Analysing the Performance of Python-based Web Services with the VyPR Framework

Joshua Heneage Dawes[1,2], Marta Han[3,2], Omar Javed[4], 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
[4] Università della svizzera italiana, Lugano, Switzerland

Abstract. In this tutorial paper, we present the current state of VyPR, a framework for the performance analysis of Python-based web services. We begin by summarising our theoretical contributions which take the form of an engineer-friendly specification language; instrumentation and monitoring algorithms; and an approach for explanation of property violations. We then summarise the VyPR ecosystem, which includes an intuitive library for writing specifications and powerful tools for analysing monitoring results. We conclude with a brief description of how VyPR was used to improve our understanding of the performance of a critical web service at the CMS Experiment at CERN.

1 Making a Start with Performance Analysis

Understanding a program's performance precisely is vital, especially when the program performs critical activities or fulfils a wide range of use cases. Analysing the performance of a program involves two key steps: *determination of expected performance*, and *investigation when performance deviates from expectation.*

To address the first step, we must construct an appropriate definition of the *performance* of a program. Such a definition usually depends on the category into which the program fits. For example, the program may perform a lot of computation with heavy memory operations and complex control-flow. In this case, performance may be characterised by the time taken to compute a result, along with paths taken through control-flow. Alternatively, the program in question may communicate frequently over a network, in which case a characterisation of its performance may be in terms of the network's stability and the latency experienced with respect to the data being transferred.

In this work, the outcome of the first step is a specification of the performance expectations, based on a combination of data from previous program runs and engineers' intuition, that can be checked during future program runs. Further, the systems with which we concern ourselves are Python-based web services. This allows us to investigate the performance of critical services employed at the CMS Experiment at CERN. We remark that we do not currently consider performance in the context of parallelism.

To address the second step, we need a way to determine what caused deviations from expected performance. In this work we call this process *explanation*, the definition of which also depends on the type of program being analysed and the domain in which it operates.

This paper summarises our efforts so far to introduce 1) a new body of theoretical work for performance analysis of programs that is based on Runtime Verification and 2) the implementation of a framework based on this theoretical work. The summary is as follows:

– In Section 2, we review existing work to frame our contributions to both the Software Engineering and Runtime Verification (RV) communities.
– In Section 3, we describe our theoretical body of work. We begin with an introduction to our specification language along with efficient instrumentation and monitoring algorithms that it admits [32]. We then introduce our approach for explaining violations of properties written in our specification language [31].
– In Section 4, we present VyPR [33,30], a framework for the performance analysis of Python web services. We developed VyPR at the CMS Experiment at CERN based on our theoretical contributions. VyPR marks a significant effort to establish RV as a software development tool.
– In Section 5, we give details of VyPR's analysis tools [30], which consist of a web application and a Python-based library.
– In Section 6, we give a step-by-step reproduction of an investigation that found interesting performance of critical software at the CMS Experiment.

This paper accompanies a tutorial delivered at the International Conference on Runtime Verification 2020, the materials for which can be found at http://pyvypr.github.io/home/rv2020-tutorial/.

2 Existing Techniques for Performance Analysis

We review well-established profiling techniques, work from the RV community that considers timing behaviour and work from the Software Engineering community on performance data visualisation.

2.1 Profiling

One of the most popular performance analysis techniques for Python, as for most languages, is *profiling*. Profilers record a quantity in which a software engineer is interested, which is usually the time taken by some operation. For example, if a software engineer wanted to understand the behaviour of a function `f` in a Python program, a profiling tool such as `cProfiler` [2] (a *deterministic profiler* [14]) could be attached. The engineer would then have the task of finding the data relevant to the function `f` in the pile of data generated by the profiler. While this search is not too much of a problem (in fact there are tools to help with this [1]), the overhead often is. This is mainly because naive profiling of entire Python programs often relies on *tracing* [13], which involves attaching a function with a specific signature to the Python interpreter and collecting data each time a relevant event is observed. To counter this overhead, *statistical profilers* [5] are sometimes used.

While these approaches are commonly used and well catered for by the standard Python interpreter (CPython [12]), they suffer from drawbacks when one needs to analyse anything other than the time taken by a function each time it is called. If one needs to check whether a constraint holds that relates multiple quantities, offline analysis quickly becomes more difficult and error-prone.

This is one of the first advantages that VyPR brings. That is, VyPR's rich specification language allows measurement and comparison of multiple quantities at runtime, including the time taken by various operations and (properties of) values held by variables. Further, rather than using tracing, VyPR instruments code based on specifications which reduces overhead. Finally, VyPR's offline analysis tools allow engineers to carry out detailed analyses, including inter-procedural analysis, inter-machine analysis and path comparison for determination of root causes of performance problems.

2.2 Runtime Verification for Performance Analysis

The existing work in RV that could be used in performance analysis is necessarily that which considers the timing behaviour of the system being monitored. Since approaches in RV can be characterised, at least in part, by their specification language, we focus on the languages which can reason about time [34].

The collection of specification languages introduced or adapted by the RV community includes temporal logics [40,38,20,15], rule systems [18], stream logics [29], event-processing logics [36] and automata [17,23]. The situations in which each can be used depend on their semantics and expressiveness.

As a first example that can reason about time, we take Metric Temporal Logic (MTL) [38], which extends Linear Temporal Logic (LTL) [40] with intervals attached to modal operators. In MTL one can write a formula such as $\phi \, \mathcal{U}_{[0,5]} \, \psi$ to mean that ϕ should hold for at most 5 units of time until ψ holds. There is also Timed Linear Temporal Logic (TLTL) [20], which extends LTL with operators that allow one to place constraints on how long *since* or *until* some observation was or will be made. Automata are used in settings both with [17] and without [23] time. In other RV work, automata are used only in the synthesis of monitors for specifications written in timed logics [20]. Stream logics, rule systems and event-processing systems are all capable of reasoning about time [29,18,36].

