform for clean environments, each individual system is created and destroyed on demand.

[7]`https://wiki.debian.org/apt`
[8]`https://chocolatey.org`
[9]`https://www.nuget.org`

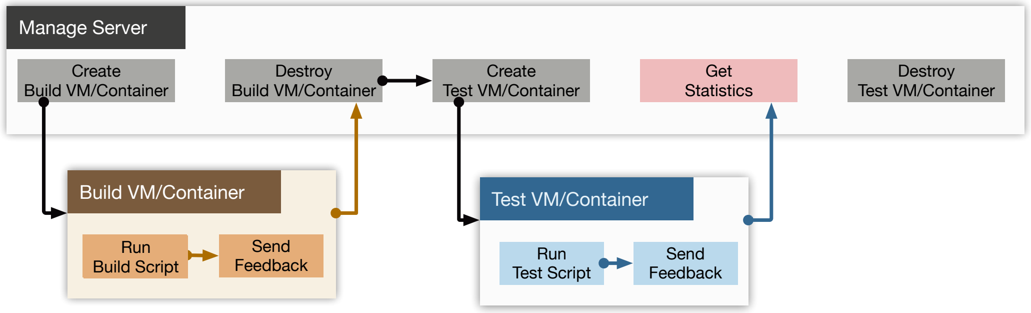


Fig. 3.3: Build and test processes in container overview

Under this situation, the life cycle of the virtual machine is almost identical to that of the testing process. The quantity of resources related to testing could be simplified to the total amount of resources used in a testing machine. Most virtualization platforms or operating systems provide API or other methods for statistical data used by the system, for example, Docker API, perf for Linux, and dumpsys command for an Android emulator or device. Therefore, the data can be collected in a unified way with little overhead, regardless of the user's build/test script.

3.4 Use case studies

This section demonstrates a use-case scenario to explain the necessity of the features described in the previous sections. For the experimental use case, we modified Drone CI[10] to provide our proposed features, and used Github as a source-code repository.

3.4.1 Android application

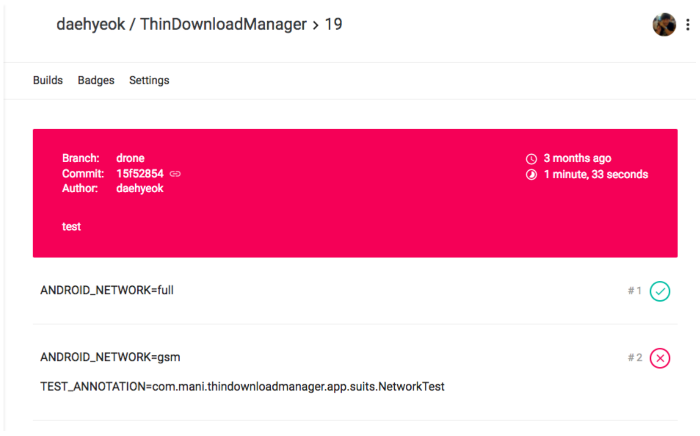
Initially, we developed an Android application that downloaded files from a given URL using HTTP/1.1 [7].

Testing with two different network conditions

First, for arguing the necessity of flexible physical environment testing in CI, we wrote a UI testing code that clicks the button on the main screen, waits until a file download is

[10]`https://github.com/drone/drone`

complete, then configures a CI run-test script with a full-speed network or GSM. Due to the low network bandwidth, a test with GSM (Global System for Mobile Communications) displays a timeout error. This problem might not be a bug; however, it could affect the user's experience, and should be closely inspected.



Branch: drone
Commit: 15f52854
Author: daehyeok

test

3 months ago
1 minute, 33 seconds

ANDROID_NETWORK=full

ANDROID_NETWORK=gsm
TEST_ANNOTATION=com.mani.thindownloadmanager.app.suits.NetworkTest

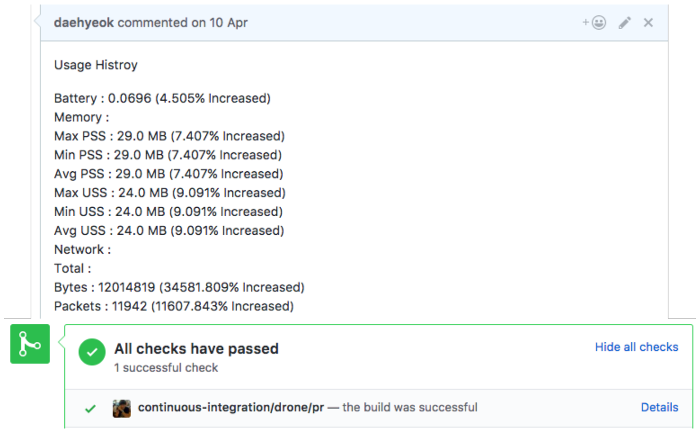
Fig. 3.4: The CI test with two different network conditions

Refactoring pull-request with resource usage

One of the common refactoring patterns involves changing a sub module to a third-party library, which is considered well designed.

Let us suppose the following scenario: The developer decides to use the third-party library instead of his own implementation with the native URLConnection API. One of the contributors made a pull-request for the above refactoring, and when his code change triggers CI tasks, the built binary successfully passes all of the user's tests. Then code readability is also better than the previous version, In the view of maintainer, no other reasons exist to deny this pull request.

With the proposed resource profiling approaches, perhaps CI could provide clues that maintainers should inspect before merging commit in above case. For example, used modules do not support an HTTP-compressed header, which could lead to CI indicating that a changed code uses more memory and network traffic. If CI reports that less than 10%



daehyeok commented on 10 Apr

```
Usage Histroy

Battery : 0.0696 (4.505% Increased)
Memory :
Max PSS : 29.0 MB (7.407% Increased)
Min PSS : 29.0 MB (7.407% Increased)
Avg PSS : 29.0 MB (7.407% Increased)
Max USS : 24.0 MB (9.091% Increased)
Min USS : 24.0 MB (9.091% Increased)
Avg USS : 24.0 MB (9.091% Increased)
Network :
Total :
Bytes : 12014819 (34581.809% Increased)
Packets : 11942 (11607.843% Increased)
```

All checks have passed
1 successful check

continuous-integration/drone/pr — the build was successful

Fig. 3.5: Example of feedback with resource usage

increased memory usages. When considered with the advantage of improved code readability, this changed amount might be ignorable. However, the traffic increase was 30 times more than the base version. The traffic change amount might be a significant factor which affects the overall quality of mobile software.

3.4.2 Python script for reading file contents

The newest version of Python demonstrates the effectiveness of the random-based environment composer. Table 3.1 contains examples of Python code and CI configuration.

```python
import platform

# MBCS (EUC-KR) encoded file path
filepath = bytearray.fromhex("c6 c4 c0 cf 2e 74 78 74")

if platform.system() == 'Windows':
    contents = open(filepath, 'r').read()
else:
    contents = open(filepath.decode("cp949"), 'r').read()
```

```
shuffle:
    Python: [2.7, 3.5, 3.6]
    OS: [Ubuntu, Windows]
    DB: [PostgreSQL, MySQL, Redis, MongoDB]
```

Table 3.1: Example of buggy python code and desirable testing environment components

Given code chunks are written for reading content from the predefined file. Prior to calling the ‘open’ function, the application checks the type of operating system to determine whether it requires converting byte array to sting. When this code is executed on a non-Windows system, such as Linux, the file path is string decoded from an EUC-KR[11] byte array to the UTF-8 string. In contrast, an unconverted byte array is passed as an argument on Windows. MBCS (MultiByte Character Set) was adopted as the default file-system encoding in the previous Python. However, Python 3.6 was patched[12] to use UTF-8 for default Windows file-system encoding. As a result, it is working well, with the exception that the platform is composed with both Python 3.6 and Windows.

Table 3.2 compares three different environment-selection strategies with case-size testing when a user wants to support three different versions of Python, or four different databases under Windows or Linux. The total number of cases to test all possible combinations is $24(3 \times 2 \times 4 = 24)$. Even though testing on every possible environment is the best practice for verifying reliability, if this code is part of the largest size of the application, the user might want to reduce the number of CI jobs triggered by every commit. User-defined configurations produce fixed environments; however, unknown problematic environments are impossible to discover.

