Python support for the Linux perf profiler

Release 3.13.3

Guido van Rossum and the Python development team

April 27, 2025

Python Software Foundation
Email: docs@python.org

Contents

author
Pablo Galindo

The Linux perf profiler is a very powerful tool that allows you to profile and obtain information about the performance of your application. `perf` also has a very vibrant ecosystem of tools that aid with the analysis of the data that it produces.

The main problem with using the `perf` profiler with Python applications is that `perf` only gets information about native symbols, that is, the names of functions and procedures written in C. This means that the names and file names of Python functions in your code will not appear in the output of `perf`.

Since Python 3.12, the interpreter can run in a special mode that allows Python functions to appear in the output of the `perf` profiler. When this mode is enabled, the interpreter will interpose a small piece of code compiled on the fly before the execution of every Python function and it will teach `perf` the relationship between this piece of code and the associated Python function using perf map files.

> **Note**
>
> Support for the `perf` profiler is currently only available for Linux on select architectures. Check the output of the `configure` build step or check the output of `python -m sysconfig | grep HAVE_PERF_TRAMPOLINE` to see if your system is supported.

For example, consider the following script:

```
def foo(n):
    result = 0
```

(continues on next page)

(continued from previous page)

```
    for _ in range(n):
        result += 1
    return result

def bar(n):
    foo(n)

def baz(n):
    bar(n)

if __name__ == "__main__":
    baz(1000000)
```

We can run `perf` to sample CPU stack traces at 9999 hertz:

```
$ perf record -F 9999 -g -o perf.data python my_script.py
```

Then we can use `perf report` to analyze the data:

```
$ perf report --stdio -n -g

# Children      Self       Samples  Command     Shared Object       Symbol
# ........  ........  ............  ..........  ..................  ...............
↪.........................
#
    91.08%     0.00%             0  python.exe  python.exe          [.] _start
            |
            ---_start
            |
                --90.71%--__libc_start_main
                        Py_BytesMain
                        |
                        |--56.88%--pymain_run_python.constprop.0
                        |          |
                        |          |--56.13%--_PyRun_AnyFileObject
                        |          |          _PyRun_SimpleFileObject
                        |          |          |
                        |          |          |--55.02%--run_mod
                        |          |          |          |
                        |          |          |           --54.65%--PyEval_EvalCode
                        |          |          |                     _PyEval_
↪EvalFrameDefault
                        |          |          |                     PyObject_
↪Vectorcall
                        |          |          |                     _PyEval_Vector
                        |          |          |                     _PyEval_
↪EvalFrameDefault
                        |          |          |                     PyObject_
↪Vectorcall
                        |          |          |                     _PyEval_Vector
                        |          |          |                     _PyEval_
↪EvalFrameDefault
                        |          |          |                     PyObject_
↪Vectorcall
                        |          |          |                     _PyEval_Vector
                        |          |          |                     |
                        |          |          |                     |--51.67%--_
```

(continues on next page)

(continued from previous page)

```
↪PyEval_EvalFrameDefault
                        |           |           |                       |           |
                        |           |           |                       |           |--
↪11.52%--_PyLong_Add
                        |           |           |                       |           | ␣
↪        |
                        |           |           |                       |           | ␣
↪        |--2.97%--_PyObject_Malloc
...
```

As you can see, the Python functions are not shown in the output, only `_PyEval_EvalFrameDefault` (the function that evaluates the Python bytecode) shows up. Unfortunately that's not very useful because all Python functions use the same C function to evaluate bytecode so we cannot know which Python function corresponds to which bytecode-evaluating function.

Instead, if we run the same experiment with `perf` support enabled we get:

```
$ perf report --stdio -n -g

# Children      Self       Samples  Command     Shared Object       Symbol
# ........  ........  ............  ..........  ..................  ...............
↪.....................................................
#
    90.58%     0.36%             1  python.exe  python.exe          [.] _start
            |
            ---_start
            |
                --89.86%--__libc_start_main
                        Py_BytesMain
                        |
                        |--55.43%--pymain_run_python.constprop.0
                        |           |
                        |           |--54.71%--_PyRun_AnyFileObject
                        |           |          _PyRun_SimpleFileObject
                        |           |           |
                        |           |           |--53.62%--run_mod
                        |           |           |           |
                        |           |           |            --53.26%--PyEval_EvalCode
                        |           |           |                      py::<module>:/
↪src/script.py
                        |           |           |                      _PyEval_
↪EvalFrameDefault
                        |           |           |                      PyObject_
↪Vectorcall
                        |           |           |                      _PyEval_Vector
                        |           |           |                      py::baz:/src/
↪script.py
                        |           |           |                      _PyEval_
↪EvalFrameDefault
                        |           |           |                      PyObject_
↪Vectorcall
                        |           |           |                      _PyEval_Vector
                        |           |           |                      py::bar:/src/
↪script.py
                        |           |           |                      _PyEval_
↪EvalFrameDefault
                        |           |           |                      PyObject_
```

(continues on next page)

(continued from previous page)

```
↪Vectorcall
                        |          |          |                    _PyEval_Vector
                        |          |          |                    py::foo:/src/
↪script.py
                        |          |          |                    |
                        |          |          |                    |--51.81%--_
↪PyEval_EvalFrameDefault
                        |          |          |                    |          |
                        |          |          |                    |          |--
↪13.77%--_PyLong_Add
                        |          |          |                    |          |  ␣
↪        |
                        |          |          |                    |          |  ␣
↪        |--3.26%--_PyObject_Malloc
```

1 How to enable perf profiling support

`perf` profiling support can be enabled either from the start using the environment variable `PYTHONPERFSUPPORT` or the `-X perf` option, or dynamically using `sys.activate_stack_trampoline()` and `sys.deactivate_stack_trampoline()`.

