Suporte do Python ao perfilador perf do Linux

Release 3.13.0b3

Guido van Rossum and the Python development team

julho 07, 2024

Python Software Foundation
Email: docs@python.org

Sumário

autor
Pablo Galindo

O perfilador perf do Linux é uma ferramenta muito poderosa que permite criar perfis e obter informações sobre o desempenho da sua aplicação. `perf` também possui um ecossistema muito vibrante de ferramentas que auxiliam na análise dos dados que produz.

O principal problema de usar o perfilador `perf` com aplicações Python é que `perf` apenas obtém informações sobre símbolos nativos, ou seja, os nomes de funções e procedimentos escritos em C. Isso significa que os nomes de funções Python e seus nomes de arquivos em seu código não aparecerão na saída de `perf`.

Desde o Python 3.12, o interpretador pode ser executado em um modo especial que permite que funções do Python apareçam na saída do criador de perfilador `perf`. Quando este modo está habilitado, o interpretador interporá um pequeno pedaço de código compilado instantaneamente antes da execução de cada função Python e ensinará `perf` a relação entre este pedaço de código e a função Python associada usando arquivos de mapa perf.

Nota: O suporte para o perfilador `perf` está atualmente disponível apenas para Linux em arquiteturas selecionadas. Verifique a saída da etapa de construção `configure` ou verifique a saída de `python -m sysconfig | grep HAVE_PERF_TRAMPOLINE` para ver se o seu sistema é compatível.

Por exemplo, considere o seguinte script:

```
def foo(n):
    result = 0
    for _ in range(n):
        result += 1
    return result
```

(continua na próxima página)

(continuação da página anterior)

```
def bar(n):
    foo(n)

def baz(n):
    bar(n)

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

Podemos executar `perf` para obter amostras de rastreamentos de pilha da CPU em 9999 hertz:

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

Então podemos usar `perf report` para analisar os dados:

```
$ 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%--_
↪PyEval_EvalFrameDefault
                        |          |          |                     |          |
                        |          |          |                     |          |--
↪11.52%--_PyLong_Add
                        |          |          |                     |          |  ␣
↪       |
                        |          |          |                     |          |  ␣
```

(continua na próxima página)

(continuação da página anterior)

```
↪       |--2.97%--_PyObject_Malloc
...
```

Como você pode ver, as funções Python não são mostradas na saída, apenas `_PyEval_EvalFrameDefault` (a função que avalia o bytecode Python) aparece. Infelizmente isso não é muito útil porque todas as funções Python usam a mesma função C para avaliar bytecode, portanto não podemos saber qual função Python corresponde a qual função de avaliação de bytecode.

Em vez disso, se executarmos o mesmo experimento com o suporte `perf` ativado, obteremos:

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

(continua na próxima página)

(continuação da página anterior)

```
↪       |
                        |          |          |                       |          |  ␣
↪       |--3.26%--_PyObject_Malloc
```

1 Como habilitar o suporte a perfilação com `perf`

O suporte à perfilação com `perf` pode ser habilitado desde o início usando a variável de ambiente `PYTHONPERFSUPPORT` ou a opção `-X perf`, ou dinamicamente usando `sys.activate_stack_trampoline()` e `sys.deactivate_stack_trampoline()`.

As funções `sys` têm precedência sobre a opção `-X`, a opção `-X` tem precedência sobre a variável de ambiente.

Exemplo usando a variável de ambiente:

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

Exemplo usando a opção `-X`:

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

Exemplo usando as APIs de `sys` em `example.py`:

```
import sys

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

non_profiled_stuff()
```

… então:

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

2 Como obter os melhores resultados

Para melhores resultados, Python deve ser compilado com `CFLAGS="-fno-omit-frame-pointer -mno-omit-leaf-frame-pointer"`, pois isso permite que os perfiladores façam o desenrolamento de pilha (ou *stack unwinding*) usando apenas o ponteiro de quadro e não no DWARF informações de depuração. Isso ocorre porque como o código interposto para permitir o suporte `perf` é gerado dinamicamente, ele não possui nenhuma informação de depuração DWARF disponível.

Você pode verificar se o seu sistema foi compilado com este sinalizador executando:

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

Se você não vir nenhuma saída, significa que seu interpretador não foi compilado com ponteiros de quadro e, portanto, pode não ser capaz de mostrar funções Python na saída de `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.

Nota: 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 --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.

Aviso: Notice that 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 the user can change the size by passing the size after comma like `--call-graph dwarf,4096`. 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.

Índice