EPJ Web of Conferences **245**, 05013 (2020) https://doi.org/10.1051/epjconf/202024505013
CHEP 2019

Analysis Tools for the VyPR Performance Analysis Framework for Python

Joshua Heneage Dawes[1,2,*], *Marta* Han[3], *Giles* Reger[1], *Giovanni* Franzoni[2], and *Andreas* Pfeiffer[2]

[1]University of Manchester, Manchester, UK
[2]CERN, Geneva, Switzerland
[3]University of Zagreb, Zagreb, Croatia

Abstract. VyPR (http://pyvypr.github.io/home/) is a framework being developed with the aim of automating as much as possible the performance analysis of Python programs. To achieve this, it uses an analysis-by-specification approach; developers specify the performance requirements of their programs (without any modifications of the source code) and such requirements are checked at runtime. VyPR then provides tools which allow developers to perform detailed analyses of the performance of their code. Such analyses can include determining the common paths taken to reach badly performing parts of code, deciding whether a single path through code led to variations in time taken by future observations, and more.
This paper describes the developments that have taken place in the past year on VyPR's analysis tools to yield a Python shell-based analysis library, and a web-based application. It concludes by demonstrating the use of the analysis tools on the CMS Experiment's Conditions Upload service.

1 Motivation for Sophisticated Performance Analysis

Gaining a precise understanding of the performance of a program is an important problem, especially when the program performs a critical activity or covers a diverse set of use cases. In order to provide a good solution, there must first be a precise definition of *performance* and such a definition depends strongly on the program being analysed. For example, if a program communicates over a network often, one could characterise its performance by how the network behaves as time and input vary. Alternatively, if a program performs a lot of computation, one could characterise its performance by quantities that are internal to the program such as values held in memory and timing information.

1.1 Existing Approaches to Performance Analysis

Existing approaches to checking the performance of a program, in particular with respect to timing, tend to focus on profiling. Profiling tools, particularly in the Python setting, include `cProfiler` [1] and `pyinstrument` [2]. While profiling has been proven through years of

*e-mail: joshua.dawes@cern.ch

operational experience in industry to be a powerful tool, it can have shortcomings. For example, if we consider a profiling technique that involves recording the time taken by some function whenever it is called, then this is perfect for analysing this single quantity, perhaps with respect to others, over time. A similar profiling technique would even be appropriate for measuring the value held by some variable during some run of a function. The offline analysis required to use such an approach to understand the performance of a program would not be difficult; it would simply be a matter of plotting the recorded quantity with respect to whatever parameters were also observed and then manually inspecting.

However, these methods can easily become infeasible if we want to check a constraint that is a composite of constraints over multiple quantities. Performing a check like this could require 1) more complex instrumentation and 2) more complex, possibly error-prone, manual analysis performed by developers. In terms of instrumentation, standard profiling approaches can already be quite intrusive, since common approaches to profiling Python code include 1) developers decorating functions whose performance is interesting and 2) tracing the program execution using the interpreter [3]. These approaches can also be unreliable, since a manual approach may result in some edge cases not being identified. Finally, manual analysis of data collected when it represents measurements made over many quantities can be difficult and inefficient.

1.2 Analysis-by-Specification

This paper begins by describing the VyPR framework (http://pyvypr.github.io/home/), which provides a way for developers to analyse the performance of their Python programs with respect to multiple non-trivial constraints. To achieve this, VyPR takes inspiration from Runtime Verification (RV) [13], which involves deciding whether a run of a program holds a given property. Applying VyPR to a program involves 1) specifying the program's expected performance; 2) applying automatic instrumentation and monitoring to check agreement with the specification; and 3) using VyPR's analysis tools to obtain an in-depth understanding of the program's performance. While there is much work on specification of program behaviour [5–7] and online monitoring [8] from the RV community, VyPR distinguishes itself with its low-level specification language [9].

