PerfCI: A Toolchain for Automated Performance Testing during Continuous Integration of Python Projects

Omar Javed
omar.javed@usi.ch
Università della Svizzera italiana,
Lugano, Switzerland

Joshua Heneage Dawes
joshua.heneage.dawes@cern.ch
University of Manchester, Manchester,
United Kingdom
CERN, Geneva, Switzerland

Marta Han
marta.han@cern.ch
University of Zagreb, Zagreb, Croatia
CERN, Geneva, Switzerland

Giovanni Franzoni
Andreas Pfeiffer
firstname.lastname@cern.ch
CERN, Geneva, Switzerland

Giles Reger
giles.reger@manchester.ac.uk
University of Manchester, Manchester,
United Kingdom

Walter Binder
walter.binder@usi.ch
Università della Svizzera italiana,
Lugano, Switzerland

ABSTRACT

Software performance testing is an essential quality assurance mechanism that can identify optimization opportunities. Automating this process requires strong tool support, especially in the case of Continuous Integration (CI) where tests need to run completely automatically and it is desirable to provide developers with actionable feedback. A lack of existing tools means that performance testing is normally left out of the scope of CI. In this paper, we propose a toolchain - PerfCI - to pave the way for developers to easily set up and carry out automated performance testing under CI. Our toolchain is based on allowing users to (1) specify performance testing tasks, (2) analyze unit tests on a variety of python projects ranging from scripts to full-blown flask-based web services, by extending a performance analysis framework (VyPR) and (3) evaluate performance data to get feedback on the code. We demonstrate the feasibility of our toolchain by using it on a web service running at the Compact Muon Solenoid (CMS) experiment at the world's largest particle physics laboratory — CERN.

Package. Source code, example and documentation of PerfCI are available: https://gitlab.cern.ch/omjaved/perfci. Tool demonstration can be viewed on YouTube: https://youtu.be/RDmXMKA1v7g. We also provide the data set used in the analysis: https://gitlab.cern.ch/omjaved/perfci-dataset.

ACM Reference Format:
Omar Javed, Joshua Heneage Dawes, Marta Han, Giovanni Franzoni, Andreas Pfeiffer, Giles Reger, and Walter Binder. 2020. PerfCI: A Toolchain for Automated Performance Testing during Continuous Integration of Python Projects. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3324884.3415288

1 INTRODUCTION

Continuous Integration (CI) [2] is the practice of integrating changes made by multiple developers into a code base [23]. It consists of automatically checking code correctness before integration. There are a number of different CI options [6, 13, 21], which execute unit tests autonomously in a remote server before finalising changes to the code.

Hence, a CI process provides a separation of concerns i.e., one service handles code version control, while a different service checks for code correctness. However, CI mostly checks the correctness of software functionality [12] which ignores checking non-functional properties, including performance issues.

Detecting performance issues is vital for software optimization. Over the years, performance issues have caused software failures, which have resulted in the abandonment of hundred-million dollar projects [19]. Furthermore, they are costly to diagnose because their consequences are not easily observable, leading to many hours or days of diagnosis time [15]. Moreover, software systems are becoming extremely complex and workloads are rapidly evolving [15]. Therefore, detecting performance bugs at an early stage of software development is extremely important. This would help provide early feedback to developers who can address these issues quickly.

To address this concern, in this paper we demonstrate two contributions. The first is a toolchain — PerfCI — which allows developers to easily carry out performance testing under CI of Python-based projects. PerfCI runs an extended version of VyPR [11], a performance analysis framework based on Runtime Verification [1], by default with the possibility of easily configuring additional analyses. The second contribution is an extension of the analysis facilities provided by VyPR to allow querying performance data with respect to individual unit test executions. This will provide actionable feedback to developers, i.e., information that allows for the identification of optimization opportunities. Hence, automation is achieved by developers requiring minimum effort to get started with setting up performance testing tasks and data analysis. Furthermore, we achieve flexibility by leveraging the existing CI pipeline without modifications.

To demonstrate the efficacy of our proposed toolchain, we apply it to a critical web service used at the CMS experiment at CERN, the world's largest particle physics laboratory. At CMS, the growing complexity of code bases along with the prodigious amounts of data generated by the CMS detector (600 MB/s [4]) make identifying software performance issues crucial.

