Instrumenting Python with Kieker

Serafim Simonov
Kiel University
stu126367@mail.uni-kiel.de

Thomas F. Düllmann
HITeC e.V.
duellmann@hitec-hamburg.de

Reiner Jung
Kiel University
reiner.jung@email.uni-kiel.de

Sven Gundlach
Kiel University
sven.gundlach@email.uni-kiel.de

Abstract

Python has become a widely used programming language in big data, machine learning, and scientific modeling. In all these domains, performance is a key factor to success and requires the ability to understand the runtime behavior of software. Therefore, we ported Kieker monitoring to Python and evaluated different approaches to introduce probes into code.

In this paper, we evaluate these approaches, show their benefits and limitations and provide a performance evaluation of the Kieker 4 Python framework.

1 Introduction

Python is a versatile programming language that supports object-oriented, functional, and imperative programming. It has mostly been used in desktop applications, but in recent years it has become a widely used in application in areas such as data analysis, big data, machine learning, and scientific modeling. It can be used as a programming language and as a host language for frameworks implemented in C or other compiled languages. Due to the script-like nature of Python, it allows users to integrate specific frameworks quickly and interactively. Tools, like Jupyter [6], are built upon these abilities and are widely used.

Performance and understanding software behavior are key issues in these application domains. This cannot be determined by means of static analysis. Therefore, we have to instrument the source code to be able to collect traces. Kieker [1, 8] already provides the basic infrastructure for collecting traces, but the seamless introduction of probes in Python is limited.

While Python provides facilities to weave and decorate methods and functions on its own, these features can also be used by other programmers in their software for their code. This affects how we can introduce instrumentation and can, in some cases, limit our ability to instrument and decorate.

In this paper, we discuss different working approaches for instrumentation, show their limits and application. In addition, we present preliminary performance measurements of our instrumentation based on the micro benchmark MooBench [3, 5].

2 Case Studies

To cover a diverse set of Python code, we used the Spyder cross-platform software development IDE and TensorFlow, a Python machine learning framework.

Due to their different structure and design, they both constitute unique challenges in terms of application monitoring and inserting instrumentation.

Spyder is an open source IDE for Python that is written in Python. It was the first multi-threaded real world application that we used to test a non-intrusive instrumentation approach, using post import weaving.

As Spyder is a desktop application, it produces one huge trace when the whole application is instrumented. Thus, we used the new architecture visualization of Kieker that does not rely on trace reconstruction, but only on calls.

TensorFlow (TF) is a framework for machine learning applications that are written in Python [4]. While the machine learning community is interested in the accuracy and other properties of the prediction performance, we want to learn more about the application performance. Therefore, we observe TF at runtime. Therefore, we utilized the Kieker Python probe and instrumentation techniques.

Unfortunately, the post import technique can only instrument parts of TF, as TF uses itself weaving and manipulates the Python Dictionary. The pre import technique provides a more comprehensive instrumentation of TF, but still has limitations.

While TF as a whole is rather difficult to instrument, we were able to get some results regarding the Keras module already, which is part of TF. As a result, we were able to extract first component dependency graphs from the Kieker traces. Due to size limitations, we omit the graphics here.

3 Instrumenting Python

Python is a very flexible and versatile programming language. It supports a range of object-oriented programming patterns that can be used to realize weaving of monitoring probes. The same facilities can also be used by others. Thus, there can be conflicts and

obstacles that limit our ability to utilize these facilities in any scenario. Therefore, we implemented two different automatic instrumentation approaches and provide the option to apply instrumentation manually for cases both automatic approaches fail.

In the following subsections, we introduce the manual instrumentation, as well as two automatic instrumentation methods based on the decorator pattern.

3.1 Manual Instrumentation

We can use the Kieker 4 Python API to instrument the Python code manually. The writing of the records is controlled by `SingleMonitoringController` class, that must be instantiated once in the beginning of the program. We can collect and store the information in record objects, e.g. `BeforeOperationEvent`. The invocation `new_monitoring_record()` with the record object passed as a parameter triggers internally a routine that writes the records either directly in to a local file or sends them via TCP to a remote receiver. Which writing method is used depends on the Kieker 4 Python configuration.