Despite the existing work on timed specification languages, we highlight that those which can reason about time (and are intended for the same setting as our work) tend to have a high level of abstraction with respect to the code being monitored. While this allows specification languages to be more expressive, it creates two problems. First, the software engineer becomes responsible for maintaining a mapping from the program being monitored to the symbols in the specification. Second, it makes interpretation of specifications difficult because a mapping must be considered at the same time.

We have previously called the combination of these problems the *separation problem* [32]. In the same work, we introduced a new specification language, Control-Flow Temporal Logic (CFTL). CFTL formulas are low-level, meaning that they require only the source code of the monitored program to make sense. While formulas must then be changed with the program source code, we remark that this has faced no resistance in practice. Further, we highlight that this fits in with RV's role as a complement to testing in the software development process.

Explanation of property violations has received little attention, though there has been some interest in recent years (our work on explanation of CFTL violations [31]; work on trace edit-distances [41]; explanation in Cyber-Physical Systems [19] and the use of *temporal implicants* [35]). Outside of RV the Fault Localisation [24,25,45,46,43] and Model Checking [22,28,44,16] communities have made significant contributions.

2.3 Performance Data Visualisation

Our experience has shown that effective presentation of data obtained by monitoring at runtime is as important as the specification and monitoring processes themselves. However most existing work on visualising performance data and runtime analytics is done outside RV and focuses on data obtained by methods other than formal verification [21,27,26].

Despite the lack of integration with formal verification, a key trend is the display of performance data in the context of the source code that generated it. The web-based analysis environment that we provide visualises data recorded by VyPR alongside the source code, similarly to contributions from the Software Engineering community. The key difference is that we have not yet looked at displaying performance data directly in the IDE because we have not yet needed to.

3 An Engineer-friendly Specification Language

Our main goal is to provide a framework with which software engineers can easily analyse the performance of their programs with minimal disruption to their existing development process. This gives rise to the requirement that, if we use a formal specification language, such a language should make the transition from natural language to formal specification as smooth as possible. Here, we give a brief description of our language (Control-Flow Temporal Logic (CFTL)) with its semantics; and our instrumentation, monitoring and explanation approaches. For full details, the reader can refer to [31,32,33].

3.1 A Representation of Programs

We consider a program P whose performance we want to analyse, written in a subset of the Python language defined in [32]. Our first step is to introduce

```
x = n
for i in range(n):
  y = i*2
a = y
f()
```

$$\begin{tikzcd}
{} \arrow[r] & {[]_{\sigma_0}} \arrow[d, "e_1"] & \\
& {[x \mapsto \text{changed}]_{\sigma_1}} \arrow[d, "e_2"] & \\
& \text{loop } {[i \mapsto \text{changed}]_{\sigma_2}} \arrow[r, "e_3"] \arrow[d, "e_6"] & {[y \mapsto \text{changed}]_{\sigma_3}} \arrow[l, bend left, "e_4"] \arrow[dl, bend left, "e_5"] \\
& \text{endLoop } {[]_{\sigma_4}} \arrow[d, "e_7"] & \\
& {[a \mapsto \text{changed}]_{\sigma_5}} \arrow[r, "e_8"] & {[f \mapsto \text{called}]_{\sigma_6}}
\end{tikzcd}$$

Fig. 1. A Python program with a for-loop with its symbolic control-flow graph.

a representation of the program P on which both the CFTL semantics and our instrumentation approach are based. We would like engineers to be able to reason about 1) the *states* created during a run of P by instructions such as assignments and function calls; and 2) the *transitions* that occur to move between such states. Since we need to consider instrumentation, which in this work we assume happens strictly before runtime, we need pre-runtime analogues of these states. We call these *symbolic states*. A symbolic state σ is a pair $\langle \rho, m \rangle$ for ρ a unique identifier of the statement in the source code that generated the symbolic state and m a map from each program variable/function to a *status* in $\{\text{undefined}, \text{changed}, \text{unchanged}, \text{called}\}$.

We then represent a program P as a directed graph whose vertices are these symbolic states, with edges representing instructions that could move the program from one state to another. We call such a graph a *symbolic control-flow graph*, denote it by $\mathsf{SCFG}(P)$ and give an example in Figure 1.

3.2 A Representation of Program Runs

We now introduce the structures over which CFTL semantics are defined, that is, our representations of program runs. To help us later with instrumentation, a representation of a run of our program P should be seen as a path through $\mathsf{SCFG}(P)$, the symbolic control-flow graph of P. To this path, we can add information from runtime, such as variable values and timestamps. We do this by augmenting symbolic states to give their runtime analogue, *concrete states*. A concrete state is a triple $\langle \sigma, v, t \rangle$ for σ a symbolic state, v a map from each program variable/function to a value and t a real-numbered timestamp. We can now represent a program run by a sequence of concrete states, which we call a *dynamic run* and denote by $\mathcal{D}$. Further, these dynamic runs have the required property that each concrete state can be associated with a symbolic state, meaning that each dynamic run forms a path through a symbolic control-flow graph.

```
authenticated = check_authentication()
if authenticated:
  in = get_auth_data()
  results = query("...", in)
  if not results:
    query("...", in)
else:
  in = get_non_auth_data()
  results = query("...", in)
```

Fig. 2. A sample Python program.

3.3 Control-Flow Temporal Logic

Control-Flow Temporal Logic (CFTL) is a linear time, temporal logic designed for writing specifications over low-level program behaviour (at the line-of-code level). We achieve this by defining its semantics over dynamic runs, which are detailed representations of program runs. CFTL formulas are in prenex normal-form, so consist of two parts: the quantifiers, and the inner-most quantifier-free formula. Finally, when a dynamic run $\mathcal{D}$ satisfies a CFTL formula φ, we write $\mathcal{D} \models \varphi$.

