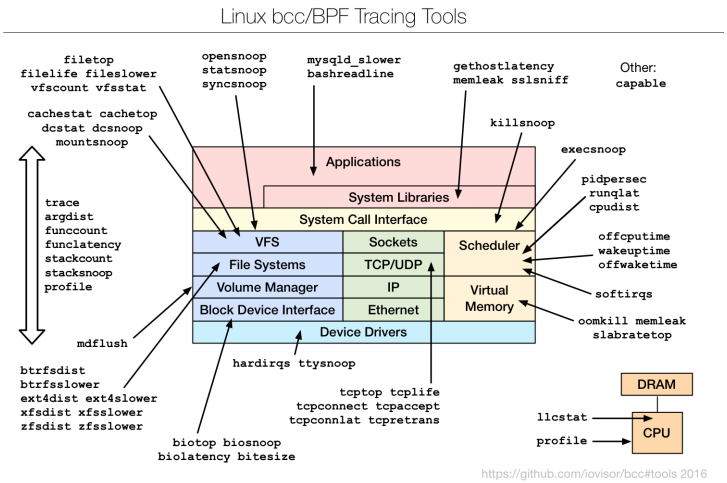
DTrace for Linux 2016

27 Oct 2016



With the final major capability for BPF tracing (timed sampling) merging in Linux 4.9-rc1, the Linux kernel now has raw capabilities similar to those provided by DTrace, the advanced tracer from Solaris. As a long time DTrace user and expert, this is an exciting milestone! On Linux, you can now analyze the performance of applications and the kernel using production-safe low-overhead custom tracing, with latency histograms, frequency counts, and more.

There have been many tracing projects for Linux, but the technology that finally merged didn't start out as a tracing project at all: it began as enhancements to Berkeley Packet Filter (BPF), aka eBPF. At first, these enhancements let BPF redirect packets to create software-defined networks. Later on, support for tracing events was added, enabling programmatic tracing in Linux.

While BPF currently lacks a high-level language like DTrace, the front-ends available have been enough for me to create many BPF tools, some based on my older DTraceToolkit. In this post I'll describe how you can use these tools, the front-ends available, and discuss where the technology is going next.

Screenshots

I've been adding BPF-based tracing tools to the open source bcc project (thanks to Brenden Blanco, of PLUMgrid, for leading bcc development). See the bcc install instructions. It will add a collection of tools under /usr/share/bcc/tools, including the following.

Tracing new processes:

```
# execsnoop
PCOMM            PID    RET ARGS
bash             15887    0 /usr/bin/man ls
preconv          15894    0 /usr/bin/preconv -e UTF-8
man              15896    0 /usr/bin/tbl
man              15897    0 /usr/bin/nroff -mandoc -rLL=169n -rLT=169n -Tutf8
man              15898    0 /usr/bin/pager -s
nroff            15900    0 /usr/bin/locale charmap
nroff            15901    0 /usr/bin/groff -mtty-char -Tutf8 -mandoc -rLL=169n -rLT=169n
groff            15902    0 /usr/bin/troff -mtty-char -mandoc -rLL=169n -rLT=169n -Tutf8
groff            15903    0 /usr/bin/grotty
```

Histogram of disk I/O latency:

```
# biolatency -m
Tracing block device I/O... Hit Ctrl-C to end.
^C
     msecs               : count     distribution
       0 -> 1            : 96       |************************************|
       2 -> 3            : 25       |*********                           |
       4 -> 7            : 29       |***********                         |
       8 -> 15           : 62       |************************            |
      16 -> 31           : 100      |****************************************|
      32 -> 63           : 62       |************************            |
      64 -> 127          : 18       |******                              |
```

