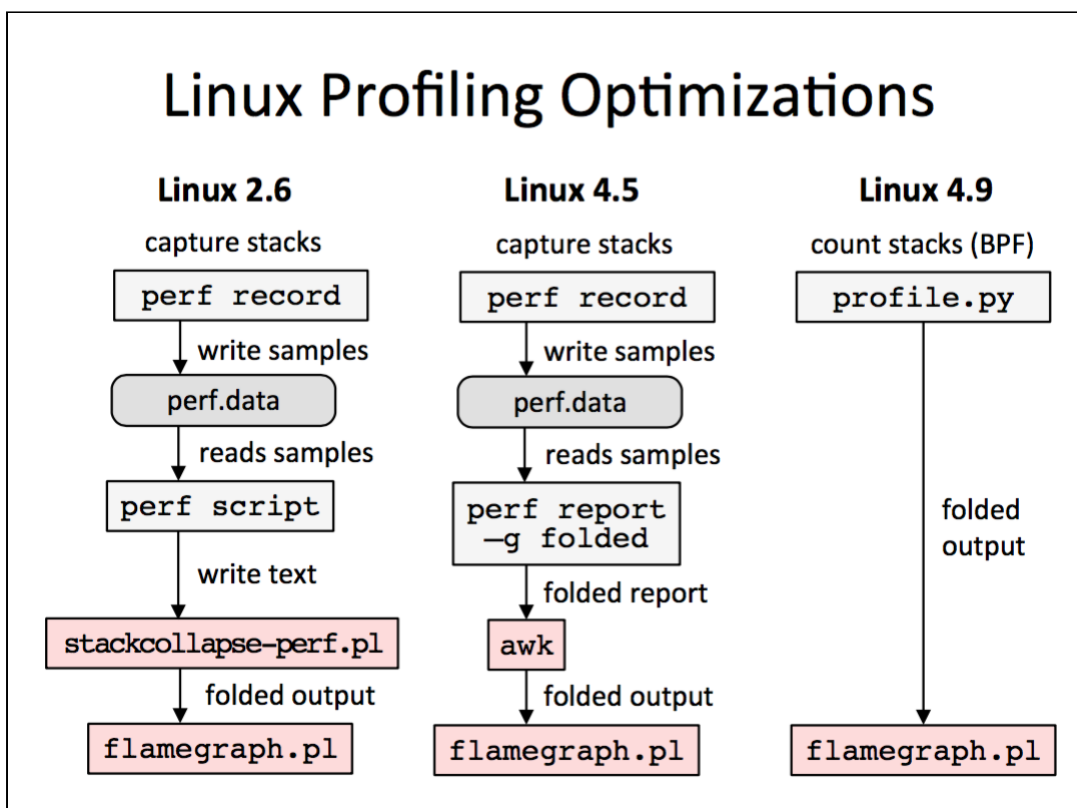


Linux 4.9's Efficient BPF-based Profiler

21 Oct 2016

BPF-optimized profiling arrived in Linux 4.9-rc1 ([patchset](#)), allowing the kernel to profile via timed sampling and summarize stack traces. Here's how I explained it recently in a talk alongside other prior perf methods for CPU [flame_graph](#) generation:



Linux 4.9 skips needing the perf.data file entirely, and its associated overheads. I wrote about this as a missing BPF feature [in March](#). It is now done.

In detail:

- **Linux 2.6:** `perf record` dumps all stack samples to a binary `perf.data` file for post-processing in user space. This is copied via a ring buffer, and `perf` wakes up an optimal number of times to read that buffer, so this has already been optimized somewhat. With low frequency sampling (<50 Hertz), it would be hard to measure the runtime overhead. The post-processing, however, could take up to several seconds of CPU time (and some disk I/O for `perf.data`) for a busy application and many CPUs.
- **Linux 4.5:** The prior use of `stackcollapse-perf.pl` was to frequency count stack traces, but really, the `perf` command has this capability, it just couldn't emit output in the folded format (stack traces on single lines, with frames delimited by ";"). Linux 4.5 added a `"-g folded"` option to `perf report`, making this workflow more efficient. I blogged about this [previously](#).
- **Linux 4.9:** BPF was enhanced (eBPF) to attach to `perf_events`, which are created by `perf_event_open()` with a `PERF_EVENT_IOC_SET_BPF` ioctl() to attach a BPF program. Voilà! Timed samples can now run BPF programs. BPF already had the capability to walk stack traces and frequency count them, added in Linux 4.6, we just needed this capability to attach it to timed samples.