For brevity, instead of giving the full syntax, which can be found in [32], we consider the sample program in Figure 2 and give examples of CFTL formulas that express properties over this program.

A simple constraint over call durations. We can express the constraint that no call to `query` should ever take more than 2 seconds by writing

$$\forall c \in \mathsf{calls}(\texttt{query}) : \mathsf{duration}(c) \leq 2$$

In this case, the quantifier $\forall c \in \mathsf{calls}(\texttt{query})$ identifies all pairs of concrete states in a dynamic run such that, in the second concrete state, the function `query` has just been called. We call c a *variable* and $\mathsf{calls}(\texttt{query})$ a *predicate*. The quantifier-free part of the formula then uses CFTL's $\mathsf{duration}$ operator. This operator takes the difference of the timestamps associated with the pair of concrete states identified by the quantifier. The final step is to assert that the observed duration is less than or equal to 2.

Place a constraint, but only in a certain case. One might want to constrain the time taken by every call to `query` in the future, but *only* if `authenticated` was set to `True`. To capture this property, one could write

$$\forall s \in \mathsf{changes}(\texttt{authenticated}) : \forall c \in \mathsf{future}(s, \mathsf{calls}(\texttt{query})) :$$
$$s(\texttt{authenticated}) = \texttt{True} \implies \mathsf{duration}(c) < 2$$

To form the second quantifier, we use CFTL's future operator to identify all occurrences of an event after whatever is identified by the preceding quantifier.

In this case, for each change of the variable `authenticated`, every future call of the function `query` after that will be identified. Then, for each pair formed by this process, the quantifier-free part of the formula will be evaluated.

Place a constraint on the time between points. Consider the constraint that the time taken to reach the next state in which `results` has been changed after the change of `authenticated` should be less than a certain value. With CFTL, one could write

$$\forall s \in \mathsf{changes}(\mathtt{authenticated}) : \mathsf{timeBetween}(s, \mathsf{next}(s, \mathsf{changes}(\mathtt{results}))) < 1$$

In this formula, we quantify over changes of `authenticated` and then, in the quantifier-free part, we introduce two new features of CFTL. The first is the `next` operator, which takes a variable along with a predicate and gives the next part of the dynamic run being considered which satisfies the predicate. The second new idea is the timeBetween operator, which takes two parts of the dynamic run and measures the time between them.

Place a constraint relating two measurements. Often, the time taken by some function call relates to the size of its input. For example, one might want to express the requirement that each call of the function `query` takes no more than 0.8 times the length of its input variable `in`. Then this could be expressed in CFTL by

$$\forall c \in \mathsf{calls}(\mathtt{query}) : \mathsf{duration}(c) \leq \mathsf{length}(\mathsf{source}(c)(\mathtt{in})) \times 0.8$$

3.4 Instrumentation

Consider the situation in which a monitoring algorithm must decide whether the formula $\forall c \in \mathsf{calls}(\mathtt{query}) : \mathsf{duration}(c) \leq 2$ holds or not, given a dynamic run generated by the program in Figure 2. Such a dynamic run would contain much more information than is needed to decide whether this formula holds. Hence, when the monitoring algorithm tries to process the dynamic run, it would first have to decide whether each concrete state was useful. In this case, most of the information in the dynamic run would be useless, and a large amount of the work done by the monitoring algorithm would be to conclude as such.

Our instrumentation approach involves 1) ensuring that the dynamic run extracted from the running program includes only the concrete states relevant to a formula by determining a subset of *critical symbolic states* before runtime; and 2) speeding up certain lookup processes that monitoring must perform when it *has* observed something that it needs.

Ensuring extraction of a conservative dynamic run is achieved with two steps. First, we inspect the quantifiers to determine *points of interest* in the program, and then we inspect the quantifier-free part of the formula to determine where measurements need to be taken. As an example, consider again the specification

$$\forall s \in \mathsf{changes}(\mathtt{authenticated}) : \mathsf{timeBetween}(s, \mathsf{next}(s, \mathsf{changes}(\mathtt{results}))) < 1$$

Determining points of interest. We first inspect the quantifiers and see that, based on $\forall s \in$ `changes`(`authenticated`), we need to find all symbolic states in the symbolic control-flow graph in which the variable `authenticated` has just been changed. In this case, we treat `authenticated` as a local primitive-type variable, so we do not consider side-effects from impure functions.

Determining instrumentation points. Once the set of points of interest has been derived, we inspect the quantifier-free part of the formula. In this case, we have a single constraint over multiple quantities that must be measured at runtime. In inspecting the timeBetween operator, we see that the first argument is the variable s, which refers to a concrete state in which the variable `authenticated` has just been changed. Hence, s refers directly to the points of interest that we have already determined, so we include these in our set of critical symbolic states.

Next, we have next(s, changes(`results`)), which refers to the next concrete state in which `results` was changed after `authenticated` was changed. Since when we perform instrumentation we often cannot be sure of the branches taken at runtime, we must follow the symbolic control-flow graph from each of the points of interest to determine each *possible* next change of `results`.

Correctness. Informally, instrumentation is correct when the dynamic run without instrumentation satisfies a CFTL formula if and only if the dynamic run with only concrete states included by instrumentation also satisfies it [32].

3.5 Monitoring

The problem that we must solve in monitoring is that of determining whether a dynamic run satisfies a CFTL formula, given incremental updates to the dynamic run (otherwise known as online monitoring).

Our monitoring approach is helped significantly by instrumentation, in that the input dynamic run is both reduced and structured by it. In particular, our instrumentation approach involves constructing a tree that allows our monitoring algorithm to quickly determine in what way some information from a dynamic run with instrumentation is relevant to the property being monitored [33].