Tracing common ext4 operations slower than 5 milliseconds:

```
# ext4slower 5
Tracing ext4 operations slower than 5 ms
TIME     COMM            PID    T BYTES   OFF_KB   LAT(ms) FILENAME
21:49:45 supervise       3570   W 18      0           5.48 status.new
21:49:48 supervise       12770  R 128     0           7.55 run
21:49:48 run             12770  R 497     0          16.46 nsswitch.conf
21:49:48 run             12770  R 1680    0          17.42 netflix_environment.sh
21:49:48 run             12770  R 1079    0           9.53 service_functions.sh
21:49:48 run             12772  R 128     0          17.74 svstat
21:49:48 svstat          12772  R 18      0           8.67 status
21:49:48 run             12774  R 128     0          15.76 stat
21:49:48 run             12777  R 128     0           7.89 grep
21:49:48 run             12776  R 128     0           8.25 ps
21:49:48 run             12780  R 128     0          11.07 xargs
21:49:48 ps              12776  R 832     0          12.02 libprocps.so.4.0.0
21:49:48 run             12779  R 128     0          13.21 cut
[...]
```

Tracing new active TCP connections (connect()):

```
# tcpconnect
PID    COMM         IP SADDR            DADDR            DPORT
1479   telnet       4  127.0.0.1        127.0.0.1        23
1469   curl         4  10.201.219.236   54.245.105.25    80
1469   curl         4  10.201.219.236   54.67.101.145    80
1991   telnet       6  ::1              ::1              23
2015   ssh          6  fe80::2000:bff:fe82:3ac fe80::2000:bff:fe82:3ac 22
```

Tracing DNS latency by tracing getaddrinfo()/gethostbyname() library calls:

```
# gethostlatency
TIME      PID    COMM           LATms HOST
06:10:24  28011  wget           90.00 www.iovisor.org
06:10:28  28127  wget            0.00 www.iovisor.org
06:10:41  28404  wget            9.00 www.netflix.com
06:10:48  28544  curl           35.00 www.netflix.com.au
06:11:10  29054  curl           31.00 www.plumgrid.com
06:11:16  29195  curl            3.00 www.facebook.com
06:11:25  29404  curl           72.00 foo
06:11:28  29475  curl            1.00 foo
```

Interval summaries of VFS operations by type:

```
# vfsstat
TIME         READ/s  WRITE/s CREATE/s   OPEN/s  FSYNC/s
18:35:32:       231       12        4       98        0
18:35:33:       274       13        4      106        0
18:35:34:       586       86        4      251        0
18:35:35:       241       15        4       99        0
```

Tracing off-CPU time with kernel and user stack traces (summarized in kernel), for a given PID:

```
# offcputime -d -p 24347
Tracing off-CPU time (us) of PID 24347 by user + kernel stack... Hit Ctrl-C to end.
^C
[...]
    ffffffff810a9581 finish_task_switch
    ffffffff8185d385 schedule
    ffffffff81085672 do_wait
    ffffffff8108687b sys_wait4
    ffffffff81861bf6 entry_SYSCALL_64_fastpath
    --
    00007f6733a6b64a waitpid
    -                bash (24347)
        4952

    ffffffff810a9581 finish_task_switch
    ffffffff8185d385 schedule
    ffffffff81860c48 schedule_timeout
    ffffffff810c5672 wait_woken
    ffffffff8150715a n_tty_read
    ffffffff815010f2 tty_read
    ffffffff8122cd67 __vfs_read
    ffffffff8122df65 vfs_read
    ffffffff8122f465 sys_read
    ffffffff81861bf6 entry_SYSCALL_64_fastpath
    --
    00007f6733a969b0 read
    -                bash (24347)
        1450908
```

Tracing MySQL query latency (via a USDT probe):

```
# mysqld_qslower `pgrep -n mysqld`
Tracing MySQL server queries for PID 14371 slower than 1 ms...
TIME(s)        PID          MS QUERY
0.000000       18608   130.751 SELECT * FROM words WHERE word REGEXP '^bre.*n$'
2.921535       18608   130.590 SELECT * FROM words WHERE word REGEXP '^alex.*$'
4.603549       18608    24.164 SELECT COUNT(*) FROM words
9.733847       18608   130.936 SELECT count(*) AS count FROM words WHERE word REGEXP '^bre.*n$'
17.864776      18608   130.298 SELECT * FROM words WHERE word REGEXP '^bre.*n$' ORDER BY word
```

Using the trace multi-tool to watch login requests, by instrumenting the pam library:

```
# trace 'pam:pam_start "%s: %s", arg1, arg2'
TIME     PID    COMM         FUNC             -
17:49:45 5558   sshd         pam_start        sshd: root
17:49:47 5662   sudo         pam_start        sudo: root
17:49:49 5727   login        pam_start        login: bgregg
```

Many tools have usage messages (-h), and all should have man pages and text files of example output in the bcc project.

