eng.uber.com

Pyflame: Uber Engineering's Ptracing Profiler for Python - Uber Engineering Blog

by Evan Klitzke

At Uber, we make an effort to write efficient backend services to keep our compute costs low. This becomes increasingly important as our business grows; seemingly small inefficiencies are greatly magnified at Uber's scale. We've found flame graphs to be an effective tool for understanding the CPU and memory characteristics of our services, and we've used them to great effect with our Go and JavaScript services. In order to get high quality flame graphs for Python services, we wrote a high-performance profiler called Pyflame, implemented in C++. In this article, we explore design considerations and some unique implementation characteristics that make Pyflame a better alternative for profiling Python code.

Deterministic Profilers

Python offers several built-in deterministic profilers via the profile and cProfile modules. The deterministic profilers in Python (profile and cProfile) work by using the sys.settrace() facility to install a trace function that's run at various points of interest, such as the start and end of each function and at the beginning of each logical line of code. This mechanism yields high-resolution profiling information, but it has a number of shortcomings.

High Overhead

The first drawback is its extremely high overhead: we commonly see it slowing down programs by 2x. Worse, we found this overhead to cause inaccurate profiling numbers in many cases. The cProfile module has difficulty accurately reporting timing statistics for methods that run very quickly because the profiler overhead itself is significant in those cases. Many engineers don't use profiling information because they can't trust its accuracy.

Lack of Full Call Stack Information

The second problem with the built-in deterministic profilers is that they don't record full call stack information. The built-in profiling modules only record information going up one stack level, which limits the usefulness of these modules. For example, when one decorator is applied to a large number of functions, the decorator frequently shows up in the callees and callers sections of the profiling output, with the true call information obscured due to the flattened call stack information. This clutter makes it difficult to understand true callee and caller information.

Lack of Services Written for Profiling

Finally, the built-in deterministic profilers require that the code be explicitly instrumented for profiling. A common problem for us is that many services weren't written with profiling in mind. Under high load, we may encounter serious performance problems with the service and want to collect profiling information quickly. Since the code isn't already instrumented for profiling, there's no way to immediately start collecting profiling information. If the load is severe enough, we may need an engineer to write code to enable a deterministic profiler (typically by adding an RPC method to turn it on and another to dump profiling data). This code then needs to be reviewed, tested, and deployed. The whole cycle might take several hours, which is not fast enough for us.

Sampling Profilers

There are also a number of third-party sampling profilers for Python. These sampling profilers typically work by installing a POSIX interval timer, which periodically interrupts the process and runs a signal handler to record stack information. Sampling profilers sample the profiled process rather than deterministically collecting profiling information. This technique is effective because the sampling resolution can be dialed up or down. When the sampling resolution is high, the profiling data is more accurate but performance suffers. For instance, the sampling resolution can be set high to get detailed profiles with a correspondingly high amount of overhead, or it can be set low to get less detailed profiles with less overhead.

A few limitations come with sampling profilers. First, they typically come with high overhead because they're implemented in Python. Python itself is not fast, especially compared to C or C++. In fact, the cProfile deterministic profiler is implemented in C for this reason. With these sampling profilers, getting acceptable performance often means setting the timer frequency to something that is relatively coarse-grained.

The other limitation is that the code needs to be explicitly instrumented for profiling, just as with deterministic profilers. Therefore, existing sampling profilers lead to the same problem as before: under high load, we want to profile some code, only to realize we have to rewrite it first.

Pyflame to the Rescue

With Pyflame, we wanted to maintain all of the possible profiling benefits:

- Collect the full Python stack, all the way to its root
- Emit data in a format that could be used to generate a flame graph
- Have low overhead

- Work with processes not explicitly instrumented for profiling

More importantly, we aimed to avoid all existing limitations. It might sound impossible to ask for all of the features without making any sacrifices. But it's not as impossible as it sounds!

Using ptrace for Python Profiling

Most Unix systems implement a special process trace system call called ptrace(2). ptrace is not part of the POSIX specification, but Unix implementations like BSD, OS X, and Linux all provide a ptrace implementation that allows a process to read and write to arbitrary virtual memory addresses, read and write CPU registers, deliver signals, etc. If you've ever used a debugger like GDB, then you've used software that's implemented using ptrace.