3.2 Post Import Weaving

Post import weaving is an approach discussed in literature [2] and used by other developers [7]. Here, functions are replaced by their decorated counterparts. The `PostImportFinder` collects modules for instrumentation and utilizes a specific module loader to load the module and modify the functions in them.

While this approach allows to instrument functions seamlessly, there are limitations. The code of the instrumented software might also employ this approach to monitor security violations [7]. Thus, modules and functions might not get instrumented as intended. During the instrumentation, the original object and its information gets lost, which could affect the execution of the software. This caused issues when instrumenting TensorFlow.

3.3 Pre Import Weaving

Pre Import Weaving aims to apply the decorators before execution to avoid this limitation. This was a central issue in the TensorFlow use case. Here, the abstract syntax tree is used to apply the decorator annotations to all functions. This process is performed by the `InstrumentOnImportFinder` automatically.

The method utilizes the standard Python library module `ast` to process the code. However, the software to be instrumented can also use this feature for its own purposes, which may affect whether the instrumentation is applied at all.

4 Performance Evaluation

Overhead is the central issue of monitoring frameworks. Therefore, we run performance measurements utilizing a Python re-implementation of the micro benchmark MooBench [3].

MooBench consists of a script to execute the experiment and a control application. The latter executes a recursive function repeatedly and measures its execution time. These measurements are repeated for different setups to identify the effect that instrumentation and logging have onto the execution time of the test function. The whole benchmark for different instrumentation frameworks [10] including Kieker 4 Python can be found on GitHub.[1]

We defined 5 different setups utilizing each of the 2 instrumentation options, including, (a) application without instrumentation, (b) application with instrumentation, but without active probes, (c) with active probes, but with a dummy writer, (d) with active probes and the text writer, and (e) with active probes using the TCP writer. In total, we had 9 different experiments, as the no instrumentation setup only needed to run once.

For comparison, we executed the corresponding setup for Kieker 4 Java, on the same machine with the same configuration parameters for the same setups, utilizing one instrumentation option, AspectJ.

The experiment was executed single-threaded on a server with an Intel Xeon E5620, 2.4 GHz with 8 GB RAM, and each experiment was repeated 10 times. The test code is a function with the recursion depth of 10 with zero additional method time, producing a trace with 10 calls. We created 200 000 traces per experiment, but only collect the latter half for measurement to ensure that the just-in-time compiler in Java has been applied, and the system is in steady state. The same applies to the Python experiment.

An overview of the results is depicted in Table 1. The benchmark runs 42 times slower in Python than in Java without instrumentation, and varies more in its execution time. This is reduced when the probe is introduced to 8 times slower, indicating that the instrumentation overhead is much lower than in Java.

As expected, the text loggers are the slowest. This is due to the high volume of text that needs to be written during logging. Binary representations are much faster, as they are more compact. Kieker 4 Python, does not include a binary writer, as we limited our effort to a TCP writer and the use of the Kieker collector (formerly known as Kieker-Data-Bridge).

Also, an interesting notion is that for the TCP and null writers, the pre import weaving is faster while we expected that both decorator approaches are similar in their overhead.

5 Conclusions

We presented the instrumentation of two case studies with Kieker 4 Python and evaluated the performance of the implementation and two of the instrumentation methods. Based on the measurements, the *pre import weaving* method results in faster execution. This is an interesting result, as we assumed that there should

[1] `https://github.com/kieker-monitoring/moobench`

Table 1: Kieker 4 Python and Java performance measurements in milliseconds in comparison. Kieker 4 Python has two result sets in addressing the two different instrumentation approaches. Kieker 4 Java used AspectJ for instrumentation. 1.q, 2.q, 3.q stand for the first, second (median) and third quartile.