Out of necessity

In 2014, Linux tracing had some kernel summary features (from ftrace and perf_events), but outside those we still had to dump-and-post-process data – a decades old technique that has high overhead at scale. You couldn't frequency count process names, function names, stack traces, or other arbitrary data in the kernel. You couldn't save variables in one probe event, and then retrieve them in another, which meant that you couldn't measure latency (or time deltas) in custom places, and you couldn't create in-kernel latency histograms. You couldn't trace USDT probes. You couldn't even write custom programs. DTrace could do all these, but only on Solaris or BSD. On Linux, some out-of-tree tracers like SystemTap could serve these needs, but brought their own challenges. (For the sake of completeness: yes, you *could* write kprobe-based kernel modules – but practically no one did.)

In 2014 I joined the Netflix cloud performance team. Having spent years as a DTrace expert, it might have seemed crazy for me to move to Linux. But I had some motivations, in particular seeking a greater challenge: performance tuning the Netflix cloud, with its rapid application changes, microservice architecture, and distributed systems. Sometimes this job involves systems tracing, for which I'd previously used DTrace. Without DTrace on Linux, I began by using what was built in to the Linux kernel, ftrace and perf_events, and from them made a toolkit of tracing tools (perf-tools). They have been invaluable. But I couldn't do some tasks, particularly latency histograms and stack trace counting. We needed kernel tracing to be programmatic.

What happened?

BPF adds programmatic capabilities to the existing kernel tracing facilities (tracepoints, kprobes, uprobes). It has been enhanced rapidly in the Linux 4.x series.

Timed sampling was the final major piece, and it landed in Linux 4.9-rc1 (patchset). Many thanks to Alexei Starovoitov (now working on BPF at Facebook), the lead developer behind these BPF enhancements.

The Linux kernel now has the following features built in (added between 2.6 and 4.9):

- Dynamic tracing, kernel-level (BPF support for kprobes)
- Dynamic tracing, user-level (BPF support for uprobes)
- Static tracing, kernel-level (BPF support for tracepoints)
- Timed sampling events (BPF with perf_event_open)
- PMC events (BPF with perf_event_open)
- Filtering (via BPF programs)
- Debug output (bpf_trace_printk())
- Per-event output (bpf_perf_event_output())
- Basic variables (global & per-thread variables, via BPF maps)
- Associative arrays (via BPF maps)
- Frequency counting (via BPF maps)
- Histograms (power-of-2, linear, and custom, via BPF maps)
- Timestamps and time deltas (bpf_ktime_get_ns(), and BPF programs)
- Stack traces, kernel (BPF stackmap)
- Stack traces, user (BPF stackmap)
- Overwrite ring buffers (perf_event_attr.write_backward)

The front-end we are using is bcc, which provides both Python and lua interfaces. bcc adds:

- Static tracing, user-level (USDT probes via uprobes)
- Debug output (Python with BPF.trace_pipe() and BPF.trace_fields())
- Per-event output (BPF_PERF_OUTPUT macro and BPF.open_perf_buffer())
- Interval output (BPF.get_table() and table.clear())
- Histogram printing (table.print_log2_hist())
- C struct navigation, kernel-level (bcc rewriter maps to bpf_probe_read())
- Symbol resolution, kernel-level (ksym(), ksymaddr())
- Symbol resolution, user-level (usymaddr())
- BPF tracepoint support (via TRACEPOINT_PROBE)
- BPF stack trace support (incl. walk method for stack frames)
- Various other helper macros and functions
- Examples (under /examples)
- Many tools (under /tools)
- Tutorials (/docs/tutorial*.md)
- Reference guide (/docs/reference_guide.md)

I'd been holding off on this post until the last major feature was integrated, and now it has been in 4.9-rc1. There are still some minor missing things we have workarounds for, and additional things we might do, but what we have right now is worth celebrating. Linux now has advanced tracing capabilities built in.