For each CI job, our random composing approach selects different environment combinations as much as a length of largest option list (database). Therefore, the size of the task is reduced from 24 times to four times.

	Number of Cases	Possibility of Fail
All Case	$3 \times 2 \times 4 = 24$	Always
Random Approach	4	58%(84/144)
Manually	Fixed by user	N/A

Table 3.2: Number of cases with example

For calculating the possibility of discovering problematic combinations, let $E_{Selected}$ be the list of all environments, where an environment is defined as a tuple of component. Therefore, we can represent $E_{Selected}$ like (Python, OS, DB).

[11] `https://tools.ietf.org/html/rfc1557#ref-EUC-KR`

[12] `https://www.python.org/dev/peps/pep-0529`

The database is not related to the above problem; we can simplify the environment as tutple of Python and OS. The number of all possible cases for $E_{Selected}$ with our approach is 144, and the number of the cases in which tuple (3.6, Windows) is included at least once is 84. Therefore, the possibility of discovering a bug is approximately 58%. A more detailed process to calculate possibility is described in the next sub section.

Compared with test all cases every time, the number of running tasks fell by nearly two fifths. However, the success rate only dropped by 42%.

Possibility of discovering bugs on example

We can represent the result of our random composer as a two-dimensional matrix in which each column presents a set of composed environments, and each row presents the category of components. The database is not related to our example problem, and to simplify calculations, we can ignore the DB row. When we named each column as a slot from 1 to 4, the function $f(category, n)$, which denotes the number of cases in each cell, can be measured as follows

$$f(category, n) = \begin{cases} size\_of\_category + 1 - n & \text{if } n \leq \text{size\_of\_category} \\ size\_of\_category & \text{if size\_of\_category} < n \end{cases} \tag{3.1}$$

Table 3.3 Table 3 presents the results of each cell, and the number of all possible cases $\prod_{\substack{c \in \{\text{Python, OS}\} \\ 1 \leq n \leq 4}} f(c, n)$ is 144.

	Slot 1	Slot 2	Slot 3	Slot 4
Python	3	2	1	3
OS	2	1	2	2

Table 3.3: Number of cases for environment composing

Python 3.6 must have occurred only once in slots 1 to 3, and might be re-selected in slot 4. Therefore, for finding a number of cases for Python 3.6 combined with Windows, we can divide the cases into the following two cases

Combined only once in slot 1 to 3 The number of cases for slot 4 is always five (all cases except Python 3.6 with Windows). The number of cases for slots 1 to 3 is $4 \times 5 \times 3 = 60$ cases.

Combined in slot 4 (regardless of the state of the other slots) $\prod_{\substack{c \in \{\text{Python, OS}\} \\ 1 \leq n \leq 3}} f(c, n)$ is 24

Table 3.4 illustrates the detailed process for calculation.

Finally, the possibility of discovering a bug using our approach in the example python problem is $84/144 = 7/12 \approx 58\%$.

	Slot 1	Slot 2	Slot 3	Slot 4
Python	1(3.6)	2	1	5
OS	1(Windows)	1	2	

(Python3.6, Windows) combined only in slot 1 (total 20)

	Slot 1	Slot 2	Slot 3	Slot 4
Python	2	1(3.6)	1	5
OS	1	1(Windows)	2	

(Python3.6, Windows) combined only in slot 2 (total 20)

	Slot 1	Slot 2	Slot 3	Slot 4
Python	2	1	1(3.6)	5
OS	2	1	1(Windows)	

(Python3.6, Windows) combined only in slot 3 (total 20)

	Slot 1	Slot 2	Slot 3	Slot 4
Python	3	2	1	1(3.6)
OS	2	1	2	1(Windows)

(Python3.6, Windows) combined in slot 4 (total 24)

Table 3.4: Number of cases for Python 3.6 is combined with Windows (total 84)

3.5 Summary

The purpose of this chapter was to design supplemental CI-service features to help developers to discover potential problems with the CI tasks. Our study was based on the following assumption: due to CI's high demand for maintenance costs, building and testing all possible conditions is almost impossible. Also, existing CI frameworks still do not provide enough configurable methods for physical environment and feedback. Therefore, we suggested a design for physical environments that is also included in CI's configurable options; then, the random-based component selector was presented to find bugs, which only occur during particular conditions. Finally, the delivered information collected by statistics data was suggested. However, these results may not be applicable to all situations. For instance, this feedback might only be useful when modified code affects most of the test units or significantly changes the software's behaviours. Additionally, although some environments cannot combine with each other, we ignored this to simplifying our research scope. Future studies are required to develop syntax for describing the complex association between each environment's components.

CHAPTER 4
DYNAMIC CI SCHEDULING

In the previous chapter, we described the reason and problem regarding the same testing tasks running in multiple environments. In this circumstance, it is clear that the effcient use of available resources in infrastructure is another key issue for CI. In this section, we propose a CI task-scheduling algorithm, which will be used to develop an effcient CI infrastructure.

4.1 Introduction

Virtualization techniques have achieved significant development, and are widely used in many computing domains. Continuous Integration is one area that has actively adopted virtualization to isolate and reuse build and test environments. In particular, modern CI tools are implemented using a container-based virtualization technique, which is more lightweight and easier to configure than system-based virtualization technologies. By using a container, CI tools can easily reproduce the same build and test environment, which makes CI services more popular than past years.

While there are several commercial CI services, many start-ups or small businesses still operate their CI services using open-source projects and public clouds or in-house services due to their operational costs. In this circumstance, multiple CI tasks run concurrently due to the limited computing resources of a CI infrastructure; therefore, the distributing method remains a major technical challenge to improving the overall performance of CI jobs.

In this chapter, we present a CI-scheduling algorithm that can dynamically allocate multiple CI tasks on different servers based on previous CI-execution results. To avoid or minimize resource competition between different CI tasks, this approach profiles their executions time to calculate remaining time of current tasks then dynamically allocates the next task to the best node. Our approach focused on implementing CI frameworks built on container technology. Frameworks should not manipulate the testing processing in the container because CI used containers and test scripts that were either predefined or provided by the user.

4.2 Background and motivation

For the CI to verify the code, it is important to providing different and isolated testing environments. The automated build and testing script should run in a clean environment, and start from scratch. To archive this goal, virtualization techniques have been widely used in past years. In particular, modern CI tools are implemented using a container-based virtualization technique. With the container, the CI can create an environment for software testing, and reuse it often for rapid iteration. Some of the popular container frameworks include Docker and rkt[1].

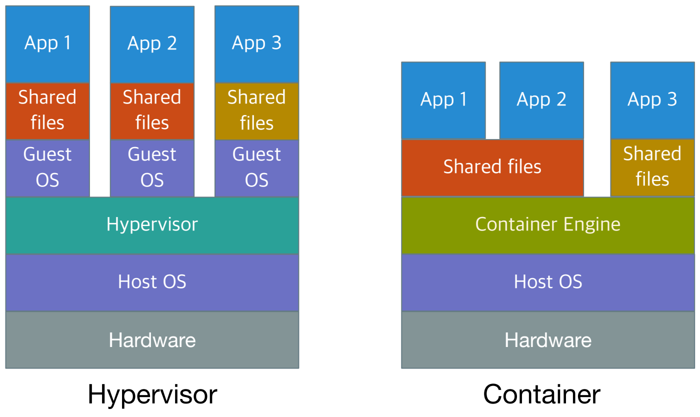


Fig. 4.1: The comparison of Hypervisor and Container

Figure 4.1 compares applications on container or Hypervisor technologies, and indicates that Container shares more common components than Hypervisor. Container consequently has less overheads and is easier to configure than system-level virtualization, also known as Hypervisor, including VMware and XEN. In the many previous researches [8], performance of container was almost the same with bare metal, and was quickly used to replace Hypervisor. However, a shared OS kernel throughout co-hosted containers contributes to the lack of

[1]`https://coreos.com/rkt`

isolation. As a result, applications on container may suffer from performance interference by other tenants. For many years, this phenomnon was neglected.