It's possible to use ptrace to implement a Python profiler. The idea is to periodically ptrace *attach* to the process, use the memory *peeking* routines to get the Python stack trace, and then *detach* from the process. Specifically with Linux ptrace, a profiler can be written using the request types PTRACE_ATTACH, PTRACE_PEEKDATA, and PTRACE_DETACH. In theory, this is pretty straightforward. In practice, it's complicated by the fact that recovering the stack trace using only the PTRACE_PEEKDATA request is very low-level and unintuitive.

First, we'll briefly cover how the PTRACE_PEEKDATA request works on Linux. This request type reads data at a virtual memory address in the traced process. The signature of the ptrace system call on Linux looks like this:

```
long ptrace(enum __ptrace_request request, pid_t pid, void *addr, void *data);
```

When using PTRACE_PEEKDATA, the following function arguments are supplied:

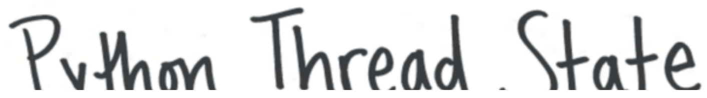
Parameter	Value
request	PTRACE_PEEKDATA
pid	The traced process ID
addr	The memory address to read
data	Unused (NULL by convention)

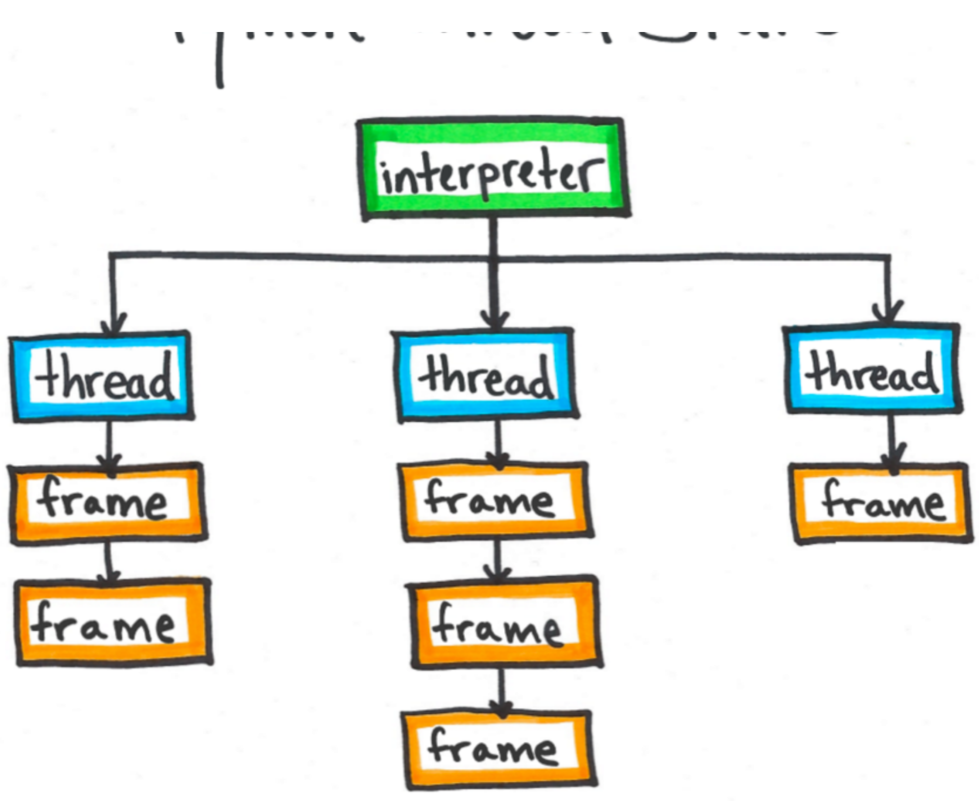
The value ptrace(2) returns is the long at that memory address. On Linux with GCC, the long type is defined to be the same as the native architecture word size, so on a 32-bit system the return value is a signed 32-bit integer, and on a 64-bit system the return value is a signed 64-bit integer.