When checking a dynamic run for satisfaction of a formula, our monitoring algorithm maintains an and-or formula tree (with extra structure to cope with CFTL formulas) for every part of the dynamic run that is identified by the formula's quantifiers. As more of the dynamic run is processed, the relevant formula trees are updated until they collapse to a truth value. The result of monitoring is then, as expected, the conjunction of these truth values. In the case that a formula tree was not collapsed because the dynamic run did not contain enough information, a number of policies could be implemented, though in practice we opt for discarding such formula trees.

3.6 Explanation by Path Comparison

We now summarise the extension of our work on CFTL to explaining why runs of a program did not satisfy a property, addressing the need identified in [42]. The full details are given in [31].

In that work, we first highlighted that dynamic runs with instrumentation applied may have the problem that their precise path through a symbolic control-flow graph can no longer be determined. Our solution to this was to modify our instrumentation process to ensure that enough concrete states are included in dynamic runs to allow the precise path taken to be reconstructed.

Once we can obtain the exact program path taken by a dynamic run in the form of a sequence of edges from a symbolic control-flow graph, our key contribution is a formalism for comparison of multiple paths. We assume that these dynamic runs generated different verdicts with respect to a CFTL property.

To enable comparison, we introduced a new technique that identifies points of disagreement in sets of paths and enables comparison of how the paths disagreed. To achieve this, we represent the regions over which paths disagree by *path parameters* and say that each path gives each of these parameters a *value* with the sub-path that it took in that region.

Finally, we extended this comparison of paths to take full advantage of CFTL's semantics. This extension enables analyses such as comparison of paths between two statements at which measurements were taken for a specification. Ultimately, this paves the way for identification of problematic regions of code based on monitoring results.

4 Translation into a Software Development Tool

The theory summarised in Section 3 has been used to implement VyPR, a framework for the performance analysis of Python programs (most often, web services) along with an ecosystem of supporting tools. VyPR provides facilities for specification of program performance, automatic instrumentation with respect to specifications, efficient online monitoring and in-depth offline analysis. More information, along with documentation, can be found at http://pyvypr.github.io/home/.

The discussion in this section draws on experience from addressing the challenges encountered when implementing VyPR. These challenges can be divided into two categories:

– Determining how to implement a tool based on the theory given so far that enables specification and then performs instrumentation and monitoring automatically.
– Determining how to enable software engineers to benefit as much as possible from the resulting tool.

4.1 A Python Library for CFTL

The first step a software engineer takes when using VyPR is writing their specifications. The process of a software engineer writing a formal specification is that of translation from a natural language specification to its equivalent in the specification language. To facilitate this process, we opted to build a Python library, PyCFTL, with which engineers can write their specifications and supply them to VyPR. To demonstrate PyCFTL, we consider the CFTL specification

$$\forall s \in \mathsf{changes}(\mathtt{x}) : \mathsf{timeBetween}(s, \mathsf{dest}(\mathsf{next}(s, \mathsf{calls}(\mathtt{query})))) < 1$$

and construct it in Python code using the PyCFTL library

```
1 Forall(s = changes("x")).\
2 Check(lambda s : (
3   timeBetween(s, s.next_call("query").result()) < 1
4 ))
```

Here, we see two key translations:

Quantifiers. The single quantifier is built at line 1 by instantiating the `Forall` class using Python's keyword argument feature to introduce the variable `s`. The predicates implemented in the PyCFTL library match those defined in the CFTL semantics, namely changes and calls.

Constraints. The constraint to be checked at each point of interest identified by the quantifier is defined between lines 2 and 4 by passing a *lambda* expression to the method `Check` defined on the `Forall` instance. The arguments of this lambda expression must match the variables bound by the quantifiers. Hence, in this case, our lambda has the single argument `s`.

The constraint definition also includes one of the key features of PyCFTL: variables bound by quantifiers contain objects that can be treated like the events they represent. For example, using the variable `s` that holds a state, we can refer to the next call along using the method `next_call`. We can then get the state immediately after that call using the method `result`.

In Figure 3 we give a further example, which demonstrates more of the PyCFTL library. Once a specification is written, the next concern to address is how to deal with Python software that contains multiple modules and classes. In the current implementation of VyPR, we wrap PyCFTL in a layer that allows software engineers to indicate to which *individual functions* in their code a given specification should apply. Figure 4 gives an example.

In Figure 4, lines 2-8 give an example of *explicit indication* and lines 10-14 give an example of *indication using compilation*. *Explicit indication* involves the software engineer manually telling VyPR which package, module and then either 1) function or 2) class/method to instrument and monitor.

This approach to specification gives rise to the problem that software engineers may not know precisely where in their system they would like to measure something. To ease the situation, we have begun development of a set of helper classes to enable *indication using compilation*. In practice, this means that, when a software engineer uses a helper class such as `Functions`, VyPR will inspect the symbolic control-flow graph of each function in a system and select it for

$$\begin{aligned}&\forall s \in \mathsf{changes}(\texttt{auth}) :\\ &\forall c \in \mathsf{future}(s, \mathsf{calls}(\texttt{query})) :\\ &\quad s(\texttt{authenticated}) = \texttt{True} \implies \mathsf{duration}(c) < 2\end{aligned}$$

```
Forall(s = changes("auth")).\
Forall(c = calls("query", after="s")).\
Check(lambda s, c : (
  If(s("auth").equals(True)).then(c.duration() < 2)
))
```

Fig. 3. Another example of the PyCFTL specification library.

```
{
  "package.module": {  # explicit indication
    "class.method": [
      Forall(c = calls("func")).Check(
        lambda c : c.duration() < 1
      )
    ]
  },
  # indication using compilation
  Functions(containing_change_of="threshold"): [
    Forall(s = changes("threshold")).Check(
      lambda s : s("threshold") < 10
    )
  ]
}
```

Fig. 4. An example of how to tell VyPR where to instrument and monitor for a specification.

monitoring if it fulfils the criteria given as an argument. Since system-wide static analysis is an obvious source of inefficiency, part of our development efforts involve caching results to mitigate the situation.