The web service to which we apply our toolchain is the CMS Conditions Uploader [8], a service that plays a vital role in the preparation of physics data for analysis. Further, the web service undergoes frequent changes to better fit the evolving needs of CMS physicists, making a CI process used for the CMS Conditions Uploader a reasonable choice for evaluation of our toolchain.

2 MOTIVATING SCENARIOS

We highlight two performance testing/analysis scenarios, each serving as motivation for the design of our toolchain.

Scenario 1. A developer wants to analyze the frequency of control-flow paths taken during runtime [17]. This is usually done through profiling techniques. In our toolchain, users can plugin existing tools as tasks (see stage 1 in Section 3.1), so profiling can be performed by adding a profiling task to the central configuration file. If such a task is added to the configuration file, unit tests are instrumented accordingly, allowing automatic collection of data for performance testing.

Profilers are often used to identify hotspots in the code. However, these hotspots could be due to actual heavy computation. Therefore, we use runtime verification (with VyPR) to further identify sections of the code where performance doesn't meet the expectation.

Scenario 2. A developer is trying to understand the classification of a performance bug i.e., deciding whether it is a bug which manifests in all inputs ("always-active") or in inputs that trigger the bug on a special condition [15] ("hard-to-detect"). These bugs will be difficult to detect if the software testing executes problematic code only once. This is usually the goal of functional testing. Our toolchain provides a unit test runner which can exercise application code many times based on different inputs. This is useful for performance analysis in stage 2 (Section 3.2) of our toolchain. Once the unit tests have been run, developers can use our extension of VyPR's analysis library to analyze performance data obtained from running the same unit test with multiple inputs as discussed in stage 3 (Section 3.3).

3 PERFCI TOOLCHAIN

PerfCI allows developers to use their existing CI setup and unit tests to identify problematic code regions and obtain actionable feedback for making optimization decisions.

To achieve this, PerfCI provides a central configuration file in which developers can define the tasks to be executed during CI and the specification for which VyPR should monitor.

Once CI is complete, we provide an extended version of VyPR's analysis library [9], giving users the ability to analyze performance data with respect to unit test results in order to derive actionable feedback.

Combining our configurable CI approach with the extended analysis library leads to the three key steps taken by PerfCI, shown in Figure 1. For brevity, we only show the unit test stage, however there can be many stages, especially if one configures many analyses. In the Figure, dotted lines represent the functioning of our toolchain: 1) the user specifies performance testing tasks; 2) VyPR monitors the given system when executed under unit tests; and 3) the user analyzes the data to get actionable feedback on the code. We now explain each of these stages in more detail.

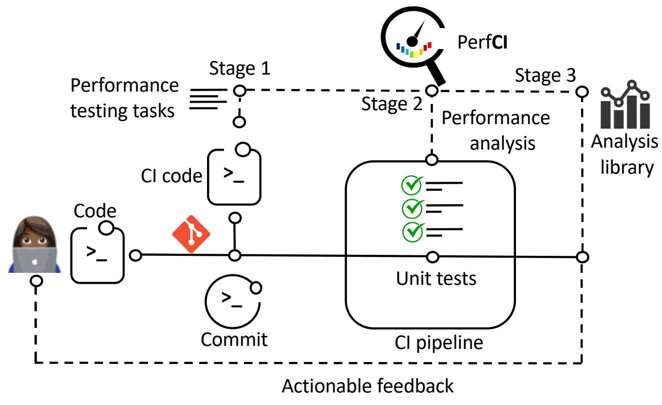

Figure 1: Overview of PerfCI function. Solid lines show the normal flow of the CI process, while dotted lines represent the operations of PerfCI.

3.1 Stage 1: Specifying performance testing tasks

One of the main features of PerfCI is the ability of the user to easily define performance testing tasks in a configuration file[1]. These tasks include 1) passing inputs to test suites; 2) attaching existing tools via plugin in order to carry out specific performance analyses; and 3) slicing the input space (useful when running performance analysis on specific sets of inputs). Users can also define performance specifications to allow the system's performance to be analyzed with respect to individual unit test executions.

Performance testing tasks are specified in a YAML file. YAML was chosen for two reasons: 1) it is easy to define performance testing tasks because of YAML's simple format, and 2) users can directly embed Python code into a YAML file. Our YAML file is placed in the root directory of the project where the base CI file exists e.g., '.gitlab-ci.yml'.

Given a YAML file, PerfCI updates the base CI file with the additional configuration. On push, a performance testing stage consisting of the tasks defined is added to the existing CI pipeline.