The `sys` functions take precedence over the `-X` option, the `-X` option takes precedence over the environment variable.

Example, using the environment variable:

```
$ PYTHONPERFSUPPORT=1 perf record -F 9999 -g -o perf.data python my_script.py
$ perf report -g -i perf.data
```

Example, using the `-X` option:

```
$ perf record -F 9999 -g -o perf.data python -X perf my_script.py
$ perf report -g -i perf.data
```

Example, using the `sys` APIs in file `example.py`:

```
import sys

sys.activate_stack_trampoline("perf")
do_profiled_stuff()
sys.deactivate_stack_trampoline()

non_profiled_stuff()
```

…then:

```
$ perf record -F 9999 -g -o perf.data python ./example.py
$ perf report -g -i perf.data
```

2 How to obtain the best results

For best results, Python should be compiled with `CFLAGS="-fno-omit-frame-pointer -mno-omit-leaf-frame-pointer"` as this allows profilers to unwind using only the frame pointer and not on DWARF debug information. This is because as the code that is interposed to allow `perf` support is dynamically generated it doesn't have any DWARF debugging information available.

You can check if your system has been compiled with this flag by running:

```
$ python -m sysconfig | grep 'no-omit-frame-pointer'
```

If you don't see any output it means that your interpreter has not been compiled with frame pointers and therefore it may not be able to show Python functions in the output of `perf`.

3 How to work without frame pointers

If you are working with a Python interpreter that has been compiled without frame pointers, you can still use the `perf` profiler, but the overhead will be a bit higher because Python needs to generate unwinding information for every Python function call on the fly. Additionally, `perf` will take more time to process the data because it will need to use the DWARF debugging information to unwind the stack and this is a slow process.

To enable this mode, you can use the environment variable `PYTHON_PERF_JIT_SUPPORT` or the `-X perf_jit` option, which will enable the JIT mode for the `perf` profiler.

Note

Due to a bug in the `perf` tool, only `perf` versions higher than v6.8 will work with the JIT mode. The fix was also backported to the v6.7.2 version of the tool.

Note that when checking the version of the `perf` tool (which can be done by running `perf version`) you must take into account that some distros add some custom version numbers including a `-` character. This means that `perf 6.7-3` is not necessarily `perf 6.7.3`.

When using the perf JIT mode, you need an extra step before you can run `perf report`. You need to call the `perf inject` command to inject the JIT information into the `perf.data` file.:

```
$ perf record -F 9999 -g -k 1 --call-graph dwarf -o perf.data python -Xperf_jit my_
↪script.py
$ perf inject -i perf.data --jit --output perf.jit.data
$ perf report -g -i perf.jit.data
```

or using the environment variable:

```
$ PYTHON_PERF_JIT_SUPPORT=1 perf record -F 9999 -g --call-graph dwarf -o perf.data␣
↪python my_script.py
$ perf inject -i perf.data --jit --output perf.jit.data
$ perf report -g -i perf.jit.data
```

`perf inject --jit` command will read `perf.data`, automatically pick up the perf dump file that Python creates (in `/tmp/perf-$PID.dump`), and then create `perf.jit.data` which merges all the JIT information together. It should also create a lot of `jitted-XXXX-N.so` files in the current directory which are ELF images for all the JIT trampolines that were created by Python.

Warning

When using `--call-graph dwarf`, the `perf` tool will take snapshots of the stack of the process being profiled and save the information in the `perf.data` file. By default, the size of the stack dump is 8192 bytes, but you can change the size by passing it after a comma like `--call-graph dwarf,16384`.

The size of the stack dump is important because if the size is too small `perf` will not be able to unwind the stack and the output will be incomplete. On the other hand, if the size is too big, then `perf` won't be able to sample the process as frequently as it would like as the overhead will be higher.

The stack size is particularly important when profiling Python code compiled with low optimization levels (like `-O0`), as these builds tend to have larger stack frames. If you are compiling Python with `-O0` and not seeing Python functions in your profiling output, try increasing the stack dump size to 65528 bytes (the maximum):

```
$ perf record -F 9999 -g -k 1 --call-graph dwarf,65528 -o perf.data python -
↪Xperf_jit my_script.py
```

Different compilation flags can significantly impact stack sizes:

- Builds with `-O0` typically have much larger stack frames than those with `-O1` or higher
- Adding optimizations (`-O1`, `-O2`, etc.) typically reduces stack size
- Frame pointers (`-fno-omit-frame-pointer`) generally provide more reliable stack unwinding

Index

E

P