2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion) | 978-1-6654-1219-3/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICSE-Companion52605.2021.00070

CIBench: A Dataset and Collection of Techniques for Build and Test Selection and Prioritization in Continuous Integration

Xianhao Jin
Computer Science
Virginia Tech
xianhao8@vt.edu

Francisco Servant
Computer Science
Virginia Tech
fservant@vt.edu

***Abstract*—Continuous integration (CI) is a widely used practice in modern software engineering. Unfortunately, it is also an expensive practice — Google and Mozilla estimate their CI systems in millions of dollars. There are a number of techniques and tools designed to or having the potential to save the cost of CI or expand its benefit - reducing time to feedback. However, their benefits in some dimensions may also result in drawbacks in others. They may also be beneficial in other scenarios where they are not designed to help. Therefore, we built CIBench, a dataset and collection of techniques for build and test selection and prioritization in Continuous Integration. CIBench is based on a popular existing dataset for CI — TravisTorrent [2] and extends it in multiple ways including mining additional Travis logs, Github commits, and building dependency graphs for studied projects. This dataset allows us to replicate and evaluate existing techniques to improve CI under the same settings, to better understand the impact of applying different strategies in a more comprehensive way.**

***Index Terms*—continuous integration, software maintenance, empirical software engineering**

I. Introduction

Continuous Integration (CI) is a popular software development practice by which developers integrate code into a shared repository several times a day [4]. However, CI gains adoption in practice, along with difficulties and pain points. As software companies adopt CI, they execute many builds for many of projects in a very frequent way. As workload increases, two main problems appear: (1) the time to receive feedback from the build process increases [7], as software builds often outnumber the available computational resources — having to wait in build queues, and (2) the computational cost of running builds also becomes very high [5].

Multiple techniques have been proposed to improve CI. Most of them have the goal of reducing either its time-to-feedback or its computational cost. Time-to-feedback-reduction techniques aim to observe failures earlier — by prioritizing failing executions over passing ones. Computational-cost-reduction techniques aim to observe failures only — by selectively running failing executions only, saving the cost of executing passing ones. These techniques may operate in two different levels of granularity, by prioritizing or selecting: test executions *e.g.,* [3], [8], or build executions *e.g.,* [7], [1].

To the extent of our knowledge, the existing techniques to improve CI have been evaluated under different settings, making it hard to compare them. Previous studies used different software projects, different metrics, and rarely compared one technique to another. However, we expect that different choices of goal, granularity, and technique design will bring different trade-offs. Empirically understanding these trade-offs will have valuable practical implications for the design of future techniques and for practitioners adopting them.

As a result, we built CIBench[1], a dataset and collection of time-to-feedback-reduction and computational-cost-reduction techniques in Continuous Integration. CIBench selects representative projects from TravisTorrent [2] and extends it in multiple ways, including mining additional Travis logs, Github commits, and building dependency graphs for selected projects. Based on our dataset, we replicated and evaluated all the existing 10 CI-improving techniques from the research literature, representing the two goals (time-to-feedback and computational-cost reduction) and the two levels of granularity (build-level and test-level) for which such techniques have been proposed. Finally, we measured the effectiveness of all techniques with 10 metrics in 3 dimensions.

CIBench is the first dataset for the comprehensive evaluation of CI-improving techniques, including a replication of 14 variants of 10 CI improving techniques and how they perform with 10 metrics. The observations and findings from the evaluation can be accessed by existing work [6].

II. The CIBench Dataset

In this section, we give an overview of our dataset, including technical briefings for better understanding of how we extended the original dataset, how we replicated existing techniques and how we evaluated these techniques.

A. Data Pre-processing in TravisTorrent

We firstly performed data preprocessing in Travis Torrent dataset [2], which includes 1,359 projects (402 Java projects and 898 Ruby projects) with data for 2,640,825 build instances. We removed "toy projects" from the data set by

[1]https://doi.org/10.5281/zenodo.4372963

978-1-6654-1219-3/21/$31.00 ©2021 IEEE
DOI 10.1109/ICSE-Companion52605.2021.00070

focusing on those that are more than one year old, and that have at least 200 builds and at least 1000 lines of source code, which is a criteria applied in multiple other works. To be able to evaluate test granularity techniques, we also filter out those projects whose build logs do not contain any test information. We focused on builds with passing or failing result, rather than error or canceled — since they can be exceptions or may happen during the initialization and get aborted immediately before the real build starts. Besides, in Travis a single push or pull request can trigger a build with multiple jobs, and each job corresponds to a configuration of the building step. We did a preliminary investigation of these builds and found that these jobs with the same build identifier normally share the same build result and build duration. Thus, as many existing papers have done, we considered these jobs as a single build. After this filtering process, we obtained 82,427 builds from 100 projects (13,464 failing builds). The result after data pre-processing can be found under folder "TravisTorrent" and the project list can be found under folder "Extended_TravisTorrent/projects".