3.2 Stage 2: Monitoring

In order to monitor the code exercised by unit tests for performance issues, we use VyPR, a framework based on Runtime Verification. Runtime Verification typically consists of performing dynamic analysis in order to decide whether a run of a program meets some specified behaviour.

Our reasons for choosing Runtime Verification are two-fold. First, it complements software testing [5]. Second, the language used for behaviour specification can be chosen such that disagreement of the program run with the specification can lead to immediate breaking

[1]Detail about actions can be found in wiki page: https://gitlab.cern.ch/omjaved/perfci/

of a CI process via generation of a false verdict. VyPR, which is being used at the CMS Experiment at CERN for performance analysis, allows such behaviour because its specification language (Control-Flow Temporal Logic [10]) ensures that, once reached, a false verdict will never change.

Since the performance of any system can vary greatly depending on the input being processed, in order for VyPR to monitor code when running under different inputs we provide a test runner for handling execution of unit tests based on different inputs. We achieve this by parameterizing the unit test runner i.e., input is passed as an argument in the test runner which can exercise the application code based on different inputs supplied. Furthermore, running unit tests repeatedly on different inputs help to identify whether performance degradation happens all the time or under specific inputs in the code exercised by the unit test.

3.3 Stage 3: Performance data insights

In order to understand the performance data, we extended the existing analysis library provided by VyPR. Once the CI process is completed, users can use the analysis library to identify and understand performance issues in the code. Users can make simple queries to get data such as all monitored functions, all failures with respect to specifications and all performance data generated during a specific unit test execution. On the other hand, users can also do more complex analysis, such as comparing different code paths taken by different inputs[2]. The advantage of using our extension of VyPR's analysis library is that users can get actionable feedback, i.e. information regarding optimization of the code and/or improvement of the performance specifications.

4 ANALYZING THE CMS CONDITIONS UPLOADER WITH PERFCI

We demonstrate the toolchain — PerfCI — by analyzing the CMS Conditions Uploader, a critical web service used at the CMS Experiment at CERN. This service is responsible for performing correctness checks on data describing the physical calibrations of the CMS detector before inserting it into a database. This process must take place if physicists are to perform any analyses, hence it is vital that the service functions correctly and quickly. Our steps for the analysis are as follows:

1. Profiling via plugin. In order for VyPR to monitor code exercised by unit tests, users can write a performance specification in the PerfCI configuration file. To suggest to developers part of their code over which they should write specifications we provide a plugin, based on the *profiler* tool, that generates call graphs with statistics on function execution time. We integrate this plugin into CI by specifying it as a task in PerfCI's configuration file, which leads to appropriate instrumentation and data collection.

A bottleneck that we found by running our toolchain can be seen in Figure 2. Specifically, there was a constructor which occupied 99% of the total time taken by the unit test execution.

Ultimately, our profiling plugin provides indication to users of where a bottleneck has occurred. However, it does not help users to understand why this bottleneck has occurred and neither does

[2]See https://pyvypr.github.io/home/analysis-library/docs/ for details.

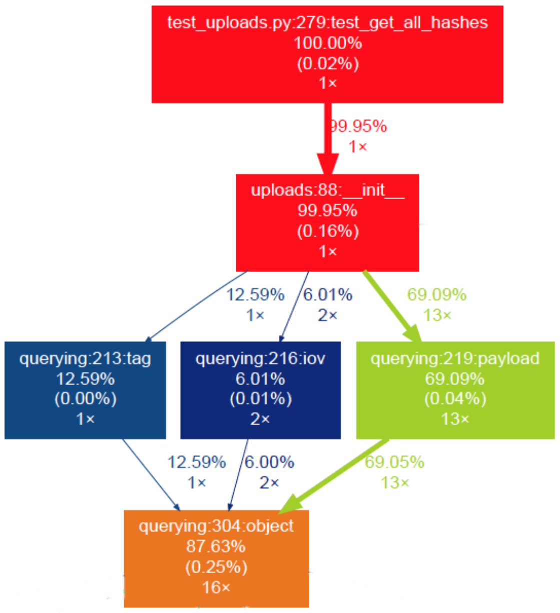

Figure 2: Bottleneck identification by profiling

```
verification_conf = {
    'CondDBFW.uploads': {
        "uploader.__init__": [
            Forall(c = calls('querying.connect')).Check(lambda c :
                timeBetween(
                    c.input(), c.next_call("close").result()
                )._in ([0 , 1])
            )
        ]
    }
}
```

Figure 3: Performance specification

it show how many code sections have been affected by this performance bottleneck. For this we perform a deeper analysis with VyPR and its analysis library.