The main contribution of this paper is a description of one of VyPR's analysis tools, designed to enable developers to easily perform in-depth analyses of their programs' performance. In particular, we describe an offline analysis library for Python that allows:

- Straightforward querying of the detailed information VyPR stores while monitoring a run of a program.
- Program path analysis in order to identify potentially problematic regions of code.
- Value analysis in order to identify problematic inputs to certain events observed at runtime.

We conclude by describing further work that is now underway. This will include extensions of the analysis tools and a web-based application to provide a visual approach to offline analysis.

2 Building Performance Specifications

Performance requirements are specified using the PyCFTL library for Python that VyPR provides. A specification written using PyCFTL is in two parts:

- *Point of interest selection*, which involves the developer giving a criteria to select points at which to check a constraint during a single run of a function.
- *Constraint building*, which involves the developer giving a constraint, built up using functions provided by PyCFTL, to check at each point of interest.

2.1 A Simple Example

We first describe the simple requirement that a call of the function `func` within a single scope never exceeds 2 seconds. We begin with the point of interest selection. In this case, our points of interest are every call to the function `func`, which we select using the code `Forall(call = calls('func'))`. VyPR will find every explicit call to `func` that is statically determinable; calls of the same function under other names must be captured using precisely those names. We must then define the requirement that each of the calls, which is bound to the variable `call` by this call to `Forall`, should take no more than 2 seconds. We can do this by building a lambda expression which takes `call`, a PyCFTL object representing a function call, as a variable: `lambda call : call.duration() <= 2`. To put this all together, we have the code:

```
Forall(call = calls('func')).Check(lambda call : call.duration() <= 2)
```

2.2 More Complex Constraints

We can build more complex specifications in multiple ways; by making point of interest selection more complex and by making the constraint we apply at each such point more complex. Our next example shows a more complex constraint.

Suppose that we require that, whenever a variable `n` is changed, the next call to a function `process` should take no more than `n*10` seconds. We first select our points of interest using `Forall(state = changes('n'))` and then define the constraint to take the variable `state` and constrain the next call to `process` found after it using

```
lambda state : state.next_call('process').duration() <= state('n')*10
```

Putting this together, we have

```
Forall(state = changes('n')).Check(lambda state : (
  state.next_call('process').duration() <= state('n')*10
))
```

From these examples, one can see that PyCFTL supports comparison of quantities with constants and other quantities measurable at runtime. It also supports arithmetic on measured numerical quantities (with the requirement that the arithmetic is performed on the right).

We now briefly show two final properties. The first captures the requirement that "every time `construct` is called, the time until the next call to `commit` should be no more than 1 second".

```
Forall(call = calls('construct')).Check(lambda call : (
  timeBetween(call.result(), call.next_call('commit').input()) <= 1
))
```

The final property shows a different way to build constraints along with how point of interest selection can use different criteria. In this case, we require that "when line 10 is traversed, if the value of `type` immediately after is `A`, every call to the function `query` after it should leave the list `res` containing at least 2 results."

```
Forall(q = state_after_line(10)).Forall(c = calls('query', after='q')).\
Check(lambda q, c : (
  If(q('type').equals('A')).then(
    c.result()('res').length() >= 2
  )
))
```

2.3 Instrumentation and Monitoring

Given any specification that is written in PyCFTL, VyPR can automatically instrument the program under scrutiny (so that a conservative amount of data is taken at runtime) and efficiently monitor for agreement with the specification at runtime [9, 10]. Instrumentation is performed by:

- Reading into memory the code of the function to which a particular property applies.
- Using Python's `ast` library to construct the abstract syntax tree (AST) of the code.
- Inserting instrumentation code at a conservative set of program points identified by static analysis.

The modified AST is then compiled to bytecode and the original source file is renamed to prevent recompilation and subsequent loss of instrumented bytecode. Monitoring is triggered by importing the VyPR package and instantiating a VyPR object which also starts a separate thread in which the monitoring algorithm runs. The instruments placed send information to this thread in order for the monitoring algorithm to decide whether the program run satisfied the specification given by the developer.

