vmprof Documentation

Release 0.4

Maciej Fijalkowski, Antonio Cuni, Sebastian Pawlus, Richard Plan

Sep 07, 2017

Contents

<[vmprof]>

vmprof is a platform to understand and resolve performance bottlenecks in your code. It includes a *lightweight profiler* for CPython 2.7, CPython 3 and PyPy and an assembler log visualizer for PyPy. Currently we support Linux, Mac OS X and Windows.

The following provides more information about CPU profiles and JIT Compiler Logs:

CHAPTER 1

CPU Profiles

vmprof is a statistical profiler: it gathers information about your code by continuously taking samples of the call stack of the running program, at a given frequency. This is similar to tools like vtune or gperftools: the main difference is that those tools target C and C-like languages and are not very helpful to profile higher-level languages which run on top of a virtual machine, while vmprof is designed specifically for them. vmprof is also thread safe and will correctly display the information regardless of usage of threads.

There are three primary modes. The recommended one is to use our server infrastructure for a web-based visualization of the result:

```
python -m vmprof --web <program.py> <program parameters>
```

If you prefer a barebone terminal-based visualization, which will display only some basic statistics:

```
python -m vmprof <program.py> <program parameters>
```

To display a terminal-based tree of calls:

```
python -m vmprof -o output.log <program.py> <program parameters>

vmprofshow output.log
```

To upload an already saved profile log to the vmprof web server:

```
python -m vmprof.upload output.log
```

For more advanced use cases, vmprof can be invoked and controlled from within the program using the given API.

Requirements

VMProf runs on x86_64 and x86. It supports Linux, Mac OS X and Windows running CPython 2.7, 3.4, 3.5 and PyPy 4.1+.

Installation

Installation of `vmprof` is performed with a simple command:

```
pip install vmprof
```

PyPi ships wheels with libunwind shared objects (this means you need a recent version of pip).

If you build VMProf from source you need to compile C code:

    sudo apt-get install python-dev

We strongly suggest using the `--web` option that will display you a much nicer web interface hosted on `vmprof.com`.

If you prefer to host your own vmprof visualization server, you need the vmprof-server package.

After `-m vmprof` you can specify some options:

- `--web` - Use the web-based visualization. By default, the result can be viewed on our server.
- `--web-url` - the URL to upload the profiling info as JSON. The default is `vmprof.com`
- `--web-auth` - auth token for user name support in the server.
- `-p period` - float that gives you how often the profiling happens (the max is about 300 Hz, rather don't touch it).
- `-n` - enable all C frames, only useful if you have a debug build of PyPy or CPython.
- `--lines` - enable line profiling mode. This mode adds some overhead to profiling, but in addition to function calls it marks the execution of the specific lines inside functions.
- `-o file` - save logs for later
- `--help` - display help
- `--config` - a ini format config file with all options presented above. When passing a config file along with command line arguments, the command line arguments will take precedence and override the config file values.

Example *config.ini* file:

```
web-url = vmprof.com
web-auth = ffb7d4bee2d6436bbe97e4d191bf7d23f85dfeb2
period = 0.1
```

API

There is also an API that can bring more details to the table, but consider it unstable. The current API usage is as follows:

Module level functions

- `vmprof.enable(fileno, period=0.00099, memory=False)` - enable writing `vmprof` data to a file described by a fileno file descriptor. Period is in float seconds. The minimal available resolution is around 1ms, we're working on improving that (note the default is 0.99ms). Passing `memory=True` will provide additional data in the form of total RSS of the process memory interspersed with tracebacks.
- `vmprof.disable()` - finish writing vmprof data, disable the signal handler

- `vmprof.read_profile(filename)` - read vmprof data from `filename` and return `Stats` instance.

`Stats` object

Stats object gives you an overview of data:

- `stats.get_tree()` - Gives you a tree of objects

`Tree` object

Tree is made of Nodes, each node supports at least the following interface:

- `node[key]` - a fuzzy search of keys (first match)
- `repr(node)` - basic details
- `node.flatten()` - returns a new tree that flattens all the metadata (gc, blackhole etc.)
- `node.walk(callback)` - call a callable of form `callback(root)` that will be invoked on each node

Why a new profiler?

There is a variety of python profilers on the market: CProfile is the one bundled with CPython, which together with lsprofcalltree.py provides good info and decent visualization; plop is an example of statistical profiler.

We wanted a profiler with the following characteristics:

- Minimal overhead, small enough that enabling the profiler in production is a viable option. Ideally the overhead should be in the range 1-5%, with the possibility to tune it for more accurate measurments
- Ability to display a full stack of calls, so it can show how much time was spent in a function, including all its children
- Good integration with PyPy: in particular, it must be aware of the underlying JIT, and be able to show how much time is spent inside JITted code, Garbage collector and normal intepretation.