2. Analysis-by-specification. In the function identified by our plugin as a bottleneck, we found that the 'uploader' constructor involves heavy computation in order to perform some validation. Such validation takes place between the initialisation of a database connection and its closure. To more closely investigate the behaviour of this code, we used VyPR to monitor for a specification which expresses the requirement that *the time elapsed from the start of the database connection initialisation to the end of the closure call should not be more than 1 second.* Figure 3 shows our specification code. The constraint of 1 second found in the specification was defined based on the observation of the uploader constructor's execution time. The results of monitoring our specification are shown by the 'verdict severity' metric (see Figure 4). This is explained in the next step.

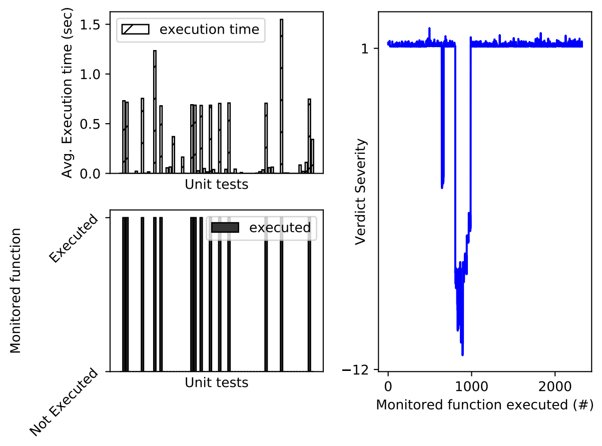

Figure 4: Performance data visualization by our CI analysis library.

3. Analyzing data with the analysis library. With our extension of the analysis library, users can select a single unit test execution and get all performance data derived from monitoring code exercised by that test execution. Figure 4 shows the execution time of all unit tests of the CMS Conditions Uploader (upper-left), and the monitored function executed by each unit test (lower-left).

We analyzed 65 unit tests written over the CMS Conditions Uploader, out of which 14 test cases (lower-left part of Figure 4) suffer from the bottleneck identified in the uploader constructor. We can see from the Figure that whenever the monitored function is executed the time of the unit test increases. This shows that the function under inspection has a severe bottleneck in the execution path of unit tests.

With the analysis library, we can also get information on the behavior of verdicts. In Figure 4 (right) we plot the *verdict severity* of our specification, which is the absolute difference between the measured value and the constraint we gave in the specification (i.e., [0,1]). This difference is further multiplied by a "characteristic function". The characteristic function gives 1 if the constraint was satisfied, and -1 otherwise. Hence, bad failures give values which are very negative as opposed to borderline failures which give values close to 0. In our case, we ran unit tests based on 100 inputs with varying size. Most of the inputs were in the order of Mbs (i.e., 89 out of 100 inputs). However, there were some inputs with large size (i.e., order of 100 Mbs). We found that such inputs result in bad failures — high negative values in the right part of Figure 4.

Remark on CI execution time. With the help of the extended analysis library, we were able to identify a slow-down in the test suite (Figure 4). From this, removing unnecessary uploader instantiations from test cases reduced total CI execution time by an average (taken across three CI runs with 100 inputs) of 38%.

5 RELATED WORK

There are a number of popular CI options available such as CircleCI [6], TravisCI [21], GitLab CI/CD [13], and many more. These CI tools allow developers to automatically execute unit tests (in a remote server) before pushing changes to the code on a public code repository service. However, these tools and services do not offer performance testing. Our tool can be used by developers to easily setup performance testing on existing CI tools and services.

There is already work on identifying, detecting and analyzing performance bugs [3, 7]. These studies target either understanding the effectiveness of performance bug detection [14] or empirically demonstrating the cause and effect of performance bugs [15]. In this paper, we draw inspiration from the existing state-of-the-art by proposing a tool that can be used by developers to analyze, detect and understand performance bugs during CI.

Existing CI services provide performance testing functionality e.g., Jenkins uses the Taurus tool for load testing [18]. Furthermore, a study has also shown the benefit of including performance benchmarking into the CI process [16, 20, 22]. On the contrary, we identify performance issues by analyzing the code exercised by unit tests.