4.2 Instrumentation and Monitoring

The instrumentation and monitoring algorithms implemented by VyPR are those described in Sections 3.4 and 3.5 respectively, extended to the setting of a system with multiple functions written across multiple files.

Instrumentation. This algorithm consists of inserting code in places appropriate for the specification given so that a dynamic run can be generated at runtime. To achieve this, a specification built using PyCFTL generates a structure in memory that VyPR can use, along with the symbolic control-flow graph of each relevant function in a system, to determine where to place new code. Placement of the code is performed statically by 1) adding new entries to the abstract syntax tree (easily obtained via Python's standard library) for each instrumentation point; 2) compiling to bytecode and 3) renaming the original source code file. This renaming prevents Python's default behaviour of recompiling bytecode when disparities between it and source code are detected.

The exact instrumentation code placed depends on the specification given. For example, if the specification calls for the measurement of a duration, VyPR will place code that records timestamps and computes their difference. The communication between instrumentation code and the monitoring algorithm is performed via a queue, since the monitoring algorithm runs asynchronously from the monitored program. For more information on how VyPR performs instrumentation, see our previous work [33].

Monitoring. This algorithm is responsible for processing the dynamic runs generated by instrumentation code, generating verdicts and sending the data to a central server for later analysis. So far, we have not needed our monitoring algorithm to feed verdict information back into the running program, so our implementation is asynchronous.

Since we work with Python, we face the challenge of the Global Interpreter Lock (GIL) [6], the mechanism used to make the Python interpreter thread-safe. Its major disadvantage is that threads running code that is not IO-heavy are not run asynchronously because the GIL is based on mutual exclusion. However, threads with IO activity can run asynchronously because the GIL is released for IO. Since we developed VyPR primarily for web services, the significant IO activity required for reading incoming requests, writing outgoing responses and performing database queries release the GIL to allow VyPR to run.

If one were to consider applying VyPR elsewhere, we have already highlighted that the current implementation of VyPR can generate high overhead if there is more work being done by monitoring than by the monitored program [32]. However, either by using multiprocessing or existing techniques for bypassing the GIL [3], we can remove this problem.

4.3 Storing Verdicts

Once our monitoring implementation generates verdicts, it sends them, along with a collection of other data, to a central repository to be stored in a relational database schema. This central repository of data, called the *verdict server*, is a separate system in itself with which VyPR communicates via HTTP. The server is responsible for:

- Storing verdicts, along with the measurements taken to reach those verdicts and paths taken by monitored functions.
- Providing an API that VyPR uses to send all of the data collected during instrumentation and monitoring.

– Providing APIs for our analysis tools (see Sections 5.1 and 5.2).
– Serving our web-based analysis environment (see Section 5.1).

The server is implemented in Python using the Flask [10] framework with SQLite [8] as a backend database. The relational database is designed to store the data generated by VyPR at runtime compactly, meaning that our analysis tools are designed to enable straightforward access of data stored in a non-trivial schema. Finally, while we have plans to move away from SQLite to use a more robust database system, SQLite has met our needs so far.

5 Analysing Monitoring Results

An area that has received little attention from the RV community is the analysis of results of program monitoring. Our experience with VyPR has shown that the analysis tools provided by a program analysis framework are as important as the specification tools.

This is especially true when one is attempting to help software engineers explain violations of their specifications. In this direction, we have found that the way in which the results of our explanation work described in Section 3.6 are presented is just as important as the way in which the explanations are derived.

We now describe the analysis tools in the VyPR ecosystem, which include 1) a web-based environment for visual inspection of monitoring results; and 2) a Python library for writing analysis scripts. The aim of these tools is to make investigation of monitoring data straightforward and facilitate engineers in finding explanations of violations of their specifications. Details on usage can be found at https://pyvypr.github.io/home/use-vypr.html.

5.1 A Web-based Analysis Environment

Since inspection of some data generated by VyPR lends itself best to a graphical user interface, part of the VyPR ecosystem is a web application. The application enables sophisticated, in-depth inspection to be performed with just a few clicks in the web browser. We now present key features.

Explicit links between specifications and source code. In our analysis environment, we allow engineers to select parts of their specifications to focus on relevant parts of their source code. This allows understanding from the engineer of precisely which part of their code they are analysing.

The performance of individual statements. Similarly to existing work on performance data visualisation performed by the Software Engineering community [21,27,26], our analysis environment displays the data from monitoring in the context of the relevant code. In fact, results from monitoring for specifications written in CFTL lend themselves easily to such a representation because of CFTL's low-level nature, meaning measurements can be linked directly to lines of source code.

Separating performance data by verdict. When viewing a plot of measurements taken during monitoring, it is reasonable for an engineer to want to see only the measurements that generated either satisfaction or violation. With the analysis environment, engineers can plot measurements and then filter them by whether the specification being monitored was satisfied or violated.

Program path highlighting. Given the path reconstruction approach described in Section 3.6, the analysis environment can highlight paths in code up to and between measurements. It can also highlight paths taken by whole function calls.

Since it is expected that performance data is visualised across many function calls, we use colour-coding to highlight source code based on how well a specification was satisfied or how severely it was violated on average.

Without the analysis environment, these tasks would require in-depth knowledge of the schema in which data is stored and non-trivial post-processing.

5.2 A Python Analysis Library

For the cases in which the analysis environment is not suitable, we built a Python library. This library helps software engineers to automate their inspection of performance data by writing scripts.