None of the existing solutions satisfied our requirements, hence we decided to create our own profiler. In particular, cProfile is slow on PyPy, does not understand the JITted code very well and is shown in the JIT traces.

How does it work?

As most statistical profilers, the core idea is to have a signal handler which periodically inspects and dumps the stack of the running program: the most frequently executed parts of the code will be dumped more often, and the post-processing and visualization tools have the chance to show the end user usueful info about the behavior of the profiled program. This is the very same approach used e.g. by gperftools.

However, when profiling an interpreter such as CPython, inspecting the C stack is not enough, because most of the time will always be spent inside the opcode dispatching loop of the virtual machine (e.g., `PyEval_EvalFrameEx` in case of CPython). To be able to display useful information, we need to know which Python-level function correspond to each C-level `PyEval_EvalFrameEx`.

This is done by reading the stack of Python frames instead of C stack.

Additionally, when on top of PyPy the C stack contains also stack frames which belong to the JITted code: the vmprof signal handler is able to recognize and extract the relevant info from those as well.

Once we have gathered all the low-level info, we can post-process and visualize them in various ways: for example, we can decide to filter out the places where we are inside the `select()` syscall, etc.

The machinery to gather the information has been the focus of the initial phase of vmprof development and now it is working well: we are currently focusing on the frontend to make sure we can process and display the info in useful ways.

Links

- vmprof-flamegraph Convert vmprof data into text format for flamegraph

CHAPTER 2

Frequently Asked Questions

- **What does <native symbol 0xdeadbeef> mean?**: Debugging information might or might not be compiled with some libraries. If you see lots of those entries you might want to compile the libraries to include dwarf debugging information. In most cases `gcc -g ...` will help. If the symbol has been exported in the shared object (on linux), `dladdr` might still be able to extract the function name even if no debugging information has been attached.
- **What do the colors on vmprof.com mean?**: For plain CPython there is no particular meaning, we might change that in the future. For PyPy we have a color coding to show at which state the VM sampled (e.g. JIT, Warmup, ...).
- **My Windows profile is malformed?**: Please ensure that you open the file in binary mode. Otherwise Windows will transform *n* to *rn*.
- **Do I need to install libunwind?**: Usually not. We ship python wheels that bundle libunwind shared objects. If you install vmprof from source, then you need to install the development headers of your distribution. OSX ships libunwind per default. If your pip version is really old it does not pull wheels and it will end up compiling from source.

CHAPTER 3

Develop VMProf

VMProf consists of several projects working together:

- vmprof-python: The PyPI package providing the command line interface to enable vmprof.
- vmprof-server: Webservice hosted at vmprof.com. Hosts and visualizes data uploaded by vmprof-python package.
- vmprof-integration: Test suite for pulling together all different projects and ensuring that all play together nicely.
- PyPy: A virtual machine for the Python programming language. Most notably it contains an implementation for the logging facility vmprof-server can display.

The following description helps you to set up a development environment on Linux. For Windows and MacOSX the instructions might be similar.

Develop VMProf on Linux

It is recommended to use Python 3.x for development. Here is a list of requirements on your system:

- python
- sqlite3
- virtualenv

Please move you shell to the location you store your source code in and setup a virtual environment:

```
$ virtualenv -p /usr/bin/python3 vmprof3
$ source vmprof3/bin/activate
```

All commands from now on assume you have the vmprof3 virutal environment enabled.

Clone the repositories

```
$ git clone git@github.com:vmprof/vmprof-integration.git
$ git clone git@github.com:vmprof/vmprof-server.git
$ git clone git@github.com:vmprof/vmprof-python.git
# on old mercurial version the following command takes ages. please use a recent␣
↪version
$ hg clone ssh://hg@bitbucket.org/pypy/pypy # optional, only if you want to hack on␣
↪pypy as well
```

VMProf Server

```
# setup django service
$ cd vmprof-server
$ pip install -r requirements/development.txt
$ python manage.py migrate
# to run the service
$ python manage.py runserver -v 3
```

VMProf Python

An optional stage. It is only necessary if you want to co develop vmprof-python with vmprof-server:

```
# install vmprof for development (only needed if you want to co develop vmprof-python)
$ cd vmprof-python
$ python setup.py develop
```

Now you are able to change both the python package and the server and see the results. Here are some more hints on how to develop this platform

Smaller Profiles

Some times it is tedious to generate a big log file and develop a new feature with it. Both for VMProf and JitLog you can generate small log files that ease development.