Table 4.1 presents the average times when CI container for GIT runs alone or with another container. Compared to with processing time when one GIT container monopolized node's resources. When two GIT containers shared a node, task time was 50% longer. In the worst case, if another CPU-intensive processing CI task (FFMpeg) was co-located in the node, the CI task times increased by almost two times.

Table 4.1: CI task time for Git

CI task	time (seconds)
1 CI task (Git)	494.4675
2 same CI tasks (Git/Git)	768.047
2 different CI tasks (Git/FFMpeg)	985.348

Therefore, it is clear that reducing interference is a key factor to improving overall CI performance. To date, however, there has been little discussion about minimizing the performance reduction that container interference causes in CI. In this thesis, we argue that the CI infrastructure could be improved when scheduling a new task to a node, with consideration for the above problem.

4.3 Approach

The primary objective of our approach was to reduce the amount of processing time in the CI infrastructure when it constructed with multiple servers. In this section, we first define the underlying problem in existing CI, and then describe our insights that led to improving performance with a time-based scheduler.

4.3.1 Problematic situation

Considering the CI scenario that is illustrated in Figure 4.2, CI frameworks try to deploy three different tasks to two nodes. When those tasks are triggered by chronological order and CI server using round-robin, Task 3 is allocated to the first node. However, if Task 2 has a shorter processing time than Task 1, the second node becomes a wasted resource.

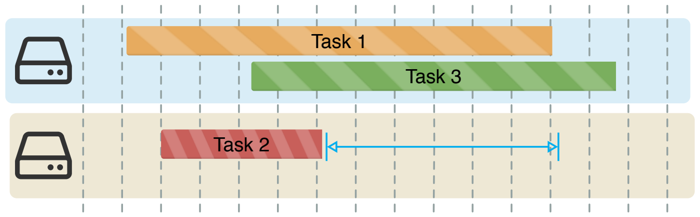


Fig. 4.2: The CI task scheduling problem

Also, as we described in section 4.2, it might increase the processing time of Tasks 1 and 3.

4.3.2 Small commit

In many software-engineering practices, developers are encouraged to commit often. For example, 'Extreme Programming' [9] which is one of the most popular software-engineering principles, proposes merging code into the code repository every few hours. Moreover, each commit should ideally have a single functional change. It allows the reviewer to check only related files for bug fixes instead of checking multiple potentially unrelated files. Based on this principle, we assume that build and test behaviors commonly do not changed dynamically. To verify this assumption, we build 30 recent pull-requests in the PHP-SRC repository.

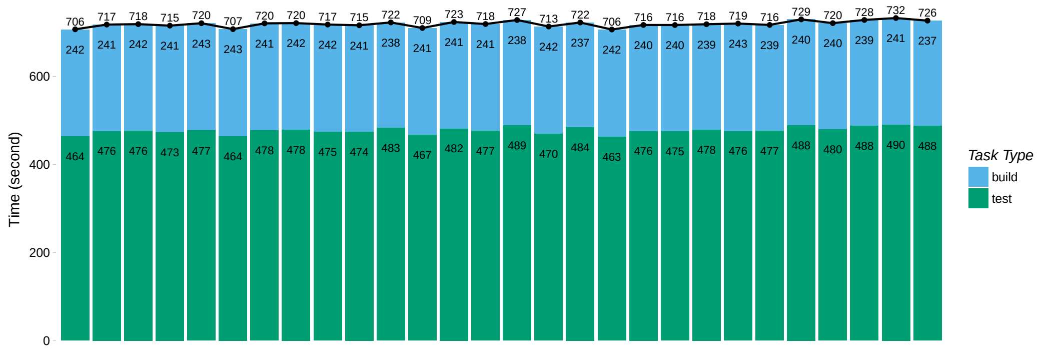


Fig. 4.3: CI task time for 30 recent PHP pull-request

Figure 4.3 illustrates CI task (build and test) time. From the chart, it can be seen that total processing time took from 706.27 seconds to 732.32 seconds, and averaged at

718.85 seconds. Therefore, even the largest time gap is less than 4% of the average time. As we expected, a significant processing time change is not found, and this result provides key insights for our approach. To improve performance, we use this characteristic of CI to predict the remaining time of the current task. In the following section, we present our scheduling algorithms.

4.3.3 Time-based dynamic scheduling overview

The main goal of the suggested scheduling approach is to dynamically schedule CI tasks while considering the capacity of each CI server. The major contribution of this chapter is that we present task-scheduling algorithms that select an appropriate CI server through the logged data. We named our proposed approach Time-Based Dynamic Scheduling (TDS). Algorithm 2 displays TDS in pseudo code.

Algorithm 2 Time-based Dynamic Selection (TDS)

```
 1: procedure TIME-BASED DYNAMIC SELECTION()
 2:     $min\_task\_count \leftarrow$ MIN($\{node.task\_count \mid node \in \{all\ nodes\}\}$)
 3:     if $min\_task\_count == TASK\_CAPACITY$ then            ▷ all servers are fulled
 4:         return $NULL$                                    ▷ wait until there is an available node
 5:     end if
 6:     $candidate\_group \leftarrow \{node \mid node.task\_count == min\_task\_count,\ node \in \{all\ node\}\}$
 7:     $candidated\_node,\ estimated\_time \leftarrow NULL,\ \infty$
 8:     $current\_time \leftarrow$ Now()
 9:     for all $node \in \{candidate\_group\}$ do
10:         for all $t \in \{all\ task\ in\ node\}$ do
11:             $avg\_time \leftarrow AverageTimes(t.project\_type)$     ▷ Previous average processed time
12:             $progessed\_time \leftarrow current\_time - t.start\_time$
13:             $remain\_time \leftarrow avg\_time - progessed\_time$
14:             if $remain\_time < estimated\_time$ then
15:                 $candidated\_node,\ estimated\_time \leftarrow node,\ remain\_time$
16:             end if
17:         end for
18:     end for
19:     return $candidated\_node$
20: end procedure
```

1. Initially, the TDS select nodes that have the least number of tasks. (line 2-6)

2. Iterating selected nodes and task in nodes, then the remaining time is calculated by the gap between the average time of previous jobs in the logged data (line 11) and progressed time until the current time (line 12).

3. Return node that have a shortest remaining time.

The basic idea behind this is to avoid resource competition by dynamically allocating CI tasks on different servers. If the competition is unavoidable, this algorithm provides an alternative that minimizes overlapped execution time by anticipating completion times. The anticipated completion time of a CI task on each server node can be computed using the profiling data obtained from previous tasks.

4.4 Continuous Integration simulation

To analyze the effects of our proposed approach and compare them with existing algorithms, we developed a simulator that mimics task triggers (for example, commit to source repository) and test scenario generator to more efficiently plan experiments in a small lab.

4.4.1 Comparison algorithms

For the comparison, we selected four different algorithms that were implemented in open-source tools such as Kubernetes or Docker Swarm.

Round Robin assigned each task to a node in circular order without any priority, except nodes that had reached limitation. Ideally, this algorithm was expected to demonstrate the best performance when one project utilizes CI because the scheduler minimizes overlapped task times.

Binpack places a new task on the most loaded node that has not reached limitation. If multiple nodes are included in the most-loaded group, unlike round-robin, the binpack algorithm selects the most recently used node. Ideally, this scheduler maximizes overlapped task times. Therefore, this algorithm was expected to demonstrate the worst performance.

Random selects a node from all of the available nodes without priority.

Spread places a new task on the least loaded node to spread tasks evenly across the nodes. If multiple nodes are included in the least loaded group, then this algorithm selects a node randomly from the group.

4.4.2 Experiment-scenario generator

To explore the different behaviors and effects of each algorithm, it was necessary to use a simulator instead of a real version-control server. The simulator allowed us to trigger tasks in the same chronological order, but with different CI algorithms to compare performance. Another benefit of this method is that we can experiment in a small testing lab. If every node is either empty or filled most of the time, regardless of the scheduling method, then most of the tasks either do not overlap or always overlap. To minimize this situation during the test, we generated testing scenarios in a probabilistic manner depending on a number of current nodes and tasks. Algorithm 3 illustrates how both the order of tasks and the wait time are generated prior to simulation. The test scenario generated the following order

1. Let task_deck be the list of all tasks which used to be the experiment. Initially, 10 tasks for every two projects are shuffled randomly (line 39-43).