In order to make writing scripts as intuitive as possible, the library is designed similarly to an Object-Relational Mapping (ORM) [7,39]. That is, the library provides classes that mirror tables in a relational schema so that each row in a table becomes an instance of the table's class. Methods defined on these instances then reflect either foreign key relationships between tables, or perform more complex queries. In practice, rather than operating directly on a database, the library communicates with the verdict server via the server's analysis API.

In addition to the ORM, the analysis library provides facilities for post-processing of data generated by VyPR. These facilities greatly simplify otherwise complex tasks, which we now describe.

Comparison of Program Paths. Each time a function monitored by VyPR is executed, program path information is generated by additional instrumentation. It is then coupled with the set of verdicts. Each measurement taken during monitoring is mapped to this path information, so paths up to and between measurements can be reconstructed.

The program path information generated is a sequence of integers, each of which being the unique ID of a more complex path condition. Measurements are mapped into this sequence of integers via a pair consisting of an *offset* and a *reaching path length*, both also integers. The offset identifies the prefix of the sequence whose path conditions lead to the branch on which the measurement was taken. The reaching path length tells us how far along this branch to traverse before the precise statement at which the measurement was taken is reached.

The algorithm that performs this transformation (described in [32]) is integrated with the ORM. The result is that 1) objects representing *function calls* define methods to reconstruct the entire path taken by those calls; and 2) objects representing *measurements* define methods that reconstruct paths up to those measurements. Classes are also provided to help identify where paths disagree and how, using the notion of *path parameters* described in Section 3.6.

Construction of Call Trees. If multiple functions in a system are monitored by VyPR, enough information about each function call is stored that the call tree can be partially reconstructed (it will be missing calls of any function that is not monitored). Since VyPR associates measurements with the function call during which they were taken, it is also possible to reconstruct the call tree occurring *during* a measurement (if the measurement is over a function call).

Path Comparison with Call Trees. Suppose that a function `f` calls a function `g`, both of which are monitored by VyPR. Suppose further that one of the specifications written over `f` is the following

```
Forall(c = calls("g")).Check(lambda c : c.duration() < 1)
```

Then, since path information exists for `g`, we can use comparison of the paths taken through `g` to explain the verdicts generated by `f`. The analysis library makes this straightforward again by implementing the notion of *path parameters*.

Call Trees over Multiple Machines. If programs on multiple machines are monitored by VyPR, with both instances of VyPR pointing to the same verdict server and synchronised via some global clock such as NTP [4], monitoring results from each machine can be combined. In particular, the analysis library's call tree reconstruction algorithm can detect when a function call from one machine took place during a function call on another. The advantage of this that we have so far used is that one can use comparison of the paths traversed on one machine to explain measurements on another.

6 VyPR in Practice

We conclude with a description of our operational experience with VyPR at the CMS Experiment [9] at the LHC [11] at CERN. We discuss our experience analysing a system to which VyPR has been applied extensively, resulting in a detailed understanding of its performance. More detail on the investigations presented here can be found in our tutorial material.

6.1 Finding Performance Problems

Our case study is a web service used at the CMS Experiment for uploading certain data needed in physics analyses to a central database. The service consists of a client program and a server; the client is responsible for reading in and performing some initial checks on data, and the server is responsible for final checks and all database operations. The investigation that we describe here is purely on the server-side, but we have used VyPR's analysis facilities to see how the two components of the service behave together. To generate the data that we use here, we replayed 4000 instances of data upload from a repository that we have been building up over the past 3 years of data collection at the LHC.

Defining Specifications. We consider a function that is responsible for performing some checks which vary depending on the existing state of the target database. Hence, there are two branches that can be taken and then a final database query. We need to investigate the time taken to get from the start of these branches, to the return of the database query at the end. Our specification for this is

```
Forall(q = changes("tag")).\
Forall(c = calls("commit", after="q")).\
Check(lambda q, c : timeBetween(q, c.result()) < 1.2)
```

This specification will actually lead to VyPR recording more data than we need because of the structure of the monitored code. In our analysis script, we deal with this.

We also need to measure the time taken by some key functions that are called on the paths of interest. VyPR associates measurements with the statements in code from which they were taken, so we write a specification placing an arbitrary constraint over the function calls that we care about. It is then easy to get the measurements that were taken along relevant paths in our monitored function.

Analysing Results. We give a brief description of how we used our analysis library to write a script which generated the plot shown in Figure 5. The script contains the following steps:

1. Select the function in the system that we need to analyse. For that function, choose the specifications.
2. Since our main specification contains two quantifiers, select the two statements in code between which we will be comparing paths.
3. Get all measurements of the time taken for runs of the function to get between the two statements that we have selected, such that the specification was violated. Additionally get all measurements of function call durations taken on the relevant paths.

Once all measurements have been obtained, we still need to separate them by which path was taken. To do this, we use the analysis library's implementation of *path parameters.* This involves:

1. *Intersecting* all of the paths that we obtain between the two statements we selected.
2. Finding the single point (in this case) where the paths disagree, which is represented by a *path parameter.*
3. Finding the *value* given to this path parameter by each path, ie, the sub-path taken in the region of disagreement.