Safety

BPF and its enhancements are designed to be production safe, and it is used today in large scale production environments. But if you're determined, you may be able to still find a way to hang the kernel. That experience should be the exception rather than the rule, and such bugs will be fixed fast, especially since BPF is part of Linux. All eyes are on Linux.

We did hit a couple of non-BPF bugs during development that needed to be fixed: rcu not reentrant, which could cause kernel hangs for funccount and was fixed by the "bpf: map pre-alloc" patchset in 4.6, and with a workaround in bcc for older kernels. And a uprobe memory accounting issue, which failed uprobe allocations, and was fixed by the "uprobes: Fix the memcg accounting" patch in 4.8 and backported to earlier kernels (eg, it's in the current 4.4.27 and 4.4.0-45.66).

Why did Linux tracing take so long?

Prior work had been split among several other tracers: there was never a consolidated effort on any single one. For more about this and other issues, see my 2014 tracing summit talk. One thing I didn't note there was the counter effect of partial solutions: some companies had found another tracer (SystemTap or LTTng) was sufficient for their specific needs, and while they have been happy to hear about BPF, contributing to its development wasn't a priority given their existing solution.

BPF has only been enhanced to do tracing in the last two years. This process could have gone faster, but early on there were zero full-time engineers working on BPF tracing. Alexei Starovoitov (BPF lead), Brenden Blanco (bcc lead), myself, and others, all had other priorities. I tracked my hours on this at Netflix (voluntarily), and I've spent around 7% of my time on BPF/bcc. It wasn't that much of a priority, in part because we had our own workarounds (including my perf-tools, which work on older kernels).

Now that BPF tracing has arrived, there's already tech companies on the lookout for BPF skills. I can still highly recommend Netflix. (If you're trying to hire *me* for BPF skills, then I'm still very happy at Netflix!.)

Ease of use

What might appear to be the largest remaining difference between DTrace and bcc/BPF is ease of use. But it depends on what you're doing with BPF tracing. Either you are:

- **Using BPF tools/metrics**: There should be no difference. Tools behave the same, GUIs can access similar metrics. Most people will use BPF in this way.
- **Developing tools/metrics**: bcc right now is much harder. DTrace has its own concise language, D, similar to awk, whereas bcc uses existing languages (C and Python or lua) with libraries. A bcc tool in C+Python may be a *lot* more code than a D-only tool: 10x the lines, or more. However, many DTrace tools used shell wrapping to provide arguments and error checking, inflating the code to a much bigger size. The coding difficulty is also different: the rewriter in bcc can get fiddly, which makes some scripts much more complicated to develop (extra bpf_probe_read()s, requiring more knowledge of BPF internals). This situation should improve over time as improvements are planned.
- **Running common one-liners**: Fairly similar. DTrace could do many with the "dtrace" command, whereas bcc has a variety of multitools: trace, argdist, funccount, funclatency, etc.
- **Writing custom ad hoc one-liners**: With DTrace this was trivial, and accelerated advanced analysis by allowing rapid custom questions to be posed and answered by the system. bcc is currently limited by its multitools and their scope.

In short, if you're an end user of BPF tools, you shouldn't notice these differences. If you're an advanced user and tool developer (like me), bcc is a lot more difficult right now.