2. CI simulator begins with the spread algorithm.

3. The function GENERATE is called, and then called again whenever any task in the CI infrastructure is completed. The role of GENERATE is to probabilistically decide whether to push the new task into the CI job que. The possibility this process depend on task_count which refers total number task in nodes or waiting in pushTask function (line 15-18). If task_deck is less than a number of nodes in infrastructure, then the new task is always pushed. However, if it is between the minimum and maximum total capacities of infrastructure, then the new task is pushed with a 40% success rate (line 24-33). This process is repeated when task_count is less than a number of nodes in infrastructure.

4. The function pushTask takes charge of pushing the new task to the CI job queue. Each task has an additional delay of between 1 and 150 seconds before being pushed. For replaying the generated task order with other algorithms, the elapsed times, from the moment the generator started, are saved in a history list (line 10).

5. The generator waits until all 20 tasks, which shuffled in the first step, are finished. Then, it returns a history list.

Algorithm 3 CI Test Scenario Generator

```
 1: $wating\_task \leftarrow 0$                              ▷ Global variable for counting sleeping task before pushed
 2: $task\_deck \leftarrow empty$                                          ▷ Global list for remained task
 3: $stat\_time \leftarrow$ NOW()                                                 ▷ Program start time
 4: $history \leftarrow empty$                                ▷ Global list to writing generated scenario

 5: function pushTask()                                                   ▷ Insert new task to CI que
 6:     $wating\_task \leftarrow wating\_task + 1$
 7:     $wait\_time \leftarrow$ RANDOM(1, 150)                           ▷ Random integer from 1 to 150
 8:     SLEEP($wait\_time$)
 9:     $t \leftarrow$ POP($task\_deck$)
10:     Append($history, (t, NOW() - start\_time)$)                                ▷ Log pushed time
11:     PUSH($CI\_TASK\_QUE, t$)
12:     $wating\_task \leftarrow wating\_task - 1$
13: end function

14: function generate()
15:     $task\_count \leftarrow watint\_task$
16:     for all $n \in \{all\ node\ in\ infrastructure\}$ do
17:         $task\_count \leftarrow task\_count + n.current\_task$
18:     end for
19:     $node\_count \leftarrow$ LENGTH(node)                            ▷ Number of node in infrastructure
20:     $maximum\_task \leftarrow node\_count \times$ TASK_CAPACITY      ▷ Maximum number of task to CI run currently
21:     if $node\_count <= maximum\_task$ OR EMPTY($task\_deck$)  then ▷ All nodes is fulled or all task is pushed
22:         return
23:     end if
24:     $barrier \leftarrow 40$
25:     if $task\_count <= node\_count$ then
26:         $barrier \leftarrow 0$
27:     else if $task\_count == maximum\_task - 1$ then
28:         $barrier \leftarrow 100$
29:     end if
30:     if $barrier <$ RANDOM(1, 100) then
31:         return
32:     end if
33:     pushTask()
34:     if $task\_count < node\_count$ then
35:         Generate()
36:     end if
37: end function

38: procedure GenerateScenario($project\_1, project\_2$)
39:     for $i \leftarrow 1, 10$ do
40:         Append($task\_deck, project\_1$)
41:         Append($task\_deck, project\_2$)
42:     end for
43:     SHUFFLE($task\_deck$)                                              ▷ randomly shuffle task_deck
44:     RUNCI($Spread$)                                       ▷ Run CI with spread scheduling algorithm
45:     Generate()
46:     repeat
47:         if $Any\ CI\ task\ is\ finished$ then
48:             Generate()
49:         end if
50:     until All CI task is done
51:     return $history$
52: end procedure
```

4.5 Evaluation

4.5.1 Experimental environment

The experimental setup is comprised of one CI-simulation server and three CI servers. Each CI server is compounded of an Amazon Web EC2 c4.xlarge instances with 16GB SSD, Ubuntu 16.04 64bits, and Docker 17.03.1-ce. Each node is limited to working with a maximum of two concurrent tasks. We used two open-source projects, Git and PHP, for our experiment. Both projects were hosted on Github and integrated with Travis CI, through their code, and the configuration for CI and build status were publically accessible. In the Travis CI, the PHP project currently runs two different build environments, and GIT works with seven different configurations. To more closely mimic our experiment to the real-world development process, we selected and followed one build condition from each project's CI configuration.

4.5.2 Baseline benchmarks

First, to investigate the best and worst cases of each CI task and base information, we manually ran CI tasks with the following benchmark cases:

- Executing one CI task on a CI server
- Executing the same two CI tasks concurrently on a CI server

Table 4.2 summarizes the statistics for the above scenarios.

		Mean	Standard deviation	Min	Max
GIT	1 CI task	303.4	2.718	302	309
	2 CI task	507.4	6.328	492	514
PHP	1 CI task	880.6	12.65	866	893
	2 CI task	1137.6	21.12	1113	1172

Table 4.2: Baseline Times (seconds)

As indicated in Table 4.2, the task time of GIT is less than half that of PHP. However, comparing the two projects, GIT received more interference from each other. When we ran two tasks concurrently, GIT increased by 204 seconds (approximately 60%); however, PHP only increased by approximately 26% (257 seconds). One reason for this difference may be that GIT uses more CPU and memory than PHP. In summary, regardless of their processing time, the depth of interference by other CI containers can vary depending on the project.

4.5.3 Experimental result

For the purpose of analysing our approach, we compared our algorithm in two different test cases. In both cases, the first step for the experiment is to generate test scenarios, followed by algorithm 3 with selected one or two projects. Then we ran four different algorithms include three comparison methods with the same scenario.

Case 1: When CI frameworks are used by only one project

First, we set up the CI simulator to work with PHP and GIT separately. Table 4.5 and Figure 4.7 compare the statistical results for these two experiments, and the detailed CI task-process times and how to allocate the tasks to each strategy for each CI experiment are illustrated in Figures 4.5 and 4.6.

		Execution Time (seconds)			
		Mean	Standard deviation	Min	Max
GIT	BINPACK	574.45	130.149	342	836
	RANDOM	491.6	82.703	337	570
	ROUND ROBIN	473.9	78.870	336	558
	SPREAD	504.2	78.532	341	571
	TDS	**474.65**	**79.457**	**337**	**558**
PHP	BINPACK	1102.3	84.591	911	1175
	RANDOM	1098.3	84.888	907	1173
	ROUND ROBIN	1088.7	93.005	898	1184
	SPREAD	1099.6	87.535	902	1180
	TDS	**1085.7**	**89.282**	**908**	**1178**

Table 4.3: Experiment results when CI frameworks are used by one project

All of the tasks are expected to have the same processing time; therefore, ideally, the task that has the earliest start time is always expected to finish the earliest. As a result, our TDS approach demonstrates the same behavior with the round robin algorithm. In both the PHP and GIT cases, TDS and round robin displayed the best result, while BINPACK always

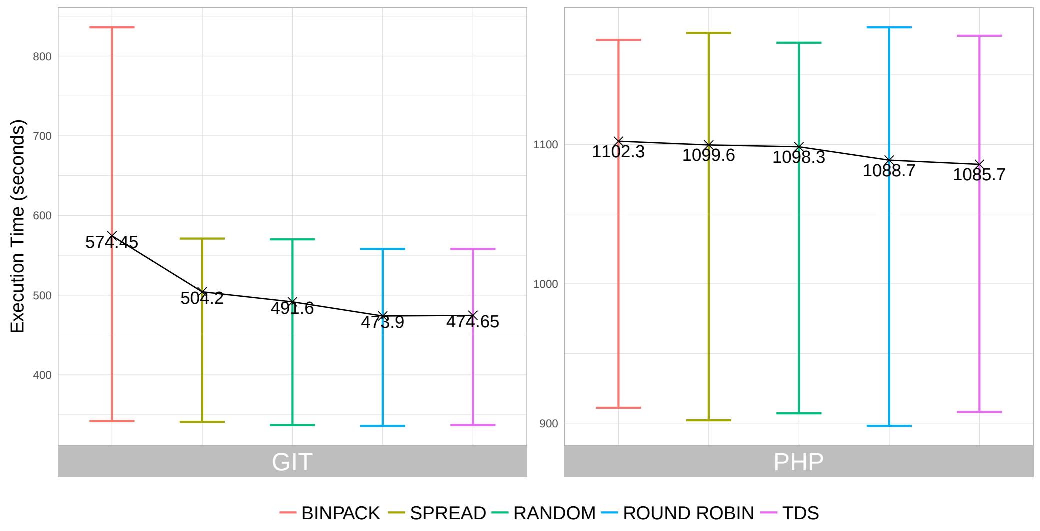


Fig. 4.4: Time range and mean value for the first experiment case

presented the worst result, which was maximized overlap time. Similarly, with the baseline benchmark, GIT demonstrated more of a difference than PHP. When comparing GIT's and BINPACK's mean times, GIT received a benefit of approximately 18%, whereas PHP only received a 2% benefit. The range of progress times with PHP do not vary significantly; however, in contrast, the progress time of BINPACK with GIT are much higher than the other four strategies.