3 Prototype Analysis Tools

The data captured by VyPR has a complex structure that is closely related to the framework's mathematical foundations. The offline analysis tools being developed so far include a Python library and a web application. The Python library operates in separation from VyPR, with most operations being possible simply by querying VyPR's verdict server and some currently requiring the source code of the relevant parts of the service that was monitored. In-depth documentation of the analysis library can be found at http://pyvypr.github.io/home/analysis-library/. Finally, the development of the web application is underway.

3.1 Object-Oriented Analysis Library

VyPR's current principle analysis tool is the VyPRAnalysis library, which provides methods that communicate with the verdict server via HTTP. The benefit of all data being stored on a central server (currently in a relational database) and accessible via an API that the server provides is that end-points can be defined to cover cases for which otherwise complex queries would be required. Such queries would make offline analysis time-consuming and error-prone, so we took the approach of developing an analysis library that takes common, complex queries and wraps them in single functions. This simplicity allows scripts written by developers using the analysis library to be written without thinking of the underlying (often heavy) computation that must happen to answer certain questions.

3.2 Some Simple Queries

Once the analysis library has been imported, some basic knowledge of the information collected by VyPR is useful. Given the simple specification

```
Forall(call = calls('func')).Check(lambda call : call.duration() < 2)
```

the information stored will allow the developer to:

- Select an HTTP request/transaction.

- Select a specific function and specification over that function.
- Select a call of the function.
- For a given point of interest (in this case, a call of `func`), see whether the constraint given was satisfied there, what the *observations* were that led to this verdict, and which parts of the code generated them. For example, in the specification we consider, a measurement is needed to decide whether `call.duration() < 2` holds. We call this measurement an *observation*.

Since all of this information is stored in a relational database, one can query in multiple directions. For example, by selecting a statement in code, a developer can see all results during monitoring that used a measurement from it.

We now give some examples of Python code using the analysis library. First, we list all of the failed verdicts generated by `func` violating the constraint defined by the specification above:

```
function = analysis.list_functions()[0]
prop = function.get_properties()[0]
call = function.get_calls()[0]
failed_verdicts = call.get_verdicts(property=prop, value=0)
```

Then, we can select a verdict and get the list of observations that were needed to reach it:

```
verdict = failed_verdicts[0]
observations = verdict.get_observations()
```

After the execution of this code, `observations` will hold a list of `Observation` objects, each of which containing the value observed, the part of the specification for which that value was observed and an indication of the point of code from which the measurement was taken.

3.3 Explaining a Verdict

Given a result concerning the satisfaction of a specification by a run of a program, it is natural to ask for an explanation of this result. One of the main examples of more complex queries for such explanation is *path comparison* [4]. This requires additional instrumentation by VyPR which results in small overhead; measurements have shown an additional 1% runtime overhead for our IO-heavy use cases. Path comparison is a powerful technique that can be used in different ways, so we present a case here in which it would be useful. Consider the specification

```
Forall(call = calls('construct')).Check(lambda call : (
  timeBetween(call.result(), call.next_call('commit').input()) <= 1
))
```

Based on this, it would be reasonable to select a call of `construct` to assign to `call` and then ask what the differences were (usually) between paths from `call.result()` to `call.next_call('commit').input()` across multiple runs of the function over which this specification was written. The path comparison machinery provided by VyPR would enable the developer to identify regions of disagreement among multiple paths between the two points. For example, if the code between the two points contained an if-statement, VyPR would be able to detect whether the paths agreed before and after the if-statement, but not during it. Ultimately, this facility allows developers to identify paths through their code that are less performant. Section 4.3 shows this working in practice.

4 Application of VyPR at the CMS Experiment

We now describe the application of VyPR to an upcoming version of the CMS Upload Service for Alignment and Calibrations data [11]. This service is responsible for uploading constants describing the alignment and calibration of the CMS detector [12] to a central database. The upload process involves frequent communication with other machines on a network in order to correctly validate proposed constants. We refer to a single execution of the upload process as a *Conditions upload*.