B. Data Extension in TravisTorrent

We extended the information in TravisTorrent of these 100 projects in multiple ways. First of all, we needed to capture the historical failure ratio and duration for each individual test. To obtain these information, we built scripts to download the raw build logs from Travis and parse them to extract all of the information about test executions, such as test name, duration and outcome. These information can be found in folder "Extended_TravisTorrent/test_info_logs" with scripts for downloading and parsing build logs. Some techniques require additional information that TravisTorrent does not provide for builds, such as the content of commit messages, changed source lines and changed file names. For that, we also mined additional information about commits in the projects' code repositories through Github. The information is covered under folder "Extended_TravisTorrent/git_info". Then, we matched each test with its corresponding test file in the project. Finally, to be able to run other techniques, we built a dependency graph for the source code of each project using a static code analysis tool (Scitool Understand[2]) to determine the paths between the source files and test files. The dependency information of projects is stored in the folder "Extended_TravisTorrent/dependency".

C. Technique Replications and Simulations

We replicated 10 techniques and simulated them in a real-world scenario, including three test prioritization techniques, one build prioritization technique, three test selection techniques, and three build selection techniques. The detailed information about replicated techniques can found in existing paper [6]. For those techniques trained within projects, we used an 11-fold, chronological variant of cross-validation. For each project, we split its chronological timeline into 11 folds. We used the first chronological fold only for testing, and we iteratively test the other 10 folds. For each testing fold, we trained on all the folds that precede it chronologically. Besides, for selection-based techniques, when a build or test is skipped, the technique will not know its outcome. Additionally, when builds are skipped, we accumulated their code changes into the subsequent build. The folder "Replication" includes the replication and simulation of all 10 existing techniques mentioned in the paper [6] based on the extended data set. We also included the simulation result of cross-project and within-project for those techniques that require to be trained.

[2]Understand Static Code Analysis Tool: https://scitools.com/

D. Technique Evaluations

After we replicated and simulated all the techniques under the same settings, we compared them based on ten metrics from three dimensions: (D1) computational-cost reduction, (D2) missed failure observation, and (D3) early feedback. To compare the techniques' cost-saving ability (D1), we measured saved numbers or duration of builds or tests. We also included the proportion of skipped failing tests or builds to better understand the undesirable side effect of cost-saving techniques skipping some of the failing executions (D2). Finally, we measured positions shifted for treated or all failing builds and build-queue-length saved to understand how selection techniques impact the failure observation (D3). We additionally included positions shifted for observed failing tests to see how prioritization techniques advance failure observations. The comparison result of the techniques based on 10 different metrics can be found under folder "Evaluation/metrics", with scripts analyzing the simulation results. The folder "data_set/Evaluation/result" includes the figures of the evaluation.

REFERENCES

[1] R. Abdalkareem, S. Mujahid, E. Shihab, and J. Rilling. Which commits can be ci skipped? *IEEE Transactions on Software Engineering*, 2019.

[2] M. Beller, G. Gousios, and A. Zaidman. Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration. In *Proceedings of the 14th working conference on mining software repositories*, 2017.

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

[4] M. Fowler and M. Foemmel. Continuous integration. *Thought-Works) http://www.thoughtworks.com/ContinuousIntegration.pdf*, 122:14, 2006.

[5] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy. The art of testing less without sacrificing quality. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 483–493. IEEE, 2015.

[6] X. Jin and F. Servant. What helped, and what did not? an evaluation of the strategies to improve continuous integration. In *Proceedings of the 43th International Conference on Software Engineering*, page To appear, 2021.

[7] J. Liang, S. Elbaum, and G. Rothermel. Redefining prioritization: continuous prioritization for continuous integration. In *Proceedings of the 40th International Conference on Software Engineering*, pages 688–698, 2018.

[8] M. Machalica, A. Samylkin, M. Porth, and S. Chandra. Predictive test selection. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 91–100. IEEE, 2019.