PHP Triggered Time (seconds)	1	35	52	76	213	965	1152	1262	1269	1398
	2254	2286	2302	2304	2340	3408	3453	3460	3479	3527
GIT Triggered Time (seconds)	1	62	104	157	431	473	480	481	635	882
	904	927	932	966	1038	1459	1495	1501	1502	1878

Table 4.4: CI task triggered time for first experiment

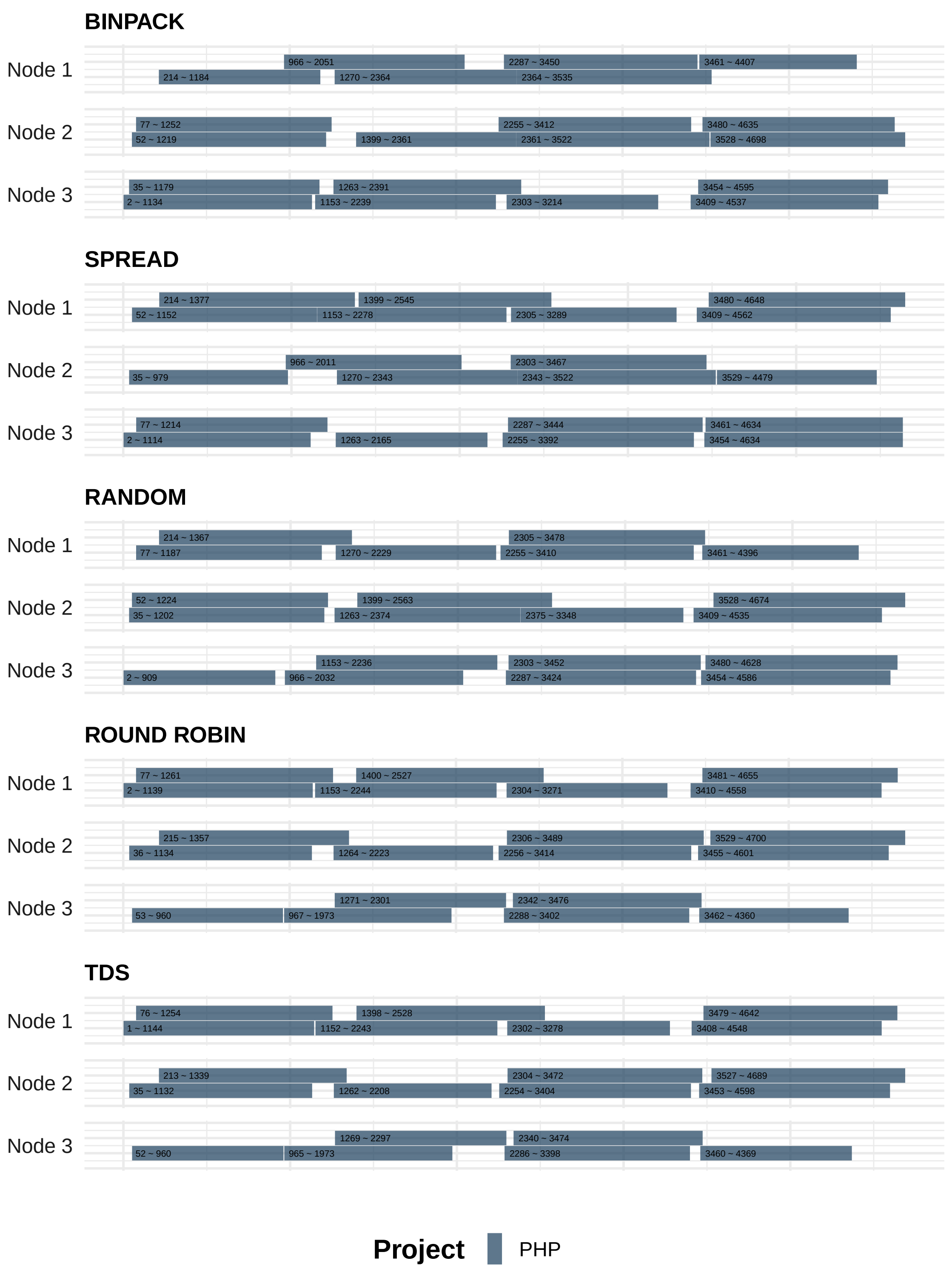


Fig. 4.5: The CI experiment time line with PHP

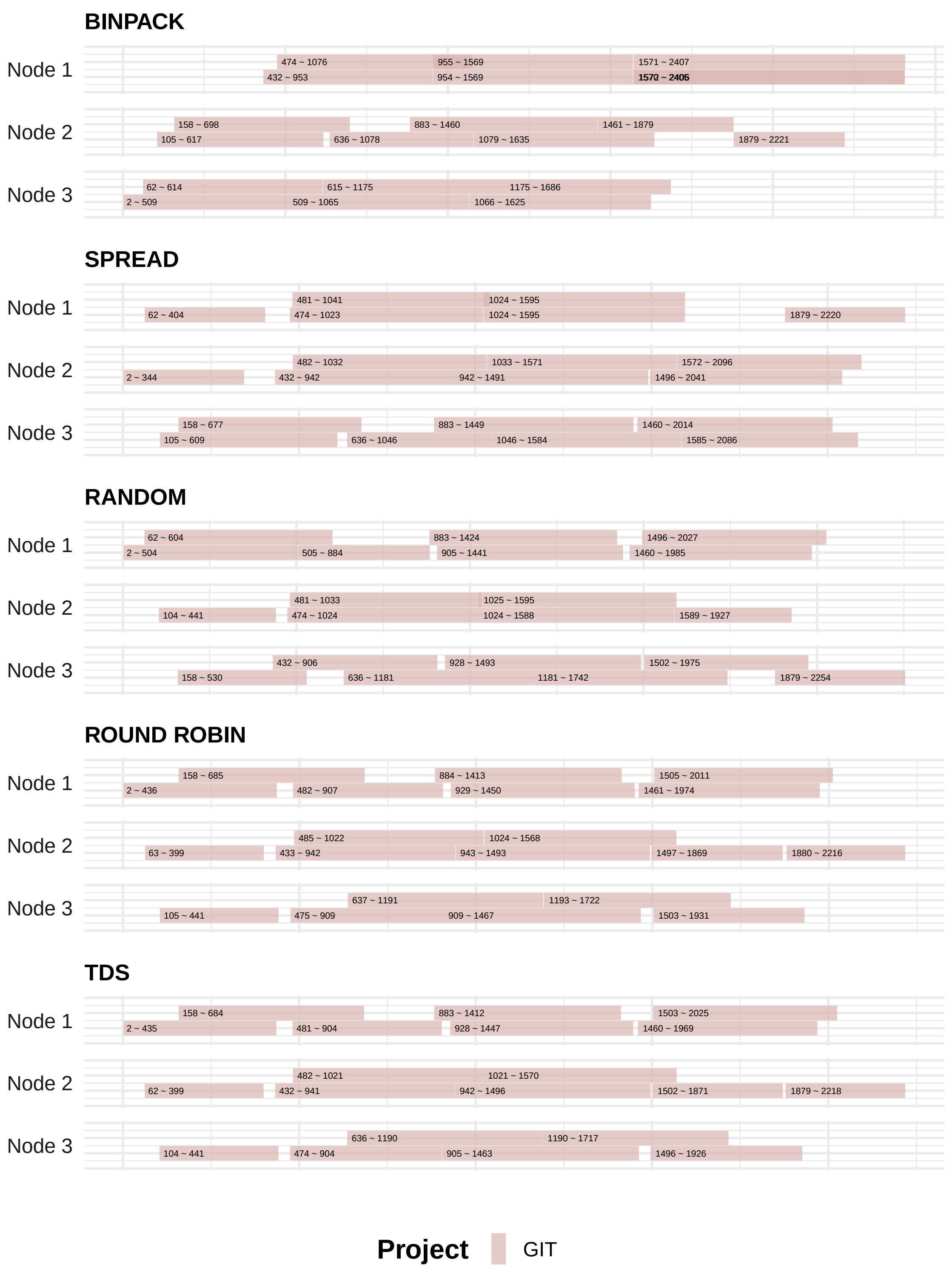


Fig. 4.6: The CI experiment time line with GIT

Case 2: When CI frameworks are shared by two different projects

Table 4.5 and Figure 4.7 provide the summary statics for the test results when the CI simulator works with PHP and GIT together.

	Execution Time (seconds)							
	Mean		Standard deviation		Min		Max	
	GIT	PHP	GIT	PHP	GIT	PHP	GIT	PHP
BINPACK	437.9	1139.8	55.277	84.842	359	984	515	1251
RANDOM	419.8	1141.8	55.599	95.926	337	1022	503	1295
SPREAD	446.1	1095.0	90.136	124.826	335	912	563	1306
ROUND ROBIN	429.8	1089.6	86.504	85.268	336	987	542	1240
TDS	**406.3**	**1092.8**	**51.549**	**90.445**	**336**	**900**	**493**	**1221**

Table 4.5: Experiment result when CI frameworks are shared by two different projects