To show a current example of the bcc Python front-end, here's the code for tracing disk I/O and printing I/O size as a histogram:

```
from bcc import BPF
from time import sleep

# load BPF program
b = BPF(text="""
#include <uapi/linux/ptrace.h>
#include <linux/blkdev.h>

BPF_HISTOGRAM(dist);

int kprobe__blk_account_io_completion(struct pt_regs *ctx, struct request *req)
{
    dist.increment(bpf_log2l(req->__data_len / 1024));
    return 0;
}
""")

# header
print("Tracing... Hit Ctrl-C to end.")

# trace until Ctrl-C
try:
    sleep(99999999)
except KeyboardInterrupt:
    print

# output
b["dist"].print_log2_hist("kbytes")
```

Note the embedded C (text=) in the Python code.

This gets the job done, but there's also room for improvement. Fortunately, we have time to do so: it will take many months before people are on Linux 4.9 and can use BPF, so we have time to create tools and front-ends.

A higher-level language

An easier front-end, such as a higher-level language, may not improve adoption as much as you might imagine. Most people will use the canned tools (and GUIs), and only some of us will actually write them. But I'm not opposed to a higher-level language either, and some already exist, like SystemTap:

```
#!/usr/bin/stap
/*
 * opensnoop.stp    Trace file open()s.  Basic version of opensnoop.
 */

probe begin
{
    printf("\n%6s %6s %16s %s\n", "UID", "PID", "COMM", "PATH");
}

probe syscall.open
{
    printf("%6d %6d %16s %s\n", uid(), pid(), execname(), filename);
}
```

Wouldn't it be nice if we could have the SystemTap front-end with all its language integration and tapsets, with the high-performance kernel built in BPF back-end? Richard Henderson of Red Hat has already begun work on this, and has released an initial version!

There's also ply, an entirely new higher-level language for BPF:

```
#!/usr/bin/env ply

kprobe:SyS_*
{
    $syscalls[func].count()
}
```

This is also promising.

Although, I think the real challenge for tool developers won't be the language: it will be knowing what to do with these new superpowers.

How you can contribute

- **Promotion**: There are currently no marketing efforts for BPF tracing. Some companies know it and are using it (Facebook, Netflix, Github, and more), but it'll take years to become widely known. You can help by sharing articles and resources with others in the industry.
- **Education**: You can write articles, give meetup talks, and contribute to bcc documentation. Share case studies of how BPF has solved real issues, and provided value to your company.
- **Fix bcc issues**: See the bcc issue list, which includes bugs and feature requests.
- **File bugs**: Use bcc/BPF, and file bugs as you find them.
- **New tools**: There are more observability tools to develop, but please don't be hasty: people are going to spend hours learning and using your tool, so make it as intuitive and excellent as possible (see my docs). As Mike Muuss has said about his ping program: "If I'd known then that it would be my most famous accomplishment in life, I might have worked on it another day or two and added some more options."
- **High-level language**: If the existing bcc front-end languages really bother you, maybe you can come up with something much better. If you build it in bcc you can leverage libbcc. Or, you could help the SystemTap BPF or ply efforts.
- **GUI integration**: Apart from the bcc CLI observability tools, how can this new information be visualized? Latency heat maps, flame graphs, and more.

Other Tracers

What about SystemTap, ktap, sysdig, LTTng, etc? It's possible that they all have a future, either by using BPF, or by becoming better at what they specifically do. Explaining each will be a blog post by itself.

And DTrace itself? We're still using it at Netflix, on our FreeBSD-based CDN.

Further bcc/BPF Reading

I've written a bcc/BPF Tool End-User Tutorial, a bcc Python Developer's Tutorial, a bcc/BPF Reference Guide, and contributed useful /tools, each with an example.txt file and man page. My prior posts about bcc & BPF include:

- eBPF: One Small Step (we later just called it BPF)
- bcc: Taming Linux 4.3+ Tracing Superpowers
- Linux eBPF Stack Trace Hack (stack traces are now officially supported)
- Linux eBPF Off-CPU Flame Graph (" ")
- Linux Wakeup and Off-Wake Profiling (" ")
- Linux Chain Graph Prototype (" ")
- Linux eBPF/bcc uprobes
- Linux BPF Superpowers
- Ubuntu Xenial bcc/BPF
- Linux bcc Tracing Security Capabilities
- Linux MySQL Slow Query Tracing with bcc/BPF
- Linux bcc ext4 Latency Tracing
- Linux bcc/BPF Run Queue (Scheduler) Latency
- Linux bcc/BPF Node.js USDT Tracing
- Linux bcc tcptop
- Linux 4.9's Efficient BPF-based Profiler