	Python				Java			
setup	mean	1.q	2.q	3.q	mean	1.q	2.q	3.q
w/o	6.908	6.860	6.892	6.932	0.166	0.16	0.160	0.167
probe inactive - post	15.131	15.096	15.139	15.188	1.820	0.857	0.900	3.595
probe inactive - pre	15.137	15.042	15.079	15.121				
null writer - post	534.177	583.261	585.929	587.558	17.844	14.473	17.306	20.950
null writer - pre	118.255	117.655	117.881	118.160				
text writer - post	2681.863	2104.617	2292.797	3236.825	232.100	223.481	226.028	228.541
text writer - pre	2772.709	2782.738	2793.667	2808.349				
tcp writer - post	1340.107	1199.775	1205.714	1661.062	12.228	9.016	12.427	14.457
tcp writer - pre	960.505	870.799	876.135	1149.236				

not be a significant performance difference between both methods, as they both utilize decorators.

We were able to apply both approaches on our case studies to some extent. However, there are limits to what can be instrumented and whether the weaving has side effects. Furthermore, modules written in C, like numpy, cannot be instrumented with Kieker 4 Python. Here it might be necessary to instrument numpy's C code with Kieker 4 C [9].

In the future, we will evaluate two additional methods to mitigate issues with function decorators and import modifications. First, we will monitor changes to the `sys.meta_path` to be able to react to changes introduced by other code and reapply instrumentation. This may help to reduce the chance of having instrumentation removed by other code, but as all facilities we use, can also be used by any other programmer, there can be cases where this method fails.

Second, we envision a method inspired by AspectJ and similar tooling to apply instrumentation before execution, i.e., create a custom Python to Python compiler that adds instrumentation to the code or extend Python itself to be able to be used in this fashion. However, this method requires extensive knowledge on Python grammar and return behavior.

Finally, we aim to improve the performance of Kieker 4 Python by adding asynchronous logging, and define an approach to combine Kieker 4 C and Python to be able to monitor modules implemented in other programming languages.

Acknowledgment Funded by KMS Kiel Marine Science - Centre for Interdisciplinary Marine Science at Kiel University and the Deutsche Forschungsgemeinschaft (DFG – German Research Foundation), grant no. HA 2038/8-1 – 425916241.

References

[1] A. van Hoorn, J. Waller, and W. Hasselbring. "Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis". In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*. Boston, Massachusetts, USA, April 22–25, 2012: ACM, Apr. 2012, pp. 247–248.

[2] B. Jones and D. Beazley. *Python Cookbook*. O'Reilly Media, Incorporated, 2012.

[3] J. Waller. "Performance Benchmarking of Application Monitoring Frameworks". PhD thesis. Faculty of Engineering, Kiel University, Dec. 2014.

[4] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015.

[5] J. Waller, N. C. Ehmke, and W. Hasselbring. "Including Performance Benchmarks into Continuous Integration to Enable DevOps". In: *SIGSOFT Softw. Eng. Notes* 40.2 (Mar. 2015), pp. 1–4.

[6] T. Kluyver et al. "Jupyter Notebooks - a publishing format for reproducible computational workflows". In: *Positioning and Power in Academic Publishing: Players, Agents and Agendas*. Ed. by F. Loizides and B. Scmidt. Netherlands: IOS Press, 2016, pp. 87–90.

[7] Boris. *Behind the Scenes: Building a Dynamic Instrumentation Agent for Python*. 2017.

[8] W. Hasselbring and A. van Hoorn. "Kieker: A monitoring framework for software engineering research". In: *Software Impacts* 5 (Aug. 2020).

[9] R. Jung, S. Gundlach, and W. Hasselbring. "Instrumenting C and Fortran Software with Kieker". In: *Symposium on Software Performance*. Ed. by S. Becker et al. CEUR Worshop Proceedings. Nov. 2021.

[10] D. G. Reichelt, S. Kühne, and W. Hasselbring. "Overhead Comparison of OpenTelemetry, inspectIT and Kieker". In: *Symposium on Software Performance 2021*. CEUR Worshop Proceedings. Nov. 2021.