Due to the problem that we presented in section 4.3.1, the round robin method no longer guarantee selection of the best node. In the GIT's results, both the random and TDS algorithms demonstrated shorter mean times than round robin. However, the random approach allocated resources unfairly; as a result, PHP still suffered resource competing.

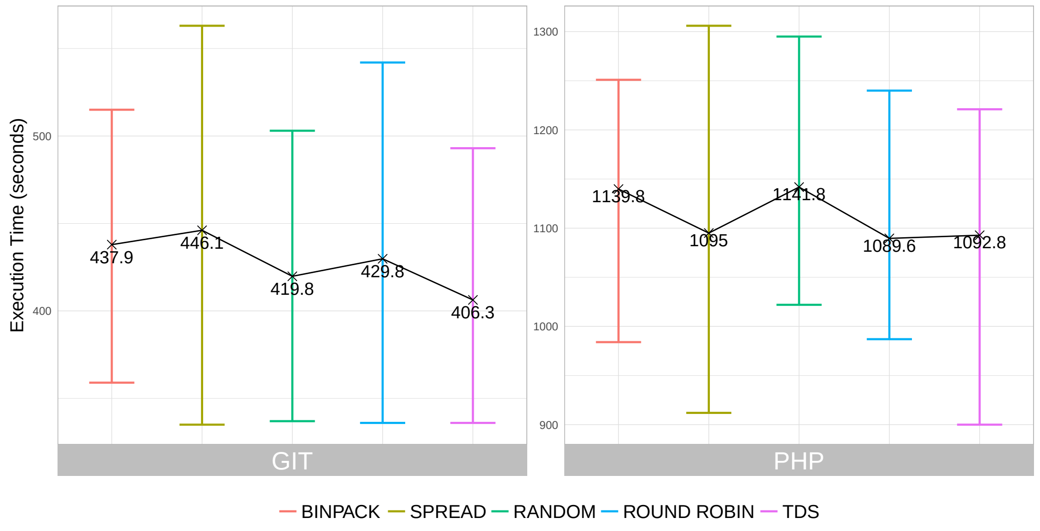


Fig. 4.7: Time range and mean value for second experiment case

Figure 4.7 verifies that even though the random strategy progressed GIT's task, which presented the second-best performance, it presented the worst performance with the PHP task. Our approach also presented slightly longer performances than round robin; however, it significantly reduced GIT's mean time – more so than other strategies. With both GIT and PHP, TDS's longest time did not exceed the longest time with other strategies. Detailed CI task-process times and how to allocate tasks to each strategy are illustrated in Figure 4.8.

Triggered Time (seconds)	2	10	51	294	359	460	899	905	952	1011
Project	PHP	GIT	GIT	PHP	PHP	GIT	GIT	PHP	GIT	PHP

Triggered Time (seconds)	1394	1427	1437	1516	1866	1891	1994	2382	2384	2699
Project	GIT	PHP	GIT	PHP	GIT	PHP	PHP	GIT	GIT	PHP