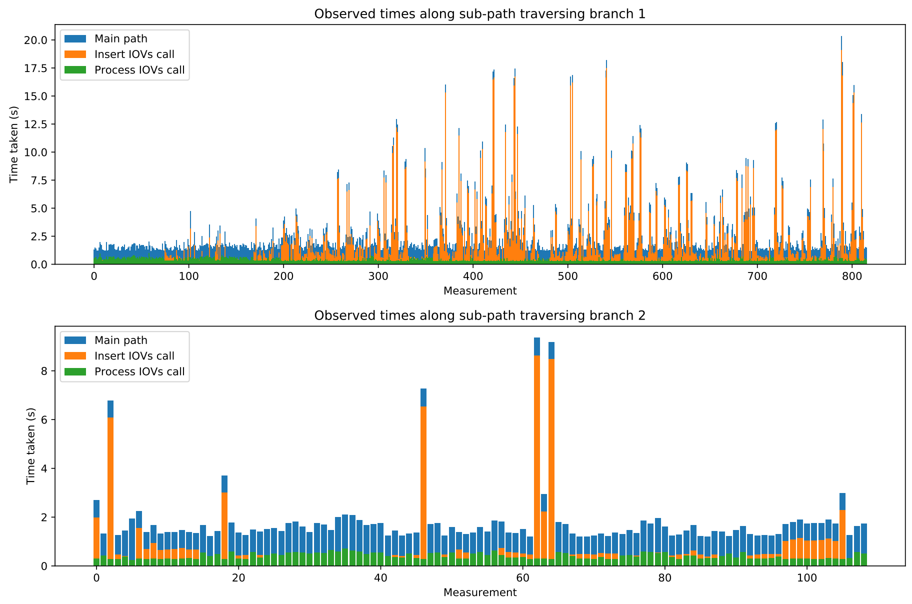

Fig. 5. The plot we built with our analysis library to show the range of time measurements for violations along two distinct paths through some source code.

It is then straightforward to group the measurements by the path that was taken, allowing Figure 5 to be divided with respect to the two possible sub-paths found. It is clear from these plots that one path taken in the region of disagreement resulted in far worse performance. Further, from the inclusion of the times taken by the two key function calls on those paths, we see that one of the two tends to be to blame for the peaks.

Ultimately, the results of this investigation and others will be fed into the development process at CMS to help consolidate the service. This will be helped by our work on applying VyPR during Continuous Integration [37].

7 Future Work

There are many directions for future work, which we divide into the following categories:

- *Instrumentation.* There are clear opportunities for improvement of VyPR's instrumentation process.
- *Analysis environment improvements.* We aim to minimise the need for the analysis library.
- *More expressive specifications.* Classes of properties such as those written over multiple functions currently cannot be expressed directly in CFTL.

– *Performance analysis in Continuous Integration.* Work is ongoing on extending VyPR to combine functional testing with performance analysis [37].

8 Conclusion

In this paper, we described the current state of VyPR, a framework for the performance analysis of Python web services. We summarised our theoretical contributions, which include an engineer-friendly specification language, along with instrumentation, monitoring and explanation strategies. We continued by describing the implementation of the VyPR framework, while we also discussed the challenges of making such a framework as useful as possible for software engineers. We introduced our analysis tools which mark a significant contribution to the area of analysing monitoring results. Finally, we summarised an investigation that improved our understanding of a critical service running at the CMS Experiment at CERN.

References

1. Call Graphs for Python. https://github.com/jrfonseca/gprof2dot
2. `cProfiler`. https://docs.python.org/2/library/profile.html#module-cProfile
3. Extending Python with C. https://docs.python.org/2.7/extending/extending.html
4. Network Time Protocol. http://www.ntp.org
5. `pyinstrument`. https://github.com/joerick/pyinstrument
6. Python's GIL. https://wiki.python.org/moin/GlobalInterpreterLock
7. SQLAlchemy for Python. https://www.sqlalchemy.org
8. SQLite. https://www.sqlite.org/index.html
9. The CMS Experiment at CERN. http://cms.cern
10. The Flask Framework. https://flask.palletsprojects.com/en/1.1.x/
11. The LHC. https://home.cern/science/accelerators/large-hadron-collider
12. The Python Programming Language. https://github.com/python/cpython
13. `trace` - the Python tracing tool. https://docs.python.org/2/library/trace.html
14. What is Deterministic Profiling? https://docs.python.org/2.4/lib/node453.html
15. Alur, R., Etessami, K., Madhusudan, P.: A temporal logic of nested calls and returns. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 467–481. Springer (2004)
16. Ball, T., Naik, M., Rajamani, S.K.: From symptom to cause: Localizing errors in counterexample traces. In: Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 97–105. POPL '03, Association for Computing Machinery, New York, NY, USA (2003). https://doi.org/10.1145/604131.604140, https://doi.org/10.1145/604131.604140
17. Barringer, H., Falcone, Y., Havelund, K., Reger, G., Rydeheard, D.: Quantified event automata: Towards expressive and efficient runtime monitors. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012: Formal Methods. pp. 68–84. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
18. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Rule-Based Runtime Verification. In: Steffen, B., Levi, G. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 44–57. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)