PeASS [20] is a tool that uses Java unit tests as performance tests in CI . It looks for performance degradation between two versions. In contrast, PerfCI is developed in a single version scenario where performance issues are identified in a single release.

6 CONCLUSION

We presented the PerfCI toolchain which allows developers to carry out performance testing under CI. PerfCI allows developers to easily setup performance actions in a configuration file. Then, it uses techniques from Runtime Verification in the form of the VyPR framework to analyze performance issues in the execution of unit tests. We extended the existing analysis library provided by VyPR to help developers to combine unit test executions with the performance data generated by exercised code. In order to demonstrate the feasibility of PerfCI, we setup our toolchain with the CMS Conditions Uploader, a critical service used at the CMS Experiment at CERN. We identified and analyzed a previously unknown performance bottleneck. In the future, we plan to extend PerfCI to work with other CI services e.g., TravisCI.

REFERENCES

[1] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. 2018. Introduction to Runtime Verification. In *Lectures on RV.*
[2] Grady Booch. 1990. Object-Oriented Design with Applications.
[3] J. Burnim, S. Juvekar, and K. Sen. 2009. WISE: Automated test generation for worst-case complexity. In *ICSE.*
[4] CERN. 2020. https://home.cern/science/computing/processing-what-record. (Accessed on 05/26/2020).
[5] Jesús Mauricio Chimento, Wolfgang Ahrendt, and Gerardo Schneider. 2018. Testing Meets Static and Runtime Verification. In *FormaliSE@ICSE.*
[6] CircleCI. 2020. https://circleci.com. (Accessed on 05/25/2020).
[7] David Daly, William Brown, Henrik Ingo, Jim O'Leary, and David Bradford. 2020. The Use of Change Point Detection to Identify Software Performance Regressions in a Continuous Integration System. In *ICPE.*
[8] Joshua H Dawes. 2017. A Python object-oriented framework for the CMS alignment and calibration data. In *Journal of Physics: Conference Series.*
[9] Joshua Heneage Dawes, Marta Han, Giles Reger, Giovanni Franzoni, and Andreas Pfeiffer. 2019. Analysis Tools for the VyPR Framework for Python. In *CHEP.*
[10] Joshua Heneage Dawes and Giles Reger. 2019. Explaining Violations of Properties in Control-Flow Temporal Logic. In *RV.*
[11] Joshua Heneage Dawes, Giles Reger, Giovanni Franzoni, Andreas Pfeiffer, and Giacomo Govi. 2019. VyPR2: A Framework for Runtime Verification of Python Web Services. In *TACAS.*
[12] T. Durieux, R. Abreu, M. Monperrus, T. F. Bissyandé, and L. Cruz. 2019. An Analysis of 35+ Million Jobs of Travis CI. In *ICSME.*
[13] GitLab. 2020. https://docs.gitlab.com/ee/ci/. (Accessed on 05/25/2020).

[14] Xue Han, Tingting Yu, and David Lo. 2018. PerfLearner: Learning from Bug Reports to Understand and Generate Performance Test Frames. In *ASE*.

[15] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and Detecting Real-World Performance Bugs. In *PLDI*.

[16] C. Laaber and P. Leitner. 2018. An Evaluation of Open-Source Software Microbenchmark Suites for Continuous Performance Assessment. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. 119–130.

[17] Rashmi Mudduluru and Murali Krishna Ramanathan. 2016. Efficient Flow Profiling for Detecting Performance Bugs. In *ISSTA*.

[18] Jenkins CI performance plugin. 2020. http://jenkinsci.github.io/performance-plugin/RunTests.html. (Accessed on 05/29/2020).

[19] The Register. 2002. https://www.theregister.co.uk/2002/07/03/1901_census_site_still_down/. (Accessed on 05/25/2020).

[20] D. G. Reichelt, S. Kühne, and W. Hasselbring. 2019. PeASS: A Tool for Identifying Performance Changes at Code Level. In *ASE*.

[21] TravisCI. 2020. https://travis-ci.org/. (Accessed on 05/25/2020).

[22] Jan Waller, Nils C. Ehmke, and Wilhelm Hasselbring. 2015. Including Performance Benchmarks into Continuous Integration to Enable DevOps. In *SIGSOFT Softw. Eng. Notes*.

[23] Fiorella Zampetti, Carmine Vassallo, Sebastiano Panichella, Gerardo Canfora, Harald Gall, and Massimiliano Di Penta. 2020. An empirical characterization of bad practices in continuous integration. In *ESE*.