Table 4.6: CI task triggered time for second experiment

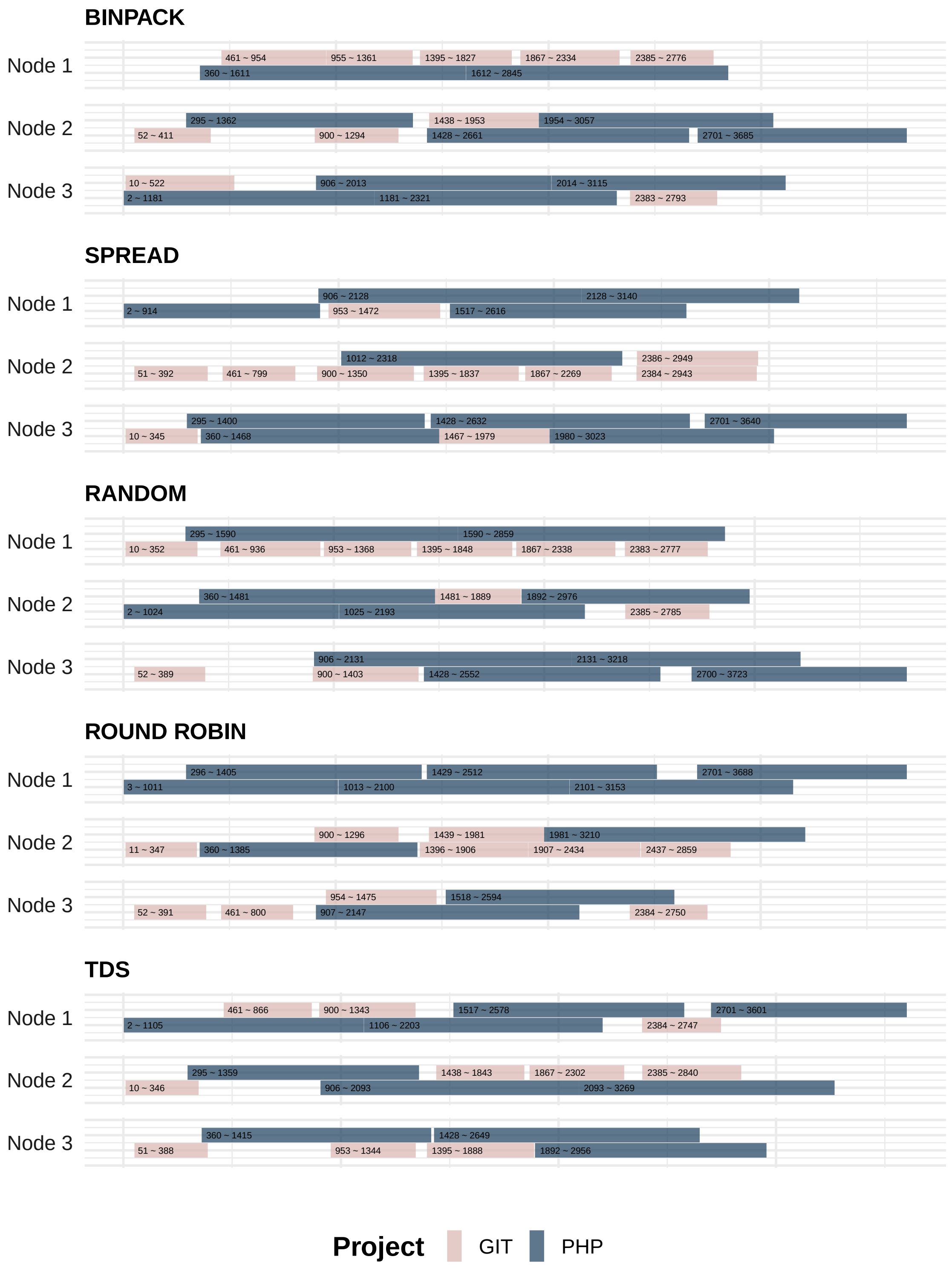


Fig. 4.8: The CI experiment time line with PHP and GIT

4.5.4 Summary

The results of this study indicate that the scale of interference from other cohosted projects is variable, and could affect performance of CI infrastructure. In our experiment with GIT and PHP, GIT's CI tasks presented more fluctuated process times than PHP. Performance decreases as overlap times increase; to avoid this, we presented time-based dynamic CI task-scheduling algorithms based on predicting remaining time, then evaluated our approach with four different algorithms that container-management tools utilize. In the first benchmark, we experimented with the same CI tasks. As described in Table 4.3, our approach, TDS, consistently demonstrated either the best or close to the best results. In the second benchmark, we experimented with different CI tasks (Git and PHP) simultaneously. As described in Table 4.5, unlike the other strategies, which demonstrated either the worst performances in both the project results or better performance in only one project, our approach presented a similar result to the first benchmark result. In summary, our approach does not guarantee the best result because the scheduling algorithm selects the most suitable CI server for the given CI task based on estimation.

4.5.5 Threats to validity

The experimental results are subject to both internal and external validity threats. Internal validity is threatened by the way in which the source repository for triggering CI task is simulated. Rather than using the real data in the source repository, we generated own testing scenarios the programmatic way, which depends on the number of current jobs. Therefore, the possibility exists that there is a gap between the experiment process and the rea- world scenario. However, as mentioned in section 4.2, algorithms could not have different behaviors when CI infrastructure is used highly intensively or occasionally. Therefore, this might affect the performance gap between the algorithms, but not the relative priority.

External validity is threatened by the build or test software's dependencies, for example, if testing depends on external modules that must be downloaded before the build or test process. Process times will change depending on network bandwidth or server status. However, we used the default CI configuration in real projects to validate our approaches. Moreover, modern CI frameworks provide a caching feature to reduce the download or duplicated-initialization process. In these cases, external dependencies might not significantly influence the total process times.

4.6 Summary

We introduced the TDS CI task-selection algorithm that estimates the CI completion time for each CI task, and then selects the most suitable CI server. Decreases in performance in CI services are usually due to resource competition between CI services on the same CI server. To minimize the side-effect of resource competition between CI services, we used previous task times to predicting a remaining time of current tasks. Compared to other methods such as round robin, binpack, and random, our approach demonstrated a shorter time for completing the task, and allowed the task to use resources fairly in shared CI infrastructure .

The limitation of the proposed approach is that instead of fully considering characteristics, it only uses a portion of the tasks in each node and an average time of previous jobs. Furthermore, it was only tested on our small test set with the simulated scenario.

In subsequent work, our approach should be verified with a realistic scenario. To enhance the scheduling algorithm with more accurate profiling and monitoring techniques to estimate CI task completion times, the plan is to consider resource usage per task to increase the accuracy of prediction times.

CHAPTER 5
RELATED WORK

To date, a number of studies [1, 2, 10–12] have indicated the usage and benefit of CI through surveys or data collection from open-source projects. The results obtained from the preliminary study of open-source projects using CI are presented by Vasilescu *et al.* [10]. They found that developers who work with CI experience more productivity. Hilton *et al.* [1] confirmed positive outcome through qualitative studies with the open-source projects [2] and proprietary projects [1]. Similarly, Debbiche *et al.* [5] interviewed major Swedish telecommunication company but contrasts with above studies focused on challenges of CI.

The practice and implementation of CI varies. Ståhl *et al.* [13] present current descriptive models that facilitate the documentation of CI practices and other researchers [14, 15] introduced visualization techniques to foster project visibility. Leading companies such as Facebook [16, 17] and Google [6, 18] also presented implementations and practices.

Practical problems owing to size and complexity are argued in multiple studies. The majority of previous studies [19–21] are based on applying different types or number of tasks to each development process, for instance, the regression-test selector-collecting test [18, 22], which related to code change, and then skipped the other test cases for verifying each commit. However, all of these cases are always tested either for the merged code branch or before releasing the product. Similarly, several studies [23] have attempted to reduce task time by improving re-usability through the reuse of previous outputs, which did not affected by code changes. Other researchers introduced techniques that change either the task order [24] in the CI task queue or the processing order of the task by prioritizing test cases according to code coverage [25] or previous historical data [26, 27] to fail and stop in early stage. Unlike previous research works, our work for task scheduling focused on improving the utilities of the CI infrastructure without modifying internal processes. Therefore, our research and other existing approaches could have a complementary relationship.

Testing as a Service is a similar concept to CI; however, it only focused on testing and implementation for the SAAS model. Recently, TAAS has received attention in relation to mobile apps due to the fragmentation problem on the mobile platform. According to a market report from opensignal [28], over 24,000 distinct Android devices produced by 1,294 manufacturers were presented in 2015. To handle the diversity of environments during the development process, in the industry, few companies lauched a commercial TAAS service for mobile devices, such as firebase test lab[1], AWS device farm[2].

Academic researchers also proposed testing the model [29, 30], platform [31, 32] and the platform, and prioritizing the devices to test [33–35]. Vilkomir[35] prove that effectiveness of random selected testing environments. In his experiment, he investigated how many devices are needed to cover device-specified bugs. He reported that tests with five randomly selected device demonstrated an 85% effectiveness and 13 different devices achieved 100% effectiveness. Although the above studies have a similar purpose to our thesis, and TAAS could be used through a CI framework, using separate services or frameworks increases a user's effort in relation to configuration and maintenance. Therefore, we attempted to improve CI frameworks, instead of using another framework or service.

To the best of our knowledge, our approach is the first work that demonstrates how to optimize the performance of container clusters for CI. Regardless of CI, most tools, such as Docker Swarm[3] and Kubernetes[4], are designed for managing multiple containers in the cloud, and support few options to select scheduling strategies. While Docker swarm and Kubernetes are widely used in the industry, their main goal is to help construct a fault-tolerance system, so new containers are considered running forever. Empirical researches reported [8, 36] that containers have a higher performance than conventional virtual machines. As a result, studies for utilizing container in software engineering are becoming increasingly popular; many researchers [37, 38] introduced container-based methods and tools for the software-development process. While the many existing studies have focused on performance aspects,

[1]`https://firebase.google.com`
[2]`https://aws.amazon.com/device-farm`
[3]`http://www.docker.com/products/docker-swarm`
[4]`http://kubernetes.io/`

interference from other tenants has not been noticed. Sharma *et al.* [39] found that container shows more performance interference by co-hosted machines than Hypervisor, and resource limitation was also less effective to reducing interference.

Similarly to our resource-profiling approach, other researchers [40, 41] introduced automated regression benchmarking, and Waller *et al.* [41] suggested including benchmarks into CI to detecting performance issues in the early stages. However, their implementation, which uses shell scripts to call benchmark tools, requires a user's effort to write separate scripts for each project.

CHAPTER 6
CONCLUSION AND DISCUSSION

Continuous integration requires increasing coverage for software-execution environments, and requires techniques for enhancing the efficency of infrastructure. In this thesis, we proposed the supplementary configuration method to improving the utility of CI by increasing the chance of discovering potential software problems with limited available computing resources. Then task-scheduling algorithms, which allocate tasks to the best node in the server cluster to improve the performance of the CI infrastructure, were presented. Through use cases and simulation, we obtained promising results, which suggest that this approach could benefit a developer's productivity and improved effectively of CI infrastructure.