4.1 Using VyPR for the CMS Conditions Upload Service

The first step in applying VyPR is to write a specification. One approach to this process, especially when the code base is already written, is to start with approximate constraints over high-level parts of the code base being analysed and iteratively refine the specification.

Using this iterative process, which is the one that has been favoured during use at CMS so far, more intuition can be gathered about the expected values of measurements taken and the relationship between quantities measurable at runtime.

Once a specification is written, VyPR's instrumentation process is currently accessible to developers via the `instrument.py` script deployed with the main VyPR source code. This reads the specification, determines the relevant parts of the service code and finally performs instrumentation.

Monitoring at runtime is triggered by importing the `Monitor` class from VyPR. Instantiation of this class starts up a monitoring thread that consumes from a queue to which instrumentation code pushes messages.

4.2 Previous Results

The application of the first prototype of VyPR to the CMS Conditions Upload Service yielded two instances of interesting behaviour [10]. One was an unexpectedly large amount of time taken by one query, and another was that the times taken by many repetitions of a particular query seemed to be inversely proportional to the time between the queries. This first application allowed us to identify initial performance problems with critical infrastructure, but also showed the feasibility of using an approach inspired by Runtime Verification [13].

4.3 New Results

The first application of the newly developed path comparison machinery was to construct plots of the time taken for the CMS Conditions Upload Service to reach one statement from another during calls of a specific function. Our specification was similar to the one discussed earlier:

```
Forall(call = calls('tag_in_destination')).Check(lambda call : (
  timeBetween(call.result(), call.next_call('commit_iovs').result()) <= 1.2
))
```

In this case, the value tested by an if-statement to decide on which path to take from `call.result()` to `call.next_call('commit_iovs').result()` was a non-trivial object. This meant that storing the value and using that to determine the branch in offline analysis was not feasible. Further, we could have modified the code being monitored to give some indication of the branch via a boolean variable, but this was not appropriate because 1) we aim for minimal intrusion and 2) for more complex control-flow, this can become difficult.

In order to construct the plots that we needed with VyPR's analysis library, we took the following steps:

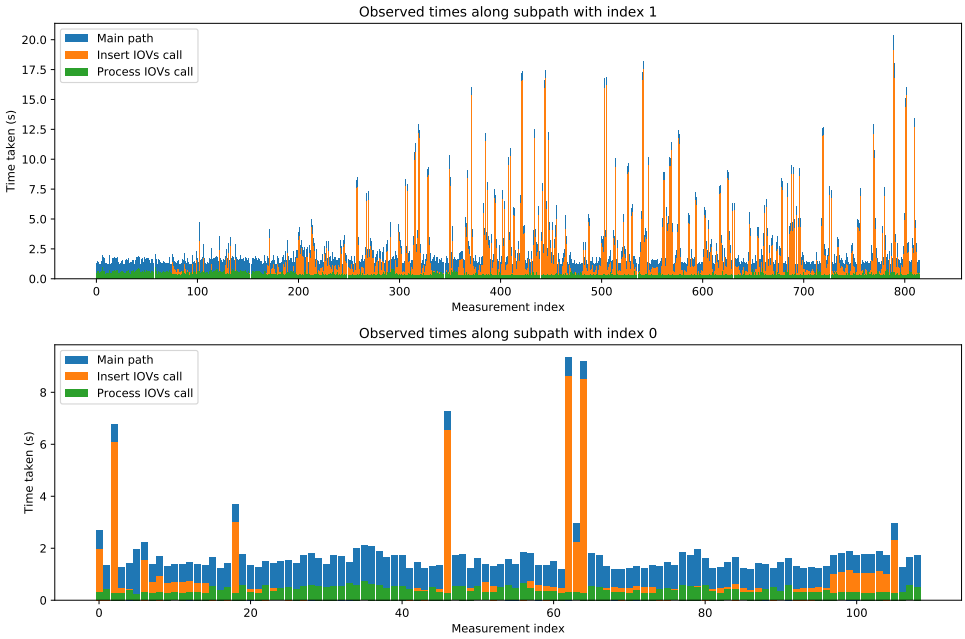

Figure 1. Plots of the time taken to get from one statement in code to another. Each plot contains measurements taken along a distinct path. The *measurement index* is the index of the measurement in the set derived for the particular path.