I've also giving a talk about bcc/BPF, at Facebook's Performance@Scale event: Linux BPF Superpowers. In December, I'm giving a tutorial and talk on BPF/bcc at USENIX LISA in Boston.

(Update: I also have a new website, Linux eBPF Tools.)

Acknowledgements

- Van Jacobson and Steve McCanne, who created the original BPF as a packet filter.

- Barton P. Miller, Jeffrey K. Hollingsworth, and Jon Cargille, for inventing dynamic tracing, and publishing the paper: "Dynamic Program Instrumentation for Scalable Performance Tools", Scalable High-performance Conputing Conference (SHPCC), Knoxville, Tennessee, May 1994.
- kerninst (ParaDyn, UW-Madison), an early dynamic tracing tool that showed the value of dynamic tracing (late 1990's).
- Mathieu Desnoyers (of LTTng), the lead developer of kernel markers that led to tracepoints.
- IBM developed kprobes as part of DProbes. DProbes was combined with LTT to provide Linux dynamic tracing in 2000, but wasn't integrated.
- Bryan Cantrill, Mike Shapiro, and Adam Leventhal (Sun Microsystems), the core developers of DTrace, an awesome tool which proved that dynamic tracing could be production safe and easy to use (2004). Given the mechanics of dynamic tracing, this was a crucial turning point for the technology: that it became safe enough to be shipped *by default in Solaris*, an OS known for reliability.
- The many Sun Microsystems staff in marketing, sales, training, and other roles, for promoting DTrace and creating the awareness and desire for advanced system tracing.
- Roland McGrath (at Red Hat), the lead developer of utrace, which became uprobes.
- Alexei Starovoitov (PLUMgrid, then Facebook), the lead developer of enhanced BPF: the programmatic kernel components necessary.
- Many other Linux kernel engineers who contributed feedback, code, testing, and their own patchsets for the development of enhanced BPF (search lkml for BPF): Wang Nan, Daniel Borkmann, David S. Miller, Peter Zijlstra, and many others.
- Brenden Blanco (PLUMgrid), the lead developer of bcc.
- Sasha Goldshtein (Sela) developed tracepoint support in bcc, developed the most powerful bcc multitools trace and argdist, and contributed to USDT support.
- Vicent Martí and others at Github engineering, for developing the lua front-end for bcc, and contributing parts of USDT.
- Allan McAleavy, Mark Drayton, and other bcc contributors for various improvements.

Thanks to Netflix for providing the environment and support where I've been able to contribute to BPF and bcc tracing, and help get them done. I've also contributed to tracing in general over the years by developing tracing tools (using TNF/prex, DTrace, SystemTap, ktap, ftrace, perf, and now bcc/BPF), and books, blogs, and talks.

Finally, thanks to Deirdré for editing another post.

Conclusion

Linux doesn't have DTrace (the language), but it now does, in a way, have the DTraceToolkit (the tools).

The Linux 4.9 kernel has the final capabilities needed to support modern tracing, via enhancments to its built-in BPF engine. The hardest part is now done: kernel support. Future work now includes more performance CLI tools, alternate higher-level languages, and GUIs.

For customers of performance analysis products, this is also good news: you can now ask for latency histograms and heatmaps, CPU and off-CPU flame graphs, better latency breakdowns, and lower-cost instrumentation. Per-packet tracing and processing in user space is now the old inefficient way.

So when are you going to upgrade to Linux 4.9? Once it is officially released, new performance tools await: apt-get install bcc-tools. For updates on bcc/BPF tools, see bcc (github) and my eBPF Tools page.

Enjoy!

Brendan

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).