Pyflame Documentation

Release 1.4.0

Evan Klitzke

Nov 13, 2017

Contents:

Pyflame is a unique profiling tool that generates flame graphs for Python. Pyflame is the only Python profiler based on the Linux ptrace(2) system call. This allows it to take snapshots of the Python call stack without explicit instrumentation, meaning you can profile a program without modifying its source code! Pyflame is capable of profiling embedded Python interpreters like uWSGI. It fully supports profiling multi-threaded Python programs.

Pyflame is written in C++, with attention to speed and performance. Pyflame usually introduces less overhead than the builtin `profile` (or `cProfile`) modules, and also emits richer profiling data. The profiling overhead is low enough that you can use it to profile live processes in production.

CHAPTER 1

Installing Pyflame

You have two options for installing Pyflame: you can try a pre-built package, or you can install from source. To build from source, you will need a C++ compiler with basic C++11 support. Pyflame is known to compile on versions of GCC as old as GCC 4.6.

1.1 Build Dependencies

Generally you'll need autotools, automake, libtool, pkg-config, and the Python headers. If you have headers for both Python 2 and Python 3 installed you'll get a Pyflame build that can target either version of Python.

1.1.1 Debian/Ubuntu

Install the following packages if you are building for Debian or Ubuntu. Note that you technically only need one of `python-dev` or `python3-dev`, but if you have both installed then you can use Pyflame to profile both Python 2 and Python 3 processes.

```
# Install build dependencies on Debian or Ubuntu.
sudo apt-get install autoconf automake autotools-dev g++ pkg-config python-dev␣
↪python3-dev libtool make
```

1.1.2 Fedora

Again, you technically only need one of `python-devel` and `python3-devel`, although installing both is recommended.

```
# Install build dependencies on Fedora.
sudo dnf install autoconf automake gcc-c++ python-devel python3-devel libtool
```

1.2 Compiling

Once you've installed the appropriate build dependencies, you can compile Pyflame like so:

```
./autogen.sh
./configure      # Plus any options like --prefix.
make
make check       # Optional, test the build! Should take < 1 minute.
make install     # Optional, install into the configure prefix.
```

The Pyflame executable produced by the `make` command will be located at `src/pyflame`. Note that the `make check` command requires that you have the `virtualenv` command installed. You can also sanity check your build with a command like:

```
# Or use -t python3, as appropriate.
pyflame -t python -c 'print(sum(i for i in range(100000)))'
```

1.2.1 Creating A Debian Package

If you'd like to build a Debian package, run the following from the root of your Pyflame git checkout:

```
# Install additional dependencies required for packaging.
sudo apt-get install debhelper dh-autoreconf dpkg-dev

# This create a file named something like ../pyflame_1.3.1_amd64.deb
dpkg-buildpackage -uc -us
```

1.3 Pre-Built Packages

Several Pyflame users have created unofficial pre-built packages for different distros. Uploads of these packages tend to lag the official Pyflame releases, so you are **strongly encouraged to check the pre-built version** to ensure that it is not too old. If you want the newest version of Pyflame, build from source.

1.3.1 Conda

Evan Klitzke maintains a Conda package of Pyflame:

```
conda install -c eklitzke pyflame
```

1.3.2 Ubuntu PPA

Trevor Joynson has set up an unofficial PPA for all current Ubuntu releases: ppa:trevorjay/pyflame.

```
sudo apt-add-repository ppa:trevorjay/pyflame
sudo apt-get update
sudo apt-get install pyflame
```

Note also that you can build your own Debian package easily, using the one provided in the `debian/` directory of this project.

1.3.3 Arch Linux

Oleg Senin has added an Arch Linux package to AUR.

CHAPTER 2

Using Pyflame

Pyflame has two distinct modes: you can attach to a running process, or you can trace a command from start to finish.

2.1 Attaching To A Running Python Process

The default behavior of Pyflame is to attach to an existing Python process. The target process is specified via its PID:

```
# Profile PID for 1s, sampling every 1ms.
pyflame -p PID
```

This will print data to stdout in a format that is suitable for usage with Brendan Gregg's `flamegraph.pl` tool (which you can get here). A typical command pipeline might be like this:

```
# Generate flame graph for pid 12345; assumes flamegraph.pl is in your $PATH.
pyflame -p 12345 | flamegraph.pl > myprofile.svg
```

You can also change the sample time with `-s`, and the sampling frequency with `-r`. Both units are measured in seconds.

```
# Profile PID for 60 seconds, sampling every 10ms.
pyflame -s 60 -r 0.01 -p PID
```

The default behavior is to sample for 1 second (equivalent to `-s 1`), taking a snapshot every millisecond (equivalent to `-r 0.001`).

2.1.1 Attaching To Docker/Containerized Processes

Pyflame knows how to do something interesting: it can attach to containerized processes from **outside the container**. It does this by directly using the setns(2) system call (which is how Docker works under the hood).

