BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2024
http://excel.fit.vutbr.cz

Improving the GraalPy Interpreter

Adam Hrbáč*

Abstract

GraalPy is a Python implementation for the Java Virtual Machine, designed for easy embedding into Java applications. This is primarily useful for adopting 3rd party Python packages in existing Java codebases. GraalPy also often has greater performance compared to CPython, the canonical implementation of Python. This work implements two major features: The async API, one of the last major missing Python features in GraalPy, used primarily for web development, allowing writing concurrent code without parallelism, using so-called colored async, where each context switch point must be explicitly annotated. It is composed of two major parts, a library providing an event loop, as well as the syntactic components of Python, providing the way with which to indicate context switches. The second feature is a subset of the tracing API, a CPython API for implementing Python debuggers, used by integrated debuggers in IDEs, coverage tools, etc. It works by analyzing the Python bytecode in order to determine whether a new line is being executed, and if so, invokes a registered callback. This callback is also used when returning a value, calling a function and raising an exception, allowing a debugger to set a breakpoint for these events. Both features are part of the GraalPy releases and have had a notable benefit to compatibility with 3rd party packages.

*xhrbac12@vut.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

GraalPy [1], an implementation of the Python programming language utilising the language implementation framework Truffle [2], is designed for embedding Python into Java applications without having to go through the C APIs of the language runtimes. In order to maximize the usability of GraalPy, it is desirable to support as much of Python as possible. Unlike similar language implementations using Truffle, notably GraalJS and TruffleRuby, the most common Python implementation, CPython [3], leaks its internals, both via the well-supported C API and the language itself [4], making it almost useless to implement Python without matching CPython implementation details as well if the Python ecosystem is desirable. Since a big reason for embedding Python into Java applications is being able to use python packages, the ecosystem is indeed desirable.

There are two major features this work implements in GraalPy, the first of which is asynchronous programming. This allows for writing concurrent code without parallelism, with the programmer explicitly syntactically annotating context switches, making it easier to avoid data races and race conditions. This requires two separate components, the syntax, providing the means by which to indicate where these context switches may happen, as well as async alternatives to the `for` loop and the `with` statement. The second part is the library providing the event loop in which this concurrency occurs. In this work, `asyncio` is ported from CPython, being part of the standard library and the most used library. Since only fairly small parts of `asyncio` are written via the CPython C API, it was not necessary to reimplement the entire library in Java, only a few key functions. The second feature is the CPython tracing API, which is primarily used to implement debuggers, and also backs the popular `coverage.py` library. While GraalPy has its own debugging API provided by Truffle, conventional editors, such as PyCharm and VSCode have their integrated debuggers written with the CPython API in mind, making the API desirable to support, despite being considered a CPython implementation detail. Although it is not reasonably possible to have both APIs behave in identical ways, CPython itself changes the behaviour heavily between minor versions, therefore minor differences are acceptable.

2. Asynchronous programming in Python

In recent years, asynchronous programming has seen a rise popularity in the Python ecosystem, primarily for web-related tasks. The inclusion of `asyncio` into the standard library has allowed the ecosystem to grow to the point when async support (that is, the support for Asynchronous programming in Python as implemented in this work) is expected in just about every relevant library. This has shifted the situation from async being a weird gimmick for specialised use cases to being the state-of-the-art way to do web development in Python. Since running web servers is a use-case of GraalPy, supporting these libraries is valuable.

The primary goal of this work is to create a baseline from which the compatibility with async-related libraries can be tested. A lot of pure-python libraries do work fine as is, but Python web servers are typically not written in pure Python. Especially `uvicorn` would be valuable, since it is the most used async Python web server. Further improvements will be more related to supporting platform-specific IO abstractions and C extensions than Python features, since it is now possible to identify these defects.

As GraalPy gets better at supporting the Python ecosystem, it gets more and more common to run into popular libraries which do not work due to lacking async support. This has created issues, since the prior stubs can often hard crash, which makes running test suites more difficult than the tests simply not passing. Thanks to the contributions of this work, async features tend to work with no additional changes if the remainder of the package does. A current example is `Flask`, a web framework with async support.

Another interesting aspect is dealing with task-local (where Task is the unit of concurrency in `asyncio`) state. Since each Task spawned from a Task should inherit the task-local state of the spawning Task, it must be possible to copy this state in constant time, and since the context stores variable names, it is useful to have hashing involved. A specific data structure is perfect here, a so-called Hash-Array Mapped Trie [5].

As part of this work, the basic implementation of the standard Python async APIs was merged into the GraalPy 23.0.2 release. Certain features are missing, but since these features are either all but impossible, or debugging tools, they are not a priority. This work has since been used to improve compatibility with certain packages, notably `httpx` and `Jinja2`.

3. The Tracing API

As was mentioned earlier, making a distinction between the implementation details of CPython and Python as the language is not useful for accessing the Python ecosystem. One of these implementation details is tracing, a CPython API for implementing debuggers and similar tools. The goal of this part of the work is supporting this API in GraalPy.

The primary goal is to create a baseline from which to improve support for libraries such as `coverage.py`, a tool for measuring test coverage, and pdb, the standard python debbuger, as well as the integrated debuggers in common Python code editors. This should allow significantly easier work with GraalPy in those editors, and saves work on porting Truffle tooling to each of the editors, which will never be quite as good as the integrated debuggers.

Due to the recent switch to the bytecode interpreter in GraalPy, it is possible to implement tracing in a very similar fashion as in CPython. However, line numbers are handled differently between CPython and GraalPy, requiring a more complex computation on the GraalPy, since only source offsets are stored. It is impossible to get identical behaviour, since the bytecode is quite different in places. Nevertheless, most programs using the tracing API should work, unless they rely on opcode events – that is, the ability to trace execution at the CPython bytecode level, since GraalPy does not use CPython bytecode.

Additionally, the support for so-called debugger jumps was added, that is, the ability for the debugger to non-deterministically change the currect executing line. This could cause interpreter stack corruption in some cases, so it is necessary to check whether the jump avoids doing so.

As part of this work, a comprehensive implementation of the tracing API of CPython was merged into the release 24.0 of GraalPy. It has since been used to improve support of GraalPy in various editors, as well as the test coverage library `coverage.py` [6]

References

[1] Oracle-Labs. Graalpy. source code. https://github.com/oracle/graalpython, Accessed 2023-06-07.

[2] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-optimizing ast interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages*, DLS '12, page 73–82, New York, NY,

USA, 2012. Association for Computing Machinery.

[3] Python-Foundation. CPython. source code. https://github.com/python/cpython, Accessed 2023-06-07.

[4] Armin Ronacher. How python was shaped by leaky internals. PyCon Russia.

[5] Phil Bagwell. Ideal hash trees. Technical report, École Polytechnique Fédérale de Lausanne, 2000.

[6] Ned Batchelder. Coverage.py. source code. https://github.com/nedbat/coveragepy, Accessed 2024-01-28.