19. Bartocci, E., Manjunath, N., Mariani, L., Mateis, C., Ničković, D.: Automatic failure explanation in cps models. In: Ölveczky, P.C., Salaün, G. (eds.) Software Engineering and Formal Methods. pp. 69–86. Springer International Publishing, Cham (2019)
20. Bauer, A., Leucker, M., Schallhart, C.: Runtime Verification for LTL and TLTL. ACM Trans. Softw. Eng. Methodol. **20**(4), 14:1–14:64 (2011). https://doi.org/10.1145/2000799.2000800, https://doi.org/10.1145/2000799.2000800
21. Beck, F., Moseler, O., Diehl, S., Rey, G.D.: In situ understanding of performance bottlenecks through visually augmented code. In: 2013 21st International Conference on Program Comprehension (ICPC). pp. 63–72 (2013)
22. Beer, I., Ben-David, S., Chockler, H., Orni, A., Trefler, R.: Explaining counterexamples using causality. Formal Methods in System Design **40**(1), 20–40 (Feb 2012). https://doi.org/10.1007/s10703-011-0132-2, https://doi.org/10.1007/s10703-011-0132-2
23. Bensalem, S., Bozga, M., Krichen, M., Tripakis, S.: Testing conformance of real-time applications by automatic generation of observers. Electronic Notes in Theoretical Computer Science **113**, 23 – 43 (2005). https://doi.org/https://doi.org/10.1016/j.entcs.2004.01.036, http://www.sciencedirect.com/science/article/pii/S157106610405251X, proceedings of the Fourth Workshop on Runtime Verification (RV 2004)
24. Birch, G., Fischer, B., Poppleton, M.R.: Fast model-based fault localisation with test suites. In: Blanchette, J.C., Kosmatov, N. (eds.) Tests and Proofs. pp. 38–57. Springer International Publishing, Cham (2015)
25. Christakis, M., Heizmann, M., Mansur, M.N., Schilling, C., Wüstholz, V.: Semantic fault localization and suspiciousness ranking. In: Vojnar, T., Zhang, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 226–243. Springer International Publishing, Cham (2019)
26. Cito, J., Leitner, P., Rinard, M., Gall, H.C.: Interactive production performance feedback in the ide. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). pp. 971–981 (2019)
27. Cito, J., Oliveira, F., Leitner, P., Nagpurkar, P., Gall, H.C.: Context-based analytics - establishing explicit links between runtime traces and source code. In: 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP). pp. 193–202 (2017)
28. Clarke, E., Jha, S., Lu, Y., Veith, H.: Tree-like counterexamples in model checking. pp. 19– 29 (02 2002). https://doi.org/10.1109/LICS.2002.1029814
29. D'Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H.B., Mehrotra, S., Manna, Z.: Lola: Runtime Monitoring of Synchronous Systems. In: 12th International Symposium on Temporal Representation and Reasoning (TIME'05). pp. 166–174. IEEE Computer Society Press (June 2005)
30. Dawes, J.H., Han, M., Reger, G., Franzoni, G., Pfeiffer, A.: Analysis Tools for the VyPR Framework for Python. In: International Conference on Computing in High Energy and Nuclear Physics, Adelaide, Australia 2019 (2019)
31. Dawes, J.H., Reger, G.: Explaining Violations of Properties in Control-Flow Temporal Logic. In: Runtime Verification, Porto, Portugal 2019 (2019)
32. Dawes, J.H., Reger, G.: 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. pp. 2206–2214 (2019). https://doi.org/10.1145/3297280.3297497, https://doi.org/10.1145/3297280.3297497

33. Dawes, J.H., Reger, G., Franzoni, G., Pfeiffer, A., Govi, G.: 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. pp. 98–114 (2019). https://doi.org/10.1007/978-3-030-17465-1_6, https://doi.org/10.1007/978-3-030-17465-1_6
34. Falcone, Y., Krstić, S., Reger, G., Traytel, D.: A taxonomy for classifying runtime verification tools. In: Colombo, C., Leucker, M. (eds.) Runtime Verification. pp. 241–262. Springer International Publishing, Cham (2018)
35. Ferrère, T., Maler, O., Ničković, D.: Trace diagnostics using temporal implicants. In: Finkbeiner, B., Pu, G., Zhang, L. (eds.) Automated Technology for Verification and Analysis. pp. 241–258. Springer International Publishing, Cham (2015)
36. Hallé, S., Varvaressos, S.: A formalization of complex event stream processing. vol. 2014, pp. 2–11 (09 2014). https://doi.org/10.1109/EDOC.2014.12
37. Javed, O., Dawes, J.H., Han, M., Franzoni, G., Pfeiffer, A., Reger, G., Binder, W.: PerfCI: A Toolchain for Automated Performance Testing during Continuous Integration of Python Projects. In: International Conference on Automated Software Engineering 2020. p. to appear (2020)
38. Koymans, R.: Specifying real-time properties with metric temporal logic. Real-Time Systems **2**(4), 255–299 (Nov 1990). https://doi.org/10.1007/BF01995674, https://doi.org/10.1007/BF01995674
39. Papadopoulos, I., Chytracek, R., Duellmann, D., Govi, G., Shapiro, Y., Xie, Z.: Coral, a software system for vendor-neutral access to relational databases pp. 495–499 (04 2006). https://doi.org/10.1142/9789812773678_0082
40. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science (sfcs 1977). pp. 46–57 (Oct 1977). https://doi.org/10.1109/SFCS.1977.32
41. Reger, G.: Suggesting edits to explain failing traces. In: Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings. pp. 287–293 (2015)
42. Reger, G.: A report of rv-cubes 2017. In: Reger, G., Havelund, K. (eds.) RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools. Kalpa Publications in Computing, vol. 3, pp. 1–9. EasyChair (2017). https://doi.org/10.29007/2496, https://easychair.org/publications/paper/MVXk
43. Reiter, R.: A theory of diagnosis from first principles. Artificial Intelligence **32**(1), 57 – 95 (1987). https://doi.org/https://doi.org/10.1016/0004-3702(87)90062-2, http://www.sciencedirect.com/science/article/pii/0004370287900622
44. Riacs, W.V., Groce, A., Groce, A., Visser, W., Visser, W.: What went wrong: Explaining counterexamples. In: In SPIN Workshop on Model Checking of Software. pp. 121–135. Springer-Verlag (2002)
45. de Souza, H.A., Chaim, M.L., Kon, F.: Spectrum-based software fault localization: A survey of techniques, advances, and challenges. CoRR **abs/1607.04347** (2016), http://arxiv.org/abs/1607.04347
46. Wotawa, F., Stumptner, M., Mayer, W.: Model-based debugging or how to diagnose programs automatically. In: Proceedings of the 15th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems: Developments in Applied Artificial Intelligence. pp. 746–757. IEA/AIE '02, Springer-Verlag, London, UK, UK (2002), http://dl.acm.org/citation.cfm?id=646864.708248