However, our study was validated by suggested use cases and a small simulation. Thus, these results may not be applicable to all situations. To apply this approach to real projects, further studies are required to improve its applicability.

In terms of directions for future research, further work could include the following:

- Develop configurational syntax to describe more complex compositions for build and test tasks.
- Develop a composing algorithm to generate environment combinations more effectively, for example, excluding or including combinations that a user manually selected or that were recently tested.
- Increase the scheduling algorithm with accurate profiling and monitoring techniques to reducing the scale of interference with overlapped time.
- Develop data-analysis techniques with resource usage to provide processed or visualized data to users instead of raw statistics.

- Experiments should be applied during real-world software-development processes, and effect and benefit should be analyzed.

REFERENCES

[1] M. Hilton, N. Nelson, D. Dig, T. Tunnell, and D. Marinov, "Continuous Integration (CI) Needs and Wishes for Developers of Proprietary Code," Corvallis, OR: Oregon State University, Dept. of Computer Science, Tech. Rep., 2016.

[2] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects," in *The 31st ieee/acm international conference*, New York, New York, USA: ACM Press, 2016, pp. 426–437.

[3] G. Booch, *Object-oriented analysis and design with applications*. Benjamin Cummings, 1994.

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

[5] A. Debbiche, M. Dienér, and R. B. Svensson, "Challenges When Adopting Continuous Integration: A Case Study," in *Product-focused software process improvement*, Cham: Springer, Cham, Dec. 2014, pp. 17–32.

[6] J. Micco, "The State of Continuous Integration Testing@ Google," Google, Tech. Rep., 2017.

[7] R. Fielding, J. . Gettys, J. Mogul, H. Frystyk, L. . Masinter, P. . Leach, and T. Berners-Lee, "RFC 2616, Hypertext Transfer Protocol – HTTP/1.1," Tech. Rep., 1999.

[8] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, *IBM Research Report: An Updated Performance Comparison of Virtual Machines and Linux Containers*. IBM Research Division, 2014.

[9] K. Beck, *Extreme Programming Explained*, ser. Embrace Change. Addison-Wesley Professional, 2000.

[10] B. Vasilescu and S. Van Schuylenburg, "Continuous integration in a social-coding world: Empirical evidence from GitHub," *... (icsme)*, pp. 401–405, 2014.

[11] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov, *Quality and productivity outcomes relating to continuous integration in GitHub*. New York, New York, USA: ACM, Aug. 2015.

[12] D. Ståhl and J. Bosch, "Experienced Benefits of Continuous Integration in Industry Software Product Development: A Case Study," in *Artificial intelligence and applications*, Calgary,AB,Canada: ACTAPRESS, 2013.

[13] D. Ståhl and J. Bosch, "Modeling continuous integration practice differences in industry software development," *Journal of systems and software*, 2014.

[14] A. Nilsson, J. Bosch, and C. Berger, "Visualizing Testing Activities to Support Continuous Integration: A Multiple Case Study," in *Agile processes in software engineering and extreme programming*, Cham: Springer, Cham, May 2014, pp. 171–186.

[15] M. Brandtner, S. C. Muller, P. Leitner, and H. C. Gall, "SQA-Profiles: Rule-based activity profiles for Continuous Integration environments," in *2015 ieee 22nd international conference on software analysis, evolution, and reengineering (saner)*, IEEE, 2015, pp. 301–310.

[16] C. Rossi, E. Shibley, S. Su, K. Beck, T. Savor, and M. Stumm, "Continuous deployment of mobile software at facebook (showcase)," in *The 2016 24th acm sigsoft international symposium*, New York, New York, USA: ACM Press, 2016, pp. 12–23.

[17] T. Savor, M. Douglas, M. Gentili, L. Williams, K. Beck, and M. Stumm, "Continuous deployment at Facebook and OANDA," in *The 38th international conference*, New York, New York, USA: ACM Press, 2016, pp. 21–30.

[18] S. Yoo, R. Nilsson, and M. Harman, "Faster fault finding at Google using multi objective regression test optimisation," *8th european software ...*, 2011.

[19] R. M. de Abreu, "Multi-Stage Continuous Integration: Leveraging Scalability on Agile Software Development," Metropolia Ammattikorkeakoulu, Tech. Rep., 2013.

[20] R. O. Rogers, "Scaling continuous integration," *... conference on extreme programming and agile ...*, 2004.

[21] S. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments," in *The 22nd acm sigsoft international symposium*, New York, New York, USA: ACM Press, 2014, pp. 235–245.

[22] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *Acm transactions on software engineering and methodology (tosem)*, vol. 6, no. 2, pp. 173–210, Apr. 1997.

[23] 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*, IEEE, 2015, pp. 797–798.

[24] M. Waterloo, "Improving the Efficiency of CI with Uber-commits," Master's thesis, Department of Computer Science, University of Nebraska–Lincoln, Department of Computer Science, University of Nebraska–Lincoln, 2016.

[25] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *Ieee transactions on software engineering*, vol. 27, no. 10, pp. 929–948, 2001.

[26] J.-M. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," in *The 24th international conference*, New York, New York, USA: ACM Press, 2002, p. 119.

[27] M. Gligoric, L. Eloussi, and D. Marinov, "Ekstazi: Lightweight Test Selection," in *2015 ieee/acm 37th ieee international conference on software engineering*, IEEE, 2015, pp. 713–716.

[28] OpenSignal, "Android fragmentation visualized," OpenSignal, Tech. Rep., Aug. 2015.

[29] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *The acm sigsoft 20th international symposium*, New York, New York, USA: ACM Press, 2012, p. 1.

[30] R. Mahmood, N. Esfahani, T. Kacem, N. Mirzaei, S. Malek, and A. Stavrou, "A white-box approach for automated security testing of Android applications on the cloud," in *2012 7th international workshop on automation of software test (ast)*, IEEE, 2012, pp. 22–28.

[31] C. Tao and J. Gao, "Modeling mobile application test platform and environment: testing criteria and complexity analysis," in *The 2014 workshop*, New York, New York, USA: ACM Press, 2014, pp. 28–33.

[32] C. Tao and J. Gao, "On building a cloud-based mobile testing infrastructure service system," *Journal of systems and software*, vol. 124, pp. 39–55, 2017.

[33] H. Khalid, M. Nagappan, and E. Shihab, "Prioritizing the devices to test your app on: A case study of android game apps," in *Proceedings of the 22nd ...*, 2014.

[34] X. Lu, X. Liu, H. Li, T. Xie, Q. Mei, and D. Hao, "PRADA: Prioritizing android devices for apps by mining large-scale usage data," in *Proceedings of the 38th ...*, IEEE, 2016, pp. 3–13.

[35] S. Vilkomir, "Multi-device coverage testing of mobile applications," *Software quality journal*, vol. 2, no. 3, p. 46, 2017.

[36] M. Raho, A. Spyridakis, M. Paolino, and D. Raho, "KVM, Xen and Docker: A performance analysis for ARM based NFV and cloud computing," *... electronic and electrical ...*, pp. 1–8, 2015.

[37] M. Rahman, Z. Chen, and J. Gao, "A Service Framework for Parallel Test Execution on a Developer's Local Development Workstation," *Service-oriented system ...*, pp. 153–160, 2015.

[38] J. Cito and H. C. Gall, *Using docker containers to improve reproducibility in software engineering research.* New York, New York, USA: ACM, May 2016.

[39] P. Sharma and L. Chaufournier, "Containers and Virtual Machines at Scale: A Comparative Study," in *Proceedings of the ...*, 2016.

[40] T. Kalibera, J. Lehotsky, D. Majda, B. Repcek, M. Tomcanyi, A. Tomecek, P. Tuma, and J. Urban, *Automated benchmarking and analysis tool.* New York, New York, USA: ACM, Oct. 2006.

[41] J. Waller, N. C. Ehmke, and W. Hasselbring, "Including Performance Benchmarks into Continuous Integration to Enable DevOps," *Acm sigsoft software engineering notes*, vol. 40, no. 2, pp. 1–4, Apr. 2015.