I developed profile.py for [bcc tools](#) as a BPF profiler (see [example output](#)), which uses this new Linux 4.9 support. Here's a truncated screenshot:

```
# ./profile.py 10
Sampling at 49 Hertz of all threads by user + kernel stack for 10 secs.
[...]
ffffff81414025 copy_user_enhanced_fast_string
ffffff817b0774 tcp_sendmsg
ffffff817dd145 inet_sendmsg
ffffff8173ea48 sock_sendmsg
ffffff8173eae5 sock_write_iter
ffffff81232db3 __vfs_write
ffffff81233438 vfs_write
ffffff812348a5 sys_write
ffffff818737bb entry_SYSCALL_64_fastpath
00007fae0a2614fd [unknown]
000000000020000 [unknown]
- iperf (21262)
    132

ffffff81414025 copy_user_enhanced_fast_string
ffffff8174e76b skb_copy_datagram_iter
ffffff817addb3 tcp_recvmsg
ffffff817dd0ae inet_recvmsg
ffffff8173e65d sock_recvmsg
ffffff8173e89a SYSC_recvfrom
ffffff8173fb9e sys_recvfrom
ffffff818737bb entry_SYSCALL_64_fastpath
00007f9c16cf58bf recv
- iperf (21266)
    200
```

The output are stack traces, process details, and a single number: the number of times this stack trace was sampled. The entire stack trace is used as a key in an in-kernel hash, so that only the summary is emitted to user space.

You can use my profile tool to output folded format directly, for flame graph generation. Here I'm also including -d, for stack delimiters, which the flame graph software will color grey:

```
# ./profile.py -df 10
[...]
iperf;[unknown];[unknown];-;entry_SYSCALL_64_fastpath;sys_write;vfs_write;__vfs_write;sock_write;
iperf;recv;-;entry_SYSCALL_64_fastpath;sys_recvfrom;SYSC_recvfrom;sock_recvmsg;inet_recvmsg;tcp_r
```

This can be read directly by flamegraph.pl -- it doesn't need stackcollapse-perf.pl (as pictured in the slide).

Here's the USAGE message for profile, which shows other capabilities:

```

# ./profile.py -h
usage: profile.py [-h] [-p PID] [-U | -K] [-F FREQUENCY] [-d] [-a] [-f]
                  [--stack-storage-size STACK_STORAGE_SIZE]
                  [duration]

Profile CPU stack traces at a timed interval

positional arguments:
  duration              duration of trace, in seconds

optional arguments:
  -h, --help            show this help message and exit
  -p PID, --pid PID     profile this PID only
  -U, --user-stacks-only
                        show stacks from user space only (no kernel space
                        stacks)
  -K, --kernel-stacks-only
                        show stacks from kernel space only (no user space
                        stacks)
  -F FREQUENCY, --frequency FREQUENCY
                        sample frequency, Hertz (default 49)
  -d, --delimited       insert delimiter between kernel/user stacks
  -a, --annotations     add _[k] annotations to kernel frames
  -f, --folded          output folded format, one line per stack (for flame
                        graphs)
  --stack-storage-size STACK_STORAGE_SIZE
                        the number of unique stack traces that can be stored
                        and displayed (default 2048)

examples:
  ./profile             # profile stack traces at 49 Hertz until Ctrl-C
  ./profile -F 99       # profile stack traces at 99 Hertz
  ./profile 5           # profile at 49 Hertz for 5 seconds only
  ./profile -f 5        # output in folded format for flame graphs
  ./profile -p 185      # only profile threads for PID 185
  ./profile -U          # only show user space stacks (no kernel)
  ./profile -K          # only show kernel space stacks (no user)

```

I have an older version of this tool in the `bcc tools/old` directory, which works on Linux 4.6 to 4.8 using an old kprobe trick, although one that had caveats.

Thanks to Alexei Starovoitov (Facebook) for getting the kernel BPF profiling support done, and Teng Qin (Facebook) for adding bcc support yesterday.

We could have developed BPF profiling sooner, but since the 2.6 style of profiling was somewhat sufficient, BPF effort has been on other higher priority areas. But now that BPF profiling has arrived, it means more than just saving CPU (and disk I/O) resources: imagine real-time flame graphs!

2 Comments

Brendan Gregg's Blog

 Login ▾ Recommend 7 Tweet Share

Sort by Best ▾

Zorba Grecu • 2 years ago

Hi,



was happy to discover this new feature and tested it. Was kind of hoping to have the off-cpu times included in the output. Tried the offcputime tool as well, but no success. Well what I would love to achieve is to have a simple tool that answers the question: which function spends more time including any system call. I wrote a simple program with main that starts several heavy threads. Main then waits on pthread_join(). for the threads to terminate. Unfortunately seems that none of the tools would be able to "interrupt" main thread that is blocked on a long pthread_join() (futex syscall).

So the question is .. is it possible to achive with bpf/bcc to write a tool that would interrupt all threads and see their stack, whatever if they are running or blocked on a kernel call.

Thanks a lot (and thanks for all your work)

 |  • Share ›**Mukul Sabharwal** • 2 years ago

Any chance this will be made available to non-root users if they want to attach a program only to threads of their own PID? Haven't looked at this too deeply so my answer may be in the code, but how does the stack walk happen: ebp or unwind info? or can the user specify which?

 |  • Share › Subscribe  Add Disqus to your siteAdd DisqusAdd  Disqus' Privacy PolicyPrivacy PolicyPrivacy Policy

You can comment here, but I can't guarantee your comment will remain here forever: I might switch comment systems at some point (eg, if disqus add advertisements).