1. Select the relevant function and specification over that function.
2. Get the list of calls of that function.
3. For each call, get the pair of observations generated by the two statements of interest.
4. For each pair of observations, reconstruct the path between them.
5. Determine the region of disagreement of the resulting set of paths.
6. For each path taken through the region of disagreement, construct a plot of the time elapsed between the two observations. In our case, there were two paths taken through the region of disagreement, so we got two plots.

The plots constructed after replaying 4000 Conditions uploads and applying this analysis procedure (while throwing away observations for which the time taken violated our specification) are given in Figure 1. It is clear that one path is often less performant than the other. In an effort to determine the root cause of the erratic performance, we used VyPR to measure the time taken by calls lying on the two paths. We see that calls to the *Insert IOVs* function seem to be problematic.

5 Future Plans

Further work on the analysis library involves writing a set of *prebuilt scripts* to deploy alongside VyPR. Using this approach, we can provide automated analysis in the background. Fi-

nally, ongoing work on the web-based analysis tool will allow visualisation of path comparison data and exploitation of the data generated by our prebuilt analysis scripts.

6 Conclusion

In this paper we have briefly described the VyPR performance analysis framework for Python programs. Our main contribution was the set of analysis tools that are currently in the prototype stage, whose first application to critical infrastructure at CMS was described.

References

[1] `cProfiler`, https://docs.python.org/2/library/profile.html#module-cProfile

[2] `pyinstrument` - *a Statistical Profiler for Python*, https://github.com/joerick/pyinstrument

[3] `trace` - *the Python tracing tool*, https://docs.python.org/2/library/trace.html

[4] J.H. Dawes, G. Reger, *Explaining Violations of Properties in Control-Flow Temporal Logic*, in *Runtime Verification - 19th International Conference, RV 2019, Porto, Portugal, October 8-11, 2019, Proceedings* (2019), pp. 202–220, `https://doi.org/10.1007/978-3-030-32079-9_12`

[5] A. Pnueli, *The temporal logic of programs*, in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)* (1977), pp. 46–57, ISSN 0272-5428

[6] H. Barringer, A. Goldberg, K. Havelund, K. Sen, *Rule-Based Runtime Verification*, in *Verification, Model Checking, and Abstract Interpretation*, edited by B. Steffen, G. Levi (Springer Berlin Heidelberg, Berlin, Heidelberg, 2004), pp. 44–57, ISBN 978-3-540-24622-0

[7] S. Hallé, S. Varvaressos, *A Formalization of Complex Event Stream Processing* (2014), Vol. 2014, pp. 2–11

[8] A. Bauer, M. Leucker, C. Schallhart, ACM Trans. Softw. Eng. Methodol. **20**, 14:1 (2011)

[9] J.H. Dawes, G. Reger, *Specification of temporal properties of functions for runtime verification*, in *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, SAC 2019, Limassol, Cyprus, April 8-12, 2019* (2019), pp. 2206–2214, `https://doi.org/10.1145/3297280.3297497`

[10] J.H. Dawes, G. Reger, G. Franzoni, A. Pfeiffer, G. Govi, *VyPR2: A Framework for Runtime Verification of Python Web Services*, in *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part II* (2019), pp. 98–114, `https://doi.org/10.1007/978-3-030-17465-1_6`

[11] J.H. Dawes, C. Collaboration, Journal of Physics: Conference Series **898**, 042059 (2017)

[12] T.C. Collaboration, Journal of Instrumentation **3**, S08004 (2008)

[13] E. Bartocci, Y. Falcone, A. Francalanza, M. Leucker, G. Reger, in *Lectures on Runtime Verification - Introductory and Advanced Topics* (2018), Vol. 10457 of *LNCS*, pp. 1–23