There are small logs generated by a python script in *vmprof-server/vmlog/test/data/loggen.py*. Use the following command to load those:

```
$ ./manage.py loaddata vmlog/test/fixtures.yaml
```

Now open your browser and redirect them to the jitlog. E.g. http://localhost:8000/#/1v1/traces

Integration Tests

This is a very important test suite to ensure that all packages work together. It is automatically run every day by travis. You can run them locally. If you happen not to run a Debian base distribution, you can provide the following shell variable to prevent the tests from downloading a Debian PyPy:

```
$ TEST_PYPY_EXEC=/path/to/pypy py.test testvmprof/
```

CHAPTER 4

Native profiling

Version 0.4+ is able to profile native functions (routines written in a different language like C) on Mac OS X and Linux. See below for a technical overview.

By default this feature is enabled. To disable native profiling add `--no-native` as a command line switch.

NOTE be sure to provide debugging symbols for your native functions, otherwise you will not see the symbol name of your e.g. C program.

Technical Design

Native sampling utilizes `libunwind` in the signal handler to unwind the stack.

Each stack frame is inspected until the frame evaluation function is encountered. Then the stack walking switches back to the traditional Python frame walking. Callbacks (Python frame -> ... C frame ... -> Python frame ->

C frame)

will not display intermediate native functions. It would give the impression that the first C frame was never called, but it will show the second C frame.

Earlier Implementation

Prior to 0.4.3 the following logic was implemented (see e.g. commit 3912330b509d). It was removed because it could not be implemented on Mac OS X (libunwind misses register/cancel functions for generated machine code).

To find the corresponding `PyFrameObject` during stack unwinding vmprof inserts a trampoline on CPython (called `vmprof_eval`) and places it just before `PyEval_EvalFrameEx`. It is a callee trampoline saving the `PyFrameObject` in the callee saved register `%rbx`. On Python 3.6+ the frame evaluation PEP 523 is utilized as trampoline.

CHAPTER 5

Profile File Formats

This project incooperates several custom tailored file formats. Most notably the CPU & Memory profile format and a file format for the JIT log.

Both share the same setup:

```
<8-bit tag><content of 32 bytes>
<8-bit tag><content of 55 bytes>
...
```

The tag decides how to proceed with the content.

CPU & Memory Profile

- Adress mapping: Matches the following pattern:

  ```
  <lang>:<symbol name>:<line>:<file>
  ```

  Most commonly lang will be *py*, but also can be *n* for native symbols. *line* is a positive integer number. *file* a path name, or '-' if no file could be found.

CHAPTER 6

JIT Compiler Logs

JitLog is a PyPy logging facility that outputs information about compiler internals. It was built primarily for the following use cases:

- Understand JIT internals and be able to browse emitted code (both IR operations and machine code)
- Track down speed issues
- Help bug reporting

This version is now integrated within the webservice vmprof.com and can be used free of charge.

Usage

The following commands show example usages:

```
# upload both vmprof & jitlog profiles
pypy -m vmprof --web --jitlog <program.py> <arguments>

# upload only a jitlog profile
pypy -m jitlog --web <program.py> <arguments>

# upload a jitlog when your program segfaults/crashes
$ pypy -m jitlog -o /tmp/file.log <program.py> <arguments>
<Segfault>
$ pypy -m jitlog --upload /tmp/file.log
```

CHAPTER 7

Jit Log Query Interface

This command line interface can be used to pretty print optimized program parts. If you are unfamiliar with the JIT compiler built into PyPy, we highly recommend reading the docs for it.

Security Warning

It is discouraged to run the query API on a remote server. As soon as the query parameter (*-q*) is parameterized, arbitrary code execution can be performed. Note that this is fine as long one can trust the user.

Basic Usage

Let's go ahead and inspect the *example.py* program in this repository. It is assumed that the reader setup vmprof for pypy already (e.g. in a virtualenv).

Now run the following command to generate the log:

```
# run your program and output the log
pypy -m vmprof -o log.jit example.py
```

This generates the file that normally is sent to vmprof.com whenever *–web* is provided.

The query interface is a the flag '-q' which incooperates a small query language. Here is an example:

```
pypy -m jitlog log.jit -q 'bridges & op("int_add_ovf")'
... # will print the filtered traces
```

Query API

Brand new. Subject to change!

loops()
Filter: Reduces the output to loops only

bridges()
Filter: Reduces the output to bridges only

func(*name*)
Filter: Selects a trace if it happens to optimize the function containing the name.

op(*name*)
Filter: Only selects a traces if it contains the IR operation name.

CHAPTER 8

Data sent to vmprof.com

We only send the bare essentials to vmprof.com. This package is no spy software.

It includes the following data:

- The full command line
- The name of the interpreter used
- Filesystem path names, function names and line numbers of to your scripts
- Generic system information (Operating system, CPU word size, ...)

If jit log data is sent (–jitlog) on PyPy the following is also included:

- Meta data the JIT compiler produces. E.g. IR operations, Machine code
- Source code snippets: vmprof.com will receive source lines of your program. Only those are transmitted that ran often enough to trigger the JIT compiler to optimize your program.

Index