If you choose to profile a process from outside the container, use the true PID, as reported by `ps` on the host (i.e. outside of the container).

You can also run Pyflame from inside containers, although this is a bit more annoying, since normally ptrace is disabled inside containers for security reasons. If you attach to a process this way, you will need to use the inside-the-container PID. You can find this by running `ps` inside of the container itself.

We recommend running Pyflame from outside containers, since it means you can keep ptrace disabled inside containers. If you want to run Pyflame inside containers, and have problems, please make sure to read the Docker notes in the *FAQ*.

2.2 Tracing Python Commands

Sometimes you want to trace a command from start to finish. An example would be tracing the run of a test suite or batch job. Pass `-t` as the **last** Pyflame flag to run in trace mode. Anything after the `-t` flag is interpreted literally as part of the command to run:

```
# Trace a given command until completion.
pyflame [regular pyflame options] -t command arg1 arg2...
```

Often `command` will be `python` or `python3`, but it could be something else, like `uwsgi` or `py.test`. For instance, here's how Pyflame can be used to trace its own test suite:

```
# Trace the Pyflame test suite, a.k.a. pyflameception!
pyflame -t py.test tests/
```

As described in the docs for attach mode, you can use `-r` to control the sampling frequency.

2.2.1 Tracing Programs That Print To Stdout

By default, Pyflame will send flame graph data to stdout. If the profiled program is also sending data to stdout, then `flamegraph.pl` will see the output from both programs, and will get confused. To solve this, use the `-o` option:

```
# Trace a process, sending profiling information to profile.txt
pyflame -o profile.txt -t python -c 'for x in range(1000): print(x)'

# Convert profile.txt to a flame graph named profile.svg
flamegraph.pl <profile.txt >profile.svg
```

2.3 Timestamp ("Flame Chart") Mode

Generally we recommend using regular flame graphs, generated by `flamegraph.pl`. However, Pyflame can also generate data with a special time stamp output format, useful for generating "flame charts" (somewhat like an inverted flame graph) that are viewable in Chrome. In some cases, the flame chart format is easier to understand.

To generate a flame chart, use `pyflame --flamechart`, and then pass the output to `utils/flame-chart-json` to convert the output into the JSON format required by the Chrome CPU profiler:

```
# Generate flame chart data viewable in Chrome.
pyflame --flamechart [other pyflame options] | flame-chart-json > foo.cpuprofile
```

Read the following Chrome DevTools article for instructions on loading a `.cpuprofile` file in Chrome 58+.

CHAPTER 3

FAQ

3.1 What Python Versions Are Supported?

Python 2 is tested with Python 2.6 and 2.7. Earlier versions of Python 2 are likely to work as well, but have not been tested.

Python 3 is tested with Python 3.4, 3.5, and 3.6. Python 3.6 introduces a new ABI for the `PyCodeObject` type, so Pyflame only supports the Python 3 versions that header files were available for when Pyflame was compiled.

It's possible for Pyflame to get confused about what Python version the target process is when profiling an embedded Python build, such as uWSGI. If you run into this issue, use the `--abi` option to force a particular Python ABI.

3.2 What Is "(idle)" Time?

In Python, only one thread can execute Python code at any one time, due to the Global Interpreter Lock, or GIL. The exception to this rule is that threads can execute non-Python code (such as IO, or some native libraries such as NumPy) without the GIL.

By default Pyflame will only profile code that holds the Global Interpreter Lock. Since this is the only thread that can run Python code, in some sense this is a more accurate representation of the profile of an application, even when it is multithreaded. If nothing holds the GIL (so no Python code is executing) Pyflame will report the time as "idle".

If you don't want to include this time you can use the invocation `pyflame -x`.

If instead you invoke Pyflame with the `--threads` option, Pyflame will take a snapshot of each thread's stack each time it samples the target process. At the end of the invocation, the profiling data for each thread will be printed to stdout sequentially. This gives you a more accurate profile in the sense that you will see what each thread was trying to do, even if it wasn't actually scheduled to run.

Pyflame may "freeze" the target process if you use this option with older versions of the Linux kernel. In particular, for this option to work you need a kernel built with waitid() ptrace support. This change was landed for Linux kernel 4.7. Most Linux distros also backported this change to older kernels, e.g. this change was backported to the 3.16 kernel series in 3.16.37 (which is in Debian Jessie's kernel patches). For more extensive discussion, see issue #55.

One interesting use of this feature is to get a point-in-time snapshot of what each thread is doing, like so:

```
# Get a point-in-time snapshot of what each thread is currently running.
pyflame -s 0 --threads -p PID
```

3.3 Are BSD / OS X / macOS Supported?

Pyflame uses a few Linux-specific interfaces, so unfortunately it is the only platform supported right now. Pull requests to add support for other platforms are very much wanted.

Someone who is proficient with low-level C systems programming can probably get BSD to work without *too much* difficulty. The necessary work to adapt the code is described in Issue #3.