There is one additional complication here. On error, ptrace(2) returns the value -1 and sets errno appropriately. However, the data at the address we’re reading could actually contain the value -1. Therefore, a return value of -1 is ambiguous: was there an error, or did that memory address really contain -1? To resolve this ambiguity when reading data, we must first clear errno and then make the ptrace request. Then, if the return value is -1, we check to see if errno was set during the ptrace call. Curiously, the ambiguity in the interpretation of the return value is an artifact of the GNU libc wrapper. The underlying system call on Linux uses the return value to signal an error, and it stores the peeked data into the data field, which must be supplied in this case.

Extracting the Thread State

Internally, Python is structured with one or more independent interpreters, and each sub-interpreter tracks one or more threads. Due to the global interpreter lock, only one thread actually runs at any given time. The currently executing thread information is held in a global variable named _PyThreadState_Current, which is normally not exported by the Python C API. From this variable, Pyflame can find the current frame object. From the current frame, the entire stack trace can be unwound. Therefore, once Pyflame locates the memory location of _PyThreadState_Current, it can recover the rest of the stack information by using PTRACE_PEEKDATA, as described above. Pyflame follows the thread state pointer to a frame object, and each frame object has a back pointer to another frame. The final frame has a back pointer to NULL. Each frame object holds fields which can be used to recover the filename, line number, and function name for the frame.





Each Python interpreter tracks one or more thread state objects, and each thread state has a pointer to a linked list of frames representing the call stack for that thread. The _PyThreadState_Current symbol points to the active thread.

The most difficult part of this is actually locating the address of _PyThreadState_Current. Depending on how the Python interpreter was compiled, there are two possibilities:

- In the default build mode, _PyThreadState_Current is a regular symbol with a well-known address in the text area that does not change. While the address doesn't change, the actual value for the address depends on what compiler is used, what compilation flags are used, etc.

- When Python is compiled with –enable-shared, the _PyThreadState_Current symbol is not built into Python itself but in a dynamic library. In this case, address space layout randomization (ASLR) means that the virtual memory address is different every time the interpreter runs.

In either case on Linux, the symbol can be located by parsing the ELF information from the interpreter (or from libpython in a dynamic build). Linux systems include a header file called elf.h that has the necessary definitions to parse an ELF file. Pyflame memory maps the file and then uses these ELF struct definitions to parse out the relevant ELF structures. If the special ELF .dynamic section indicates that the build links against libpython, then Pyflame proceeds to parse that file. Next, it locates the _PyThreadState_Current symbol in the .dynsym ELF section, either from the Python executable itself or from libpython, depending on the build mode.

For dynamic Python builds, the address of _PyThreadState_Current has to be augmented with the ASLR offset. This is done by reading /proc/PID/maps to get the virtual memory mapping offsets for the process. The offset from this file is added to the value read from libpython to get the true virtual memory address for the symbol.

Interpreting Frame Data

In the source code for the Python interpreter, you see regular C syntax for dereferencing pointers and accessing struct fields:

```
// frame has type void*
void *f_code = (struct _frame*)frame->f_code;

void *co_filename = (PyCodeObject*)f_code->co_filename;
```

Instead, Pyflame has to use ptrace to read from the Python process's virtual memory space and manually implement pointer dereferencing. The following is a representative snippet of code from Pyflame that emulates the code in the previous code listing:

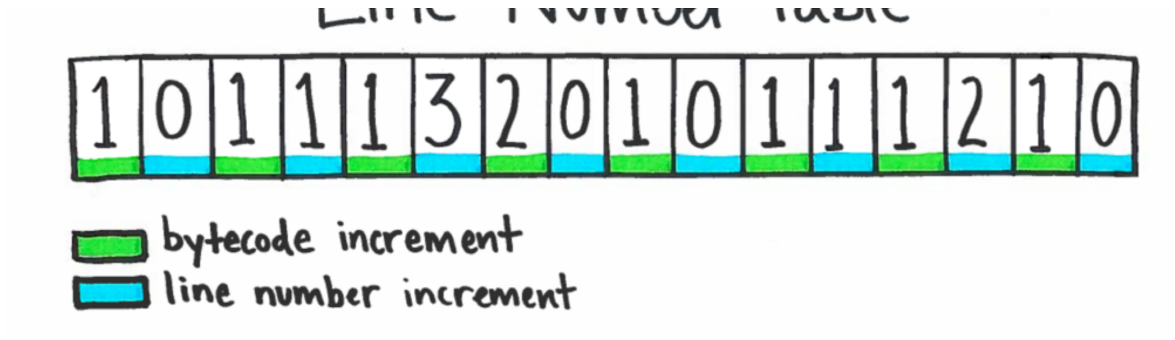
```
const long f_code = PtracePeek(pid, frame + offsetof<(_frame,
f_code));
const long co_filename =

    PtracePeek(pid, f_code + offsetof(PyCodeObject, co_filename));
```

Here, a helper method called PtracePeek() implements the call to ptrace with the PTRACE_PEEKDATA parameter and handles error checking. Pointers are represented as unsigned longs, and the offsetof macro is used to compute struct offsets. The ptrace code in Pyflame is more verbose than regular C code, but the logical structure of the two code listings is exactly the same.

The code to actually extract filenames and line numbers is interesting. Python 2 stores filenames using a type called PyStringObject, which simply stores the string data inline (at a fixed offset from the head of the struct). Python 3 has much more complicated string handling due to the internal unification of the string type and unicode types. For strings that contain only ASCII data, the raw string data can be found inline in the struct in much the same way. Pyflame currently only supports all ASCII filenames on Python 3.

Implementing the line number decoding for Pyflame was one of the more challenging parts of developing Pyflame. Python stores the line number data in an interesting data structure called the "line number table," in a field in the code object called f_lnotab. There's a file called lnotab_notes.txt in the Python source code that explains the exact data structure. First, know that the Python interpreter works by translating regular Python code to a lower-level bytecode representation. Typically, one line of Python code expands to many bytecode instructions. Bytecode instructions therefore typically advance much more quickly than lines of code. Instead of storing and updating a line number field in each frame, the Python interpreter uses a compressed data structure that associates bytecode offsets to line number offsets. The bytecode-to-line-number data structure is computed once for each code object. The line number can be computed implicitly for any bytecode instruction.



The line number table is an array with interleaved bytecode and line number increments. The line number for a given bytecode address is computed by keeping a running sum of both bytecode increments and line number increments, and stopping at the desired address.

Profiling Dockerized Services/Containers

At Uber, we run most of our services in Linux containers using Docker. One of the interesting challenges of building Pyflame was making it work with Linux containers. Typically, processes on the host cannot interact with containerized processes. However, in most cases the root user can ptrace containerized processes, and this is how we run Pyflame in production at Uber.

Docker containers use mount namespaces to isolate filesystem resources between the host and the container. Pyflame has to access files inside the container to access the correct ELF file and compute symbol offsets. Pyflame enters the container's mount namespace using the setns(2) system call. First, Pyflame compares /proc/self/ns/fs to /proc/PID/ns/fs. If they differ, Pyflame enters the process's mount namespace by calling open(2) on /proc/PID/ns/fs and then calling setns(2) on the resulting file descriptor. By retaining an open file descriptor to the original /proc/self/ns/fs, Pyflame can subsequently return to its original namespace (i.e., escape the container).

Like What You're Reading? Try Pyflame Yourself!

We've found Pyflame to be an extremely useful tool for profiling Python code at Uber and finding inefficient code paths to optimize. We're releasing Pyflame today as free software, under the Apache 2.0 license. Please try it out and let us know if you find any bugs. And, as always, we love getting pull requests, so please send them if you have improvements.

Evan Klitzke is a staff software engineer within Uber Engineering's core infrastructure group. He joined Uber just over 4 years ago in September 2012.

Photo Credits for Header: "Flame" by Liz West, licensed under CC-BY 2.0. Image cropped for header dimensions.

Like what you're reading? Sign up for our newsletter for updates from the Uber Engineering blog.