By comparison, it is probably *much more* work to get Pyflame working on macOS. The current code assumes that the host uses ELF object/executable files. Apple uses a different object file format, called Mach-O, so porting Pyflame to macOS would entail doing all of the work to port Pyflame to BSD, *plus* additional work to parse Mach-O object files. That said, the Mach-O format is documented online (e.g. here), so a sufficiently motivated person could get macOS support working.

3.4 What Are These Ptrace Permissions Errors?

Because it's so powerful, the `ptrace(2)` system call is often disabled or severely restricted. In order to use ptrace, these conditions must be met:

- You must have the `SYS_PTRACE` capability (which is denied by default within Docker images).
- The kernel must not have `kernel.yama.ptrace_scope` set to a value that is too restrictive.

In both scenarios you'll also find that `strace` and `gdb` do not work as expected.

3.4.1 Ptrace Errors Within Docker Containers

By default Docker images do not have the `SYS_PTRACE` capability. If you want it enabled, invoke `docker run` using the `--cap-add SYS_PTRACE` option:

```
# Allows processes within the Docker container to use ptrace.
docker run --cap-add SYS_PTRACE ...
```

You can also use capsh(1) to list your current capabilities:

```
# You should see cap_sys_ptrace in the "Bounding set".
capsh --print
```

You do not need to run Pyflame from within a Docker container. If you have sufficient permissions (i.e. you are root, or the same UID as the Docker process) Pyflame can be run from outside a container to inspect a process inside a container. This is better for security, since you can keep ptrace disabled in the container.

3.4.2 Ptrace Errors Outside Docker Containers Or When Not Using Docker

If you're not in a Docker container, or you're not using Docker at all, ptrace permissions errors are likely related to you having too restrictive a value set for the `kernel.yama.ptrace_scope` sysfs knob.

Debian Jessie ships with `ptrace_scope` set to 1 by default, which will prevent unprivileged users from attaching to already running processes.

To see the current value of this setting:

```
# Prints the current value for the ptrace_scope setting.
sysctl kernel.yama.ptrace_scope
```

If you see a value other than 0 you may want to change it. Note that by doing this you'll affect the security of your system. Please read the relevant kernel documentation for a comprehensive discussion of the possible settings and what you're changing. If you want to completely disable the ptrace settings and get "classic" permissions (i.e. root can ptrace anything, unprivileged users can ptrace processes with the same user id) then use:

```
# Use this if you want "classic" ptrace permissions.
sudo sysctl kernel.yama.ptrace_scope=0
```

3.4.3 Ptrace With SELinux

If you're using SELinux, you may have problems with ptrace. To check if ptrace is disabled:

```
# Check if SELinux is denying ptrace.
getsebool deny_ptrace
```

If you'd like to enable it:

```
# Enable ptrace under SELinux.
setsebool -P deny_ptrace 0
```

CHAPTER 4

Contributing

We love getting pull requests and bug reports! This section outlines some ways you can contribute to Pyflame.

4.1 Hacking

This section will explain the Pyflame code for people who are interested in contributing source code patches.

A good way to start understanding the code is to read the two blog posts (linked on the main docs page) written by Evan Klitzke. They cover the basics about how Pyflame works, and have some helpful information about how the code is organized.

The code style in Pyflame (mostly) conforms to the Google C++ Style Guide. Additionally, all of the source code is formatted with clang-format. There's a `.clang-format` file checked into the root of this repository which will make `clang-format` do the right thing. Different clang releases may format the source code slightly differently, as the formatting rules are updated within clang itself. Therefore you should eyeball the changes made when formatting, especially if you have an older version of clang.

If you are changing any of the low-level C++ bits, and end up with a broken build, you may want to try by getting the following command working before testing with the full test suite:

```
# Sanity check Pyflame.
pyflame -t python -c 'print(sum(i for i in range(100000)))'
```

To run the full test suite locally:

```
# Run the Pyflame test suite.
make check
```

If you change any of the Python files in the `tests/` directory, please run your changes through YAPF before submitting a pull request.

4.2 How Else Can I Help?

Patches are not the only way to contribute to Pyflame! Bug reports are very useful as well. If you file a bug, make sure you tell us the exact version of Python you're using, and how to reproduce the issue.

CHAPTER 5

Websites

- Project homepage (this documentation)
- Source code at Github

CHAPTER 6

Blog Posts

Some existing articles and blog posts on Pyflame include:

- Pyflame: Uber Engineering's Ptracing Profiler For Python by Evan Klitzke (2016-09)
- Pyflame Dual Interpreter Mode by Evan Klitzke (2016-10)
- Using Uber's Pyflame and Logs to Tackle Scaling Issues by Benoit Bernard (2017-02)
- Building Pyflame on Centos 6 (Chinese) by Faicker Mo (2017-04)

If you write a new post about Pyflame, please let us know and we'll add it here!